

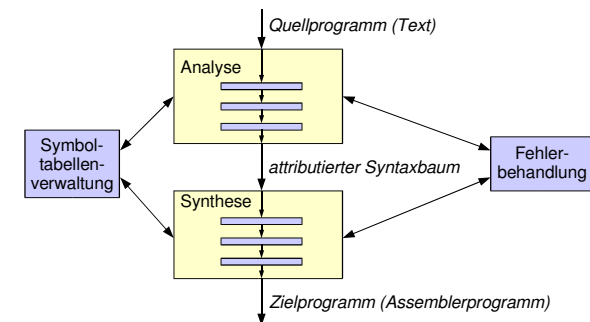
Wiederholung

Inhalte der Vorlesung

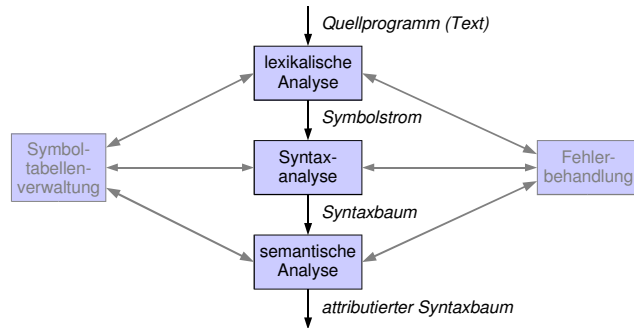
1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. Syntaxanalyse und der Parser-Generator yacc
6. Syntaxgesteuerte Übersetzung
7. Übersetzungssteuerung mit make

1. Einführung

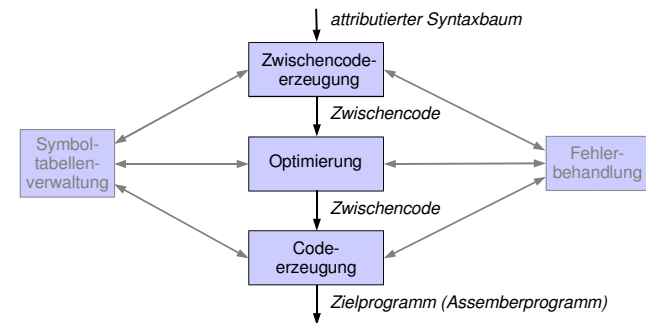
Die Phasen der Übersetzung



Die Analysephasen



Die Synthesephasen



2. Lexikalische Analyse

Lexikalische Analyse

position := initial + rate * 60

position := initial + rate * 60

Bezeichner := Bezeichner + Bezeichner * Zahl
(Zuweisungs- (Addit.- (Mult.-
 symbol) Symbol) Symbol)
 "position" "initial" "rate" 60

- Gruppierung der Eingabezeichen in Lexeme
 - Leerzeichen werden entfernt
- Zuordnung Lexem → Symbol
 - Symbole (Token): Bezeichner, :=, +, *, Zahl, ...
 - Symbol hat z.T. Wert als Attribut

Äquivalenz regulärer Ausdrücke und endlicher Automaten

- die Sprache jedes regulären Ausdrucks kann von einem endlichen Automaten erkannt werden
 - auch Umkehrung gilt:
die Sprache eines endlichen Automaten kann mit einem regulären Ausdruck beschrieben werden
 - Automat: im allgemeinen nicht deterministisch
- jeder nichtdeterministische endliche Automat kann in einen deterministischen übersetzt werden
- Folgerung:
für jeden regulären Ausdruck kann *automatisch* ein *effizienter* erkennender Automat erzeugt werden

Einfügen und Suchen von Bezeichnern

- Bezeichner werden als Zahl verschlüsselt
 - effizienter Gleichheitstest
 - Abbildungen als Felder implementierbar
 - effizient
- Verfahren: Streuspeicher (Hashing)
 - Tabelleneinträge willkürlich, aber deterministisch und möglichst gleichmäßig in „Eimer“ verteilen
 - Standardverfahren
 - siehe Standard-C-Library: `hcreate()`, `hsearch()`
 - siehe Vorlesung „Praktische Informatik 2“
 - siehe Drachenbuch, Kap 7.6

3. Der Textstrom-Editor sed

Einfache typische Anwendungen

- kleine Textmodifikation
 - Betreff in Vacation-Text einsetzen
- Formatierung einer Liste etwas ändern
 - festes Präfix in einer Dateiliste entfernen

Die Grundkommandos von sed

- **Syntax:**
`sed [-e Editierkommando] [-f Scriptdatei] [-n] [Datei ...]`
- **Editierkommando:**
Adresse Funktion
- **Funktion:**
`s / Ausdruck / Ersetzung / [g][p]` (substitute) → ersetzt auf Ausdruck passenden Text
`d` (delete) → löscht aktuelle Zeile, nichts wird gedruckt
`!` Funktion (not) → führt Funktion aus, falls Adresse nicht paßt
- **Adresse:**
(leer)
`/ Ausdruck /`
- **Ausdruck:**
ein regulärer Ausdruck wie bei grep
- **Modus:**
`g` (global) ersetzt alle passenden Texte, nicht nur den ersten
`p` (print) falls Ersetzung stattgefunden hat, drucke Zeile (trotz Option `-n`)

Typische Anwendungen von regulären Ausdrücken

- variable Textanteile erkennen
 - variables Präfix in einer Dateiliste entfernen
- interessante Textanteile extrahieren
 - Liste aller Benutzer aus `/etc/passwd` extrahieren
 - Liste aller „echten“ Benutzer aus `/etc/passwd` extrahieren

4. Der Scanner-Generator lex

Aufbau einer lex-Datei

Definitionen

`%%`

Regeln

`%%`

Unterprogramme } optional



Einsetzen des Benutzernamens: Lösung

- username.l:

```
%{
#include <stdio.h>
#include <stdlib.h>
}%
%option main
%%
"<username>" printf("%s", getenv("USER"));
```

-
- Regel: vorne Muster, hinten Aktion
 - Default-Regel: druckt aktuelles Zeichen
 - Definitionen:
 - C-Definitionen in %{, %}
 - lex-Option hinter %option

Kommunikation mit einem Parser

- Unterbrechung der Scan-Schleife:
`return n;`
 - Nummer des Symbols in n
 - weitere Informationen in globalen Variablen (`ytext, ...`)
- Fortsetzung der Scan-Schleife:
`yylex();`
- Symbolnummern in gemeinsamer Header-Datei

Demo

5. Syntaxanalyse und der Parser-Generator yacc

Spezifikation kontextfreier Grammatiken mit BNF

- Backus-Naur-Form:
Notation für kontextfreie Grammatiken
- Regeln geschrieben als
 $a ::= \dots$
- Terminale in Anführungszeichen: '+'
- Verkettung durch Hintereinanderschreiben

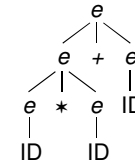
Erweiterung zu EBNF

- zusätzlich Abkürzungen:
 - Alternative: Regeln mit gleichem Nichtterminal links zusammengefaßt als
 $a ::= \dots \mid \dots \mid \dots$
 - [...]: Option, null- oder einmal
 - {...}: Wiederholung, null- bis beliebig mal
 - (...): Klammerung

Mehrdeutigkeit: Beispiel

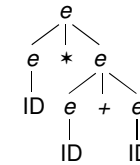
- erste Linksableitung

- $e \Rightarrow e + e$
 $\Rightarrow e * e + e$
 $\Rightarrow ID * e + e$
 $\Rightarrow ID * ID + e$
 $\Rightarrow ID * ID + ID$



- zweite Linksableitung

- $e \Rightarrow e * e$
 $\Rightarrow ID * e$
 $\Rightarrow ID * e + e$
 $\Rightarrow ID * ID + e$
 $\Rightarrow ID * ID + ID$



Aufbau einer yacc-Datei

Definitionen

%%

Regeln

%%

Unterprogramme } optional

Demo



Der katonische Leuchtturm: Lösung

- leuchtturm.y:

```
#!/
#include <stdio.h>
void yyerror(char *);
%}
%token ROT WEISS SCHREIBFEHLER
%%
katonischerturm: helfer helfer helfer helfer
helfer:
    ; mann
    | frau
    ;
mann:
    ; ROT WEISS ROT
    ;
frau:
    ; WEISS ROT WEISS
    ;
%%
void yyerror(char *msg) {
}
int main() {
    printf("Wie sieht der Leuchtturm aus, Sir? ");
    if(yyparse() == 0)
        printf("Wir liegen vor der Insel Kadonien, Sir!\n");
    else
        printf("Wir liegen *nicht* vor der Insel Kadonien, Sir!\n");
    return 0;
}
```

Absteigende Analyse versus aufsteigende Analyse

- **Absteigende Analyse** (top-down-Analyse)
 - Knoten des Ableitungsbaums werden von der Wurzel her konstruiert
 - leichter von Hand zu programmieren
 - „Praktische Informatik 2“
- **Aufsteigende Analyse** (bottom-up-Analyse)
 - Knoten des Ableitungsbaums werden von den Blättern her konstruiert
 - größere Klasse von Grammatiken
 - seltener Grammatiktransformation von Hand nötig
 - yacc

Prinzip der aufsteigenden Analyse

- „reduzieren“ des Eingabeworts auf das Startsymbol
 - Reduktionsschritt:
 - wenn: ein Teilwort = rechte Seite einer Regel
 - ersetze Teilwort durch Symbol auf linker Seite
 - richtige Wahl des Teilworts nötig
 - liefert Rechtsableitung in umgekehrter Reihenfolge
 - Name auch: „shift-reduce-Syntaxanalyse“



Implementierung der aufsteigenden Analyse mit einem Stack

- **Idee:**
 - lies jeweils ein Zeichen und packe es oben auf den Stack
 - wenn ein passendes Muster im Stack steht, wende eine Grammatikregel rückwärts an
 - Stack:
 - repräsentiert die bisher gelesene Eingabe
 - bereits erkannte Teile sind zu Nichtterminalen verdichtet



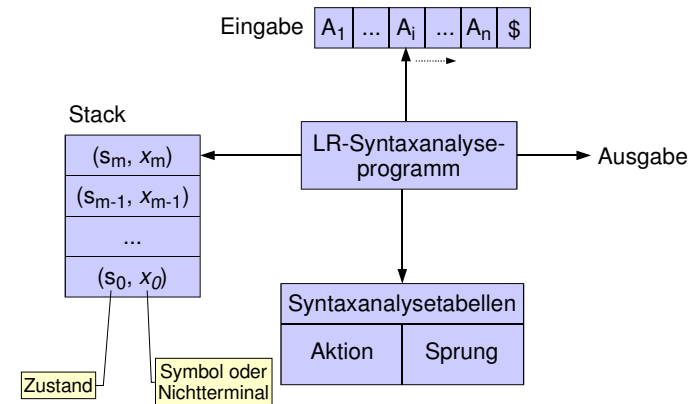
Stack-Implementierung: Beispiel

Stack	Eingabe
	ABCDE
A	BBCDE
AB	BCDE
Aa	BCDE
AaB	CDE
AaBC	DE
Aa	DE
AaD	E
Aab	E
AabE	
s	

Grundoperationen eines Shift-Reduce-Parsers

1. schieben (shift)
 - nächstes Eingabesymbol \rightarrow Stack
 2. reduzieren (reduce)
 - Stack: oberste Symbole \rightarrow Nichtterminal
 3. akzeptieren
 4. Fehler melden
- Satz:
Beim Reduzieren reicht es, nur die obersten Symbole auf dem Stack zu betrachten
 - Beweis: Drachenbuch

LR-Parser



Typische Konflikte

- mehrdeutige Grammatik
 - Konflikt in Ausdrücken
 - Konflikt in if-then-else
 - Konflikt in verschachtelten Listen
 - Konflikt in überlappenden Alternativen
- Nicht-LALR(1)-Grammatik
 - Konflikt wegen begrenzter Vorschau
 - Konflikt wegen Nicht-LALR-Grammatik

Präzedenzen: Idee

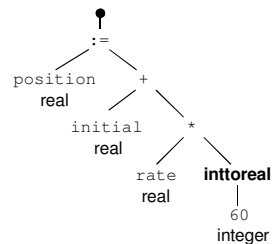
- manchmal umständlich:
Grammatik eindeutig bzw. LALR(1) machen
 - Beispiel: Ausdrücke
- Alternative:
Konflikt bleibt in Grammatik,
aber Parser nimmt immer „richtige“ Wahl
 - explizit definieren,
welche Wahl „richtig“ ist:
Präzedenz

Behandlung von Syntaxfehlern

- **schlecht:**
 - Abbruch bei erstem Fehler
- **besser:**
 - fehlerhafte Stelle überspringen
 - Analyse wieder aufsetzen
 - nach weiteren Fehlern suchen
- **Annahme:**
 - es gibt Stellen, wo die Analyse wieder aufsetzen kann
 - Beispiel: Semikolon (Ende einer Anweisung)
 - nur Heuristik

6. Syntaxgesteuerte Übersetzung

Attributierter Syntaxbaum



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

Konstruktion expliziter Syntaxbäume

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

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>

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

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 (3)

• calc-int.y

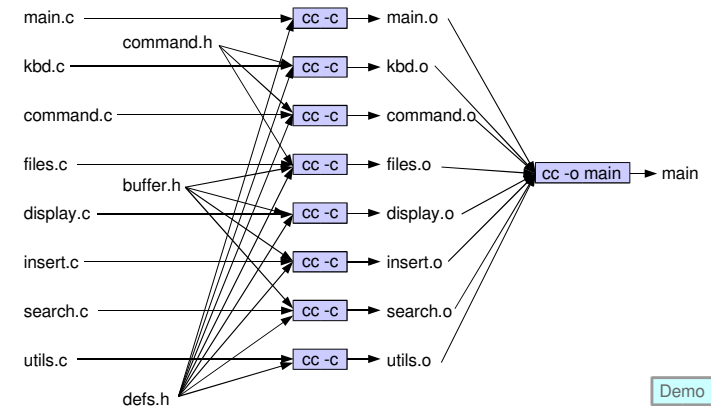
```

%{
#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();
}
    
```

7. Übersetzungssteuerung mit make

Motivation

Beispiel: Übersetzung eines kleinen Editors



Grundlagen von Makefiles

• editor-simple/Makefile

```
# Makefile for "edit".
edit : main.o kbd.o command.o display.o \
      insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c
display.o : display.c defs.h buffer.h
      cc -c display.c
insert.o : insert.c defs.h buffer.h
      cc -c insert.c
search.o : search.c defs.h buffer.h
      cc -c search.c
files.o : files.c defs.h buffer.h command.h
      cc -c files.c
utils.o : utils.c defs.h
      cc -c utils.c

clean :
      rm edit main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```



Algorithmus von make

- Aufruf: make
- Vorgehen von make:
 - Default-Ziel: erstes Ziel
 - hier: edit
 - Analyse der Abhängigkeiten
 - edit hängt von 8 Objekt-Dateien ab
 - rekursiv weitere Abhängigkeiten
 - schließlich existierende Dateien ohne Regeln
 - Sortieren der Abhängigkeiten
 - Ausführen

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