Lexeme = Strings von Zeichen → Symbole

Symbole → Symbolklassen / Token

lexikalische Analyse: Zerlegung in Lexeme → Symbolfolge

$w \in \Sigma^*$ ≙ String, Programmcode

↳ <u>Zerlegung</u> in $\alpha_1 \dots \alpha_n$ (α_i ≙ Regex)
hier: lm-Zerlegung (längsten Match finden)
ist eindeutig

↳ <u>Analyse</u>: Interpretation $\alpha_1 \dots \alpha_n$ als Tokenfolge $(T_{i_1} \dots T_{i_j})$
hier: Konvention erster Match → flm-Analyse
ist eindeutig, ex. im Fall von lm.

Sonst: Analyse nicht eindeutig

Backtrack-Automat berechnet eindeutige flm-Analyse, wenn diese existiert. Im „normal" Mode Match finden, dann im „backtrack" Mode versuchen Match zu erweitern. Wenn unproduktiver Zustand erreicht von back → normal, bzw. normal → lexer wechseln.
Worst-case Komplexität: $O(|w|^2)$ $(\alpha_1 = abc,\ \alpha_2 = (abc)^* d,\ w = (abc)^m)$

Syntax: analyse: l-Ableitung: Top-Down
r-Ableitung: Bottom-Up

TD-Analyseautomat (nicht-deterministisch) berechnet l-Ableitung

CB Kap 2 – Syntaxanalyse

Parser: Zerlegung der Symbolfolge vom Scanner in syntaktische Einheiten → Baumstruktur

Beschreibung mit CFGs, Links-/Rechtsanalyse, G eindeutig wenn für jedes Wort aus L(G) genau eine Links-/Rechtsableitung existiert

normal: CYG-Algorithmus (n^2 Platz / n^3 Zeit), hier lookahead: linear

TD: Linksanalyse / BU: gespiegelte Rechtsanalyse

Generalvoraussetzung: G ist reduziert: jedes Nichtterm erreichbar und produktiv

TD-Analyse: Kellerautomat mit Eingabe, Ausgabe (Regelnummern) und Keller (Spitze links, Satzformen).

Wenn linkes Kellersymbol = Eingabesymbol → Vergleichsschritt (deterministisch)

sonst Ableitungsschritt (nichtdeterministisch: $A \to \beta | \gamma$)

⇒ Determinismus durch k-lookahead → LL(k)-Grammatik

$\beta \in first_k(\alpha)$: Menge der Worte die durch α ableitbar sind, mit $|\beta| = k$, oder $|\beta| < k$ wenn Ende erreicht.

Problem des Rechtskontextes ($A \to \alpha\beta$), deshalb

$\beta \in follow_k(A)$: Menge der Worte die hinter A ableitbar, mit $|\beta| = k$

Für $k = 1$: lookahead-Menge einer Regel $A \to \beta$: $la(A \to \beta) = fi(\beta\, fo(A))$

$G \in LL(1)$ ⇔ besteht LL(1)-Test, d.h. la-Mengen von Alternativen sind disjunkt

$$A \to \beta | \gamma \Rightarrow la(A \to \beta) \cap la(A \to \gamma) = \emptyset$$

DTA durch Verwendung la-Mengen, action-Funktion: pop/accept/error

Parser: Berechnung la / Analysetabelle / Eindeutigkeit prüfen.

Problem Links-Rekursion: $A \to A\alpha | \beta$ ⇒ Kopf bleibt stehen $\neq LL(k)$

Beseitigung: $A \to \beta A'$, $A' \to \alpha A' | \varepsilon$

Allgemein: jede CFG → Greibach Normalform (GNF), aber nicht unbedingt LL(k)

Links-faktorisieren: $A \to \alpha\beta | \alpha\gamma \Rightarrow A \to \alpha A'$, $A' \to \beta | \gamma$

DTA: linearer Aufwand ($|w|+1$ Vergleichsschritte) $|N| \cdot (|w|+1)$ Transitionen

Kellerlänge max: $\max\{|\alpha| \mid A \to \alpha\} \cdot |N| \cdot (|w|+1)$

Rekursive-Descent-Parser: Prozedur je Nichtterm. Impliziter Keller

durch Rekursion

BU-Analyse: Automat mit Eingabe, Ausgabe, Keller (Spitze rechts)

Transitionen: shift (Eingabe-Symbol auf Keller schreiben)

reduce (Keller-Symbole ersetzen durch linke Regelseite und Regelnummer auf Ausgabe)

ist nicht deterministisch: reduce oder shift? / wieviel ersetzen / welche Regel / Ende?

letzteres Problem durch Startsepariertheit: $S' \to S$, keine andere Regel mit S'.

$(\varepsilon, w, \varepsilon) \vdash^* (S, \varepsilon, z)$

$\Rightarrow$ Determinismus durch k-lookahead: LR(k)

k=0: LR(0)-Auskünfte (für eine Regel und Keller)

Menge: LR(0)-Menge / Information (für einen Keller)

↳ endlich ∀ Kel $G \in LR(0)$ ⇝ keine widersprüchlichen Informationen → genau eine Reduceauskunft oder nur Shift-Auskünfte

Berechnung durch Potenzmengenkonstruktion aus $S' \to .S$ und ε-Abschluß.

Erhalten goto-Funktion, die für LR(0)-Menge + Symbol neue LR(0)-Menge ergibt

action-Funktion, die für LR(0)-Menge sagt: red. / shift / accept / error

→ muß eindeutig sein.

Analyseautomat: $(I_0, w, \varepsilon) \vdash^* (I_0 I, \varepsilon, z) \vdash (\varepsilon, \varepsilon, z_0)$

hier auf Keller nur Nummern der LR(0)-Mengen.

Transition: shift: linkes Symbol von Eingabe löschen, LR(0)-Nummer entspr. goto-Fkt auf Keller schreiben.

reduce$_i$: i auf Ausgabe, Anzahl der Symbole in Regel i nehmen und von Keller löschen, linke Regelseite von rechtes Kellersymbol ergeben mit goto-Fkt. neues Kellersymbol.

Bei Konflikten: lookahead $\Rightarrow$ SLR(1) kann ~~s/r~~-Konflikte lösen

$\boxed{A \to \cdot aB,\ A \to c\cdot}$ shift bei a, reduce wenn Eingabe ∈ fo(A)

immernoch Konflikte möglich LR(1): lookahead in Auskunft aufnehmen fi(+$)

$\boxed{S \to .S,\$}$ ∈ LR(1)(ε), bei Erweiterung Rechtskontext beachten: $\boxed{A \to \cdot B+,\$,\ B \to \cdot a,+}$

Problem: große Tabellen → falten äquivalenter Information (wo LR(0)-Anteile gleich)

↳ LALR(1) hier: spätere Fehlererkennung und r/r-Konflikte möglich,

aber keine s/r-Konflikte, da shift-Entscheidung aus LR(0)-Anteil folgt

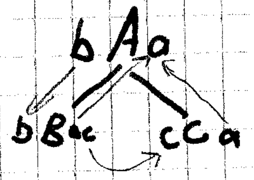
und der ist lt. Definition der Faltung identisch!

CB - Semantik / Attributierung

synthetisch : BU-Berechnung , Indexe 0 links / ≥ 1 rechts
inherited : TD-Berechnung , Indexe ≥ 1 links / 0 rechts

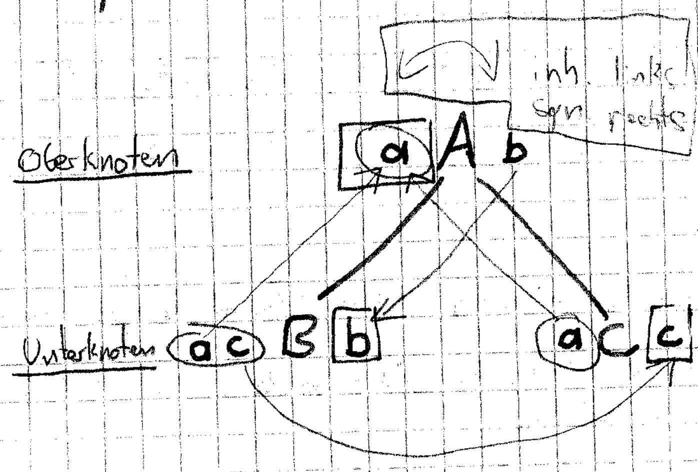
Innenvariablen: auf der linken Seite, also synthetische 0 / inherited ≥ 1
Außenvariablen: auf rechter Seite

Oberknoten : linke Regelseite (Indexe 0)
Unterknoten: rechte Regelseite (Index ≥ 1)



Beispiel: $A \rightarrow BC$

$a.0 = a.1 + a.2$
$b.1 = b.0$
$c.2 = c.1$



○ : synthetische
□ : Innenvariablen

Nur Kanten von Innen- zu Außenvariablen (⇒ keine Zirkularität innerhalb einer Regel)

Zirkularitäten entstehen durch Verklebung der Regeln im Ableitungsbaum

Abhängigkeitsgraph: Mengen synthetischer Var., die von inherited abhängen.
Im Oberknoten : •□○ , im Unterknoten •□○ → •□○ (= Deckel)

Abhängigkeitsmenge: $D(A,t)$ mit $\{(\alpha,\alpha') \mid \alpha \xrightarrow{A} \alpha' \text{ in } t\}$
Abhängigkeitssystem: $\mathcal{D}(A)$ mit $\{D(A,t) \mid A \text{ ist Wurzel von } t\}$
Induktiver Test: $\pi = A \rightarrow w \Rightarrow D[\pi] \in \mathcal{D}(A)$
$\pi = A \rightarrow A_1 A_2 A_3 \Rightarrow D[\pi; A_1 A_2 A_3] \in \mathcal{D}(A)$

Jeweils Abhängigkeiten im Oberknoten, durch Berücksichtigung der Abhängigkeiten der Unterknoten betrachten ($\alpha.0 \xrightarrow{A} \alpha'.0$) (Transitive Hülle bilden)
wenn mit Deckel trans ($\alpha.k \xrightarrow{A} \alpha.k$), $k \geq 1$ folgt ⇒ zirkulär
[...] bricht nach endlich vielen Schritten ab.

stark nicht-zirkulär : Vereinigen von ~~Att~~ Regeln $A \to a|b|c$ und gemeinsame Betrachtung $\Rightarrow$ polynomielle Laufzeit, hinreichendes Kriterium.

Berechnung von nicht zirkulären Grammatiken : synthetisches Attribut ≙ Funktion mit ~~att~~ inheriten als Parameter

<u>Spezialfall</u> : SAS : nur synth. Attr., Berechnung BU bei Syntaxanalyse, durch Attributwerte in zweite Kellerspalte. $\Rightarrow$ Ausgabe bei ACCEPT

$\Rightarrow$ AST (abstrakter Syntaxbaum) ~~st~~ statt Grammatik entsprechende Regeln mit Zuweisungen, Bedingungen, etc.

<u>mk-f</u> $((a_1, h_1), \ldots (a_n, h_n)) := (a, h)$

LAS : ~~inherits~~ inherite nur von synthetischen die links stehen

Berechnung durch Baumreise, mit jeden Knoten zweimal besuchen.

bei TD-Analyse:

1.) Ableitungsschritte als Expansion auf Keller (Punkt in Regel markiert Stelle)
neue Produktion in Keller, in zweiter Spalte berechnete Attribute des Oberknotens $\Rightarrow$ inherite

2.) Vergleichsschritte (Match) ohne Attributberechnung

3.) Reduce (Punkt am rechten Rand) : synthetische Berechnung und nach oben durchreichen, Kellereintrag entfernen.

CB4 – Zwischencodeerzeugung

BPS: Mehrfachdeklarationen und Static Scope (Dekl.-Umgebung, nicht Aufruf-Umgebung entscheidend). Programm ≙ Block ≙ Dekl. + Anweisungsteil

Zustandsraum: Programmcounter + Datenkeller (Spitze rechts) + Prozedurkeller (Sp. links)
Prozedurkeller: Aktivierungsblöcke mit stat. Verweis / dyn. Verweis / Rücksprungsadr. / lokale Variablen
Berechnung st. Verweis über base Funktion: base (ρ, dif)
≙ Anzahl Bytes um dif-Aktivierungsblöcke zu überspringen.

⟦CALL (ca, dif, loc)⟧ mit ca ≙ neue Codeadresse, dif ≙ Leveldiff., loc ≙ # lok. Variablen

$(m, d, \rho) \longrightarrow (ca, d, sv; dv : ra : lv : \rho)$

neue Adr. (ca), unverändert (d), neuer Prozedurkeller (sv; dv : ra : lv), alter P-Keller (ρ)

mit lv ≙ 0 : 0 : … : 0 (loc = # lokaler Variablen)

ra ≙ m+1 (nach CALL-Befehl weitermachen)

dv ≙ loc+2 (lok. Variablen und ra überspringen)

sv ≙ base (ρ, dif) + dv (akt. und weitere Aktivierungsblöcke überspringen)

⟦RET⟧ Rücksprung: vom Prozedurkeller die ra holen und Block löschen (anhand dv)

$(m, d, \rho.1 \ldots \rho.t) \longrightarrow (\rho.3, d, \rho.2 \ldots \rho.t)$

an Pos 3 steht ra (ρ.3), unverändert (d), ρ.2 beginnt bei ρ.2 (dv) + 2

⟦LOAD (dif, off)⟧ Speicherinhalt auf Datenkeller mit dif ≙ Leveldifferenz und off ≙ Nummer der Variable

$(m, d, \rho) \longrightarrow (m+1, d : nv, \rho)$

nächster Befehl (m+1), nv auf Keller (d : nv), unverändert (ρ)

mit $nv = \rho.$ base $(\rho, dif) + 2 + off$

zum A-Block indem deklariert (base (ρ, dif)), dv+ra überspringen (2), Offset (off)

STORE äquivalent: löscht vom Datenkeller und ändert ent. Position im Prozedurkeller

LIT z: legt z auf Datenkeller und führt nächste Anweisung aus.

Symboltabelle: Bezeichner (const, var, proc) + Parameter (Wert / Level, off / Adr, Level, size)

up ($\Delta_c \Delta_v \Delta_p$, st, a, l) = hintereinander updaten für c/v/p, wenn Bez. verschieden.

Deklarationen, alte st, [illegible] Adr, Level … up (Δ_c … up (Δ … up (Δ_p, st, a, l) st …)

für $\Delta := \varepsilon$ keine Änderung

für const : $I_1 = z_1 \dots I_n = z_n$ updaten mit $I_1 / (\text{const}, z_1) \dots I_n / (\text{const}, z_n)$

für var : $I_1 = \dots I_n$ updaten mit $I_1 / (\text{var}, l, 1) \dots I_n (\text{var}, l, n)$

(l: Level, 1 … n: Offset)

für proc : $I_1 = B_1 \dots I_n = B_n$ updaten mit $I_1 / (\text{proc}, a_1, l, size(B_1)) \dots I_n / (\text{proc}, a_n, l, size(B_n))$

(a_1: freie Adressen, l: Level, size: # lok. Variablen in B)

Startzustand $s = (1, \varepsilon, 0{:}0{:}0 : z_1 \dots z_n)$

- 1: Programm startet bei Adresse 1
- ε: DK ist leer
- 0:0:0: PK mit SV / dV / ra
- $z_1 \dots z_n$: lokale (hier Eingabevar).

Symboltabelle entsprechend $st_{I/O}(I_j) \hat{=} (\text{var}, 0, j)$

(var: Variable, 0: Level 0, j: Offset 1 … n)

Funktion trans die in Zwischencode transformiert

trans (in/out $I_1 \dots I_n$, B) :=
1: CALL (a_Γ, 0, size(B)) — a_Γ: Startadresse von Γ, 0: Level 0, size(B): # lok. Var.
2: JMP 0 — Standard Stopmarke
bt (B, $st_{I/O}$, a_Γ, 1)

(Block $B := \Delta\Gamma$, $st_{I/O}$: initiale Symtab, a_Γ: Startadresse, 1: Level 1)

Blocktrans: Dekl. übersetzen (Prozeduren) + Code/Anweisungen + RET

$bt(\Delta\Gamma, st, a, l) := dt(\Delta, up(\Delta, st, a_{new}, l), a_{new}, l), ct(\Gamma, up(\Delta, st, a_{new}, l), a, l), RET$

Dekltrans: nur für Proc. (jeweils bt aufrufen):

$dt(\Delta_c \Delta_v \Delta_p, st, a, l) := dt(\Delta_p, st, a, l)$ mit $\Delta_p = \text{proc } I_1 B_1 \dots I_n B_n \hat{=} bt(B_1, st, a_1, l+1) \dots bt(B_n, st, a_n, l+1)$

$st(I_j) := (\text{proc}, a_j, l+1, size(B_j))$

Zuweisung: Ausdruck auswerten (et) und in P-Keller: $ct(I := E, st, a, l) = et(E, st, a, l), STORE(l - dl, off)$ hier $st(I) := (\text{var}, dl, off)$

Proc.aufruf: CALL benutzen: $ct(I(), st, a, l) = CALL(ca, l - dl, loc)$, mit $st(I) := (\text{proc}, ca, dl, loc)$

Bedingung: bool'schen Ausdruck auswerten, entsprechende Verzweigungen und Code erzeugen

while: bool'schen Ausdruck, Rücksprung, Code, Rücksprung hier Auswertung

Ausdruck (arith): $z \hat{=} LIT\ z$ (auf DK), $I \hat{=}$ LIT oder LOAD auf DK, $+ \hat{=} 2 * et$, ADD (Postfixnotation)

Ausdruck (bool): $< \hat{=} 2 * et$, LT / $\neg \hat{=} sbt$, NOT / $\wedge \hat{=} 2 * sbt$, AND

sbt $\hat{=}$ strikt boolean $\Longleftrightarrow$ nbt $\hat{=}$ (non strikt / sequential) / nct $\hat{=}$ non strict ct (while/if)

Jumping code: kürzere Laufzeit, längerer Code, veränderte Semantik (Überspringen von ∞-Berechnungen)

$nbt(E, st, a, a_t, a_f, l) \Longleftrightarrow sbt(E, st, a, l)$

nbt ($E_1 < E_2$) wie sbt, aber anschließend Sprung nach Adresse für true / false

nbt (not) : Vertauschen von true und false — $nbt(\text{not } E, st, a, a_t, a_f, l) := nbt(E, st, a, a_f, a_t, l)$

nbt (and / or) : 1 * nbt mit überspringen des folgenden nbt's im Fall false / true

nct (if) : nbt mit Teil für nct's für true und false, nach true nct ans Ende springen

nct (while) : nbt mit Rücksprung für false, nct, JMP – Beginn