4. Die lexikalische Analyse

oder

C- bzw. Ratfor Programmteile in %{…%}:

- C-/Ratfor-Programmcode dazwischen
- wird unverändert (ohne Klammern) in generierte Datei übertragen
- Code kann in erster Spalte beginnen
- z.B. Präprozessoranweisungen
- Beispiel:

```
%{
#include<stdio.h>
#define MAX 100
%}
%%
….
```

4. Die lexikalische Analyse

Reguläre Definitionen:

- außerhalb von %{…..%}
- beginnen in erster Spalte
- Format: Name Ersatztext
- später: für {Name} wird Ersatztext eingesetzt
- Beispiel:

```
Z  [0-9]
Za        {Z}+
Bk        [a-z]
Bg  [A-Z]
%%
….{Bk}|{Bg}….
```

Aufgabe: Bezeichner, Integer, Float von Folie 67 mittels regulärer Definitionen kürzer ausdrücken

4. Die lexikalische Analyse

Startbedingungen:

- Zur (De-)Aktivierung bestimmter lex-Regeln zu bestimmten Zeitpunkten
- Startbedingungen (Zustände) im Definitionsteil definieren:
- %start name1 name2 name3 …
- soll Regel nur in bestimmten Zustand aktiv sein: <*name*> vor Regel setzen
- Umschalten auf anderen aktiven Zustand : BEGIN *zustandsname*
- Grundzustand: BEGIN 0
- Beispiel:

```
%Start   EINS ZWEI DREI
%%
^1          {ECHO; BEGIN EINS;}
^2          {ECHO; BEGIN ZWEI;}
^3          {ECHO; BEGIN DREI;}
\n          {ECHO; BEGIN 0;}
<EINS>a   printf(“A“);
<ZWEI>a   printf(“b“);
<DREI>a   printf(“z“);
%%
```

4. Die lexikalische Analyse

Zeichensatz-Tabellen:

- Standard für lex: ASCII-Code als Eingabe
- anderer Code als Eingabe: lex durch Zeichensatz-Tabelle umkonfigurieren (%T…….%T; dazwischen zeilenweise Zahl/String-Paare)

Verstellen lex-interner Tabellengrößen:

- %n *zahl* ändert erl. Zahl der Zustände (Default: 4000)
- %a *zahl* ändert erl. Zahl der Übergänge (Default: 16000)
- %e *zahl* ändert Zahl der erl. Parse-tree Knoten (Def: 8000)
- usw.
- lex –v liefert Statistik über Benutzung lex-interner Tabellen

4. Die lexikalische Analyse

Teil 3: Benutzerdefinierte Routinen

- üblicherweise Funktionsdefinitionen
- i.d.R. Funktionen der Aktionen des Regelteils
- werden unverändert ans Ende des generierten Programms kopiert
- alle Sprachkonstrukte der Wirtsprache erlaubt
- yywrap()
- Aufgabe: lex-Programm, welches Eingaben verschlüsselt:
  alle Buchstaben werden durch den 3.Nachfolger ersetzt, entsprechend x durch a, y durch b, z durch c. Am Ende der Eingabe wird ausgeben, wieviele Groß- bzw. Kleinbuchstaben jeweils verschlüsselt wurden.

4. Die lexikalische Analyse

Teil 2: lex-Regeln

- Format: regulärer Ausdruck Aktion(en)
- Aktion(en): C-/Ratfor-Code
- entspricht gelesener String einem reg. Ausdruck: Ausführen der Anweisung(en)
- Vorschriften:
  - wird String im Eingabetext von keinem Pattern abgedeckt: Ausgabe auf Standard-Out
  - es wird längstmöglicher String gesucht, der durch ein Pattern abgedeckt ist
  - decken mehrere Pattern String ab, wird zuerst angegebenes ausgewählt

4. Die lexikalische Analyse

Teil 2: lex-Regeln

Beispiele:

```
[  \t]                  ;        /* überliest Blanks, Tabs */
[0-9]+                  {return INT;}
[0-9]+\.[0-9]*          {return FLOAT;}
\;                      {return SEPARATOR;}
[\+\*\/\-\<\>=]         {return OPERATOR;}
#[^#]*#                 {return KOMMENTAR;}
[a-zA-Z][a-zA-Z0-9]*    {return BEZEICHNER;}
```

4. Die lexikalische Analyse

Metazeichen und Escape-Sequenzen in reg. Ausdrücken:

- Metazeichen: \ ^ $. [ ] | ( ) * + ? { } “ / % < >
- Escape-Sequenzen: \b (Backspace), \f (Formfeed), \n (Newline), \r (Return), \t (Tabulator), \\ (Backslash), \‘ (Hochkomma), \“ (Anführungszeichen), *ddd* (Oktalzahl)
- ^ Anfang einer Zeile
- $ Ende einer Zeile
- . bel. Zeichen (außer \n)
- [abcde] Klasse der Zeichen a,b,c,d,e
- [a-z0-9] Klasse aller Zeichen von a bis z und 0 bis 9
- [^A-Z] alle Zeichen außer A bis Z

4. Die lexikalische Analyse

Operatoren in reg. Ausdrücken:

- | Alternation: A | B deckt A oder B ab
- Konkatenation: AB deckt A unmittelbar gefolgt von B ab
- *, +, ?: null oder bel. viele, eins oder bel. viele, null oder eins
- (A) genauso wie A
- Wiederholung z{n,m}: z in bel. Anzahl zwischen n und m
- {name} hierfür den im Definitionsteil definierten Ausdruck setzen
- Kontextsensitivität r1/r2: deckt Strings ab, die r1 erfüllen, wenn auf sie String folgt, der r2 erfüllt

Priorisierung der Operatoren:

- | vor { } vor Konkatenation vor * + ? vor ()