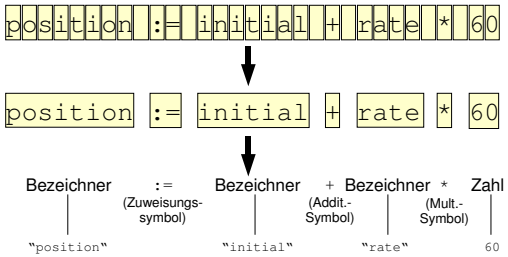
2. Lexikalische Analyse

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex (2 Termine)
5. Syntaxanalyse und der Parser-Generator yacc (3 T.)
6. Semantische Analyse
7. Transformation und Code-Erzeugung (?)
8. Übersetzungssteuerung mit make

2. Lexikalische Analyse

Lexikalische Analyse



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

Struktur der lexikalischen Analyse

1. Erkennung
   - Gruppierung in Lexeme
2. Transformation
   - effiziente Codierung als Zahlen
     - insbesondere der Bezeichner

- ist Grundstruktur vieler Phasen
  - Analyse → Synthese

Die Begriffe Symbol, Lexem, Muster

Begriff	Beispiele
Symbol	Bezeichner
Lexem	`position`, `initial`, `rate`
Muster	ein Buchstabe, gefolgt von Buchstaben und Ziffern

Lexikalische vs. syntaktische Analyse

- lexikalische und syntaktische Struktur könnten zusammen beschrieben werden
- Trennung aber sinnvoll
  - die Phasen werden einfacher
    - Beispiel: Leerraumbehandlung nur im Lexer
    - Sprachdefinition wird übersichtlicher
  - effizienter
  - Einkapselung möglicher Änderungen
    - Eingabealphabet
      - Beispiel: ASCII vs. Unicode
      - Beispiel: Sprache Z mit LaTeX- und Email-Markup

2. Lexikalische Analyse

Definition reguläre Ausdrücke

- über einem Alphabet Σ

1. ε ist regulärer Ausdruck
2. sei $a \in \Sigma$, dann:
   a ist regulärer Ausdruck
3. seien *r*, *s* reguläre Ausdrücke, dann:
   a) (*r*)|(*s*) ist regulärer Ausdruck
   b) (*r*)(*s*) ist regulärer Ausdruck
   c) (*r*)* ist regulärer Ausdruck
   d) (*r*) ist regulärer Ausdruck

Vermeidung von Klammern

- Konventionen:
  - *: höchste Priorität, links-assoziativ
  - Konkatenation: zweithöchste Priorität, links-assoziativ
  - |: niedrigste Priorität, links-assoziativ
- Beispiele:
  - (*a*)|((*b*)*(*c*)) ist äquivalent zu *a*|*b***c*
  - ((*a*)(*b*))(*c*) ist äquivalent zu *abc*
  - ((a)*)* ist äquivalent zu a**

Bezeichnete reguläre Sprachen

- seien ε, a, r, s obige reguläre Ausdrücke, dann:

1. ε bezeichnet {ε} — leeres Wort
2. *a* bezeichnet {*a*} — Einzelzeichen
3. bezeichnen r, s die Sprachen L(r), L(s), dann:
   a) (r)|(s) bezeichnet L(r) ∪ L(s) — Alternative
   b) (r)(s) bezeichnet L(r)L(s) — Konkatenation
   c) (r)* bezeichnet (L(r))* — optionale Wiederholung
   d) (r) bezeichnet L(r) — zusätzliche Klammerung

Beispiele für reguläre Ausdrücke und die bezeichneten Sprachen

- L(*a*|*b*) = {*a*, *b*}
- L((*a*|*b*)(*a*|*b*)) = {*aa*, *ab*, *ba*, *bb*}
- L(*a**) = {ε, *a*, *aa*, *aaa*, ...}

Reguläre Definitionen

- Namen für häufig benötigte Ausdrücke
  - $d_1 \rightarrow r_1$
    $d_2 \rightarrow r_2$
    ...
  - Beispiele:
    buchstabe → A | B | ... | Z | a | b | ... | z
    ziffer → 0 | 1 | ... | 9
- rechte Seite:
  regulärer Ausdruck oder *vorher* definierter Name

Abkürzungen

- *(r)*+: ein- oder mehrmaliges Auftreten
  - gleiche Priorität und Assoziativität wie (r)*
  - Definiert durch: $r^* = r^+ \mid \varepsilon$ und $r^+ = rr^*$
- (*r*)?: null- oder einmaliges Auftreten
  - gleiche Priorität und Assoziativität wie (r)*
  - Definiert durch: $r? = r \mid \varepsilon$
- [*abc*]: Zeichenklassen
  - a, b, c, ...: beliebig viele Symbole des Alphabets
  - Definiert durch: [*abc*] = a | b | c
  - weitere Abkürzung: [a-z] = a | b | ... | z

Beispiele für Abkürzungen

- L((hallo)+) = {hallo, hallohallo, hallohallohallo, ...}
- L(hunde?) = {hund, hunde}
- die Menge aller Bezeichner in C:
  [A-Za-z_][A-Za-z0-9_]*

2. Lexikalische Analyse

Definition: Endlicher Automat

- A = (S, Σ, *move*, s_0, *F)* mit
  - endliche Zustandsmenge *S*
  - (endliches) Eingabealphabet Σ
  - Transitionsrelation *move* ⊆ *(S x Σ x S)*
  - ein Startzustand $s_0 \in S$
  - Menge von akzeptierenden Zuständen $F \subseteq S$

Äquivalenz regulärer Ausdrücke und endlicher Automaten

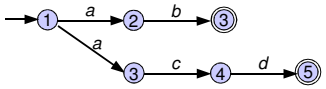
- die Sprache jedes regulären Ausdrucks kann von einem endlichen Automaten erkannt werden
  - auch Umkehrung gilt: die Sprache eines endlichen Automaten kann mit einem regulären Ausdruck beschrieben werden
  - Automat: im allgemeinen nicht deterministisch
- jeder nichtdeterministische endliche Automat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

Erkennen einer Sprache mit einem endlichen Automaten

- Wort ist in Sprache genau dann, wenn: es gibt einen Pfad im Automaten vom Anfangszustand zu einem Endzustand, an dessen Kanten die Zeichen des Wortes stehen

Beispiel: Erkennen einer Sprache mit einem endlichen Automaten

endlicher Automat:

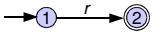


akzeptierte Sprache:

{ab, acd}

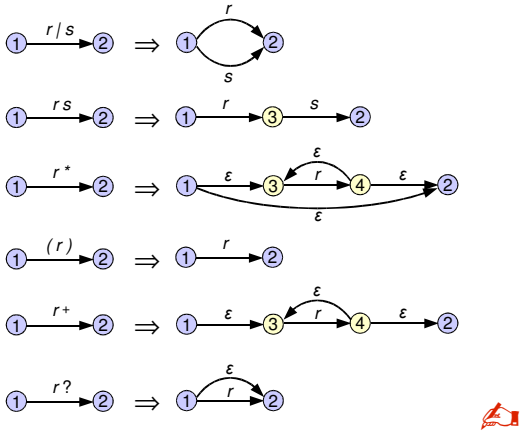
Umwandlung regulärer Ausdrücke in endliche Automaten

1. schreibe gesamten regulären Ausdruck an die einzige Kante des folgenden Automaten:

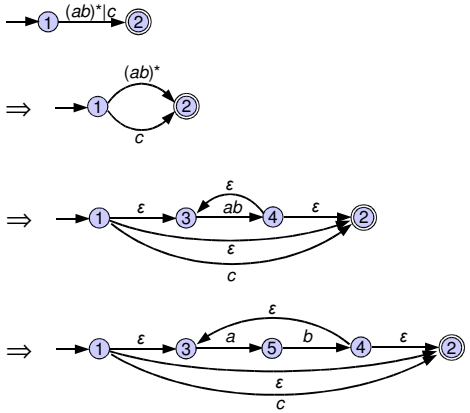


2. falls eine Kante mit mehr als einem einzelnen Zeichen beschriftet ist, führe dort eine Graphersetzung (s.u.) mit dem äußersten regulären Operator aus
3. falls keine solche Kante da, terminiere, sonst gehe zu 2.

Graphersetzungen



Beispiel einer Umwandlung



Implementierung eines nichtdet. endl. Automaten

- Problem:
  - von jedem Zustand ggf. viele Wege möglich
- Lösungsidee:
  - Menge von möglichen Zuständen statt nur ein Zustand
  - aufpassen bei ε-Kanten!
  - Details im Drachenbuch

Umwandlung nichtdeterministischer in deterministische Automaten

- nichtdet. Automaten:
  - ☹ Zeitbedarf: O(|e| x |r|)
  - ☺ Platzbedarf: O(|r|)

  |r|: Länge d. regulären Ausdrucks
  |e|: Länge der Eingabe
- det. Automaten:
  - ☺ Zeitbedarf: O(|e|) (plus Zeit zum Umwandeln, nur 1x)
  - ☹ Platzbedarf: $O(2^{|r|})$
- Idee: Potenzmengenkonstruktion NEA → DEA
  - gleiches Alphabet
  - Menge von mögl. NEA-Zuständen ↔ 1 DEA-Zustand
  - geeignete DEA-Transitionen

Operationen für die Potenzmengenkonstruktion

s: NEA-Zustand
T: Menge von NEA-Zuständen
a: Eingabezeichen

- ε-Hülle(*s*)
  - Menge der NEA-Zustände, die von *s* aus allein über ε-Kanten erreichbar sind
- ε-Hülle(*T*)
  - Menge der NEA-Zustände, die von einem $s \in T$ aus allein über ε-Kanten erreichbar sind
- move(*T*,*a*)
  - Menge der NEA-Zustände, die von einem $s \in T$ aus über eine *a*-Kante erreichbar sind

Konstruktion eines DEA aus einem NEA

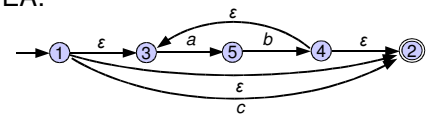
1. erzeuge den Startzustand des DEA und benenne ihn mit T_0, T_0 = ε-Hülle(s_0)
2. nimm einen unmarkierten Zustand T des DEA
3. markiere T
4. Für alle Eingabezeichen *a*:
   - 4.1 U := ε-Hülle(move(*T*,*a*))
   - 4.2 falls nötig, erzeuge den DEA-Zustand U
   - 4.3 erzeuge eine Transition von T mit *a* nach U
5. Wenn es unmarkierte Zustände gibt, gehe zu 2., sonst terminiere

Akzeptierende Zustände des DEA

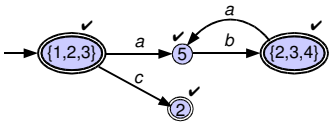
- ein Zustand *T* des DEA ist akzeptierend, falls mindestens einer der NEA-Zustände $s \in T$ akzeptierend ist

Beispiel einer DEA-Konstruktion

NEA:



resultierender DEA (in 5 Schritten konstruieren):



Implementierung eines deterministischen endl. Automaten

- Programmcode, Standard-Muster
- tabellengesteuert

Implementierung durch Programmcode

```
state=S0; term=0;
while ((state!=FAIL)&& !term) {
    switch (state) {
        case S0:
            switch (nextChar()) {
                case 'a': state=S7; break;
                /* ... */
                default: state=FAIL; break;
            }
        case S1:
            switch (nextChar()) {
                case 'a': state=S9; break;
                /* ... */
                case EOF: term=1; break;
                default: state=FAIL; break;
            }
        /* ... */
    }
}
if (term)
    ... akzeptiert
else
    ... akzeptiert nicht
```


Implementierung, tabellengesteuert

```
state=S0;
while ((state!=FAIL)) {
    if ((c=nextChar())==EOF) break;
    state=transTab[state][c];
}
if ((state!=FAIL)&&(acceptTab[state]))
    ... akzeptiert
else
    ... akzeptiert nicht
```

Anpassung für Lexemerkennung

- Termination nicht am Eingabeende, sondern sobald ein Lexem erkannt
- neuer Aufruf für nächstes Lexem

Mehrdeutigkeiten

- Probleme:
  - "`:`" und "`=`" oder "`:=`"?
  - "`/`" und "`*`" oder "`/*`"?
- Lösung: längstes Präfix (longest match)
  - Termination erst, wenn nächstes Zeichen nirgends mehr paßt
  - Bedingung: in der Syntax dürfen z.B. "`:`" und "`=`" nicht aufeinander folgen

Mehrdeutigkeiten (2)

- Problem: "`1.23456`" und "`1..10`"
  - Zahl: [0-9]+(.[0-9]+(E(+|-)?[0-9]+)?)?
  - wegen „longest match“ wird "`1`" in "`1..10`" nicht erkannt
    - am akzeptierenden Zustand vorbeigelaufen
- Lösung: Vorschau (lookahead)
  - bei Termination rücksetzen auf letzten akzeptierenden Zustand
  - ab da gelesene Zeichen müssen zurückgelegt werden
    - oben: max. 1 Zeichen
    - Fortran: viele Zeichen

Erkennung der Schlüsselworte

1. für jedes Schlüsselwort ein regulärer Ausdruck
2. Schlüsselworte als Bezeichner erkannt, später aussortiert

- sehr sinnvoll: reservierte Schlüsselworte
  - Bezeichner ∉ Schlüsselworte
  - Negativbeispiel PL/I:
    `IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;`

2. Lexikalische Analyse

Fehlerbehandlung

- nur wenige Fehler lexikalisch erkennbar
  - Beispiel: rechter Begrenzer fehlt
    - String, Kommentar, ...
- mögliche Recovery-Strategien:

  jeweils ein Zeichen wird
  - überlesen
    - Beispiel: „panische“ Fortsetzung
      - bis wieder gültiges Zeichen
      - einfachste Strategie
  - eingefügt
    - fehlendes Zeichen
  - ausgetauscht

2. Lexikalische Analyse

Transformation von Lexemen (Sieben, Screening)

- Transformationsregeln
  - Layout, Kommentar:
    unterdrücken
  - Pragmat:
    unterdrücken + Zustand des Übersetzers ändern
    - Beispiel: `#include "file.h"`
  - Trennsymbol, Operator:
    als Symbol kodieren
    - Beispiele: `;` und `+`
  - Literal:
    als Symbol kodieren + in internen Wert umrechnen
  - Bezeichner:
    als Symbol kodieren + eindeutig verschlüsseln

2. Lexikalische Analyse

Symboltabellenverwaltung

Nr.	Name	Typ	...
1	`position`	real	...
2	`initial`	real	...
3	`rate`	real	...
4	...		...

- sammelt und speichert die Attribute der Bezeichner
  - Typ
  - Gültigkeitsbereich
  - bei Prozedurnamen: Anzahl & Typen der Argumente, ...
  - Details zum Speicherbereich (bei Codeerzeugung)
  - ...

Anforderungen an die Symboltabellenverwaltung

- effizientes Einfügen von Bezeichnern in Tabelle
- effizientes Suchen von Bezeichnern in Tabelle
  - effizientes Vergleichen von Bezeichnern
- effizientes Setzen von Attributen für Bezeichner
- effizientes Lesen von Attributen für Bezeichner

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

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex (2 Termine)
5. Syntaxanalyse und der Parser-Generator yacc (3 T.)
6. Semantische Analyse
7. Transformation und Code-Erzeugung (?)
8. Übersetzungssteuerung mit make