

zeigen, dass regulär ist \rightarrow reguläre Grammatik angeben
| regulären Ausdruck angeben
| regulären Automaten erstellen

Wozu kann Übersetzer genutzt werden?

- Erzeugen von Maschinencode
- Programmtransformation in andere Programmiersprache
- Programmanalyse und Füllen von Datenbank mit Programminformationen

1. Lexikalische Analyse (Scanner):

- Einlesen des Programmtextes
- Überlesen von Kommentaren und Leerzeichen
- Erkennen von Schlüsselwörtern (z.B. if, while,...)
- Erkennen von Bezeichnern (Variablennamen) und Funktionen
- Erkennen von Zahlen, Strings und Operatoren (+,-,*,...)

⇒ Erzeugt daraus Tokens bestehend aus Token-Typ (Schlüsselwort, Operator, etc) und optionalem Wert
Ausgabe ist endliche Folge von Tokens

2. Syntaktische Analyse (Parser):

- Eingabe ist die endliche Folge von Tokens

- Prüft ob syntaktisch korrekt

⇒ gibt erst Ja oder Nein aus ob syntaktisch korrekt
danach einen Syntaxbaum

3. Semantische Analyse:

Prüft sonstige Bedingungen z.B.:

- Stimmt Typ der rechten Seite einer Zuweisung mit linker überein bzw. kann umgewandelt werden

- Sind alle Variablen definiert

- Sind sie definiert bevor sie genutzt werden

- usw.

⇒ gibt wieder Syntaxbaum (graphische Darstellung zur Ableitung eines Wortes) aus

4. Transformation:

Eingabe ist Syntaxbaum

Wandelt diesen in verschiedene Zwischensprachen (interne Darstellungen) um, bis gewünschtes Ergebnis erreicht

Andere Phasen:

- Optimierung

- Codegenerierung

5. Assemblierung (Assembler):

6. Binden (Linker):

Alle vom Assembler generierte Objekt-Dateien werden in eine große executable Datei verbunden

Ein-/Ausgaben der Phasen:

Scanner:

ein: Datei mit Zeichen (Quellcode)
aus: Folge von Tokens mit Attributen

Parser:

ein: Folge von Tokens mit Attributen
aus: Abstrakter Syntaxbaum

Transformationsphase:

ein: Abstrakter Syntaxbaum
aus: Kontrollflussgraph

Codeerzeugung:

ein: Kontrollflussgraph
aus: Programm in Maschinensprache (in ASCII-Datei)

Assemblierer:

?

ein: Maschinencode in mnemonischer Darstellung
aus: Quellprogramm in semantisch äquivalentem Objektcodeformat

Binder (Linker):

ein: Maschinencode von Programm und Bibliotheken im Objektcodeformat
aus: ausführbares Programm

Altklausuraufgaben:

Welche Rolle spielt Grammatik? generieren, akzeptieren

Dient zur Spezifikation:

- der Sprache für Benutzer (Anleitung zum Generieren eines Satz)
- des Übersetzers für Sprache (Anleitung Konstruktion eines Akzeptors)

Zusammenhang endlicher Automat und regulären Grammatiken (G)

Für jede G gibt es einen A , der die von G erzeugte Sprache akzeptiert ($L(G) = L(A)$)

Konstruktion für Beweis:

Sei $G = (N, T, P, Z)$, dann ist $A = (T, Q_N \cup \{f\}, R, Z, \{f\})$
fehlt noch ein Teil (die R des A)

Zusammenhang deterministisch und nicht-det. endl. Automat

DEA: Jedes Paar (q, t) hat höchstens einen Folgezustand q'

NEA: Es gibt ein Paar (q, t) , das mehrere Folgezustände besitzt

Zusammenhang: Zu jedem NEA gibt es einen DEA, welcher die gleiche Sprache akzeptiert

Arten von Fehlern (erkennt der Compiler diese?)

Syntaxfehler:

if $x == 0$) $x++;$

wird vom Compiler erkannt

Fehler der statischen Semantik:

int y ; $y(); \Rightarrow$ called object is not a function

wird vom Compiler erkannt

Fehler der dynamischen Semantik:

int x; scanf("%d", &x); x = 1/x; \Rightarrow Division durch 0?

wird zur Übersetzungszeit nicht erkannt, da x dynamisch zur Laufzeit zugewiesen

Logische Fehler:

x = 0; aber gemeint war x=1;

erkennt Compiler nicht, da syntaktisch und semantisch korrekt

Warnungen:

if (x=0) \Rightarrow Hinweis, dass vermutlich logischen Fehler,
da Wertzuweisung in Anweisung

erkennt Compiler, muss aber kein Fehler sein

Wie wird Quellprogramm intern dargestellt?

Tokens und Attribute?

Definitionen:

Abgeschlossenheit:

Nach Operation liegt Ergebnis wieder in Menge
z.B. $\mathbb{N} \quad 0-2 = -2$ nicht abgeschlossen da $-2 \notin \mathbb{N}$

Reguläre Grammatik:

$G = (N, T, P, z)$ Nichtterminale, Terminaler, Produktionen, Startsymbol

Sei V (Vokabular) = $N \cup T$, P haben Form:

- rechts regulär $A \rightarrow a$ oder $A \rightarrow bB$ oder $A \rightarrow \epsilon$

- links regulär $A \rightarrow a$ oder $A \rightarrow Bb$ oder $A \rightarrow \epsilon$

beide mit $A, B \in N$ und $a, b \in T$

Sprache einer Grammatik:

$L = \{w \in T^* \mid z \Rightarrow^* w\}$ w = Wort, L = Sprache

Endlicher Automat:

$A = (T, Q, R, q_0, F)$, $T \cap Q = \emptyset$?

Terminale, Zustände, Regeln, Startzustand, Finalzustände

Regeln mit Form $q_t \rightarrow q'$ oder $q \rightarrow q'$ mit $q, q' \in Q, t \in T$

Von Automat akzeptierte Sprache:

$L(A) = \{w \in T^* \mid q_0 w \Rightarrow^* q, q \in F\}$

Wort ist Folge an T , start über beliebig viele Schritte zu Finalzustand

Eindeutigkeit einer Grammatik:

Eine Grammatik ist eindeutig, wenn es für jedes ableitbare Wort genau eine (Links-) Rechtsableitung gibt

Rechtsableitung:

Immer das am weitesten rechts stehende N ersetzen

Chomsky-Hierarchie: ($A, B \in N$)

Typ-0 (rekursiv)

$U \rightarrow V$ mit $U \in V^+$ und $V \in V^*$

Typ-1 (kontextsensitiv)

$XAY \rightarrow XWY$ mit $X, Y \in V^*$ und $W \in V^+$

ist $Z \rightarrow \epsilon \in P$ darf Z nicht in rechter Seite einer Regel vorkommen

Typ-2 (kontextfrei)

$A \rightarrow w$ mit $w \in V^*$ (durch beliebig viele T ersetzbar, dahe w)

Erkennbar in $O(n^2)$ wenn eindeutig, sonst $O(n^3)$

Typ-3 (regulär)

$A \rightarrow a$ oder $A \rightarrow \epsilon$ mit $a \in T$ (nur durch ein T ersetzbar)

rechts-regulär: $A \rightarrow aB$

links-regulär: $A \rightarrow Ba$

Typ-3 Grammatik darf nicht gleichzeitig rechts- und links-reguläre Produktionen enthalten

Erkennbar in $O(n)$

Formale Sprachen Allgemeines:

Normalform:

nie $A \rightarrow a$ sondern $A \rightarrow aA$ und $A \rightarrow \epsilon$

Für jede rechtsreguläre gibt es äquiv. linksreg.

Rechts- bzw. Linkslinear:

statt ersetzen durch ein T ersetzen durch ganzes $w \in T^*$

Semi-Thue-System:

P nur in eine Richtung durchführbar statt bidirektional

Mehrdeutige Grammatik:

Gibt mind. ein Wort für das es mehr als eine Rechts-/Linksableitung gibt

Eindeutige Grammatik:

Für jedes Wort nur eine Rechts-/Linksableitung

Entscheidbarkeitsprobleme:

Erkennungsproblem (Wortprob.): ist $w \in L_G$?

Leerheitsprob.: ist $L_G = \emptyset$?

Äquivalenzprob.: gilt $L_{G_1} = L_{G_2}$?

Mehrdeutigkeitsprob.: ist G mehrdeutig?

Sprachtyp	Entscheidbar
0	-
1	Wort
2	Wort, Leer
3	Wort, Leer, Äquiv, Mehr

Abgeschlossenheit von Sprachen:

Typ L_1, L_1 und L_2	$L_1 \cup L_2$	$L_1 \cap L_2$	\bar{L}	$L_1 L_2$	L^*
0	j	j n	j	j	j
1	j	j j	j	j	j
2	j	n n	j	j	j
3	j	j j	j	j	j

j = Resultat ist gleiche Sprachklasse

Turingmaschine

$$T = (V, Q, q_0, F, \text{PROG})$$

V: endliches Alphabet mit $s \in V$

Q: Zustände mit initialen Zustand q_0

F: Finalzustände $F \subseteq Q$

A: Aktionen ($W_S, -, R, L$)

Turing-Berechenbarkeit

Eine Funktion f heißt Turing berechenbar, wenn es eine Turing Maschine T gibt, die für jede Eingabe terminiert und ein Ergebnis ausgibt.

Halteproblem

Kann ein Algorithmus für ein beliebiges Programm P mit der Eingabe E entscheiden, ob P über E nach endlich vielen Schritten anhält oder unendlich weiterläuft.²

\Rightarrow unlösbar / nicht entscheidbar

dadurch Grenzen der Programmverifikation:

unmöglich P zu schreiben, das zuverlässig alle anderen P

auf z.B. Fehlerfreiheit oder endloses Laufen prüft

Turingmaschine (TM_a)

Allgemeine Turingmaschine :

- beidseitig unendliches langes Band mit Speicherzellen
- Hat ein Schreib-Lese Kopf, kann immer nur eine Aktion ausführen
- Regeln werden wie folgt definiert

$$s, q \rightarrow a, q'$$

s: Symbol auf das der Kopf zeigt

q: aktueller Zustand

a: Aktion

q': Zustand in den übergegangen wird

Turing-These Aussage: (nicht die These nur was sie impliziert)

Es kann kein Rechenmodell geben, das mehr berechnen kann als die bisherigen Modelle

Computer können alles berechnen was prinzipiell berechenbar (wenn genug Zeit und Speicher)

Variationen

Turing Maschine mit erweitertem Befehlssatz (TM_{xi}):

Neue Befehle L_s und R_s, schreibe s und gehe noch Links bzw. Rechts

TM_a und TM_{xi} können sich gegenseitig simulieren => äquivalent

Turing Maschine mit Macros :

Ein Macro ist definiert als M = (s₀, q_b, q_e, Q_M, PROB_M)

In PROB_M darf q_b nur auf der linken und q_e auf der rechten Seite einer Regel vorkommen



Turing Maschine mit eingeschränktem Alphabet (TM_a)

Umfasst nur Symbole $\{0, 1\}$

TM_a und TM_{on} können sich gegenseitig simulieren \Rightarrow äquivalent

Einseitig beschränkte Turingmaschine (TM_→)

Band ist nur in eine Richtung unendlich

Kopf kann nicht über linkes Ende hinaus bewegt werden

TM_a und TM_→ können sich gegenseitig simulieren \Rightarrow äquivalent

Linear beschränkte Turing Maschine TM_{↑...↓}

Darf nur so viele Zellen besetzen wie Eingabe umfasst

Symbole, l' und -l' begrenzen bzw. umrahmen die Eingabe

Der Kopf darf nicht darüber hinaus

Jede TM_{↑...↓} ist eine TM_a, aber nicht jede TM_a kann zu TM_{↑...↓} transf. werden, daher nicht äquiv.

Separates Eingabe-, Ausgabe- und Arbeitsband

Ein- und Ausgabe einseitig unendlich

Ein: keine Schreibbefehle, von l nach r lesen

Aus: keine Lesebefehle, von l nach r schreiben

Arbeitsbänder: zweiseitig unendlich, zu Beginn leer,
keine weiteren Einschränkungen

k-Sprungige TM fehlt

0	1	0	1	0	
0	0	1	1	0	
1	1	0	1	1	

$\alpha_1 \alpha_2 \alpha_3 \alpha_4 \alpha_5$

Endliche Automaten:

Allgemein:

$$A = (T, Q, R, q_0, F)$$

\equiv (Terminale, Zustände, Regeln, Startzustand, Finalzustände)

$(T \cup Q, R)$ ist Semi-Thue-System

$$T \cap Q = \emptyset$$

Für jede reguläre Grammatik G gibt es einen endlichen Automaten A, mit $L(G) = L(A)$.

Beweis (konstruktiv):

Form der

$$\begin{aligned} G &= (N, T, P, Z) \\ A &= (T, Q, R \cup \{f\}, R, q_0, \{F\}) \\ R = &\{q_X \rightarrow f \mid X \geq e \in P\} \cup \\ &\{q_X \rightarrow q_Y \mid X \geq Y \in P\} \cup \\ &\{q_X \rightarrow f \mid X \geq t \in P\} \cup \\ &\{q_X \rightarrow q_Y \mid X \geq t; Y \in P\} \cup \end{aligned}$$

q' oder $q \rightarrow q'$ mit $q, q' \in Q, t \in T$

Akzeptierte Sprache: $L(A) = \{w \in T^* \mid q_0 w \xrightarrow{*} q, q \in F\}$

DEA: Jedes Paar (q, t) hat höchstens ein q'

NEA: Mind. ein Paar (q, t) hat mehrere q'

Darstellungsformen:

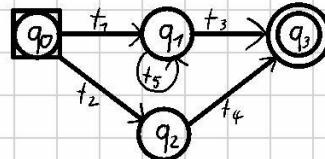
Zustandsübergangsdiagramm:

Startzustand q_0

Zustand q_1

Finalzustand q_3

Hinweis: wenn Loop in Diagramm
Sprache unendlich



Übergangstabelle:

	t_1	t_2	t_3	t_4
q_0	q_1	q_2	$q_{1,2}$	
q_1		q_2		q_3
q_2		q_2		q_3
q_3	q_4			

Wenn in jeder Zelle nur ein q' DEA
wenn mehrere (siehe (q_0, t_3) NEA)

Zusammenhänge:

Beweis

- Für jede reguläre G gibt es einen A mit $L(G) = L(A)$ und umgekehrt

- Für jeden regulären Ausdruck X gibt es einen A: $L(X) = L(A)$

- Zu jedem NEA gibt es einen DEA

- Es gibt einen eindeutigen minimalen EA

- NEA, OEA, Typ-3-Sprachen und reguläre Ausdrücke sind äquiv.

- bricht Berechnung ab, wenn kein Folgezustand oder Eingabe vollständig eingelesen

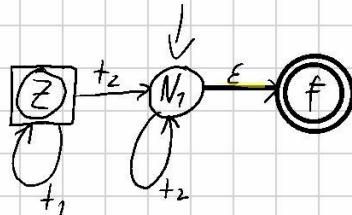
Konstruieren eines EA aus einer G:

$$G = (N, T, P, z) \quad P = (z \rightarrow t_1 z, z \rightarrow t_2 N_1, \\ N_1 \rightarrow t_2 N_2, N_2 \rightarrow \epsilon)$$

$A = (T, N \cup \{f\}, R, z, F)$ = Terminate, N sind Zustände + Finalzustand f , Regeln, Start-Nichtterm. ist Startzustand, Finalzustände

$$R = (z t_1 \rightarrow z, z t_2 \rightarrow N_1, N_1 t_2 \rightarrow N_2, N_1 \epsilon \rightarrow f)$$

Wie P nur T mit auf linker Seite

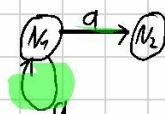


ist NEA wenn:

- ϵ -Übergang

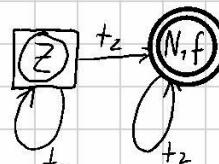
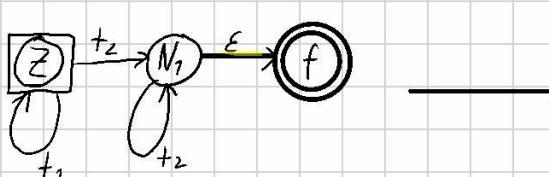
oder

- zwei mögliche Übergänge von einem Zustand mit gleicher Bedingung

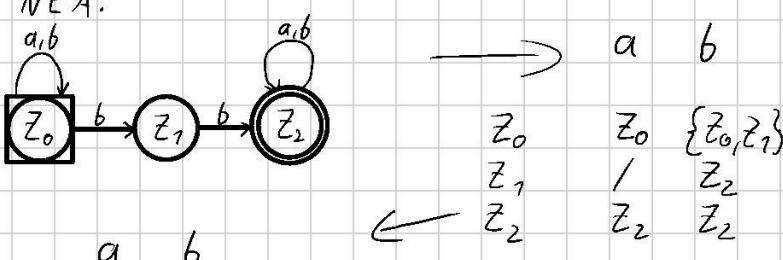


NEA in DEA umwandeln

bei ϵ -Übergang die zwei dadurch verbundenen Zustände zu einem kombinieren

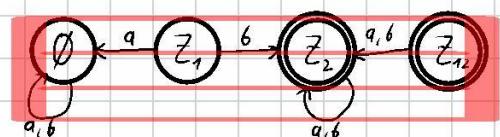
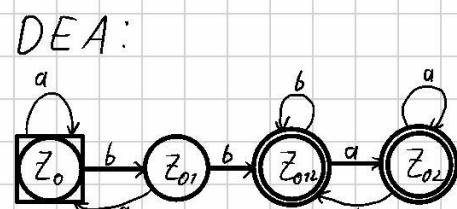


wenn mehrere Übergänge mit gleicher Bedingung NEA:



Z_0	Z_0	$\{Z_0, Z_1\}$
Z_1	\emptyset	Z_2
Z_2	Z_2	Z_2
Z_{01}	Z_0	Z_{012}
Z_{02}	Z_{02}	Z_{012}
Z_{12}	Z_2	Z_2
Z_{012}	Z_{012}	Z_{012}
\emptyset	\emptyset	\emptyset

alle die alten Endzustand Z_2 enthalten \rightarrow werden ebenfalls Endzustände



Dieser Teil kann weggelassen werden da nicht von Startzustand erreichbar

LL-Automaten & LL-Grammatik

Allgemein

- Linksrekursion:
 $E ::= E + E$

- Linksgleichheit:

$S ::= \text{if } E \text{ then } S$
| $\text{if } E \text{ then } S \text{ else } S$

LL bedeutet Eingabe von Links nach rechts und erzeugt Linksableitung

Probleme, die nur bei LL-A auftreten und Erstellung von Def. LL-A verhindern:

Linksrekursion und Linksgleichheit

LL(1)-Bedingung:

für $\forall X \in N$ mit Produktionen $X ::= \alpha$ und $X ::= \beta$ und $\alpha \neq \beta$ gilt:

$$\text{FIRST}(\alpha \text{ FOLLOW}(X)) \cap \text{FIRST}(\beta \text{ FOLLOW}(X)) = \emptyset$$

wenn Bedingung erfüllt, bauen von det. LL-Parsen bzw. det. vorausschauendem Top-Down-Parser möglich

$\text{FIRST}(S)$: Menge der terminalen Wörter (der Länge 1), die aus S abgeleitet werden können

$\text{FOLLOW}(S)$: Menge der terminalen Wörter (der Länge 1), die rechts vom Symbol S in Ableitung stehen können

LL-Kellerautomat Def.:

$$A = (T, \{q\}, R, q, \{q\}, NUT, \mathbb{Z})$$

Kellerinhalte sind T und N der G

$$R = \{ X_q \rightarrow x_n \dots x_1 q \mid X ::= x_1 \dots x_n \in P \}$$

U

$$\{ t q t \rightarrow q \mid t \in T \}$$

Eliminieren von Linksrekursion

$$A ::= A\alpha \mid \beta_1 \quad \rightarrow \quad A ::= \beta_1 A' \\ A' ::= \alpha A' \mid \epsilon$$

Eliminieren von Linksgleichheit

Ordnen Sie den verschiedenen Phasen eines Compilers deren Eingabe/Ausgabe zu:

Ausgabe des Assemblierers	<input checked="" type="checkbox"/> * Zum Quellprogramm im semantisch äquivalenten Programm im Objektcodeformat
Eingabe des Binders	<input checked="" type="checkbox"/> * Maschinencode von Programmen sowie Programmbibliotheken im Objektcodeformat
Ausgabe des Scanners	<input checked="" type="checkbox"/> * Folge von Tokens mit Attributen
Ausgabe des Binders	<input checked="" type="checkbox"/> * Ausführbares Programm
Eingabe der Transformationsphase	<input checked="" type="checkbox"/> * Abstrakter Syntaxbaum
Eingabe des Assemblierers	<input checked="" type="checkbox"/> * Programm in MaschinenSprache, idR, gespeichert in ASCII-Datei
Ausgabe der Transformationsphase	<input checked="" type="checkbox"/> * Programm in MaschinenSprache, idR, gespeichert in ASCII-Datei
Ausgabe des Parsers	<input checked="" type="checkbox"/> * Programm in MaschinenSprache, idR, gespeichert in ASCII-Datei
Ausgabe der der Codeerzeugung	<input checked="" type="checkbox"/> * Programm in MaschinenSprache, idR, gespeichert in ASCII-Datei
Eingabe der der Codeerzeugung	<input checked="" type="checkbox"/> * Abstrakter Syntaxbaum
Eingabe des Scanners	<input checked="" type="checkbox"/> * Datei mit ASCII-Daten, die im Objektcodeformat dargestellt werden
Eingabe des Parsers	<input checked="" type="checkbox"/> * Folge von Tokens mit Attributen

First Menge berechnen

- Alle N und T in Tabelle aufschreiben

$$E ::= i E'$$

- Alle T in Tabelle schreiben

$$E' ::= i E E' \mid \epsilon$$

- Für alle N Produktion durchgehen

First	E	E'	+	i
	i	ϵ	+	i

und First(N) in Spalte von N

eintragen

kontextfreie Sprache wird in

- $O(n^2)$ erkannt wenn eindeutig
- $O(n^3)$ erkannt wenn mehrdeutig

REX-Syntax	Bedeutung
a	ein Zeichen
abc	die Zeichenfolge abc
"abc"	die Zeichenfolge abc
\n	das NewLine-Zeichen
\\"	ein "-Zeichen
{abc}	Zeichenmenge
{a-z}	Zeichenbereich
-{abc}	Komplement

4a. falls p eine Produktion der Art: $Y \rightarrow \alpha X$ ist

 ➡ füge FOLLOW(Y) zu FOLLOW(X) hinzu.

 Merksatz: Steht X rechts aussen, dann füge FOLLOW der linken Seite der Produktion zu FOLLOW von X hinzu.

4b. falls p eine Produktion der Art: $Y \rightarrow \alpha X \beta$ und $\epsilon \notin \text{FIRST}(\beta)$ ist

 ➡ füge FIRST(β) zu FOLLOW(X) hinzu.

4c. falls p eine Produktion der Art: $Y \rightarrow \alpha X \beta$ und $\epsilon \in \text{FIRST}(\beta)$ ist

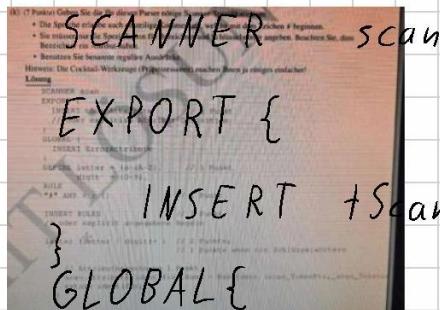
 ➡ füge sowohl FIRST(β)\{\epsilon\} als auch FOLLOW(Y) zu FOLLOW(X) hinzu.

 Merksatz: Ist ϵ in FIRST(β) enthalten, dann füge FIRST(β) ohne ϵ als auch FOLLOW der linken Seite der Produktion zu FOLLOW von X hinzu.

Scanner Altklausur-Aufgabe

Für Parser nötige Scanner-Spezifikation angeben

- einzeilige Kommentare mit # erlaubt
- nur Spezifikationen für Bezeichner und Schlüsselwörter angeben
 beachte: Bezeichner haben Attribut
- benutze benannte reguläre Ausdrücke



INSERT ErrorAttribute

}

```
DEFINE letter = {a-zA-Z}.
        digit = {0-9}.
```

RULE

"#" ANY +: {} // Kommentare

INSERT RULES // keine Ahnung

(letter | digit)* : // Bezeichner

{
 scan.Attribute.identifier.Name = MakeIdent(scan_TokenPtr, scan_TokenLength);
}
return identifier;

Top-Down - Parser (Rekursiver Abstiegsparser)

Wie funktioniert ein Top-Down-Parser? Welche Mengen müssen gebildet werden?

Für jedes Token/Nichtterminal wird Funktion definiert, die Teil der Eingabe akzeptiert (beginnend mit aktuellem Token)

Funktion liefert true, wenn Eingabeteil aus N abgeleitet werden kann bzw. dem Token entspricht, sonst false

Man muss First und Follow-Mengen bestimmen, damit vorausschauend entschieden werden kann, welche P aus Menge möglicher P eines N ausgewählt werden muss

Was bedeuten diese Mengen?

First(S): Menge der terminalen Wörter (Länge 1), die aus S abgeleitet werden können

Follow(S): Menge der terminalen Wörter (Länge 1), die rechts von dem Symbol S in einer Ableitung stehen können

Wie lautet die $LL(1)$ -Bedingung

Grammatik G ist $LL(1)$, wenn für $A := \alpha$ und $A = \beta$ mit ($\alpha \neq \beta$) gilt:

$$\text{First}(\alpha \text{ Follow}(A)) \cap \text{First}(\beta \text{ Follow}(A)) = \emptyset$$

Kann für linksrekursive G Top-Down-Parser erstellt werden?

Nein, aber kann transformiert werden und dadurch Linksrekursion aufgehoben werden, dass dann Parser erstellt werden kann.

(evtl. trotzdem nicht möglich, wenn mehrdeutig???)

LR-Automat

Regeln umgedreht aufschreiben

wenn $K \rightarrow (K)$ dann $(K)q \rightarrow Kq$

zusätzlich Shift-Regeln für jedes Terminal

$qi \rightarrow iq$

am Ende Finalisierung

$Kq\# \rightarrow q\#$ statt leer zu R wenn Ergebnis auf Stack bleiben soll

Bsp.: $1+2*3$ akzeptieren

R1	$E + Eq \rightarrow Eq$	S	$qi \rightarrow iq$
R2	$E - Eq \rightarrow Eq$	S	$q+ \rightarrow +q$
R3	$E * Eq \rightarrow Eq$	S	$q- \rightarrow -q$
R4	$(E)q \rightarrow Eq$	S	$q* \rightarrow xq$
R5	$Eq\# \rightarrow q\#$		
R6	$iq \rightarrow Eq$		

S $q 1+2*3\#$

R6 $i q 1+2*3\#$

S $E q 1+2*3\#$

S $E+ q 1+2*3\#$

R6 $E+ i q 1+2*3\#$

S $E+E q 1+2*3\# \leftarrow$ hier Shift oder Reduce möglich (Shift-Reduce-Konflikt)

S $E+E* q 1+2*3\# \quad$ bei linkassoziativität Reduce bei rechts shift

R6 $E+E*i q 1+2*3\#$

* vor + also Shift

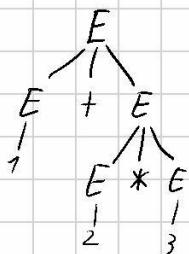
R3 $E+E*E q 1+2*3\#$

R1 $E+E q 1+2*3\#$

R5(F) $E q 1+2*3\#$

$q 1+2*3\#$

Ableitungsbaum dazu wäre



Bei Shift statt
Reduce mehr auf
Stack

wird von unten nach oben
verarbeitet

Reduce-Reduce-Konflikt wenn mehrere Regeln zutreffen
meist Tippfehler, Lösung ist First-Rule-Match oder Mehrzeichen-Vorausschau

Shift-Reduce-Konflikt wenn Shift oder Reduce möglich
Lösung sind Vorrangregeln oder einfach immer Shift

LARK-Spezifikationen angeben (Bsp. Taschenrechner)

PARSER	expr	// Parsername
SCANNER	expr-scan	// Scannername
PREC	LEFT '-' '+'	// Vorrangregeln weiter unten hat höheren Vorrang
	LEFT '*' '/'	// LEFT/RIGHT/NONE gibt Assoziativität an
:	NONE UNARY	// unäres -
	RIGHT '^'	

RULE

expr = <
= expr '+' expr.
= expr '-' expr.
= expr '*' expr.
= expr '/' expr.
= expr '^' expr.
= '-' expr PREC UNARY. // hier PREC angeben da Token, bei anderen ist
= '+' expr PREC UNARY. // Zeichen selbst Token
= '(' expr ')'.
= int-const.
= float-const.
= identifier.
>.

int-const: [Value: long] {Value:= 0}.

float-const: [Value: double] {Value:= 0.0}.

identifier: [Name: tIdent] {Name:= Noident}.

Mögliche Frage:

Wie Vorrang gelöst? hier durch explizite Angabe der Vorrangregeln
(siehe PREC)

Nötige Scanner-Spezifikation angeben:

Kellerautomat

```

bool f_i ()
{
    if (CurToken == i_tok) {
        curToken = scan_GetToken();
        return true;
    } else {
        // Syntaxfehler
        return false;
    }
}

bool f_eof ()
{
    if (CurToken == scanEofToken) {
        return true;
    } else {
        // Syntaxfehler
        return false;
    }
}

bool f_E1()
{
    if (CurToken ∈ {i}) {
        return f_i();
    }
    // Syntaxfehler
    return false;
}

bool f_E2()
{
    if (CurToken ∈ {*}) {
        return f_mult() && f_E1() && f_E2();
    }
    if (CurToken ∈ (#)) {
        return true;
    }
    // Syntaxfehler
    return false;
}

int main(...)

{
    int CurToken = scan_GetToken();
    ...
    if (f_E() && f_eof()) {
        // ok
    } else {
        // Syntaxfehler
    }
}

```

chen (nicht sicher)
ptieren nur reguläre)

$A = (T, \{q\}, R, q, \{q\}, N \cup T, \epsilon)$ $R = \{qt \rightarrow tq | t \in T\} \cup \{x_1 \dots x_n q \rightarrow Xq | X = x_1 \dots x_n \in P\} \cup \{tqt \rightarrow q\}$



AST Spezifikation

EXPRS = <

no-exprs =
exprs = Next: EXPRS REV.
 Expr: EXPR .

>.

// Baumaufbau in der Grammatik, linksrekursiv, da leere Liste links
exprs = <

= e: expr
{tree:=mexprs(mno_exprs(), e:tree); } ← wird generiert
= l: exprs ',' e: expr
{tree:= mexprs(l:tree, e:tree); }

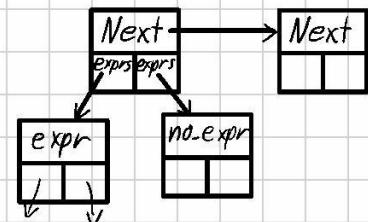
>.

// Deklaration der Nichtterminal Attribute
MODULE attributes

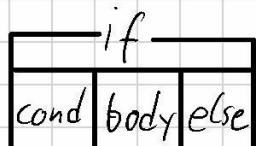
expr = [tree : tEXPR]
exprs = [tree : tEXPRS]

END attributes

mexprs gibt Baum zurück



If-else als AST



Namenskonvention:
name: typ → deklarieren
name::Var → Var-Zugriff

s: if-stmt(..) ← alle param übergeben

:- cond: double := evalExpr(s::cond)

IF cond
Then

interprete Stmt(s::body)

ELSE

interprete Stmt(s::else) ← kann auch leer
sein (else optional)

END

interprete Stmt(s::Next

