Reguläre Sprachen, kontextfreie Grammatiken, LL-Parser

BC George (HSBI)

Motivation

Was muss ein Compiler wohl als erstes tun?

- Text einlesen
- Zerlegen in Bausteine
  - Keywords
  - Namen

    ⋮

Themen für heute

- Endliche Automaten
- Reguläre Sprachen
- PDAs
- Kontextferie Sprachen
- LL-Parser

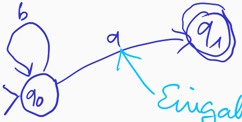
Endliche Automaten

State machines

Def.: Ein **deterministischer endlicher Automat** (DFA) ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit

- Q : endliche Menge von **Zuständen**
- Σ : Alphabet von **Eingabesymbolen** Eingabezeichen
- δ : die **Übergangsfunktion** $(Q \times \Sigma) \to Q$, δ kann partiell sein
  akt. Zustand ↪ neuer Zustand
- $q_0 \in Q$: der **Startzustand**
- $F \subseteq Q$: die Menge der **Endzustände**

Ein Wort wird akzeptiert:
- Resteingabe leer
- Endzustand



Eingabezeichen

Anfangszustand

ε = leeres Wort

Nichtdeterministische endliche Automaten

Def.: Ein **nichtdeterministischer endlicher Automat** (NFA) ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit

- Q : endliche Menge von **Zuständen**
- Σ : Alphabet von **Eingabesymbolen**
- δ : die **Übergangsfunktion** $(Q \times \Sigma) \to \underline{\mathcal{P}(Q)}$, δ kann partiell sein
- $q_0 \in Q$: der **Startzustand**
- $F \subseteq Q$: die Menge der **Endzustände**

Potenzmenge : Die Menge aller Teilmengen von Q

Akzeptierte Sprachen

Def.: Sei A ein DFA oder ein NFA. Dann ist **L(A)** die von A akzeptierte Sprache, d. h.

$L(A) = \{Wörter\ w \mid \delta^*(q_0, w) \in F\}$

$\delta^* = \delta$ $*$-mal angewendet

Wozu NFAs im Compilerbau?

Pattern Matching (Erkennung von Schlüsselwörtern, Bezeichnern, . . .) geht mit NFAs.

NFAs sind so nicht zu programmieren, aber:

Satz: Eine Sprache L wird von einem NFA akzeptiert $\Leftrightarrow$ L wird von einem DFA akzeptiert.

D. h. es existieren Algorithmen zur → Subset construction

- Umwandlung von NFAs in DFAS
- Minimierung von DFAs

→ Hopcrofts Algorithmus

Reguläre Sprachen

Reguläre Ausdrücke

Def.: Induktive Definition von **regulären Ausdrücken** (regex) und der von ihnen repräsentierten Sprache **L**:

- Basis:
    - ϵ und $\emptyset$ sind reguläre Ausdrücke mit $L(\epsilon) = \{\epsilon\}$, $L(\emptyset) = \emptyset$
    - Sei a ein Symbol $\Rightarrow$ a ist ein regex mit $L(a) = \{a\}$
- Induktion: Seien E, F reguläre Ausdrücke. Dann gilt:
    - $E + F$ ist ein regex und bezeichnet die Vereinigung $L(E + F) = L(E) \cup L(F)$
    - EF ist ein regex und bezeichnet die Konkatenation $L(EF) = L(E)L(F)$
    - E^* ist ein regex und bezeichnet die Kleene-Hülle $L(E^*) = (L(E))^*$
    - (E) ist ein regex mit $L((E)) = L(E)$

auch |

$L^+ = L^* \setminus \{\epsilon\}$

Vorrangregeln der Operatoren für reguläre Ausdrücke: *, Konkatenation, +

Formale Grammatiken

Satz → S P O

$\in N$ S → Artikel Substantiv

Artikel → a | the | ...

T

Def.: Eine *formale Grammatik* ist ein 4-Tupel $G = (N, T, P, S)$ aus

Variable

- N: endliche Menge von **Nichtterminalen**
- T: endliche Menge von **Terminalen**, $N \cap T = \emptyset$
- $S \in N$: **Startsymbol**
- P: endliche Menge von **Produktionen** der Form

$$X \to Y \text{ mit } X \in (N \cup T)^* N (N \cup T)^*, Y \in (N \cup T)^*$$

Ableitungen

Def.: Sei $G = (N, T, P, S)$ eine Grammatik, sei $\alpha A \beta$ eine Zeichenkette über $(N \cup T)^*$ und sei $A \rightarrow \gamma$ eine Produktion von G.

Wir schreiben: $\alpha A \beta \Rightarrow \alpha \gamma \beta$ ($\alpha A \beta$ leitet $\alpha \gamma \beta$ ab).

Def.: Wir definieren die Relation $\overset{*}{\Rightarrow}$ induktiv wie folgt:

- Basis: $\forall \alpha \in (N \cup T)^* \alpha \overset{*}{\Rightarrow} \alpha$ (Jede Zeichenkette leitet sich selbst ab.)
- Induktion: Wenn $\alpha \overset{*}{\Rightarrow} \beta$ und $\beta \Rightarrow \gamma$ dann $\alpha \overset{*}{\Rightarrow} \gamma$

Def.: Sei $G = (N, T, P, S)$ eine formale Grammatik. Dann ist $L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$ die von G erzeugte Sprache.

Reguläre Grammatiken

Def.: Eine **reguläre (oder type-3-) Grammatik** ist eine formale Grammatik mit den folgenden Einschränkungen:

- Alle Produktionen sind entweder von der Form
  - $X \to aY$ mit $X \in N, a \in T, Y \in N$ (*rechtsreguläre* Grammatik) oder exor
  - $X \to Ya$ mit $X \in N, a \in T, Y \in N$ (*linksreguläre* Grammatik)
- $X \to \epsilon$ ist erlaubt

Satz: Die von endlichen Automaten akzeptiert Sprachklasse, die von regulären Ausdrücken beschriebene Sprachklasse und die von regulären Grammatiken erzeugte Sprachklasse sind identisch und heißen **reguläre Sprachen**.

Reguläre Sprachen

{ { . . . { . . . } . . { . . . }

welche öffnende Klammer wird hier geschlossen?

- einfache Struktur
- Matchen von Symbolen (z. B. Klammern) nicht möglich, da die fixe Anzahl von Zuständen eines DFAs die Erkennung solcher Sprachen verhindert.

Wozu reguläre Sprachen im Compilerbau?

- Reguläre Ausdrücke
  - definieren Schlüsselwörter und alle weiteren Symbole einer Programmiersprache, z. B. den Aufbau von Gleitkommazahlen
  - werden (oft von einem Generator) in DFAs umgewandelt
  - sind die Basis des *Scanners* oder *Lexers*

Ein Lexer ist mehr als ein DFA

- Ein **Lexer**
  - wandelt mittels DFAs aus regulären Ausdrücken die Folge von Zeichen der Quelldatei in eine Folge von sog. Token um
  - bekommt als Input eine Liste von Paaren aus regulären Ausdrücken und Tokennamen, z. B. ("while", WHILE)
  - Kommentare und Strings müssen richtig erkannt werden. (Schachtelungen)
  - liefert Paare von Token und deren Werte, sofern benötigt, z. B. (WHILE, _), oder (IDENTIFIER, "radius") oder (INTEGERZAHL, "334")

Wie geht es weiter?

- Ein **Parser**
  - führt mit Hilfe des Tokenstreams vom Lexer die Syntaxanalyse durch
  - basiert auf einer sog. kontextfreien Grammatik, deren Terminale die Token sind
  - liefert die syntaktische Struktur in Form eines Ableitungsbaums (**syntax tree**, **parse tree**), bzw. einen **AST** (abstract syntax tree) ohne redundante Informationen im Ableitungsbaum (z. B. Semikolons)
  - liefert evtl. Fehlermeldungen

Kellerautomaten
(Push-Down-Automata, PDAs)

Kellerautomaten (Push-Down-Automata, PDAs)

Einordnung: Erweiterung der Automatenklasse DFA um einen Stack

Def.: Ein **Kellerautomat** (PDA) $P = (Q,\ \Sigma,\ \Gamma,\ \delta,\ q_0,\ \perp,\ F)$ ist ein Septupel aus

Q:	eine endliche Menge von Zuständen
Σ:	eine endliche Menge von Eingabesymbolen
Γ: (groß Gamma)	ein endliches Kelleralphabet
δ:	die Übergangsfunktion
	$\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \to 2^{Q \times \Gamma^*} = \mathcal{P}(Q \times \Gamma^*)$ $\Leftarrow$ nichtdeterministisch
q_0:	der Startzustand
$\perp \in \Gamma$:	der anfängliche Kellerinhalt, symbolisiert den leeren Keller ($\perp$ = bottom)
$F \subseteq Q$:	die Menge von Endzuständen

Abbildung 1: Definition eines PDAs

Ein PDA ist per Definition nichtdeterministisch und kann spontane Zustandsübergänge durchführen.

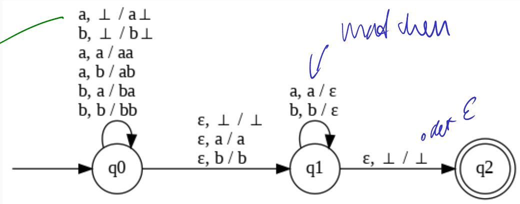
Was kann man damit akzeptieren?

Strukturen mit paarweise zu matchenden Symbolen.

Bei jedem Zustandsübergang wird ein Zeichen (oder ϵ) aus der Eingabe gelesen, ein Symbol von Keller genommen. Diese und das Eingabezeichen bestimmen den Folgezustand und eine Zeichenfolge, die auf den Stack gepackt wird. Dabei wird ein Symbol, das später mit einem Eingabesymbol zu matchen ist, auf den Stack gepackt.

Soll das automatisch vom Stack genommene Symbol auf dem Stack bleiben, muss es wieder gepusht werden.

Ein PDA für $L = \{ww^R \mid w \in \{a, b\}^*\}$:



matchen

oder ε

$a, \perp \ / a\perp$

Eingabe

top of stack

neuer top of stack, statt $\perp \rightarrow$

a

$\perp$

Deterministische PDAs

Def. Ein PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ ist *deterministisch* $:\Leftrightarrow$

- $\delta(q, a, X)$ hat höchstens ein Element für jedes $q \in Q, a \in \Sigma$ oder $(a = \epsilon$ und $X \in \Gamma)$.
- Wenn $\delta(q, a, x)$ nicht leer ist für ein $a \in \Sigma$, dann muss $\delta(q, \epsilon, x)$ leer sein.

Deterministische PDAs werden auch *DPDAs* genannt.

Der kleine Unterschied

Satz: Die von DPDAs akzeptierten Sprachen sind eine echte Teilmenge der von PDAs akzeptierten Sprachen.

Reguläre Sprachen sind eine echte Teilmenge der von DPDAs akzeptierten Sprachen.

Kontextfreie Grammatiken und Sprachen

Kontextfreie Grammatiken

Def. Eine *kontextfreie (cf-)* Grammatik ist ein 4-Tupel $G = (N, T, P, S)$ mit N, T, S wie in (formalen) Grammatiken und P ist eine endliche Menge von Produktionen der Form:

$X \to Y$ mit $X \in N, Y \in (N \cup T)^*$.

$\Rightarrow, \stackrel{*}{\Rightarrow}$ sind definiert wie bei regulären Sprachen.

Nicht jede kontextfreie Grammatik ist eindeutig

Def.: Gibt es in einer von einer kontextfreien Grammatik erzeugten Sprache ein Wort, für das mehr als ein Ableitungsbaum existiert, so heißt diese Grammatik *mehrdeutig*. Anderenfalls heißt sie *eindeutig*.

Satz: Es ist nicht entscheidbar, ob eine gegebene kontextfreie Grammatik eindeutig ist.

Satz: Es gibt kontextfreie Sprachen, für die keine eindeutige Grammatik existiert.

Kontextfreie Grammatiken und PDAs

Satz: Die kontextfreien Sprachen und die Sprachen, die von PDAs akzeptiert werden, sind dieselbe Sprachklasse.

Satz: Eine von einem DPDA akzeptierte Sprache hat eine eindeutige Grammatik.

Def.: Die Klasse der Sprachen, die von einem DPDA akzeptiert werden, heißt Klasse der *deterministisch kontextfreien (oder LR(k)-) Sprachen*.

Vorgehensweise im Compilerbau: Eine Grammatik für die gewünschte Sprache definieren und schauen, ob sich daraus ein DPDA generieren lässt (automatisch).

Bsp: Damenring-Ehe-Problem

Syntaxanalyse

Was brauchen wir für die Syntaxanalyse von Programmen?

- einen Grammatiktypen, aus dem sich manuell oder automatisiert ein Programm zur deterministischen Syntaxanalyse erstellen lässt
- einen Algorithmus zum sog. Parsen von Programmen mit Hilfe einer solchen Grammatik

Arten der Syntaxanalyse

Die Syntax bezieht sich auf die Struktur der zu analysierenden Eingabe, z. B. einem Computerprogramm in einer Hochsprache. Diese Struktur wird mit formalen Grammatiken beschrieben. Einsetzbar sind Grammatiken, die deterministisch kontextfreie Sprachen erzeugen.

- Top-Down-Analyse: Aufbau des Parse trees von oben nach unten
  - Parsen durch rekursiven Abstieg
  - tabellengesteuertes LL-Parsing
- Bottom-Up-Analyse: LR-Parsing

Bevor wir richtig anfangen...

Def.: Ein Nichtterminal A einer kontextfreien Grammatik G heißt *unerreichbar*, falls es kein $a, b \in (N \cup T)^*$ gibt mit $S \stackrel{*}{\Rightarrow} aAb$. Ein Nichtterminal A einer Grammatik G heißt *nutzlos*, wenn es kein Wort $w \in T^*$ gibt mit $A \stackrel{*}{\Rightarrow} w$.

Def.: Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt *reduziert*, wenn es keine nutzlosen oder unerreichbaren Nichtterminale in N gibt.

Bevor mit einer Grammatik weitergearbeitet wird, müssen erst alle nutzlosen und dann alle unerreichbaren Symbole eliminiert werden. Wir betrachten ab jetzt nur reduzierte Grammatiken.

Algorithmus: Rekursiver Abstieg

Algorithmus: Rekursiver Abstieg

Hier ist ein einfacher Algorithmus, der (indeterministisch) top-down Ableitungen vom Nonterminal X aufbaut:

$B \to BB$

$B \to CDE$

Eingabe: Ein Nichtterminal X und das nächste zu verarbeitende Eingabezeichen *a*.

RecursiveDescent(X, a) $//X \in N, a \in T$
    for a production, $X \to Y_1 Y_2 \dots Y_n$
        for i = 1 to n
            if $Y_i \in N$
                RecursiveDescent(Y_i, a)
            else if $Y_i \neq a$
                error

$Y_i \in T$

$C \to C\gamma$

Linksrekursion

$\Rightarrow$ Algorithmus

Abbildung 2: Recursive Descent-Algorithmus

$Y_i \in N \cup T$

Tabellengesteuerte Parser: LL(k)-Grammatiken

First-Mengen

LL:

Der Ableitungsbaum enthält nur Linksableitungen

Eingabe von links nach rechts lesen

$S \rightarrow A \mid B \mid C$

hier anfangen

Welche Produktion nehmen?

Wir brauchen die "terminalen k-Anfänge" von Ableitungen von Nichtterminalen, um eindeutig die nächste zu benutzende Produktion festzulegen. k ist dabei die Anzahl der sog. *Vorschautoken*.

Def.: Wir definieren *First* - Mengen einer Grammatik wie folgt:

- $a \in T^*, |a| \leq k : First_k(a) = \{a\}$
- $a \in T^*, |a| > k : First_k(a) = \{v \in T^* \mid a = vw, |v| = k\}$
- $\alpha \in (N \cup T)^* \backslash T^* : First_k(\alpha) = \{v \in T^* \mid \alpha \overset{*}{\Rightarrow} w, \text{mit } w \in T^*, First_k(w) = \{v\}\}$

Linksableitungen

nach oben packen

Def.: Bei einer kontextfreien Grammatik G ist die *Linksableitung* von $\alpha \in (N \cup T)^*$ die Ableitung, die man erhält, wenn in jedem Schritt das am weitesten links stehende Nichtterminal in α abgeleitet wird.

Man schreibt $\alpha \overset{*}{\Rightarrow}_l \beta$.

Follow-Mengen

/ oder einem $\beta \in (N \cup T)^*$

Manchmal müssen wir wissen, welche terminalen Zeichen hinter einem Nichtterminal stehen können.

Def. Wir definieren *Follow* - Mengen einer Grammatik wie folgt:

$\forall \beta \in (N \cup T)^*$:

k Symbole hinter β

$$Follow_k(\beta) = \{w \in T^* \mid \exists \alpha, \gamma \in (N \cup T)^* \text{ mit } S \overset{*}{\Rightarrow}_l \alpha\beta\gamma \text{ und } w \in First_k(\gamma)\}$$

$$\left.\begin{array}{l} S \to AB \mid C \\ A \to \varepsilon \mid \ldots \end{array}\right\}$$

hier muss zur Entscheidung, ob S nach AB oder nach C abgeleitet werden muss, auch First(B) berücksichtigt werden

LL(k)-Grammatiken

Def.: Eine kontextfreie Grammatik $G = (N,\ T,\ P,\ S)$ ist genau dann eine *LL(k)*-Grammatik, wenn für alle Linksableitungen der Form:

$S \overset{*}{\Rightarrow}_l\ wA\gamma \Rightarrow_l w\alpha\gamma \overset{*}{\Rightarrow}_l wx$

und

$S \overset{*}{\Rightarrow}_l wA\gamma \Rightarrow_l w\beta\gamma \overset{*}{\Rightarrow}_l wy$

mit $(w, x, y \in T^*, \alpha, \beta, \gamma \in (N \cup T)^*, A \in N)$ und $First_k(x) = First_k(y)$ gilt:

$\alpha = \beta$, d. h. die Ableitung von A ist eindeutig

LL(1)-Grammatiken

Das hilft manchmal:

Für $k = 1$: G ist $LL(1) : \forall A \to \alpha, A \to \beta \in P, \alpha \neq \beta$ gilt:

1. $\neg\exists a \in T : \alpha \overset{*}{\Rightarrow}_l a\alpha_1$ und $\beta \overset{*}{\Rightarrow}_l a\beta_1$
2. $((\alpha \overset{*}{\Rightarrow}_l \epsilon) \Rightarrow (\neg(\beta \overset{*}{\Rightarrow}_l \epsilon)))$ und $((\beta \overset{*}{\Rightarrow}_l \epsilon) \Rightarrow (\neg(\alpha \overset{*}{\Rightarrow}_l \epsilon)))$
3. $((\beta \overset{*}{\Rightarrow}_l \epsilon)$ und $(\alpha \overset{*}{\Rightarrow}_l a\alpha_1)) \Rightarrow a \notin Follow(A)$
4. $((\alpha \overset{*}{\Rightarrow}_l \epsilon)$ und $(\beta \overset{*}{\Rightarrow}_l a\beta_1)) \Rightarrow a \notin Follow(A)$

Die ersten beiden Zeilen bedeuten:

α und β können nicht beide ϵ ableiten, $First_1(\alpha) \cap First_1(\beta) = \emptyset$

Die dritte und vierte Zeile bedeuten:

$(\epsilon \in First_1(\beta)) \Rightarrow (First_1(\alpha) \cap Follow_1(A) = \emptyset)$

$(\epsilon \in First_1(\alpha)) \Rightarrow (First_1(\beta) \cap Follow_1(A) = \emptyset)$

LL(k)-Sprachen

Die von *LL(k)*-Grammatiken erzeugten Sprachen sind eine echte Teilmenge der deterministisch parsbaren Sprachen.

Die von *LL(k)*-Grammatiken erzeugten Sprachen sind eine echte Teilmenge der von *LL(k+1)*-Grammatiken erzeugten Sprachen.

Für eine kontextfreie Grammatik G ist nicht entscheidbar, ob es eine *LL(1)* - Grammatik G' gibt mit $L(G) = L(G')$.

In der Praxis reichen $LL(1)$ - Grammatiken oft. Hier gibt es effiziente Parsergeneratoren, deren Eingabe eine LL(k)- (meist LL(1)-) Grammatik ist, und die als Ausgabe den Quellcode eines (effizienten) tabellengesteuerten Parsers generieren.

$O(n)$

Algorithmus: Konstruktion einer LL-Parsertabelle

Eingabe: Eine Grammatik G = (N, T, P, S)

Ausgabe: Eine Parsertabelle *P*

ANTLR parst LL* Grammatiken

$a \in First(\alpha)$

$$
\begin{array}{l}
\texttt{for each production } X \to \alpha \\
\quad \texttt{for each } a \in First(\alpha) \\
\quad\quad \texttt{add } X \to \alpha \texttt{ to } P[X, a] \\
\\
\quad \texttt{if } \epsilon \in First(\alpha) \\
\quad\quad \texttt{for each } b \in Follow(\alpha) \\
\quad\quad\quad \texttt{add } X \to \alpha \texttt{ to } P[X, b] \\
\quad\quad\quad \texttt{if } \epsilon \in First(\alpha) \texttt{ and } \perp \in Follow(X) \\
\quad\quad\quad\quad \texttt{add } X \to \alpha \texttt{ to } P[A, \perp]
\end{array}
$$

Endezeichen des Inputs

	a
S	$S \to \alpha$

Abbildung 3: Algorithmus zur Generierung einer LL-Parsertabelle

Hier ist $\perp$ das Endezeichen des Inputs. Statt *First*$_1(\alpha)$ und *Follow*$_1(\alpha)$ wird oft nur *First*(α) und *Follow*(α) geschrieben.

Statt *First*$_1(\alpha)$ und *Follow*$_1(\alpha)$ wird oft nur *First*(α) und *Follow*(α) geschrieben.}

LL-Parsertabellen

Rekursive Programmierung bedeutet, dass das Laufzeitsystem einen Stack benutzt (bei einem Recursive-Descent-Parser, aber auch bei der Parsertabelle). Diesen Stack kann man auch "selbst programmieren", d. h. einen PDA implementieren. Dabei wird ebenfalls die oben genannte Tabelle zur Bestimmung der nächsten anzuwendenden Produktion benutzt. Der Stack enthält die zu erwartenden Eingabezeichen, wenn immer eine Linksableitung gebildet wird. Diese Zeichen im Stack werden mit dem Input gematcht.

Algorithmus: Tabellengesteuertes LL-Parsen mit einem PDA

Eingabe: Eine Grammatik G = (N, T, P, S), eine Parsertabelle P mit $w\ \perp$ als initialem Kellerinhalt

Ausgabe: Wenn $w \in L(G)$, eine Linksableitung von w, Fehler sonst

```
a = next_token()
X = top of stack  // entfernt X vom Stack

while X ≠⊥

  if X = a
    a = next_token()

  else if X ∈ T
    error

  else if P[X,a] leer
    error

  else if P[X,a] = X → Y1 Y2 ... Yk
    process_production(X → Y1 Y2 ... Yk)
    push(Y1 Y2 ... Yk) //Y1 = top of stack

  X = top of stack
```

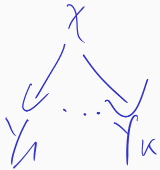


Abbildung 4: Algorithmus zum tabellengesteuerten LL-Parsen

Wrap-Up

Das sollen Sie mitnehmen

- Definition und Aufgaben von Lexern
- DFAs und NFAs
- Reguläre Ausdrücke
- Reguläre Grammatiken
- Die Struktur von gängigen Programmiersprachen lässt sich nicht mit regulären Ausdrücken beschreiben und damit nicht mit DFAs akzeptieren.
- Das Automatenmodell der DFAs wird um einen endlosen Stack erweitert, das ergibt PDAs.
- Kontextfreie Grammatiken (CFGs) erweitern die regulären Grammatiken.
- Deterministisch parsebare Sprachen haben eine eindeutige kontextfreie Grammatik.
- Es ist nicht entscheidbar, ob eine gegebene kontextfreie Grammatik eindeutig ist.

LICENSE