

1 Einleitung

Einleitung

- Compiler = Programm: Source code → Machine code

Issues

- Korrektheit (Compilerverifikation, proof-carrying code)
- Effizienz von generiertem Code (Programmanalyse und Optimierung)
- Effizienz vom Compiler (fortgeschrittene Algorithmen und Datenstrukturen: bootstrapping)

Aspekte von Programmiersprachen

syntax Wie sieht das Programm aus?

semantics Was ist die Bedeutung des Programms.

Pragmatics Länge und Verständlichkeit des Programms, Erlernbarkeit der Programmiersprache, Eignung für bestimmte Anwendungen

Compiler-Phase

Lexikalische Analyse (Scanner)

- Erkennung von Symbolen, Trennzeichen (delimiters) und Kommentaren
- durch Reguläre Ausdrücke und endliche Automaten.

Syntaktische Analyse (Parser)

- festlegen der hierarchischen Programmstruktur
- durch kontextfreie Grammatiken und Kellerautomaten (pushdown automata)

Semantische Analyse

- überprüfen von Kontextabhängigkeiten, Datentypen, ...
- durch Attribut-Grammatiken

Erzeugung von Zwischencode

- übersetzen in (zielunabhängigen) Zwischencode
- durch Baumübersetzung (tree translation)

Code Optimierung um Laufzeit und/oder Speicherbedarf zu verbessern

Erzeugung von Maschinencode ans Zielsystem gebunden

Zusätzlich Optimierung von Zielcode, Symboltabelle und Fehlerbehandlung

Klassifizierung der Compilerphasen

Analyse lexikalische, syntaktische, semantische Analyse

Synthese Erzeugung von (Zwischen- / Maschinen-) Code + Optimierung.

Alternativ:

Frontend maschinenunabhängige Teile (Analyse + Zwischencode + maschinenunabhängige Optimierungen)

Backend maschinenabhängige Teile (Erzeugung + Optimierung von Maschinencode)

n-pass Compiler

Anzahl an Läufen durch den Sourcecode (heute im Allgemeinen nur noch 1 Durchlauf: one-pass)

2 Lexikalische Analyse

Lexikalische Struktur Programm P ist Zeichensequenz:

Ω endl. Zeichenmenge (Unicode/ASCII/etc.)

$a, b, c, \dots \in \Omega$ Zeichen (= lexikalische Atome oder auch Lexeme)

$P \in \Omega^*$ Quellprogramm (nicht jedes $w \in \Omega^*$ ist gültiges Programm)

Vereinfachung für Benutzer: Natürliche Sprache für keywords, Bezeichner; Leerzeichen, Einrückungen; Kommentare und Compiler Anweisungen.

1. Symbole werden durch eine Sequenz von *Lexemen* (lexikalische Atome) dargestellt.
2. \Rightarrow Lexikalische Analyse muss zuerst P in eine Sequenz von Lexemen zerlegt werden.
3. Unterschiede von ähnlichen Lexemen sind meist egal (Bezeichner, ...)
 - Lexeme sind in *Symbolklassen* gruppiert (Bezeichner/Zahlen/...)
 - Symbolklassen werden abstrakt durch *tokens* dargestellt
 - Symbole werden durch zusätzliche Attribute identifiziert (Bezeichnernamen, numerische Werte)
 - \Rightarrow Symbol = (token, attribute) zB (int,5)
 - \Rightarrow zweites Ziel ist es eine Lexemsequenz in eine Symbolsequenz zu verwandeln.

Das Ziel der lexikalischen Analyse ist der Zerlegung des Quellprogramms in eine Lexemsequenz und dessen Transformation in eine Symbolsequenz.

Wichtige Symbolklassen

Bezeichner

- Variablennamen, Konstanten, Typen, Prozeduren, Klassen, ...
- i.A. Sequenz von Buchstaben und Zahlen, Anfang mit Buchstaben
- i.A. keine keywords, mögliche Längenbeschränkung

Keywords

- Bezeichner mit vorgegebener Bedeutung
- Darstellung von Kontrollstrukturen (`while`), Operatoren (`and`), ...

Zahlen Zahlsequenz, `+`, `-`, Buchstaben (exponential/hexadezimal Darstellung)

einfache Symbole

- ein spezielles Zeichen: `+`, `*`, `<`, `(`, `,`, ...
- jedes ist eigene Symbolklasse: `plus`, ...

zusammengesetzte Symbole

- zwei oder mehr Zeichen: `:=`, `**`, `<=`, ...
- jedes ist eigene Symbolklasse: `gets`, ...

white spaces

- Leerzeichen, Tabs, Zeilenumbrüche, ...
- i. A. um Symbole zu trennen (Ausnahme: FORTRAN)

symbol = (token, attribute)

token binäre Bezeichnung der Symbolklasse (`id`, `gets`, `plus`, ...)

attribute zusätzliche Informationen für spätere Compilerphasen:

- Referenzen zur Symboltabelle
- Werte von Zahlen
- ...
- normalerweise leer für einfache (singleton) Symbolklassen

Symbolklassen sind *Reguläre Mengen*:

- Spezifiziert durch *reguläre Ausdrücke*
- Erkennbar durch endliche Automaten
- erlaubt automatische Erzeugung von scannern (`[f]lex`)

Zeit- und Platzkomplexität der DFA Methode (RegEx \rightarrow Thompsonautomat \rightarrow ... \rightarrow DFA)

1. Kleen: $\mathcal{O}(|\alpha|)$
2. Potenzmengenkonstruktion: $\mathcal{O}(2^{|\mathfrak{A}_\alpha|})$ ($|\mathfrak{A}_\alpha| := |Q|$), $\mathfrak{A}_\alpha =$ Automaten
3. Wortproblem: $\mathcal{O}(|\omega|)$

\Rightarrow Gutes Laufzeitverhalten, aber hoher Platzbedarf (auch exponentielle Zeit in Konstruktionsphase).

Zeit- und Platzkomplexität der NFA Methode

Platz $\mathcal{O}(|\alpha|)$ zum Speichern von T (Transitionen)

Zeit $\mathcal{O}(|\alpha| \cdot |\omega|)$ in der Schleife müssen $|T|$ Zustände berücksichtigt werden.

⇒ Tauschen von exponentiellem Platzbedarf für langsamere Laufzeit.

Method	Space	Time
DFA	$\mathcal{O}(2^{ \mathcal{A}_\alpha })$	$\mathcal{O}(\omega)$
NFA	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha \cdot \omega)$

In der Praxis taucht aber das exponentielle Aufblasen der DFA Methode in realen Anwendungen nicht auf (⇒ [f]lex)
Verbesserung der NFA Methode durch Zwischenspeichern der Transitionen: $\hat{\delta}(T, a) \Rightarrow$ Kombination beider Methoden.

Das erweiterte Matching Problem Gegeben $\alpha_1, \dots, \alpha_n \in RE_\Omega$ und $\omega \in \Omega^*$, entscheide ob eine Zerlegung von ω bezüglich $\alpha_1, \dots, \alpha_n$ existiert und gebe eine entsprechende Analyse an.

Sicherstellung der Eindeutigkeit Zwei Ansätze:

1. longest match (maximal munch tokenization)

- für Eindeutigkeit von Zerlegungen
- macht Lexeme so lange wie möglich
- motiviert durch Anwendungen: normalerweise ist jedes (nichtleere) Präfix eines Bezeichners auch ein Bezeichner.

2. first match

- für Eindeutigkeit der Analyse
- wähle first matching RE in der gegebenen Ordnung

FLM-Analyse

Eingabe Ausdruck $\alpha_1, \dots, \alpha_n \in RE_\Omega$, Tokens $\{T_1, \dots, T_n\}$ und Wort ω

Verfahren

1. $\forall i \in \{1, \dots, n\}$ konstruiere Automaten $\mathcal{A}_i \in DFA_\Omega | L(\mathcal{A}_i) = \llbracket \alpha_i \rrbracket$ (DFA-Methode)
2. konstruiere den *Produktautomaten* $\mathcal{A} \in DFA_\Omega | L(\mathcal{A}) = \bigcup_{i=1}^n \llbracket \alpha_i \rrbracket$
3. partitioniere die Endzustandsmenge von \mathcal{A} um die first-match Bedingung zu erfüllen (F_1, \dots, F_n) .
4. Erweitere den daraus entstehenden DFA zu einem *backtracking-DFA*, welcher dem longest-match Prinzip folgt, und lass diesen auf ω laufen.

Ausgabe FLM-Analyse von ω , falls diese Existiert.

Backtracking DFA Ziel ist es \mathfrak{A} zu einem Backtracking-DFA \mathfrak{B} zu erweitern, welcher zu dem Eingabeband zwei Zeiger hinzufügt. Einen *backtracking-Kopf* um den letzten match zu markieren und einen *lookahead* um den longest-match zu bestimmen. Eine Konfiguration von \mathfrak{B} besteht aus drei Teilen ($\Delta := \{T_1, \dots, T_n\}$ (Tokenmenge)):

1. *mode* $m \in \{N\} \uplus \Delta$
 - $m = N$ (normal): suche first-match (kein Endzustand erreicht)
 - $m = T \in \Delta$: Token T wurde erkannt, suche nach möglichen längeren Match.
2. *Eingabeband* $vq\omega \in \Omega^* \cdot Q \cdot \Omega^*$
 - v : lookahead Teil der Eingabe ($v \neq \epsilon \Rightarrow m \in \Delta$)
 - a : erstes Zeichen von ω
 - q : aktueller Zustand von \mathfrak{A}
 - ω : restliche Eingabe
3. *Ausgabeband* $W \in \Delta^* \cdot \{\epsilon, \text{lexerr}\}$
 - Δ^* : Folge von bisher erkannten Tokens
 - lexerr : ein lexikalischer Fehler ist aufgetreten (z.B. erreichen eines nichtproduktiven Zustandes, ...)

BACKTRACKING DFA

Konfigurationsmenge von \mathfrak{B} : $(\{N\} \uplus \Delta) \times \Omega^* \cdot Q \cdot \Omega^* \times \Delta^* \cdot \{\epsilon, \text{lexerr}\}$

Startkonfiguration: $(N, q_0\omega, \epsilon)$ Transitionen (mit $q' := \delta(q, a)$):

- normal mode: nach match suchen

$$(N, qa\omega, W) \vdash \begin{cases} (N, q'\omega, W), & \text{if } q' \in P \setminus F \\ (T_i, q'\omega, W), & \text{if } q' \in F^{(i)} \\ \mathbf{output: } W \cdot \text{lexerr}, & \text{if } q' \notin P \end{cases}$$

- backtrack mode: suche nach longest-match

$$(T, vqa\omega, W) \vdash \begin{cases} (T, vaq'\omega, W), & \text{if } q' \in P \setminus F \\ (T_i, q'\omega, W), & \text{if } q' \in F^{(i)} \\ (N, q_0va\omega, WT), & \text{if } q' \notin P \end{cases}$$

- Eingabeende

$$\begin{aligned} (N, q, W) &\vdash \mathbf{output:} W \cdot \text{lexerr} && \text{if } q \in P \setminus F \\ (T, q, W) &\vdash \mathbf{output:} WT && \text{if } q \in F \\ (T, vaq, W) &\vdash (N, q_0va, WT) && \text{if } q \in P \setminus F \end{aligned}$$

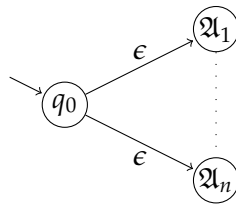
$$(N, q_0\omega, \epsilon) \vdash^* \begin{cases} W \in \Delta^*, & \text{if } W \text{ ist die FLM-Analyse von } \omega \\ W \cdot \text{lexerr}, & \text{if keine FLM-Analyse von } \omega \text{ existiert} \end{cases}$$

Anmerkungen:

- Zeitkomplexität: $\mathcal{O}(|\omega|^2)$ im worst case
- Kann durch die *tabular method* verbessert werden (good to know)

Backtracking NFA

1. $\mathcal{A}_i \in NFA_{\Omega}$
2. "Produkt"automat:



3. Partitionierung der Endzustände
4. Backtrackingautomat wie bei DFA

3 Syntaktische Analyse

3.1 Vorlesung 5

Syntaktische Analyse Das Ziel der syntaktischen Analyse ist es die syntaktische Struktur eines Programms, durch eine Folge von Token gegeben, nach einer CFG zu bestimmen (parser).

Syntaxbäume, Ableitungen und Wörter Die Beziehung zwischen Syntaxbäumen, Ableitungen und erzeugten Worten ist nicht eindeutig.

1. Ein Syntaxbaum repräsentiert i.A. mehrere Ableitungen
2. Eine Ableitung kann i.A. durch mehrere Bäume repräsentiert werden
3. Ein Wort kann i.A. durch mehrere Ableitungen erzeugt werden
4. Ein Wort kann mehrere Syntaxbäume haben

Aber:

1. Jeder Syntaxbaum erzeugt genau ein Wort (Konkatenation der Blätter)
2. Jeder Syntaxbaum gehört zu genau einer Linksableitung (leftmost) und umgekehrt
3. Jeder Syntaxbaum gehört zu genau einer Rechtsableitung (rightmost) und umgekehrt

Reduced CFG Eine Grammatik $G \in \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ wird *reduced* genannt, wenn für jedes $A \in N$ ein $\alpha, \beta \in X^*$ und $w \in \Sigma^*$ existiert, so dass

- $S \Rightarrow_* \alpha A \beta$ (A reachable) and
- $A \Rightarrow_* w$ (A productive)

2 Ansätze Beim Parsen von CFGs gibt es 2 Ansätze (Syntaxbaum)

1. Top-Down Analyse: Root \rightarrow Leaves
2. Bottom-Up Analyse: Leaves \rightarrow Root

3.2 Vorlesung 6

Nondeterministic Pushdown Automaton (PDA)

1. $G \in CFG_\Sigma$, PDA soll $L(G)$ akzeptieren und zugehörige Linksableitung berechnen:
2. Entfernen von Nichtdeterminismus durch Einführung von *lookahead*. $G \in LL(k) \Leftrightarrow G$ wird von PDA erkannt mit Lookahead von k .

NONDETERMINISTIC PDA-KONFIGURATIONEN

- Σ : Eingabealphabet; X : Kelleralphabet; $[p]$: Ausgabealphabet
- Konfigurationen: $\Sigma^* \times X^* \times [p]^*$ (top of pushdown to the left)
expansion $\pi(i) = A \rightarrow \beta \Rightarrow (\omega, A\alpha, z) \vdash (\omega, \beta\alpha, zi)$
matching $\forall a \in \Sigma, (a\omega, a\alpha, z) \vdash (\omega, \alpha, z)$
- Startkonfiguration: $(\omega, S, \epsilon); \omega \in \Sigma^*$
- Endkonfiguration: $\{\epsilon\} \times \{\epsilon\} \times [p]^*$

	$((a) * b, E, \epsilon)$	$\vdash ((a) * b, T, 2)$
	$\vdash ((a) * b, T * F, 23)$	$\vdash ((a) * b, F * F, 234)$
$G_{AE} : E \rightarrow E + T T$	(1,2) $\vdash ((a) * b, (E) * F, 2345)$	$\vdash (a) * b, E) * F, 2345)$
$T \rightarrow T * F F$	(3,4) $\vdash (a) * b, T) * F, 23452)$	$\vdash (a) * b, F) * F, 234524)$
$F \rightarrow (E) a b$	(5,6,7) $\vdash (a) * b, a) * F, 2345246)$	$\vdash () * b,) * F, 2345246)$
	$\vdash (*b, *F, 2345246)$	$\vdash (b, F, 2345246)$
	$\vdash (b, b, 23452467)$	$\vdash (\epsilon, \epsilon, 23452467)$

Lookahead und $fi_k(\alpha)$ Hinzufügen von Lookahead von k Zeichen um Nichtdeterminismus von NTA(G) aufzulösen.
 \Rightarrow Festlegen der Expansion der Reduktion auf die nächsten k Symbole.

3.3 Vorlesung 7

LL(k)
 Linksanalyse mit k -Lookahead berechnen

$G \in LL(k)$, G hat Eigenschaft $LL(k)$ wenn für alle Linksableitungen der Form: $S \Rightarrow_l^* wA\alpha$ $\left\{ \begin{array}{l} \Rightarrow_l w\beta\alpha \Rightarrow_l^* wx \\ \Rightarrow_l w\gamma\alpha \Rightarrow_l^* wy \end{array} \right.$

gilt: $\beta = \gamma$ ■

$G \in LL(k) \Leftrightarrow$ der Linksableitungsschritt für $wA\alpha$ durch die nächsten k Symbole nach w eindeutig festgelegt ist.

$$\begin{aligned}
 \mathbf{fi / fo} \quad & fi = first_1, fo = follow_1, \Sigma_\epsilon = \Sigma \cup \{\epsilon\} \\
 fi(A) = & \begin{cases} \{a\} \cup \{\dots\} : A \rightarrow a | \dots, \text{ mit } a \in \Sigma \\ \{\epsilon\} \cup \{\dots\} : A \rightarrow \epsilon | \dots \\ fi(B) \cup fi(C) : A \rightarrow B | C \\ fi(C) : A \rightarrow BC \wedge B \rightarrow \epsilon \\ fi(B) \cup fi(C) : A \rightarrow BC \wedge B \rightarrow \epsilon | \dots \\ fi(B) : A \rightarrow BC \wedge B \not\rightarrow \epsilon \end{cases} \\
 fo(A) = & \begin{cases} \epsilon \in fo(S) \\ A \rightarrow \alpha B \beta \ a \in fo(B) \Leftrightarrow a \in fi(\beta) \\ A \rightarrow \alpha B \beta \ x \in fo(B) \Leftrightarrow \epsilon \in fi(\beta) \wedge x \in fo(A) \end{cases}
 \end{aligned}$$

la-Mengen

1. $la\text{-Mengen} = la(A \rightarrow \beta) := fi(\beta \cdot fo(A)) \subseteq \Sigma_\epsilon$
 $\epsilon \in la(A \rightarrow \beta) \Leftrightarrow \beta \Rightarrow^* \epsilon$ und $\epsilon \in fo(A)$
2. $\forall a \in \Sigma, a \in la(A \rightarrow \beta) \Leftrightarrow a \in fi(\beta) \vee (\beta \Rightarrow^* \epsilon \text{ und } a \in fo(A))$
 $G \in LL(1) \Leftrightarrow \forall$ Regelpaare der Form $A \rightarrow \beta | \gamma \in P$ mit $\beta \neq \gamma$ gilt: $la(A \rightarrow \beta) \wedge la(A \rightarrow \gamma) = \emptyset$

3.4 Vorlesung 8

BEISPIEL 8.4

$$L(G'_{AE}) = L(G_{AE})$$

$$\begin{aligned}
 G'_{AE} : E & \rightarrow TE' \\
 E' & \rightarrow +TE' | \epsilon \\
 T & \rightarrow FT' \\
 T' & \rightarrow *FT' | \epsilon \\
 F & \rightarrow (E) | a | b
 \end{aligned}$$

$A \in N$	$fi(A)$	$fo(A)$
E	$\{(\cdot, a, b)\}$	$\{\epsilon, \cdot\}$
E'	$\{+, \epsilon\}$	$\{\epsilon, \cdot\}$
T	$\{(\cdot, a, b)\}$	$\{+, \epsilon, \cdot\}$
T'	$\{*, \epsilon\}$	$\{+, \epsilon, \cdot\}$
F	$\{(\cdot, a, b)\}$	$\{*, +, \epsilon, \cdot\}$

$A \rightarrow \beta \in P$	$la(A \rightarrow \beta) = fi(\beta \cdot fo(A))$
$E \rightarrow TE'$	$\{(\cdot, a, b)\}$
$E' \rightarrow +TE'$	$\{+\}$
$E' \rightarrow \epsilon$	$\{\epsilon, \cdot\}$
$T \rightarrow FT'$	$\{(\cdot, a, b)\}$
$T' \rightarrow *FT'$	$\{*\}$
$T' \rightarrow \epsilon$	$\{+, \epsilon, \cdot\}$
$F \rightarrow (E)$	$\{(\cdot)\}$
$F \rightarrow a$	$\{a\}$
$F \rightarrow b$	$\{b\}$

$\Rightarrow G'_{AE} \in LL(1)$

Deterministisches Top-Down Parsing und $DTA(G)$

1. Sicherstellen $G \in LL(1)$ (Berechnen der LA-Mengen)
 2. $NTA(G)$ (Nichtdeterministischen Top-Down Parsing Automat) konstruieren
 3. Benutze 1-Symbol LA um Expansionsschritte eindeutig zu machen:
 - $(a\omega, A\alpha, z) \vdash (a\omega, \beta\alpha, zi)$, if $\pi(i) = A \rightarrow \beta \wedge a \in la(\pi(i))$
 - $(\epsilon, A\alpha, z) \vdash (\epsilon, \beta\alpha, zi)$, if $\pi(i) = A \rightarrow \beta \wedge \epsilon \in la(\pi(i))$
 - wie vorher: $(a\omega, a\alpha, z) \vdash (z, a, z)$
- $\Rightarrow DTA(G)$ (Deterministic Top-Down Parsing Automat)

Lookahead hat 2 Vorteile

- Beseitigung des Nichtdeterminismus
- Syntaxfehler werden eher entdeckt

Einführen von Action-Funktionen $act: \Sigma_\epsilon \times X_\epsilon \rightarrow \{(\alpha, i) | \pi(i) = A \rightarrow \alpha\} \cup \{pop, accept, error\}$

G'_{AE} mit entsprechenden la-Mengen: (empty = error)

act	E	E'	T	T'	F	a	b	()	*	+	ϵ
a	(TE', 1)		(FT', 4)		(a, 8)	pop					
b	(TE', 1)		(FT', 4)		(b, 9)		pop				
((TE', 1)		(FT', 4)		((E), 7)			pop			
)		(ϵ , 3)		(ϵ , 6)				pop			
*				(*FT', 5)					pop		
+		(+TE', 2)		(ϵ , 6)						pop	
ϵ		(ϵ , 3)		(ϵ , 6)							accept

Linksanalyse von $(a) * b$

((a) * b,	E	, ϵ)	⊢	() * b,	E')T'E'	, 1471486)
⊢	((a) * b,	TE'	, 1)	⊢	() * b,)T'E'	, 14714863
⊢	((a) * b,	FT'E'	, 14)	⊢	(*b,	T'E'	, 14714863
⊢	((a) * b,	(E)T'E'	, 147)	⊢	(*b,	*FT'E'	, 147148635
⊢	(a) * b,	E)T'E'	, 147)	⊢	(b,	FT'E'	, 147148635
⊢	(a) * b,	TE')T'E'	, 1471)	⊢	(b,	bT'E'	, 1471486359
⊢	(a) * b,	FT'E')T'E'	, 14714)	⊢	(ϵ ,	T'E'	, 1471486359
⊢	(a) * b,	aT'E')T'E'	, 147148)	⊢	(ϵ ,	E'	, 14714863596
⊢	() * b,	T'E')T'E'	, 147148)	⊢	(ϵ ,	ϵ	, 147148635963

Transformation zu $LL(1)$

Transform G in $G' \in LL(1)$

1. Entfernen der Linksrekursion
2. Linksfaktorisierung
 - Erhält die Semantik, aber nicht die Syntax
 - Nicht jede Grammatik kann in $LL(1)$ umgewandelt werden.

Linksrekursion

- $A \Rightarrow^+ A\alpha$.
- G Linksrekursiv $\Rightarrow G \notin \bigcup_{k \in \mathbb{N}} LL(k)$

direkte Linksrekursion ersetzen durch Rechtsrekursion

$A \rightarrow A\alpha|\beta_1|\dots|\beta_n; \alpha \neq \epsilon, \beta_i \neq A \dots$ ersetzen mit

$$A \rightarrow \beta_1 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha A' | \epsilon$$

$G_{AE} :$	$E \rightarrow E + T$	$ $	T		
	$T \rightarrow T * F$	$ $	F		
	$F \rightarrow (E)$	$ $	a	$ $	b

	$E \rightarrow TE'$				
	$E' \rightarrow +TE'$	$ $	ϵ		
	$T \rightarrow FT'$				
	$T' \rightarrow *FT'$	$ $	ϵ		
	$F \rightarrow (E)$	$ $	a	$ $	b

indirekte Linksrekursion

$$A \rightarrow A_1 \alpha_1 \dots$$

$$A_1 \rightarrow A_2 \alpha_2 \dots$$

$$\vdots$$

$$A_{n-1} \rightarrow A_n \alpha_n \dots$$

$$A_n \rightarrow A \beta \dots$$

in Greibach Normalform umwandeln, mit Produktionen der Art:

$$A \rightarrow a B_1 \dots B_n; B_i \neq S$$

$$S \rightarrow \epsilon$$

Linksfaktorisierung Anwenden auf Produktion der Form $A \rightarrow \alpha\beta|\alpha\gamma$. (Verursachen Probleme wenn α länger als β).
Transformation: Verzögern der Entscheidung durch Linksfaktorisierung durch:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta | \gamma$$

3.5 Vorlesung 9

Komplexität von LL(1) Parsing

- LL(1) parsing hat Komplexität $O(|w|)$

Sei $n \leq |w| \cdot |N| + 1$, $|w|$ Wortlänge, $|N|$ Anzahl an Produktionen. Erklärung: Pro Zeichen werden aufgrund fehlender Linksrekursion maximal alle Produktionen einmal angewendet.

Error Handling

Error configuration:

- $(aw, A\alpha, z)$ Terminal a wird gelesen, existiert aber nicht in der la -Menge des Nichtterminals A
- $(\epsilon, A\alpha, z)$ Eingabe leer (ϵ), aber noch Nichtterminale auf dem Stack vorhanden und $\epsilon \notin la(A)$
- $(aw, b\alpha, z)$ Terminale können nicht gepopt werden.
- $(\epsilon, b\alpha, z)$ ϵ und Terminale können nicht gepopt werden
- (aw, ϵ, z) Wort nicht leer, Automat am Ende

Problem: Schwer zu implementieren.

Eigenschaften von gutem Error-handling:

- Unabhängig von Fehlern läuft der Parser weiter
- Möglichst viele Fehler sollen korrekt erkannt werden
- Keine Folgefehler
- Komplexität nimmt nicht zu

Panic Mode: Bei Fehlern werden weitere Eingabezeichen solange ignoriert, bis ein korrektes Zeichen gelesen wird. (Z.B. das Ende einer Anweisung, eines Blocks etc...)

3.6 Vorlesung 10

Bottom-Up Parsing und $LR(k)$ -Grammatiken

Top-Down-Parsing: Linksableitung

- Bottom-Up: Umgekehrte Rechtsableitung
- Durchlaufe Wort von Links nach Rechts
- shift: shifte Eingabesymbole auf den Pushdown
- reduce: Ersetze die rechte Seite einer Produktion durch seine links Seite (= Umgekehrte Expansionsschritte)
- Entfernen des Nichtdeterminismus durch *lookahead* von k Symbolen auf der Eingabe

Bottom-up Automat

- Konfigurationen: $\Sigma^* \times X^* \times [p]^*$
- Transitionen: $w \in \Sigma^*, \alpha \in X^*, z \in [p]^*$
- Shift: $(aw, \alpha, z) \vdash (w, \alpha a, z)$, if $a \in \Sigma$
- Reduce: $(w, \alpha\beta, z) \vdash (w, \alpha A, zi)$, if $\pi(i) = A \rightarrow \beta$
- Anfangskonfiguration: (w, ϵ, ϵ) für $w \in \Sigma^*$
- Endkonfigurationen: (ϵ, S, z)
- Lösung: z von rechts nach links gelesen

BEISPIEL 10.3: BOTTOM-UP PARSING VON $(a) \times b$:

G_{AE} :	$E \rightarrow E + T T$	(1,2)	$($	$(a) \times b$	$,$	ϵ	$,$	ϵ	$)$
	$T \rightarrow T \times F F$	(3,4)	\vdash	$($	$a) \times b$	$,$	$($	$,$	ϵ
	$F \rightarrow (E) a b$	(5,6,7)	\vdash	$($	$) \times b$	$,$	$(a$	$,$	ϵ
			\vdash	$($	$) \times b$	$,$	$(F$	$,$	6
			\vdash	$($	$) \times b$	$,$	$(T$	$,$	64
			\vdash	$($	$) \times b$	$,$	$(E$	$,$	642
			\vdash	$($	$\times b$	$,$	(E)	$,$	642
			\vdash	$($	$\times b$	$,$	F	$,$	6425
			\vdash	$($	$\times b$	$,$	T	$,$	64254
			\vdash	$($	b	$,$	$T \times$	$,$	64254
			\vdash	$($	ϵ	$,$	$T \times b$	$,$	64254
			\vdash	$($	ϵ	$,$	$T \times F$	$,$	642547
			\vdash	$($	ϵ	$,$	T	$,$	6425473
			\vdash	$($	ϵ	$,$	E	$,$	64254732

LR(k)

DEFINITION 10.5: START-SEPERATED CFG

Eine Grammatik G wird start-seperated genannt, wenn S nur in Produktionen der Form $S \rightarrow A$ ($S \neq A$) vorkommt.

Ziel: Entfernen des restlichen Nichtdeterminismus von $NBA(G)$ durch einen lookahead von $k \in \mathbb{N}$ Symbolen auf der Eingabe.

DEFINITION 10.7: LR(k) GRAMMATIK

Sei $G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ start-seperated und $k \in \mathbb{N}$. Dann ist $G \in LR(k)$ wenn für alle Rechtsableitungen der Form

$$S \begin{cases} \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ \Rightarrow_r^* \alpha' A' w' \Rightarrow_r \alpha \beta v \end{cases}$$

so dass aus $first_k(w) = first_k(v)$ folgt und dass $\alpha = \alpha'$, $A = A'$ und $w = v$

Anmerkungen:

- Wenn $G \in LR(k)$, so ist die Reduktion von $\alpha\beta w$ zu $\alpha A w$ immer durch $\alpha\beta first_k(w)$ bestimmt.
- Damit kann ein $NBA(G)$ in $(w, \alpha\beta, z)$ entscheiden, ob shift oder reduce angewendet werden soll und im zweiten Fall, wie reduce angewendet werden soll.

KOROLLAR 10.8: LR(0) GRAMMATIK

Sei $G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$. $G \in LR(0)$, wenn für alle Rechtsableitungen der Form

$$S \begin{cases} \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta w \\ \Rightarrow_r^* \alpha' A' w' \Rightarrow_r \alpha \beta v \end{cases}$$

folgt dass $\alpha = \alpha'$, $A = A'$ und $w = v$

Ziel ist das Erzeugen einer Information mit der der Nichtdeterminismus gelöst werden kann.

DEFINITION 10.9: LR(0) ITEMS UND SETS

$G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ sei start-separated durch $S' \rightarrow S$ und $S' \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha \beta_1 \beta_2 w$ ($A \rightarrow \beta_1 \beta_2 \in P$).

- $[A \rightarrow \beta_1 \beta_2]$ wird LR(0) item für $\alpha \beta_1$ genannt.
- Für $\gamma \in X^*$ bezeichnet $LR(0)(\gamma)$ die Menge aller LR(0) items für γ , genannt *LR(0) set/information* von γ
- $LR(0)(G) := \{LR(0)(\gamma) | \gamma \in X^*\}$

Korollar 10.10

- Für jedes $\gamma \in X^*$ ist $LR(0)(\gamma)$ endlich
- $LR(0)(G)$ ist endlich
- Das item $[A \rightarrow \beta \cdot] \in LR(0)(\gamma)$ kennzeichnet die mögliche Reduktion $(w, \alpha \beta, z) \vdash (w, \alpha A, zi)$ mit $\pi(i) = A \rightarrow \beta$ und $\gamma = \alpha \beta$
- Das item $[A \rightarrow \beta_1 \cdot \gamma \beta_2] \in LR(0)(\gamma)$ kennzeichnet einen möglichen shift

DEFINITION 10.11: LR(0)-KONFLIKTE

Sei $G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ und $I \in LR(0)(G)$.

- I hat einen shift/reduce-Konflikt, wenn Produktionen der Form $A \rightarrow \alpha_1 \alpha_2$ und $B \rightarrow \beta$ existieren mit $[A \rightarrow \alpha_1 \cdot \alpha_2], [B \rightarrow \beta \cdot] \in I$
- I hat einen reduce/reduce-Konflikt, wenn Produktionen der Form $A \rightarrow \alpha$ und $B \rightarrow \beta$ existieren mit $A \neq B$ oder $\alpha \neq \beta$ so dass $[A \rightarrow \alpha \cdot], [B \rightarrow \beta \cdot] \in I$

Lemma 10.12 $G \in LR(0) \Leftrightarrow$ kein $I \in LR(0)(G)$ enthält Items mit Konflikten

Theorem 10.13: Berechnung von LR(0)-sets

$G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ sei start-separated durch $S' \rightarrow S$ und reduziert

- $LR(0)(\epsilon)$ ist das letzte set so dass
 - $[S' \rightarrow \cdot S] \in LR(0)(\epsilon)$ und
 - wenn $[A \rightarrow \cdot B\gamma] \in LR(0)(\epsilon)$ und $B \rightarrow \beta \in P$, dann ist auch $[B \rightarrow \cdot \beta] \in LR(0)(\epsilon)$
- $LR(0)(\alpha Y)$ ($\alpha \in X^*, Y \in X$) ist das letzte set so dass
 - wenn $[A \rightarrow \gamma_1 \cdot Y\gamma_2] \in LR(0)(\alpha)$, dann ist $[A \rightarrow \gamma_1 Y \cdot \gamma_2] \in LR(0)(\alpha Y)$ und
 - wenn $[A \rightarrow \gamma_1 \cdot B\gamma_2] \in LR(0)(\alpha Y)$ und $B \rightarrow \beta \in P$, dann ist auch $[B \rightarrow \cdot \beta] \in LR(0)(\alpha Y)$

BEISPIEL 10.14

<p>G: $S' \rightarrow S$ $S \rightarrow B \mid C$ $B \rightarrow aB \mid b$ $C \rightarrow aC \mid c$</p>	<p>$I_0 := LR(0)(\epsilon):$ $[S' \rightarrow \cdot S] [S \rightarrow \cdot B] [S \rightarrow \cdot C] [S \rightarrow \cdot aB]$ $[B \rightarrow \cdot b] [C \rightarrow \cdot aC] [C \rightarrow \cdot c]$</p> <p>$I_1 := LR(0)(S):$ $[S' \rightarrow S \cdot]$</p> <p>$I_2 := LR(0)(B):$ $[S \rightarrow B \cdot]$</p> <p>$I_3 := LR(0)(C):$ $[S \rightarrow C \cdot]$</p> <p>$I_4 := LR(0)(a):$ $[B \rightarrow a \cdot B] [C \rightarrow a \cdot C] [B \rightarrow \cdot aB]$ $[B \rightarrow \cdot b] [C \rightarrow \cdot aC] [C \rightarrow \cdot c]$</p> <p>$I_5 := LR(0)(b):$ $[B \rightarrow b \cdot]$</p> <p>$I_6 := LR(0)(c):$ $[C \rightarrow c \cdot]$</p> <p>$I_7 := LR(0)(aB):$ $[B \rightarrow aB \cdot]$</p> <p>$I_8 := LR(0)(aC):$ $[B \rightarrow aC \cdot]$</p> <p>$LR(0)(aa) = LR(0)(a) = I_4$ $LR(0)(ab) = LR(0)(b) = I_5$ $LR(0)(ac) = LR(0)(c) = I_6$ $I_9 = LR(0)(\gamma) = \emptyset \forall$ übrigen Fälle</p>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Keine Konflikte $\Rightarrow G \in LR(0)$

3.7 Vorlesung 11

LR(0)-Parsing

GOTO FUNCTION

Die goto-Funktion $LR(0)(G) \times X \rightarrow LR(0)(G)$ ist definiert durch
 $goto(I, Y) = I' \Leftrightarrow \exists \gamma \in X^*$, so dass $I = LR(0)(\gamma)$ und $I' = LR(0)(\gamma Y)$

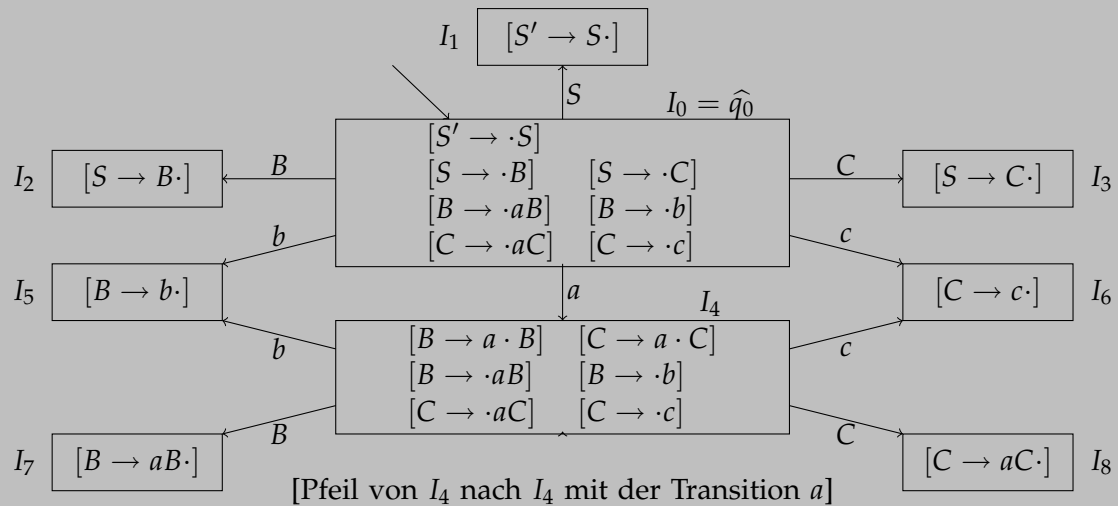
BEISPIEL 11.2 (SIEHE 10.14)

goto	S	B	C	a	b	c
I_0	I_1	I_2	I_3	I_4	I_5	I_6
I_1						
I_2						
I_3						
I_4		I_7	I_8	I_4	I_5	I_6
I_5						
I_6						
I_7						
I_8						
I_9						

Ziel: Erzeugen von $LR(0)(G)$ und goto-Funktion durch Potenzmengenkonstruktion, G start – seperated durch $S' \rightarrow S$ und reduziert. Konstruiere NFA $\mathfrak{A}(G)$

- Zustandsmenge $Q := \{[A \rightarrow \beta_1 \cdot \beta_2] \mid A \rightarrow \beta_1 \beta_2 \in P\}$ ($LR(0)$ -items von G)
- Eingabealphabet: $X := N \cup \Sigma$
- Transitionen: $\delta : Q \times X_\epsilon \rightarrow \mathfrak{P}(Q)$
 - $\delta([A \rightarrow \beta_1 \cdot Y\beta_2], Y) \ni [A \rightarrow \beta_1 Y \cdot \beta_2]$
 - $\delta([A \rightarrow \beta_1 \cdot B\beta_2], \epsilon) \ni [B \rightarrow \cdot \beta]$ wenn $B \rightarrow \beta \in P$
- Start: $q_0 := [S' \rightarrow \cdot S] \in Q$. Ende: $F := \emptyset$

BEISPIEL 11.5



Die LR(0) Action Funktion

Legt die shift-reduce Entscheidung fest und entfernt den Nichtdeterminismus.

DEFINITION 11.6

$$act(I) : LR(0)(G) \rightarrow \{red\ i | i \in [p]\} \cup \{shift, accept, error\}$$

$$act(I) := \begin{cases} red\ i & \text{if } \pi(i) = A \rightarrow \alpha \text{ und } [A \rightarrow \alpha \cdot] \in I (i \neq 0) \\ shift & \text{if } [A \rightarrow \alpha_1 \cdot \alpha_2] \in I \\ accept & \text{if } [S' \rightarrow S \cdot] \in I \\ error & \text{if } I = \emptyset \end{cases}$$

Korollar 11.7: $G \in LR(0) \Leftrightarrow act$ ist wohldefiniert

act und $goto$ bilden die LR(0)-Parsingtabelle von G

BEISPIEL 11.8: LR(0) PARSINGTABELLE (SIEHE 11.2)

goto	act	S	B	C	a	b	c
I_0	shift	I_1	I_2	I_3	I_4	I_5	I_6
I_1	accept						
I_2	red1						
I_3	red2						
I_4	shift		I_7	I_8	I_4	I_5	I_6
I_5	red4						
I_6	red6						
I_7	red3						
I_8	red5						
I_9	error						

Deterministischer LR(0) Parsing Automat

- Startkonfiguration: (w, I_0, ϵ) mit $I_0 := LR(0)(\epsilon)$

- Endkonfigurationen: $\{\epsilon\} \times \{\epsilon\} \times \Delta^*$

shift: $(aw, \alpha I, z) \vdash (w, \alpha I I', z)$ wenn $act(I) = shift$ und $goto(I, a) = I'$

reduce: $(w, \alpha I I_1 \dots I_n, z) \vdash (w, \alpha I I', z_i)$ wenn $act(I_n) = red_i$ $\pi(i) = A \rightarrow Y_1 \dots Y_n$ und $goto(I, A) = I'$

- Transitionen:

accept: $(\epsilon, I_0 I, z) \vdash (\epsilon, \epsilon, z_0)$ wenn $act(I) = accept$

error: $(w, \alpha I, z) \vdash (\epsilon, \epsilon, z_{error})$ wenn $act(I) = error$

BEISPIEL 11.10: LR(0) PARSING VON aac , SIEHE 3.7

\vdash	(aac, I_0, ϵ)
\vdash	$(ac, I_0 I_4, \epsilon)$
\vdash	$(c, I_0 I_4 I_4, \epsilon)$
\vdash	$(\epsilon, I_0 I_4 I_4 I_6, \epsilon)$
\vdash	$(\epsilon, I_0 I_4 I_4 I_8, \epsilon)$
\vdash	$(\epsilon, I_0 I_4 I_8, \epsilon)$
\vdash	$(\epsilon, I_0 I_3, \epsilon)$
\vdash	$(\epsilon, I_0 I_1, \epsilon)$
\vdash	$(\epsilon, \epsilon, \epsilon)$

Theorem 11.11 Korrektheit des LR(0) Parsing Automaten Wenn $G \in LR(0)$, dann ist der LR(0) Parsing Automat deterministisch und für jedes $w \in \Sigma^*$ und $z \in \{0, \dots, p\}^*$ gilt

$(w, I_0, \epsilon) \vdash (\epsilon, \epsilon, z) \Leftrightarrow \overleftarrow{z}$ ist eine Rechtsanalyse von w

3.8 Vorlesung 12

SLR(1) Parsing I

BEISPIEL 12.1

$$G_{AE} \quad E' \rightarrow E \quad E \rightarrow E + T | T$$

$$T \rightarrow T * F | F \quad F \rightarrow (E) | a | b$$

LR(0)(G_{AE}) mit Konflikten:

I_0 :	$[E' \rightarrow \cdot E]$	$[E \rightarrow \cdot E + T]$	I_1 :	$[E' \rightarrow E \cdot]$	$[E \rightarrow E \cdot + T]$
	$[E \rightarrow \cdot T]$	$[T \rightarrow \cdot T * F]$	I_2 :	$[E \rightarrow T \cdot]$	$[T \rightarrow T \cdot * F]$
	$[T \rightarrow \cdot F]$	$[F \rightarrow \cdot (E)]$	I_3 :	$[T \rightarrow F \cdot]$	
	$[F \rightarrow \cdot a]$	$[F \rightarrow \cdot b]$			
I_4 :	$[F \rightarrow (\cdot E)]$	$[E \rightarrow \cdot E + T]$	I_5 :	$[F \rightarrow a \cdot]$	
	$[E \rightarrow \cdot T]$	$[E \rightarrow \cdot T * F]$	I_6 :	$[F \rightarrow b \cdot]$	
	$[T \rightarrow \cdot F]$	$[F \rightarrow \cdot (E)]$	I_7 :	$[E \rightarrow E + \cdot T]$	$[T \rightarrow \cdot T * F]$
	$[F \rightarrow \cdot a]$	$[F \rightarrow \cdot b]$		$[T \rightarrow \cdot F]$	$[F \rightarrow \cdot (E)]$
				$[F \rightarrow \cdot a]$	$[F \rightarrow \cdot b]$
I_8 :	$[T \rightarrow T * \cdot F]$	$[F \rightarrow \cdot (E)]$	I_9 :	$[F \rightarrow (E \cdot)]$	$[E \rightarrow E \cdot + T]$
	$[F \rightarrow \cdot a]$	$[F \rightarrow \cdot b]$	I_{10} :	$[E \rightarrow E + T \cdot]$	$[T \rightarrow T \cdot * F]$
I_{11} :	$[T \rightarrow T * F \cdot]$		I_{12} :	$[F \rightarrow (E) \cdot]$	

Ziel: Entfernen von Konflikten mittels des ersten Eingabesymbol (la)

Beobachtung:

- $[A \rightarrow \beta_1 \cdot a \beta_2] \in LR(0)(\alpha \beta_1)$: shift nur bei lookahead a
- $[A \rightarrow \beta \cdot] \in LR(0)(\alpha \beta) \Rightarrow S' \Rightarrow_r^* \alpha A x w \Rightarrow_r \alpha \beta x w$: reduce mit $A \rightarrow \beta$ nur wenn lookahead $x \in fo(A)$

BEISPIEL 12.2 (SIEHE 12.1)

$A \in N$	$fo(A)$
E'	$\{\epsilon\}$
E	$\{+,), \epsilon\}$

- $I_1 = \{[E' \rightarrow E \cdot], [T \rightarrow E \cdot + T]\}$:
 - accept on lookahead ϵ
 - shift on lookahead $+$
- $I_2 = \{[E \rightarrow T \cdot], [T \rightarrow T \cdot * F]\}$:
 - red_2 on lookahead $+,), \epsilon$
 - shift on lookahead $*$
- $I_{10} = \{[E \rightarrow E + T \cdot], [T \rightarrow T \cdot * F]\}$:
 - red_1 on lookahead $+,), \epsilon$
 - shift on lookahead $*$

SLR(1) Parsing II

DEFINITION 12.3

Die SLR(1) action Funktion

$$\text{act}: LR(0)(G) \times \Sigma_\epsilon \rightarrow \{\text{red}_i | i \in [p]\} \cup \{\text{shift}, \text{accept}, \text{error}\}$$

ist definiert durch

$$\text{act}(I, x) := \begin{cases} \text{red}_i & \text{if } \pi(i) = A \rightarrow \alpha, [A \rightarrow \alpha \cdot] \in I (i \neq 0) \text{ und } x \in fo(A) \\ \text{shift} & \text{if } [A \rightarrow \alpha_1 \cdot x \alpha_2] \in I \text{ und } x \in \Sigma \\ \text{accept} & \text{if } [S' \rightarrow S \cdot] \in I \text{ und } x = \epsilon \\ \text{error} & \text{sonst} \end{cases}$$

DEFINITION 12.4: SLR(1) GRAMMATIK

$G \in SLR(1) \Leftrightarrow$ die SLR(1)-action-Funktion von G wohldefiniert ist. ($G \in CFG_\Sigma$)

SLR(1)-act-Funktion und LR(0)-goto-Funktionen formen zusammen SLR(1)-Parsing-Tabelle

BEISPIEL 12.5 (SIEHE 12.1)

$A \in N$	$fo(A)$
E'	$\{\epsilon\}$
E	$\{+,), \epsilon\}$
T	$\{+, *,), \epsilon\}$
F	$\{+, *,), \epsilon\}$

	act						goto									
	+	*	()	a	b	ϵ	E	T	F	+	*	()	a	b
I_0			shift		shift	shift		I_1	I_2	I_3				I_4	I_5	I_6
I_1	shift						accept				I_7					
I_2	red ₂	shift			red ₂		red ₂							I_8		
I_3	red ₄	red ₄			red ₄		red ₄									
I_4			shift		shift	shift		I_9	I_2	I_3				I_4	I_5	I_6
I_5	red ₆	red ₆			red ₆		red ₆									
I_6	red ₇	red ₇			red ₇		red ₇									
I_7			shift		shift	shift			I_{10}	I_3				I_4	I_5	I_6
I_8			shift		shift	shift				I_{11}				I_4	I_5	I_6
I_9	shift				shift						I_7				I_{12}	
I_{10}	red ₁	shift			red ₁		red ₁							I_8		
I_{11}	red ₃	red ₃			red ₃		red ₃									
I_{12}	red ₅	red ₅			red ₅		red ₅									

SLR(1) Parsing Automat

Wie bei LR(0) (Def. 11.9) mit folgenden Transitionen:

- shift $(aw, \alpha I, z) \vdash (w, \alpha I', z)$ if $\text{act}(I, a) = \text{shift}$ und $\text{goto}(I, a) = I'$
- reduce_a $(aw, \alpha I I_1 \dots I_n, z) \vdash (aw, \alpha I', zi)$ if $\text{act}(I_n, a) = \text{red}_i$, $\pi(i) = A \rightarrow Y_1 \dots Y_n$ und $\text{goto}(I, A) = I'$
- reduce _{ϵ} $(\epsilon, \alpha I I_1 \dots I_n, z) \vdash (\epsilon, \alpha I', zi)$ if $\text{act}(I_n, \epsilon) = \text{red}_i$, $\pi(i) = A \rightarrow Y_1 \dots Y_n$ und $\text{goto}(I, A) = I'$
- accept $(\epsilon, I_0 I, z) \vdash (\epsilon, \epsilon, z0)$ if $\text{act}(I, \epsilon) = \text{accept}$
- error_a $(aw, \alpha I, z) \vdash (\epsilon, \epsilon, zerror)$ if $\text{act}(I, a) = \text{error}$
- error _{ϵ} $(\epsilon, \alpha I, z) \vdash (\epsilon, \epsilon, zerror)$ if $\text{act}(I, \epsilon) = \text{error}$

LR(1) Parsing

Problem: Nicht alle Konflikte können mittels fo-Mengen beseitigt werden.

BEISPIEL 12.6

$G_{LR} : S' \rightarrow S$

$S \rightarrow L = R \quad |R$

$L \rightarrow *R \quad |a$

$R \rightarrow L$

$I_0 := LR(0)(\epsilon) : [S' \rightarrow \cdot S] [S \rightarrow \cdot L = R] [S \rightarrow \cdot R] [L \rightarrow \cdot * R] [L \rightarrow \cdot a] [R \rightarrow \cdot L]$

$I_1 := LR(0)(S) : [S' \rightarrow S \cdot]$

$I_2 := LR(0)(L) : [S \rightarrow L \cdot = R] [R \rightarrow L \cdot]$

$I_3 := LR(0)(R) : [S \rightarrow R \cdot]$

$LR(0)(G_{LR}) : I_4 := LR(0)(*) : [L \rightarrow * \cdot R] [R \rightarrow \cdot L] [L \rightarrow \cdot * R] [L \rightarrow \cdot a]$

$I_5 := LR(0)(a) : [L \rightarrow a \cdot]$

$I_6 := LR(0)(L =) : [S \rightarrow L = \cdot R] [R \rightarrow \cdot L] [L \rightarrow \cdot * R] [L \rightarrow \cdot a]$

$I_7 := LR(0)(*R) : [L \rightarrow *R \cdot]$

$I_8 := LR(0)(*L) : [R \rightarrow L \cdot]$

$I_9 := LR(0)(L = R) : [S \rightarrow L = R \cdot]$

Die Konflikte in I_2 sind nicht SLR(1)-lösbar weil $= \in fo(R)$

Beobachtung: Nicht jedes Element von $fo(A)$ darf einem Vorkommen von A folgen, da es sonst zu Konflikten kommen kann. (shift/reduce)

\Rightarrow Anpassen der LR(0)-Items durch Hinzufügen von möglichem Lookahead

DEFINITION 12.7: LR(1) ITEMS UND SETS

- $\alpha\beta_1\beta_2aw \Rightarrow [A \rightarrow \beta_1 \cdot \beta_2, a] \in LR(1)(\alpha\beta_1)$
- $\alpha\beta_1\beta_2 \Rightarrow [A \rightarrow \beta_1 \cdot \beta_2, \epsilon] \in LR(1)(\alpha\beta_1)$

KOROLLAR 12.8

- $LR(1)(\gamma)$ beinhaltet $LR(0)(\gamma)$
- $[A \rightarrow \beta_1 \cdot \beta_2, x] \in LR(1)(G) \Rightarrow x \in fo(A)$

DEFINITION 12.9: LR(1)-KONFLIKTE

- shift/reduce-Konflikt: $[A \rightarrow \alpha_1 \cdot a\alpha_2, x], [B \rightarrow \beta \cdot, a] \in I$: Lookahead von reduce ist gleich dem zu shiftenden Symbol a .
- reduce/reduce-Konflikt: $[A \rightarrow \alpha \cdot, x], [B \rightarrow \beta \cdot, x] \in I$: Gleicher Lookahead x

Lemma: $G \in LR(1) \Leftrightarrow$ Kein $I \in LR(1)(G)$ hat Items mit Konflikten

Berechnen von LR(1)-Mengen

Erweitern der Berechnung der LR(0)-Mengen (Siehe 10.13) um rechte Kontexte beizubehalten.

THEOREM 12.11

Sei $G = \langle N, \Sigma, P, S \rangle \in CFG_{\Sigma}$ start separated durch $S' \rightarrow S$ und reduziert.

- $LR(1)(\epsilon)$ ist das letzte Set mit
 - $[S' \rightarrow \cdot S, \epsilon] \in LR(1)(\epsilon)$
 - wenn $[A \rightarrow \cdot B\gamma, x] \in LR(1)(\epsilon)$, $B \rightarrow \beta \in P$ und $y \in fi(\gamma x) \Rightarrow [B \rightarrow \cdot \beta, y] \in LR(1)(\epsilon)$
- $LR(1)(\alpha Y)$ mit $\alpha \in X^*$, $Y \in X$ ist das letzte Set mit
 - wenn $[A \rightarrow \gamma_1 \cdot Y\gamma_2, x] \in LR(1)(\alpha Y) \Rightarrow [A \rightarrow \gamma_1 Y \cdot \gamma_2, x] \in LR(1)(\alpha Y)$
 - wenn $[A \rightarrow \gamma_1 \cdot B\gamma_2, x] \in LR(1)(\alpha Y)$, $B \rightarrow \beta \in P$ und $y \in fi(\gamma_2 x) \Rightarrow [B \rightarrow \cdot \beta, y] \in LR(1)(\alpha Y)$

BEISPIEL 12.12

G_{LR} : $S' \rightarrow S$
 $S \rightarrow L = R|R$
 $L \rightarrow *R|a$
 $R \rightarrow L$

$LR(1)(G_{LR}) : [S' \rightarrow \cdot S, \epsilon] \in LR(1)(\epsilon)$, $[A \rightarrow \cdot B\gamma, x] \in LR(1)(\epsilon)$, $B \rightarrow \beta \in P$, $y \in fi(\gamma x)$
 $\Rightarrow [B \rightarrow \cdot \beta, y] \in LR(1)(\epsilon)$

$I'_0 := LR(1)(\epsilon)$ $[S' \rightarrow \cdot S, \epsilon][S \rightarrow \cdot L = R, \epsilon][S \rightarrow \cdot R, \epsilon][L \rightarrow \cdot *R, \epsilon][L \rightarrow \cdot a, \epsilon][R \rightarrow \cdot L, \epsilon]$
 $I'_1 := LR(1)(S)$ $[S' \rightarrow S \cdot, \epsilon]$
 $I'_2 := LR(1)(L)$ $[S \rightarrow L \cdot = R, \epsilon][R \rightarrow L \cdot, \epsilon]$
 $I'_3 := LR(1)(R)$ $[S \rightarrow R \cdot, \epsilon]$
 $I'_4 := LR(1)(*)$ $[L \rightarrow * \cdot R, \epsilon][R \rightarrow \cdot L, \epsilon][L \rightarrow \cdot *R, \epsilon][L \rightarrow \cdot a, \epsilon][L \rightarrow * \cdot R, \epsilon][R \rightarrow \cdot L, \epsilon]$
 $I'_5 := LR(1)(a)$ $[L \rightarrow a \cdot, \epsilon][L \rightarrow a \cdot, \epsilon]$
 $I'_6 := LR(1)(L =)$ $[S \rightarrow L = \cdot R, \epsilon][R \rightarrow \cdot L, \epsilon][L \rightarrow \cdot *R, \epsilon][L \rightarrow \cdot a, \epsilon]$
 $I'_7 := LR(1)(*R)$ $[L \rightarrow *R \cdot, \epsilon][L \rightarrow *R \cdot, \epsilon]$
 $I'_8 := LR(1)(*L)$ $[R \rightarrow L \cdot, \epsilon][R \rightarrow L \cdot, \epsilon]$
 $I'_9 := LR(1)(L = R)$ $[S \rightarrow L = R \cdot, \epsilon]$
 $I'_{10} := LR(1)(L = L)$ $[R \rightarrow L \cdot, \epsilon]$
 $I'_{11} := LR(1)(L = *)$ $[L \rightarrow * \cdot R, \epsilon][R \rightarrow \cdot L, \epsilon][L \rightarrow \cdot *R, \epsilon][L \rightarrow \cdot a, \epsilon]$
 $I'_{12} := LR(1)(L = a)$ $[L \rightarrow a \cdot, \epsilon]$
 $I'_{13} := LR(1)(L = *R)$ $[L \rightarrow *R \cdot, \epsilon]$
 $I'_{14} := \emptyset$

In I'_2 shiften bei = bzw. reduzieren bei $\epsilon \Rightarrow G_{LR} \in LR(1)$

DEFINITION 12.13: LR(1) ACTION FUNKTION

Die LR(1) action Funktion

$$\text{act}: LR(1)(G) \times \Sigma_{\epsilon} \rightarrow \{\text{redi} | i \in [p]\} \cup \{\text{shift}, \text{accept}, \text{error}\}$$

ist definiert durch

$$\text{act}(I, x) := \begin{cases} \text{redi} & \text{if } \pi(i) = A \rightarrow \alpha \text{ und } [A \rightarrow \alpha \cdot, x] \in I \\ \text{shift} & \text{if } [A \rightarrow \alpha_1 \cdot x\alpha_2, y] \in I \text{ und } x \in X \\ \text{accept} & \text{if } [S' \rightarrow S \cdot, \epsilon] \in I \text{ und } x = \epsilon \\ \text{error} & \text{sonst} \end{cases}$$

Für alle $G \in CFG_{\Sigma}$ gilt: $G \in LR(1) \Leftrightarrow$ die action Funktion wohldefiniert ist.

DEFINITION 12.15: LR(1)-GOTO-FUNKTION

Die Funktion goto: $LR(1)(G) \times X \rightarrow LR(1)(G)$ ist bestimmt durch

$$\text{goto}(I, Y) = I' \Leftrightarrow \text{es existiert ein } \gamma \in Y^* \text{ mit } I = LR(1)(\gamma) \text{ und } I' = LR(1)(\gamma Y)$$

Die Konstruktion erfolgt analog zu $LR(0)$ durch Potenzmengenkonstruktion. act und goto ergeben zusammen die $LR(1)$ -parsing Tabelle von G .

Der $LR(1)$ -parsing Automat ist definiert wie bei $SLR(1)$ mit den gleichen Transitionen (siehe 12.12).

BEISPIEL 12.16

$LR(1)(G_{LR})$	$act/goto _{\Sigma_\epsilon}$			S	L	R
	*	=	a			
I'_0	shift/ I'_4		shift/ I'_5	I'_1	I'_2	I'_3
I'_1						
I'_2		shift/ I'_5				
I'_3						
I'_4	shift/ I'_4		shift/ I'_5		I'_8	I'_7
I'_5		red ₄				
I'_6	shift/ I'_1		shift/ I'_{12}		I'_{10}	I'_9
I'_7		red ₃				
I'_8		red ₅				
I'_9						
I'_{10}						
I'_{11}	shift/ I'_{11}		shift/ I'_{12}		I'_{11}	I'_{13}
I'_{12}						
I'_{13}						

empty = error / \emptyset

LR(1)-parsing von $a = *a$:

(a = *a	, I'_0	, ϵ)
⊢	= *a	, $I'_0 I'_5$, ϵ)
⊢	= *a	, $I'_0 I'_2$, 4)
⊢	*a	, $I'_0 I'_2 I'_6$, 4)
⊢	a	, $I'_0 I'_2 I'_6 I'_{11}$, 4)
⊢	ϵ	, $I'_0 I'_2 I'_6 I'_{11} I'_{12}$, 4)
⊢	ϵ	, $I'_0 I'_2 I'_6 I'_{11} I'_{10}$, 44)
⊢	ϵ	, $I'_0 I'_2 I'_6 I'_{11} I'_{13}$, 445)
⊢	ϵ	, $I'_0 I'_2 I'_6 I'_{10}$, 4453)
⊢	ϵ	, $I'_0 I'_2 I'_6 I'_9$, 44535)
⊢	ϵ	, $I'_0 I'_1$, 445351)
⊢	ϵ	, ϵ	, 4453510)

3.9 Vorlesung 13

LALR(1)-parsing

Beseitigen von Konflikten mittels LR(1) zu teuer. Beachte: Mögliche Redundanz durch Erhalt von LR(0) Mengen in LR(1) Mengen.

DEFINITION 13.1: LR(0)-ÄQUIVALENZ

Sei $Ir_0 : LR(1)(G) \rightarrow LR(0)(G)$ definiert durch

$$Ir_0(I) := \{[A \rightarrow \beta_1 \cdot \beta_2] \mid [A \rightarrow \beta_1 \cdot \beta_2, x] \in I\}.$$

Zwei Sets $I_1, I_2 \in LR(1)(G)$ werden LR(0)-äquivalent genannt ($I_1 \sim_0 I_2$) wenn $Ir_0(I_1) = Ir_0(I_2)$, sie also bis auf den Lookahead gleich sind:

$$\begin{aligned} I_1[A \rightarrow \beta_1 \cdot \beta_2, x] \\ I_2[A \rightarrow \beta_1 \cdot \beta_2, y] \\ \Rightarrow I_1 \sim_0 I_2 \end{aligned}$$

BEISPIEL 13.2 (SIEHE 12.6) LR(0) vs LR(1)

$G_{LR} : S' \rightarrow S \quad S \rightarrow L = R \mid R \quad L \rightarrow *R \mid a \quad R \rightarrow L$

$LR(0)(G_{LR}) :$

$$\begin{aligned} I_0 &:= LR(0)(\epsilon) : & [S' \rightarrow \cdot S] [S \rightarrow \cdot L = R] [S \rightarrow \cdot R] [L \rightarrow \cdot * R] [L \rightarrow \cdot a] [R \rightarrow \cdot L] \\ I_1 &:= LR(0)(S) : & [S' \rightarrow S \cdot] \\ I_2 &:= LR(0)(L) : & [S \rightarrow L \cdot = R] [R \rightarrow L \cdot] \\ I_3 &:= LR(0)(R) : & [S \rightarrow R \cdot] \\ I_4 &:= LR(0)(*) : & [L \rightarrow * \cdot R] [R \rightarrow \cdot L] [L \rightarrow \cdot * R] [L \rightarrow \cdot a] \\ I_5 &:= LR(0)(a) : & [L \rightarrow a \cdot] \\ I_6 &:= LR(0)(L =) : & [S \rightarrow L = \cdot R] [R \rightarrow \cdot L] [L \rightarrow \cdot * R] [L \rightarrow \cdot a] \\ I_7 &:= LR(0)(*R) : & [L \rightarrow *R \cdot] \\ I_8 &:= LR(0)(*L) : & [R \rightarrow L \cdot] \\ I_9 &:= LR(0)(L = R) : & [S \rightarrow L = R \cdot] \end{aligned}$$

$LR(1)(G_{LR}) :$

$$\begin{aligned} I'_0 &:= LR(1)(\epsilon) & [S' \rightarrow \cdot S, \epsilon] [S \rightarrow \cdot L = R, \epsilon] [S \rightarrow \cdot R, \epsilon] [L \rightarrow \cdot * R, =] [L \rightarrow \cdot a, =] [R \rightarrow \cdot L, \epsilon] \\ & & [L \rightarrow \cdot * R, \epsilon] [L \rightarrow \cdot a, \epsilon] \\ I'_1 &:= LR(1)(S) & [S' \rightarrow S \cdot, \epsilon] \\ I'_2 &:= LR(1)(L) & [S \rightarrow L \cdot = R, \epsilon] [R \rightarrow L \cdot, \epsilon] \\ I'_3 &:= LR(1)(R) & [S \rightarrow R \cdot, \epsilon] \\ I'_4 &:= LR(1)(*) & [L \rightarrow * \cdot R, =] [L \rightarrow * \cdot R, \epsilon] [R \rightarrow \cdot L, =] [L \rightarrow \cdot * R, =] [L \rightarrow \cdot a, =] [R \rightarrow \cdot L, \epsilon] \\ & & [L \rightarrow \cdot * R, \epsilon] [L \rightarrow \cdot a, \epsilon] \\ I'_5 &:= LR(1)(a) & [L \rightarrow a \cdot, =] [L \rightarrow a \cdot, \epsilon] \\ I'_6 &:= LR(1)(L =) & [S \rightarrow L = \cdot R, \epsilon] [R \rightarrow \cdot L, \epsilon] [L \rightarrow \cdot * R, \epsilon] [L \rightarrow \cdot a, \epsilon] \\ I'_7 &:= LR(1)(*R) & [L \rightarrow *R \cdot, =] [L \rightarrow *R \cdot, \epsilon] \\ I'_8 &:= LR(1)(*L) & [R \rightarrow L \cdot, =] [R \rightarrow L \cdot, \epsilon] \\ I'_9 &:= LR(1)(L = R) & [S \rightarrow L = R \cdot, \epsilon] \\ I'_{10} &:= LR(1)(L = L) & [R \rightarrow L \cdot, \epsilon] \\ I'_{11} &:= LR(1)(L = *) & [L \rightarrow * \cdot R, \epsilon] [R \rightarrow \cdot L, \epsilon] [L \rightarrow \cdot * R, \epsilon] [L \rightarrow \cdot a, \epsilon] \\ I'_{12} &:= LR(1)(L = a) & [L \rightarrow a \cdot, \epsilon] \\ I'_{13} &:= LR(1)(L = *R) & [L \rightarrow *R \cdot, \epsilon] \\ I'_{14} &:= \emptyset \end{aligned}$$

$$\Rightarrow \quad I'_4 \sim_0 I'_{11} \quad I'_5 \sim_0 I'_{12} \quad I'_7 \sim_0 I'_{13} \quad I'_8 \sim_0 I'_{10}$$

Die Anzahl der LR(1) Mengen ohne LR(0) äquivalenz entspricht der Anzahl der LR(0) Mengen.

LALR(1)

Idee: Vereinigen von LR(0)-äquivalenten LR(1) Mengen. Dabei werden die Lookaheadinformationen beibehalten, was allerdings zu möglichen Konflikten führt. ■

BEISPIEL 13.5 (SIEHE 13.2)

LR(0)(G_{LR}) wie in 13.2

$$\begin{aligned}
 I''_0 = I'_0 : & \quad [S' \rightarrow \cdot S, \epsilon][S \rightarrow \cdot L = R, \epsilon][S \rightarrow \cdot R, \epsilon][L \rightarrow \cdot * R, =][L \rightarrow \cdot a, =][R \rightarrow \cdot L, \epsilon] \\
 & \quad [L \rightarrow \cdot * R, \epsilon][L \rightarrow \cdot a, \epsilon] \\
 I''_1 = I'_1 : & \quad [S' \rightarrow S \cdot, \epsilon] \\
 I''_2 = I'_2 : & \quad [S \rightarrow L \cdot = R, \epsilon][R \rightarrow L \cdot, \epsilon] \\
 I''_3 = I'_3 : & \quad [S \rightarrow R \cdot, \epsilon] \\
 I''_4 = I'_4 \cup I''_{11} : & \quad [L \rightarrow * \cdot R, = / \epsilon][R \rightarrow \cdot L, = / \epsilon][L \rightarrow \cdot * R, = / \epsilon][L \rightarrow \cdot a, = / \epsilon] \\
 I''_5 = I'_5 \cup I''_{12} : & \quad [L \rightarrow a \cdot, = / \epsilon] \\
 I''_6 = I'_6 : & \quad [S \rightarrow L = \cdot R, \epsilon][R \rightarrow \cdot L, \epsilon][L \rightarrow \cdot * R, \epsilon][L \rightarrow \cdot a, \epsilon] \\
 I''_7 = I'_7 \cup I''_{13} : & \quad [L \rightarrow * R \cdot, = / \epsilon] \\
 I''_8 = I'_8 \cup I''_{10} : & \quad [R \rightarrow L \cdot, = / \epsilon] \\
 I''_9 = I'_9 : & \quad [S \rightarrow L = R \cdot, \epsilon]
 \end{aligned}$$

DEFINITION 13.6: LALR(1) ACTION FUNKTION

Die LALR(1) action Funktion

$$\text{act}: LALR(1)(G) \times \Sigma_\epsilon \rightarrow \{\text{red}_i | i \in [p]\} \cup \{\text{shift}, \text{accept}, \text{error}\}$$

ist definiert durch

$$\text{act}(I, x) := \begin{cases} \text{red}_i & \text{if } \pi(i) = A \rightarrow \alpha, [A \rightarrow \alpha \cdot, x] \in I \\ \text{shift} & \text{if } [A \rightarrow \alpha_1 \cdot x \alpha_2, y] \in I \text{ und } x \in \Sigma \\ \text{accept} & \text{if } [S' \rightarrow S \cdot, \epsilon] \in I \text{ und } x = \epsilon \\ \text{error} & \text{sonst} \end{cases}$$

Eine Grammatik $G \in CFG_\Sigma$ hat LALR(1)-Eigenschaft, wenn ihre LALR(1) action Funktion wohldefiniert ist. Zum Beispiel ist $G_{LR} \in LALR(1)$ (Beispiel 13.5)

Sei $G \in CFG_\Sigma$ und $I_1, I_2 \in LR(1)(G)$ mit $I_1 \sim_0 I_2$. Dann gilt für jedes $Y \in X$: $\text{goto}(I_1, Y) \sim_0 \text{goto}(I_2, Y)$

BEISPIEL 13.10 (SIEHE 13.5)

LALR(1)(G_{LR})	act/goto Σ_ϵ				goto N		
	*	=	a	ϵ	S	L	R
I''_0	shift/ I''_4		shift/ I''_5		I''_1	I''_2	I''_3
I''_1				accept			
I''_2		shift/ I''_6		red_5			
I''_3				red_2			
I''_4	shift/ I''_4		shift/ I''_5			I''_8	I''_7
I''_5		red_4		red_4			
I''_6	shift/ I''_4		shift/ I''_5			I''_8	I''_9
I''_7		red_3		red_3			
I''_8		red_5		red_5			
I''_9				red_1			

empty = error / \emptyset

Das Mergen von $LR(1)$ -Sets kann neue Konflikte hervorrufen:

BEISPIEL 13.11

$G : S' \rightarrow S$
 $S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
 $A \rightarrow c$
 $B \rightarrow c$

$LR(1)(G)$:

$LR(1)(\epsilon) \quad [S' \rightarrow \cdot S, \epsilon][S \rightarrow \cdot aAd, \epsilon][S \rightarrow \cdot bBe, \epsilon][S \rightarrow \cdot aBe, \epsilon][S \rightarrow \cdot bAe, \epsilon]$
 $LR(1)(S) \quad [S' \rightarrow S \cdot, \epsilon]$
 $LR(1)(a) \quad [S \rightarrow a \cdot Ad, \epsilon][S \rightarrow a \cdot Ad, \epsilon][A \rightarrow \cdot c, d][B \rightarrow \cdot c, e]$
 $LR(1)(b) \quad [S \rightarrow b \cdot Bd, \epsilon][S \rightarrow b \cdot Ae, \epsilon][B \rightarrow \cdot c, d][A \rightarrow \cdot c, e]$
 $LR(1)(aA) \quad [S \rightarrow aA \cdot d, \epsilon]$
 $LR(1)(aB) \quad [S \rightarrow aB \cdot e, \epsilon]$
 $LR(1)(ac) \quad [A \rightarrow c \cdot, d][B \rightarrow c \cdot, e]$
 $LR(1)(bB) \quad [S \rightarrow bB \cdot d, \epsilon]$
 $LR(1)(bA) \quad [S \rightarrow bA \cdot e, \epsilon]$
 $LR(1)(bc) \quad [B \rightarrow c \cdot, d][A \rightarrow c \cdot, e]$
 $LR(1)(aAd) \quad [S \rightarrow aAd \cdot, \epsilon]$
 $LR(1)(aBe) \quad [S \rightarrow aBe \cdot, \epsilon]$
 $LR(1)(bBd) \quad [S \rightarrow bBd \cdot, \epsilon]$
 $LR(1)(bAe) \quad [S \rightarrow bAe \cdot, \epsilon]$

- Keine Konflikte $\Rightarrow G \in LR(1)$
- $LR(1)(ac) \sim_0 LR(1)(bc)$, aber $LR(1)(ac) \cup LR(1)(bc)$ hat Konflikte $G \notin LALR(1)$

NAIVER ALGORITHMUS ZUM KONSTRUIEREN DES $LALR(1)$ -PARSER

1. Konstruiere $LR(1)(G)$
2. Vereinige $LR(0)$ -äquivalent zu $LR(1)$ -Mengen

Problem: Enormer Platzbedarf in ungünstigen Fällen. Es gibt aber 'bessere' Algorithmen.

Bottom-Up Parsing mehrdeutiger Grammatiken

LEMMA 13.12

Wenn $G \in CFG_{\Sigma}$ mehrdeutig ist $\Rightarrow G \notin \bigcup_{k \in \mathbb{N}} LR(k)$

BEISPIEL 13.13 (SIMPLE ARITHMETISCHE AUSDRÜCKE)

$G : E' \rightarrow E$
 $E \rightarrow E + E \mid E * E \mid a$

Wobei $*$ vor $+$ ausgerechnet wird und G linksassoziativ ist: $a + a * a + a = (a + (a * a)) + a$

$LR(0)(G) :$

$I_0 := LR(0)(\epsilon) : [E' \rightarrow \cdot E][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot a]$
 $I_1 := LR(0)(E) : [E' \rightarrow E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_2 := LR(0)(a) : [E \rightarrow a \cdot]$
 $I_3 := LR(0)(E+) : [E \rightarrow E + \cdot E][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot a]$
 $I_4 := LR(0)(E*) : [E \rightarrow E * \cdot E][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot a]$
 $I_5 := LR(0)(E + E) : [E \rightarrow E + E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_6 := LR(0)(E * E) : [E \rightarrow E * E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$

Konflikte:

- I_1 : $SLR(1)$ -lösbar (reduce bei ϵ , shift bei $+/*$)
- I_5, I_6 : nicht $SLR(1)$ -lösbar ($+, * \in fo(E)$)

Lösung:

- $I_5 : act(I_5, *) := shift, +$ linksassoziativ $\Rightarrow act(I_5, +) := red_1$
- $I_6 : act(I_6, +) := red_2, *$ linksassoziativ $\Rightarrow act(I_6, *) := red_2$

BEISPIEL 13.14: DANGLING else

$G : S' \rightarrow S$
 $S \rightarrow iSeS \mid iS \mid a$

Mehrdeutigkeit: $iaea := 1.) i(iaea)$ (standard) oder 2.) $i(ia)ea$

$LR(0)(G) :$

$I_0 := LR(0)(\epsilon) : [S' \rightarrow \cdot S][S \rightarrow \cdot iSeS][S \rightarrow \cdot iS]$
 $I_1 := LR(0)(S) : [S' \rightarrow S \cdot]$
 $I_2 := LR(0)(i) : [S \rightarrow i \cdot SeS][S \rightarrow i \cdot S][S \rightarrow \cdot iSeS][S \rightarrow \cdot iS][S \rightarrow \cdot a]$
 $I_3 := LR(0)(a) : [S \rightarrow a \cdot]$
 $I_4 := LR(0)(iS) : [S \rightarrow iS \cdot eS][S \rightarrow iS \cdot]$
 $I_5 := LR(0)(iSe) : [S \rightarrow iSe \cdot S][S \rightarrow \cdot iSeS][S \rightarrow \cdot iS][S \rightarrow \cdot a]$
 $I_6 := LR(0)(iSeS) : [S \rightarrow iSeS \cdot]$

Konflikte:

- $I_4 : e \in fo(S)$ nicht $SLR(1)$ -lösbar

Lösung:

- $act(I_4, e) := shift$

Error-Handling

Korrigiere die Präfix-Eigenschaft des LR -Parsings. Syntaktische Fehler werden an der ersten möglichen Stelle erkannt.

Panic Mode:

1. Durchsuche Pushdown nach Menge I mit definiertem $goto(I, A)$
2. Lösche Eingangssymbole, bis $x \in fo(A)$ auftritt
3. push goto (I, A)
4. weitermachen wie normal

Alternativ:

- Untersuche jeden Eintrag in der Parsingtable
- Entscheide, was der wahrscheinlichste Programmfehler ist
- Entwerfe Wiederherstellungsverfahren, das den Pushdown und die erste Eingabe verändert

Dabei ist zu beachten, dass keine Endlosschleifen entstehen.

BEISPIEL 13.15 (ERWEITERUNG VON 13.13)

$G: E' \rightarrow E(0)$
 $E \rightarrow E + E \mid E * E \mid (E) \mid a(1-4)$

Wobei $*$ vor $+$ ausgerechnet wird und G linksassoziativ ist: $a + a * a + a = (a + (a * a)) + a$

$LR(0)(G):$

$I_0(\epsilon): [E' \rightarrow \cdot E][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot (E)][E \rightarrow \cdot a]$
 $I_1(E): [E' \rightarrow E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_2((): [E \rightarrow (\cdot E)][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot (E)][E \rightarrow \cdot a]$
 $I_3(a): [E \rightarrow a \cdot]$
 $I_4(E+): [E \rightarrow E + \cdot E][E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot (E)][E \rightarrow \cdot a]$
 $I_5(E*): [E \rightarrow \cdot E + E][E \rightarrow \cdot E * E][E \rightarrow \cdot (E)][E \rightarrow \cdot a]$
 $I_6((E): [E \rightarrow (E \cdot)][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_7(E + E): [E \rightarrow E + E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_8(E * E): [E \rightarrow E * E \cdot][E \rightarrow E \cdot + E][E \rightarrow E \cdot * E]$
 $I_9((E)): [E \rightarrow (E) \cdot]$

$SLR(1)$ -Parsing von $a+$:

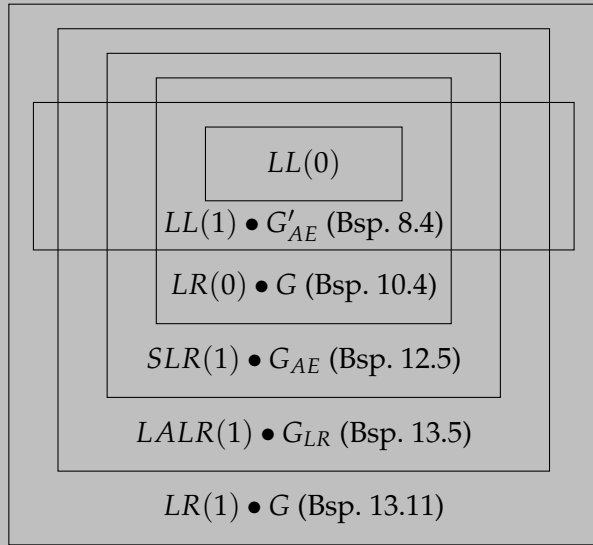
$(a+) , I_0 , \epsilon) \vdash (+) , I_0 I_3 , \epsilon)$
 $\vdash (+) , I_0 I_1 , 4) \vdash () , I_0 I_1 I_4 , 4)$
 $\vdash (\epsilon , I_0 I_1 I_4 , 4) \vdash (\epsilon , I_0 I_1 I_4 I_3 , 4)$
 $\vdash (\epsilon , I_0 I_1 I_4 I_7 , 44) \vdash (\epsilon , I_0 I_1 , 441)$
 $\vdash (\epsilon , \epsilon , 4410)$

Error Routinen:

- $e1: a$ oder $($ erwartet, aber $+, *, \epsilon$ gelesen
 - push $goto(I, a) = I_3$
 - Ausgabe: missing operand
- $e2: unerwartetes)$
 - $)$ von der Eingabe löschen
 - Ausgabe: unbalanced right paranthesis
- $e3: +, *$ oder ϵ erwartet, aber $($ oder a gefunden
 - push $goto(I, +) = I_4$
 - Ausgabe: missing operator
- $e4:)$ erwartet, aber ϵ gefunden
 - push $goto(I,)) = I_9$
 - Ausgabe: missing right paranthesis

	$act/goto _{\Sigma_\epsilon}$						$goto _N$
	a	$+$	$*$	$($	$)$	ϵ	E
I_0	shift/ I_3	$e1$	$e1$	shift/ I_2	$e2$	$e2$	I_1
I_1	$e3$	shift/ I_4	shift/ I_5	$e3$	$e2$	accept	
I_2	shift/ I_3	$e1$	$e1$	shift/ I_2	$e2$	$e1$	I_6
I_3	$e3$	red4	red4	$e3$	red4	red4	
I_4	shift/ I_3	$e1$	$e1$	shift/ I_2	$e2$	$e1$	I_7
I_5	shift/ I_3	$e1$	$e1$	shift/ I_2	$e2$	$e1$	I_8
I_6	$e3$	shift/ I_4	shift/ I_5	$e3$	shift/ I_9	$e4$	
I_7	$e3$	red1	shift/ I_5	$e3$	red1	red1	
I_8	$e3$	red2	red2	$e3$	red2	red2	
I_9	$e3$	red3	red3	$e3$	red3	red3	

ÜBERBLICK GRAMMATIKALISCHE KLASSEN

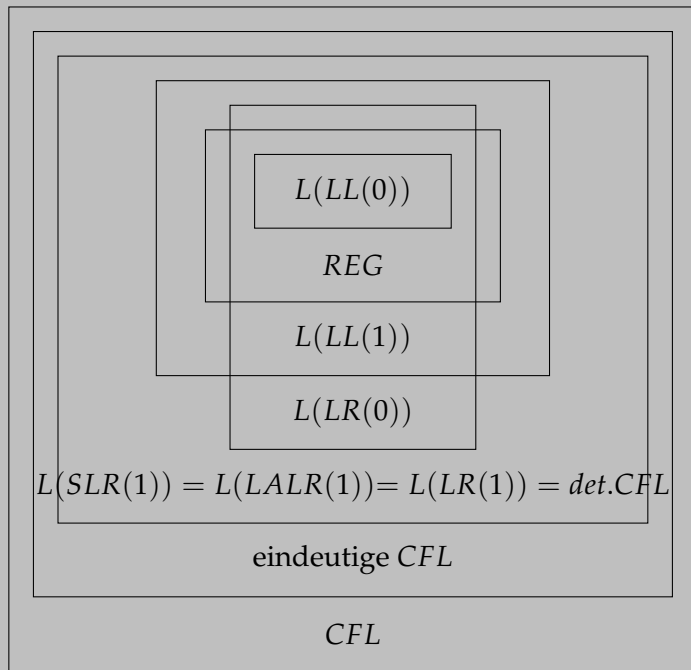


Insbesondere:

- $LL(k) \subsetneq LL(k+1)$
- $LR(k) \subsetneq LR(k+1)$
- $LL(k) \subsetneq LR(k)$

$\forall k \in \mathbb{N}$

ÜBERBLICK SPRACHKLASSEN



Insbesondere:

- $L(LL(k)) \subsetneq L(LL(k+1)) \subsetneq L(LR(k))$
- $L(LR(k)) \subsetneq L(LR(1))$

$\forall k \in \mathbb{N}$

AUSDRUCKSFÄHIGKEIT VON LL- UND LR-GRAMMATIKEN

Einfachheit	LL wins:	Einfaches Parsing, einfach zu debuggen
Allgemein	LALR wins:	Es gilt 'fast' $LL(1) \subseteq LALR(1)$. Nur konstruierte Gegenbeispiele, LL verlangt Entfernen von Linksrekursion und -faktorisierung.
Sem. Aktionen	LL wins:	siehe semantische Analyse. Actions können überall im Parser platziert werden, ohne Konflikte zu verursachen. In LALR gibt es explizite ϵ -Transitionen, welche Konflikte verursachen können.
Error Handling	LL wins:	Top-Down-Ansatz liefert Infos über den Kontext
Parser-Grösse	equal:	Beide linear in der Länge des Eingabeprogramms

⇒ Wenn möglich, wähle LL (Abhängig von der verfügbaren Grammatik und dem vorhandenen Programm)

4 Semantische Analyse

4.1 Kapitel 15

Beyond Syntax

Um effizienten Code zu erzeugen, muss ein Compiler viele Fragen klären:

- Gibt es nicht deklarierte Bezeichner?
- Gibt es deklarierte, aber nicht benutzte Bezeichner?
- Ist ein Bezeichner ein Skalar, ein Array oder eine Prozedur? Welchen Typs?
- Wird x vor seiner Benutzung definiert?
- Ist She-Ra \in Princess of Power?
- Ist ein Ausdruck Typ-konsistent?
- Wo sollen die Werte eines Bezeichners gespeichert werden? (Register, Stack, Heap)
- Referenzieren zwei Bezeichner dieselbe Adresse? (Aliasing)
- ...

Statische Semantik

Bezieht sich auf Eigenschaften der Programmkonstrukte

- Wahr für jedes Vorkommen dieses Konstrukts in jeder Programmausführung (static)
- Kann zur Compilezeit entschieden werden
- Kontext-sensitiv $\Rightarrow \notin CFG$ (Semantik)

Attributierte Grammatiken

Statische Semantik bezieht sich auf Eigenschaften von Programmkonstrukten, die für jede Ausführung des Programms gleich bleiben (static) und zur Compilezeit entschieden werden können. Allerdings sind sie kontextsensitiv und können somit nicht durch kontextfreie Grammatiken ausgedrückt werden.

Wir wollen aber Eigenschaften für Programme, die zwar abhängig vom Kontext, dafür aber Laufzeitunabhängig sind. Dafür werden kontextfreie Grammatiken um Semantik ergänzt, die dem Syntaxbaum Attributswerte zuordnen. Das Ergebnis ist dann ein Attributierter Syntaxbaum. Die semantische Analyse entspricht dann der Auswertung dieser Attribute.

Es gibt zwei Arten von Attributen:

- Synthetisch (bottom-up)
- Inherent (top-down)

Dabei wird jede Produktionsregel der Grammatik um semantische Regeln ergänzt.

Definition: Attribute Grammar

Sei $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ mit $X := N \uplus \Sigma$

- $Att = Syn \uplus Inh$ sei eine Menge von Attributen, $V = \bigcup_{\alpha \in Att} V^\alpha$ sei eine Vereinigung von Wertemengen
- $att : X \rightarrow \mathfrak{B}(Att)$ sei eine Attributzuweisung, $syn(Y) := att(Y) \cap Syn$ und $inh(Y) := att(Y) \cap Inh$ für jedes $Y \in X$
- Jede Produktion $\Pi = Y_0 \rightarrow Y_1 \dots Y_r \in P$ legt die Menge $Var_\Pi := \{\alpha.i \mid \alpha \in att(Y_i), i \in \{0, \dots, r\}\}$ der *Attributvariablen* von Π mit den Teilmengen der *inneren und äußeren Variablen* fest:

$$In_\Pi := \{\alpha.i \mid (i = 0, \alpha \in syn(Y_i)) \text{ oder } (i \in [r], \alpha \in inh(Y_i))\}$$

$$Out_\Pi := Var_\Pi \setminus In_\Pi$$

- Eine *semantische Regel* von Π ist eine Gleichung der Form $\alpha.i = f(\alpha_1.i_1, \dots, \alpha_n.i_n)$ mit $n \in \mathbb{N}, \alpha.i \in In_\Pi, \alpha_j.i_j \in Out_\Pi, f : V^{\alpha_1} \times \dots \times V^{\alpha_n} \rightarrow V^\alpha$
- Für jedes $\Pi \in P$ sei E_Π eine Menge mit genau einer semantischen Regel für jede innere Variable von Π und sei $E := (E_\Pi \mid \Pi \in P)$

Dann ist $\mathfrak{A} := \langle G, E, V \rangle$ eine *Attributgrammatik*, $\mathfrak{A} \in AG$

Beispiel

$\mathfrak{A}_B \in AG$ für binäre Zahlen:

- Attribute: $Att = Syn \uplus Inh$ mit $Syn = \{v, l\}$ und $Inh = \{p\}$
- Wertemengen: $V^v = \mathbb{Q}, V^l = \mathbb{N}, V^p = \mathbb{Z}$

• Attributzuweisungen:

$Y \in X$	N	L	B	0	1
$syn(Y)$	$\{v\}$	$\{v, l\}$	$\{v\}$	\emptyset	\emptyset
$inh(Y)$	\emptyset	$\{p\}$	$\{p\}$	\emptyset	\emptyset

- **Attributvariablen:**

$\Pi \in P$	$N \rightarrow L$	$N \rightarrow L.L$	$L \rightarrow B$	$L \rightarrow LB$	$B \rightarrow 0$	$B \rightarrow 1$
In_Π	$\{v.0, p.1\}$	$\{v.0, p.1, p.3\}$	$\{v.0, l.0, p.1\}$	$\{v.0, l.0, p.1, p.2\}$	$\{v.0\}$	$\{v.0\}$
Out_Π	$\{v.1\}$	$\{v.1, v.3\}$	$\{v.1, p.0\}$	$\{v.1, v.2, l.1, p.0\}$	$\{p.0\}$	$\{p.0\}$

Definition: Attribution of syntax trees

Sei $\mathfrak{A} = \langle G, E, V \rangle \in AG$ und t sei ein syntax tree von G mit Knotenmenge K

- K legt die Menge der *Attributvariablen von t* fest: $Var_t := \{\alpha.k \mid k \in K \text{ gelabeled mit } Y \in X, \alpha \in att(Y)\}$
- Sei k_0 ein (innerer) Knoten, an dem die Darstellung $\Pi = Y_0 \rightarrow Y_1 \dots Y_r \in P$ gilt und sei $k_1, \dots, k_r \in K$ die jeweiligen Nachfolgeknoten. Das *Attribut-Gleichungssystem* E_{k_0} von k_0 wird aus E_Π gewonnen durch das Ersetzen jedes Attributindex $i \in \{0, \dots, r\}$ durch k_i
- Das Attribut-Gleichungssystem von t ist gegeben durch $E_t := \bigcup \{E_k \mid k \text{ ist innerer Knoten von } t\}$

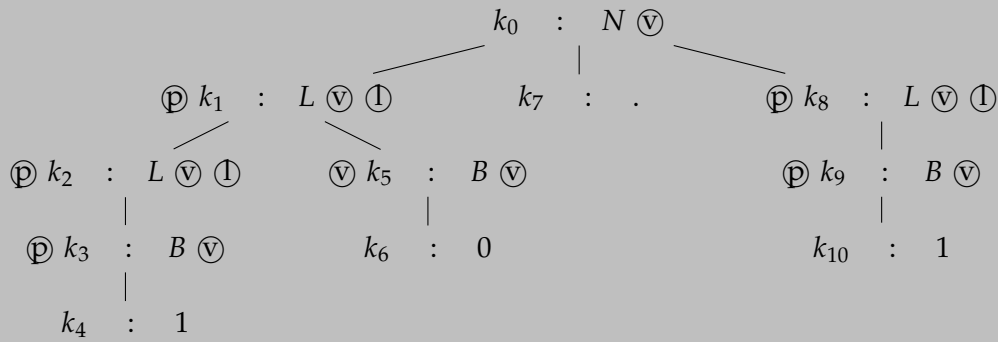
Corollar

Für jedes $\alpha.k \in Var_t$ ohne die geerbten Attributvariablen am Root und ohne die synthetischen Attribute an den Blättern von t , beinhaltet E_t genau eine Gleichung mit der linken Seite $\alpha.k$

Annahmen:

- Das Startsymbol hat keine geerbten Attribute: $inh(S) = \emptyset$
- Synthetische Attribute von Terminalsymbolen werden durch den Scanner erstellt

Beispiel



$$E_{N \rightarrow L.L}: \begin{aligned} v.0 &= v.1 + v.3 \\ p.1 &= 0 \\ p.3 &= -l.3 \end{aligned}$$

subset \rightarrow

$$E_{k_0}: \begin{aligned} v.k_0 &= v.k_1 + v.k_8 \\ p.k_1 &= 0 \\ p.k_8 &= -l.k_8 \end{aligned}$$

$$E_{L \rightarrow LB}: \begin{aligned} v.0 &= v.1 + v.2 \\ l.0 &= l.1 + 1 \\ p.1 &= p.0 + 1 \\ p.2 &= p.0 \end{aligned}$$

subset \rightarrow

$$E_{k_1}: \begin{aligned} v.k_1 &= v.k_2 + v.k_5 \\ l.k_1 &= l.k_2 + 1 \\ p.k_2 &= p.k_1 + 1 \\ p.k_5 &= p.k_1 \end{aligned}$$

4.2 Kapitel 16

Definition: Lösung eines Attribut-Gleichungssystems

Eine Lösung des Gleichungssystems E_t mit dem Syntaxbaum t ist eine Abbildung $v : Var_t \rightarrow V$, so dass für jedes $\alpha.k \in Var_t$ und $\alpha.k = f(\alpha.k_1, \dots, \alpha.k_n) \in E_t$ gilt, dass $v(\alpha.k) = f(v(\alpha.k_1), \dots, v(\alpha.k_n))$. Das Gleichungssystem kann keine, genau eine oder viele Lösungen haben.

Definition: Zirkularität von Attributgrammatiken

Ziel: Eindeutige Lösbarkeit des Gleichungssystems \Rightarrow Vermeidung von Zirkularität

Eine attributierte Grammatik ist *zirkulär*, wenn eine Attributvariable des Syntaxbaums t von sich selbst abhängig ist (E_t rekursiv). Sonst: *nicht zirkulär*.

In_Π : Alle Variablen, die auf den linken Seiten von Produktionen stehen

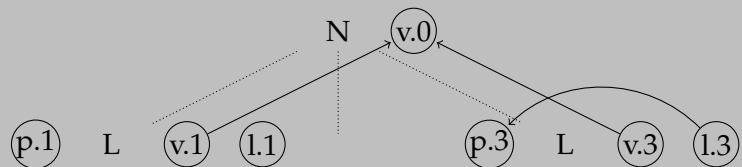
Out_Π : $Var_\Pi \setminus In_\Pi$

Zyklische Abhängigkeiten können auf dem Produktionslevel wegen der Aufteilung von Var_Π in In_Π und Out_Π nicht auftreten.

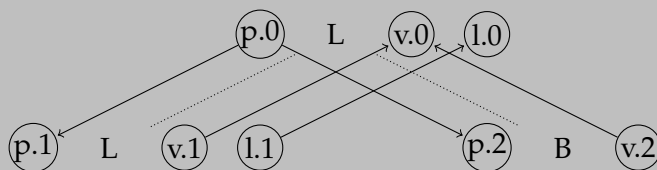
Zu jeder Produktion existiert ein azyklischer Abhängigkeitsgraph.

Beispiel

$$1. \quad \begin{aligned} N: &\rightarrow L.L \Rightarrow D_{N \rightarrow L.L} \\ v.0 &= v.1 + v.3 \\ p.1 &= 0 \\ p.3 &= -l.3 \end{aligned}$$



$$2. \quad \begin{aligned} L: &\rightarrow LB \Rightarrow D_{N \rightarrow LB} \\ v.0 &= v.1 + v.2 \\ l.0 &= l.1 + 1 \\ p.1 &= p.0 + 1 \\ p.2 &= p.0 \end{aligned}$$

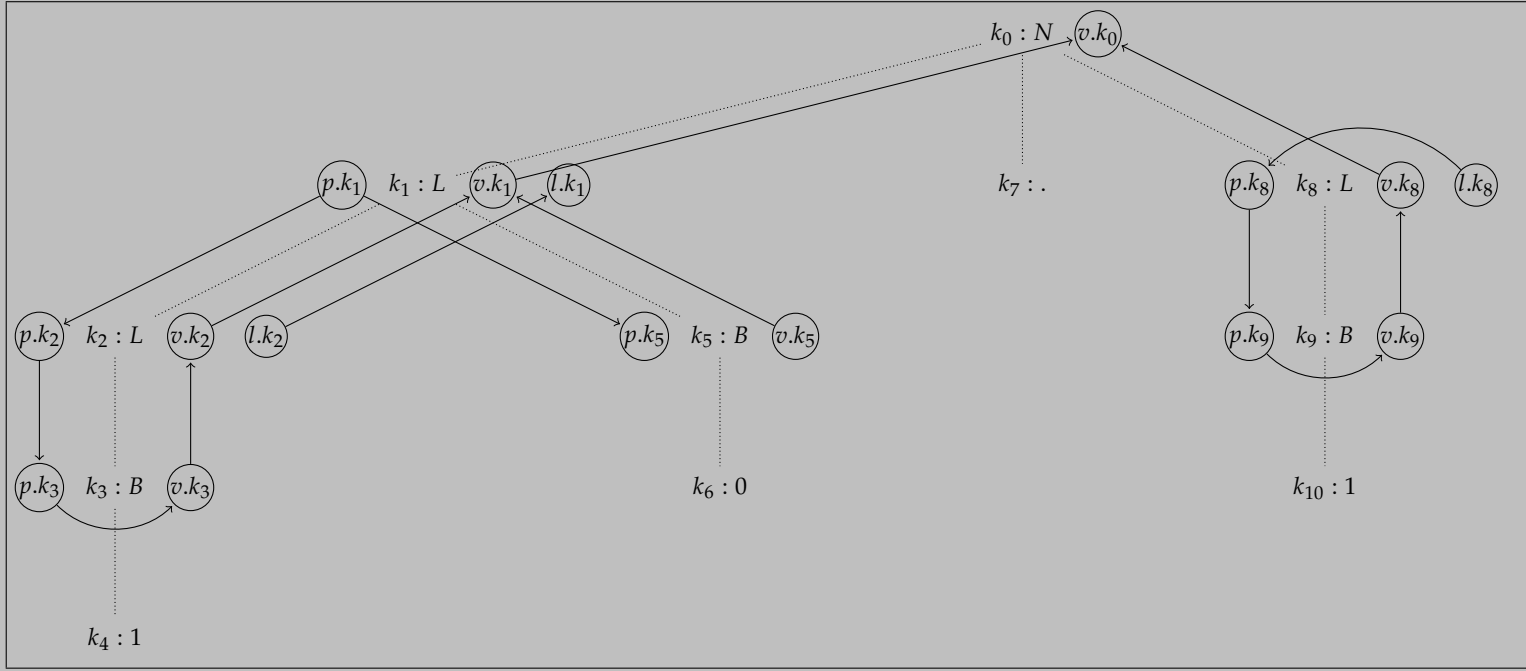


Definition: Tree Dependency Graph Der Abhängigkeitsgraph D_t des Syntaxbaums t entsteht durch zusammenkleben der Abhängigkeitsgraphen der einzelnen Produktionen.

D_t wird *zyklisch* genannt, wenn ein $x \in Var_t$ existiert mit $x \rightarrow_t^+ x$

Eine Attributgrammatik ist *zyklisch genau* dann wenn ein Syntaxbaum t existiert, so dass D_t zyklisch ist.

Beispiel



Beobachtung

Ein Zyklus in einem Abhängigkeitsgraphen D_t wird verursacht, wenn

- für mindestens ein $i \in [r]$ gibt es Attribute in $syn(A_i)$, die von $inh(A_i)$ abhängen
- eine Abhängigkeit in E_t das obere Ende des Zyklus bildet

Definition: Attributabhängigkeit Sei $\mathfrak{A} = \langle G, E, V \rangle \in AG$ mit $G = \langle N, \Sigma, P, S \rangle$.

- Wenn t ein Syntaxbaum ist mit Wurzel-Label $A \in N$ und Wurzelknoten k , $\alpha \in syn(A)$ und $\beta \in inh(A)$ so dass $\beta.k \rightarrow_t^+ \alpha.k$, dann ist α abhängig von β unter A in t (Notation: $\beta \xrightarrow{A} \alpha$ in t)
- Für jeden Syntaxbaum mit Wurzel-Label $A \in N$ gilt $D(A, t) := \{(\beta, \alpha) \in inh(A) \times syn(A) \mid \beta \xrightarrow{A} \alpha \text{ in } t\}$
- Für alle $A \in N$ gilt $\mathfrak{D}(A) := \{D(A, t) \mid t \text{ Syntaxbaum mit Wurzel-Label } A\} \subseteq \mathfrak{B}(Inh \times Syn)$

4.3 Kapitel 17

Algorithmus: Zirkularitätstest für Attributgrammatiken

Eingabe: $\mathfrak{A} = \langle G, E, V \rangle \in AG$ mit $G = \langle N, \Sigma, P, S \rangle$

Ablauf:

1. für jedes $A \in N$, konstruiere $\mathfrak{D}(A)$ iterativ wie folgt:
 - a) wenn $\pi = A \rightarrow w \in P$ dann $D[\pi] \in \mathfrak{D}(A)$
 - b) wenn $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ und $D_i \in \mathfrak{D}(A_i)$ für jedes $i \in [r]$ dann $D[\pi; D_1, \dots, D_r] \in \mathfrak{D}(A)$
2. teste, ob \mathfrak{A} zyklisch ist, indem zuerst überprüft wird, ob $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ und $D_i \in \mathfrak{D}(A_i)$ für jedes $i \in [r]$ existiert, so dass die folgende Relation zyklisch ist:

$$\rightarrow_{\pi} \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in D_i\}$$

(mit $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

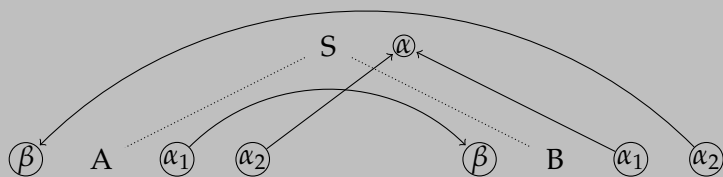
Ausgabe: *ja* oder *nein*

Eine Attributgrammatik ist zirkulär, wenn der Algorithmus mit *ja* antwortet.

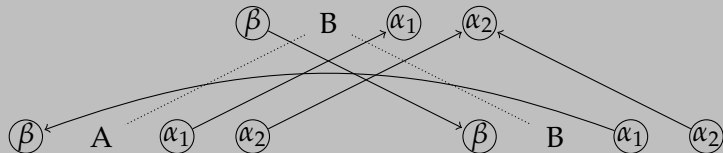
Die Komplexität dieses Algorithmusses ist exponentiell in der Grösse der Attributgrammatik.

Beispiel

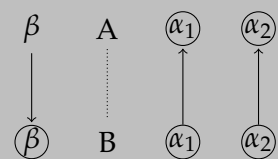
$D_{S \rightarrow AB}$:



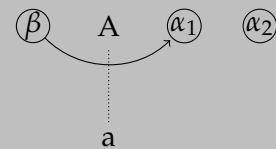
$D_{B \rightarrow AB}$:



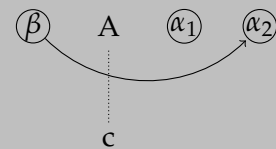
$D_{A \rightarrow B}$:



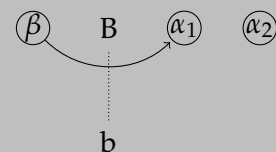
$D_{A \rightarrow a}$:



$D_{A \rightarrow c}$:



$D_{B \rightarrow b}$:



Zirkularitätstest vereinfachen, Definition Attributabhängigkeit

Idee: keine Unterscheidung zwischen Attributabhängigkeiten verschiedener Syntaxbäume

Für jedes $A \in N$ gilt, dass $\mathfrak{D}'(A) := \{(\beta, \alpha) \mid \beta \xrightarrow{A} \alpha \text{ in einigen Syntaxbäumen mit Wurzel-Label } A \subseteq Inh \times Syn\}$

Algorithmus: Strong circularity test for attribute grammars

Sei $\mathfrak{A} = \langle G, E, V \rangle \in AG$ mit $G = \langle N, \Sigma, P, S \rangle$

Ablauf:

1. für jedes $A \in N$, konstruiere iterativ $\mathfrak{D}'(A)$ wie folgt:
 - a) wenn $\pi = A \rightarrow w \in P$, dann $D[\pi] \subseteq \mathfrak{D}'(A)$
 - b) wenn $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$, dann $D[\pi : \mathfrak{D}'(A_1), \dots, \mathfrak{D}'(A_r)] \subseteq \mathfrak{D}'(A)$
2. teste, ob $\pi = A \rightarrow w_0 A_1 w_1 \dots A_r w_r \in P$ existiert, so dass die folgende Relation zyklisch ist: $\rightarrow_\pi \cup \bigcup_{i=1}^r \{(\beta.p_i, \alpha.p_i) \mid (\beta, \alpha) \in \mathfrak{D}'(A_i)\}$ (mit $p_i := \sum_{j=1}^i |w_{j-1}| + i$)

Ausgabe: *ja* oder *nein*

Eine Attributgrammatik ist zirkulär, wenn der Algorithmus mit *ja* antwortet.

- Der Zirkularitätstest läuft exponentiell in Bezug auf die Größe der Attributgrammatik.
- Der starke Zirkularitätstest läuft polynomiell in Bezug auf die Größe der Attributgrammatik.
- Jeder stark nicht-zyklische Attributgrammatik ist nicht-zyklisch
- Es gibt nicht-zyklische Attributgrammatiken, die nicht stark nicht-zyklisch sind

4.4 Kapitel 18**Attributauswertung**

Given:

- (stark) nichtzyklische Attributgrammatik $\mathfrak{A} = \langle G, E, V \rangle \in AG$
- Syntaxbaum t von G
- Bewertung $v : Syn_\Sigma \rightarrow V$ mit $syn_\Sigma := \{\alpha.k \mid k \text{ labelled durch } a \in \Sigma, \alpha \in syn(a)\} \subseteq Var_t$

Ziel:

Erweitere v zu einer (Teil-) Lösung $v : Var_t \rightarrow V$

Methoden:

1. Topologisches Sortieren von D_t (starten mit Attributvariablen die nur von synthetischen Attributen, die von Terminalen abhängen und fahre fort bei erfolgreicher Substitution)
2. Rekursive Funktionen (für stark nichtzyklische AGs)
3. Spezialfälle: S-attributierte Grammatiken (yacc), L-attributierte Grammatiken

Algorithmus (Evaluating durch topologisches Sortieren)

Eingabe: nichtzyklische $\mathfrak{A} = \langle G, E, V \rangle \in AG$, Syntaxbaum t von G , Berechnung $v : Syn_\Sigma \rightarrow V$

Prozedur:

1. sei $Var := Var_t \setminus Syn_\Sigma$ (zu Berechnende Attribute)
2. while $Var \neq \emptyset$ do
 - a) sei $x \in Var$, so dass $\{y \in Var \mid y \rightarrow_t x\} = \emptyset$
 - b) sei $x = f(x_1, \dots, x_n) \in E_t$
 - c) sei $v(x) := f(v(x_1), \dots, v(x_n))$
 - d) sei $Var := Var \setminus \{x\}$

Output: Ergebnis $v : Var_t \rightarrow V$

Attributauswertung durch rekursive Funktionen

Nur für stark nichtzyklische Attributgrammatiken

Ablauf:

- für jedes $A \in N$ und $\alpha \in \text{syn}(A)$, definiere Berechnungsfunktion $g_{A,\alpha}$ mit den folgenden Parametern:
 - der Knoten von t , an dem α berechnet wird (der durch A gelabelled wird) und
 - alle geerbten Attribute von A von denen α (potentiell) abhängt (das ist $\{\beta \in \text{inh}(A) \mid (\beta, \alpha) \in \mathfrak{D}'(A)\}$)
- gegeben sei ein Syntaxbaum t mit Wurzelknoten k_0 , berechne $g_{S,\alpha}(k_0)$ für jedes $\alpha \in \text{syn}(S)$

Ergebnis: Berechnet synthetische Attributvariablen am Wurzelknoten von t und alle Attributvariablen, von denen sie wirklich abhängen (entsprechend zu E_t)

Definition von Evaluierungsfunktionen

Für jedes $A \in N$ und $\alpha \in \text{syn}(A)$ sei(en)

- $\mathfrak{D}'(A) \subseteq \text{inh}(A) \times \text{syn}(A)$ wie es durch den strong circularity test berechnet wird
- $\text{inh}(A, \alpha) := \{\beta \in \text{inh}(A) \mid (\beta, \alpha) \in \mathfrak{D}'(A)\}$
- $A \rightarrow \gamma_1 \mid \dots \mid \gamma_m$ alle A -Produktionen in P

Dann ist $g_{A,\alpha}$ gegeben durch: $g_{A,\alpha}(k_0, \text{inh}(A, \alpha)) :=$ **case production applied at k_0 of**

\vdots
 $A \rightarrow \gamma_j : \text{eval}(\alpha.0)$
 \vdots
end

mit

$$\text{eval}(\alpha.i) := \begin{cases} \alpha & \text{wenn } \alpha \in \text{inh}(A), i = 0 \\ f(\text{eval}(\alpha_1.i_1), \dots, \text{eval}(\alpha_n.i_n)) & \text{wenn } \alpha.i \in \text{In}_{A \rightarrow \gamma_j}, \alpha.i = f(\alpha_1.i_1, \dots, \alpha_n.i_n) \in E_{A \rightarrow \gamma_j} \\ g_{Y_i, \alpha}(k_i, \text{eval}(\beta_1.i), \dots, \text{eval}(\beta_l.i)) & \text{wenn } \alpha \in \text{Syn}, i > 0, Y_i \in N, \text{inh}(Y_i, \alpha) = \{\beta_1, \dots, \beta_l\} \\ v(\alpha.i) & \text{wenn } \alpha \in \text{Syn}, i > 0, Y_i \in \Sigma \end{cases}$$

mit $\gamma_j = Y_1 \dots Y_r$ und k_i bezeichnet den i ten Nachfolger von k_0

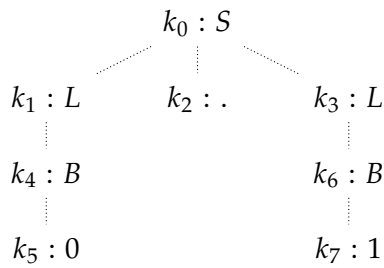
Beispiel 1

$$\begin{array}{l}
 G'_B: \\
 S \rightarrow L \quad v.0 = v.1 \\
 \quad \quad \quad p.1 = 0 \\
 S \rightarrow L.L \quad v.0 = v.1 + v.3 \\
 \quad \quad \quad p.1 = 0 \\
 \quad \quad \quad p.3 = -l.3 \\
 L \rightarrow B \quad v.0 = v.1 \\
 \quad \quad \quad l.0 = 1 \\
 \quad \quad \quad p.1 = p.0 \\
 L \rightarrow LB \quad v.0 = v.1 + v.2 \\
 \quad \quad \quad l.0 = l.1 + 1 \\
 \quad \quad \quad p.1 = p.0 + 1 \\
 \quad \quad \quad p.2 = p.0 \\
 B \rightarrow 0 \quad v.0 = 0 \\
 B \rightarrow 1 \quad v.0 = 2^{p.0}
 \end{array}
 \quad
 \begin{array}{l}
 g_{S,v}(k_0) = \text{case production}(k_0) \text{ of} \\
 S \rightarrow L : g_{L,v}(k_1, 0) \\
 S \rightarrow L.L : g_{L,v}(k_1, 0) + g_{L,v}(k_3, -g_{L,l}(k_3)) \\
 \text{end} \\
 g_{L,v}(k_0, p) = \text{case production}(k_0) \text{ of} \\
 L \rightarrow B : g_{B,v}(k_1, p) \\
 L \rightarrow LB : g_{L,v}(k_1, p + 1) + g_{B,v}(k_2, p) \\
 \text{end} \\
 g_{L,l}(k_0) = \text{case production}(k_0) \text{ of} \\
 L \rightarrow B : 1 \\
 L \rightarrow LB : g_{L,l}(k_1) + 1 \\
 \text{end} \\
 g_{B,v}(k_0, p) = \text{case production}(k_0) \text{ of} \\
 B \rightarrow 0 : 0 \\
 B \rightarrow 1 : 2^p \\
 \text{end}
 \end{array}$$

$A \in N$	S	L	B
$\mathfrak{D}'(A)$	\emptyset	$\{(p, v)\}$	$\{(p, v)\}$

Beispiel 2

Syntaxbaum t :



$$g_{S,v}(k_0) = g_{L,v}(k_1, 0) + g_{L,v}(k_3, -g_{L,l}(k_3)) = g_{B,v}(k_4, 0) + g_{L,v}(k_3, -g_{L,l}(k_3))$$

$$= 0 + g_{L,v}(k_3, -g_{L,l}(k_3)) = 0 + g_{B,v}(k_6, -g_{L,l}(k_3)) = 0 + 2^{-g_{L,l}(k_3)} = 0 + 2^{-1} = 0.5$$

Beispiel: Warum stark nichtzirkulär?

Wenn die Attributgrammatik nicht stark zirkulär ist, kann die Konstruktion der Evaluierungsfunktion fehlschlagen

$S \rightarrow A$	$\alpha.0 = \alpha_2.1$	Definition von $g_{S,\alpha}$:
	$\beta_1.1 = \alpha_1.1$	$g_{S,\alpha}(k_0)$
	$\beta_2.1 = \alpha_2.1$	$= eval(\alpha.0)$
$A \rightarrow a$	$\alpha_1.0 = \beta_2.0$	$= eval(\alpha_2.1)$
	$\alpha_2.0 = 2$	$= g_{A,\alpha_2}(k_1, eval(\beta_1.1))$
$A \rightarrow b$	$\alpha_1.0 = 1$	$= g_{A,\alpha_2}(k_1, eval(\alpha_1.1))$
	$\alpha_2.0 = \beta_1.0$	$= g_{A,\alpha_2}(k_1, g_{A,\alpha_1}(k_1, eval(\beta_2.1)))$
		$= g_{A,\alpha_2}(k_1, g_{A,\alpha_1}(k_1, eval(\alpha_2.1)))$
		\Rightarrow terminiert nicht $\frac{1}{2}$

$$\mathcal{D}'(A) = \{(\beta_2, \alpha_1), (\beta_1, \alpha_2)\}$$

L-Attributed Grammars I

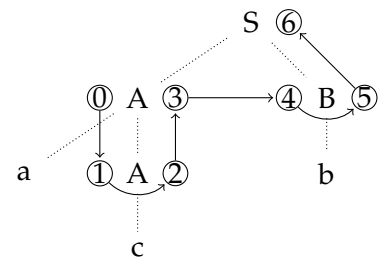
In einer L-attribuierten Grammatik sind Attributabhängigkeiten auf der rechten Seite von Produktionen nur *von links nach rechts* erlaubt.

Jede Grammatik $\mathcal{A} \in LAG$ ist nicht-zyklisch.

Example: L-attribuierte Grammatiken II

L-attribuierte Grammatik t :

$S \rightarrow AB$	$i.0 = 0$
	$i.2 = s.1 + 1$
	$s.0 = s.2 + 1$
$A \rightarrow aA$	$i.2 = i.0 + 1$
	$s.0 = s.2 + 1$
$A \rightarrow c$	$s.0 = i.0 + 1$
$B \rightarrow b$	$s.0 = i.0 + 1$



Evaluation of L-attributed Grammars

Beobachtung 1: der Syntaxbaum einer L-attribuierten Grammatik kann attribuiert werden durch eine *Tiefensuche, von-links-nach-rechts tree traversal* mit zwei Besuchen bei jedem Knoten

1. *top-down*: Evaluierung der *geerbten* Attribute
2. *bottom-up*: Evaluierung der *synthetischen* Attribute

Beobachtung 2: Besuchssequenz passt prima zu *parsing*

1. *top-down*: Expansionsschritt
2. *bottom-up*: Reduktionsschritt

Definition: Parsing automaton with attribute evaluation Sei $\mathfrak{A} = \langle G, E, V \rangle \in LAG$ mit $G = \langle N, \Sigma, P, S \rangle \in LL(1)$. Der *paring automaton with attribute evaluation* von \mathfrak{A} ist definiert durch die folgenden Komponenten:

- Eingabealphabet Σ
- Pushdown Alphabet $\Gamma := \bigcup_{\pi \in P \cup \{\rightarrow S\}} (LR(0)_\pi(G) \times Val_\pi)$ mit
 - $LR(0)_\pi(G) := \{[A \rightarrow \delta_1 \cdot \delta_2] \mid \pi = A \rightarrow \delta_1 \delta_2\}$ und
 - $Val_\pi := \{v \mid v : Out_\pi \rightarrow V\}$
- Konfigurationen $\Sigma^* \times \Gamma^*$
 - Eingangskonfiguration: $(w, ([\rightarrow \cdot S], v_\emptyset))$
 - Finale Konfiguration: $\{(\epsilon, ([\rightarrow S \cdot], v)) \mid v \in Val_{\rightarrow S}\}$
- Transitionen:
 - expand: (berechne geerbte Attribute des erweiterten Symbols)

if $x \in laB \rightarrow \delta'$ then

$$(xw, ([A \rightarrow Y_1 \dots Y_{i-1} \cdot B\delta], v)\gamma) \vdash (xw, ([B \rightarrow \cdot \delta'], v')([A \rightarrow Y_1 \dots Y_{i-1} \cdot B\delta], v)\gamma)$$

where $v'(\beta.0) := f(v(\alpha_1.i_1), \dots, v(\alpha_n.i_n))$ for $\beta \in inh(B)$ and $\beta.i = f(\alpha_1.i_1, \dots, \alpha_n.i_n) \in E_{A \rightarrow Y_1 \dots Y_{i-1} B \delta}$

match:

$$(aw, ([A \rightarrow \delta_1 \cdot a\delta_2], v)\gamma) \vdash (w, ([A \rightarrow \delta_1 a \cdot \delta_2], v)\gamma)$$

reduce: (berechne synthetische Attribute des reduzierten Symbols)

$$(w, ([B \rightarrow \delta' \cdot], v')([A \rightarrow Y_1 \dots Y_{i-1} \cdot B\delta], v)\gamma) \vdash (w, ([A \rightarrow Y_1 \dots Y_{i-1} B \cdot \delta], v'')\gamma)$$

where $v'' := v[\alpha.i \rightarrow f(v'(\alpha_1.i_1), \dots, v'(\alpha_n.i_n))]$ for $\alpha \in syn(B)$ and $\alpha.0 = f(\alpha_1.i_1, \dots, \alpha_n.i_n) \in E_{B \rightarrow \delta'}$

Example: Parsing with Attribute Evaluation

Syntaxbaum: siehe Example: L-attributierte Grammatiken II

acb

$[\rightarrow \cdot S]$	—
-------------------------	---

$\vdash acb$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—				
$[\rightarrow \cdot S]$	—								
$[S \rightarrow \cdot AB]$	—								
$\vdash acb$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr><tr><td>$[A \rightarrow \cdot aA]$</td><td>$i.0 = 0$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—	$[A \rightarrow \cdot aA]$	$i.0 = 0$		
$[\rightarrow \cdot S]$	—								
$[S \rightarrow \cdot AB]$	—								
$[A \rightarrow \cdot aA]$	$i.0 = 0$								
$\vdash cb$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr><tr><td>$[A \rightarrow a \cdot A]$</td><td>$i.0 = 0$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—	$[A \rightarrow a \cdot A]$	$i.0 = 0$		
$[\rightarrow \cdot S]$	—								
$[S \rightarrow \cdot AB]$	—								
$[A \rightarrow a \cdot A]$	$i.0 = 0$								
$\vdash cb$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr><tr><td>$[A \rightarrow a \cdot A]$</td><td>$i.0 = 0$</td></tr><tr><td>$[A \rightarrow \cdot c]$</td><td>$i.0 = 1$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—	$[A \rightarrow a \cdot A]$	$i.0 = 0$	$[A \rightarrow \cdot c]$	$i.0 = 1$
$[\rightarrow \cdot S]$	—								
$[S \rightarrow \cdot AB]$	—								
$[A \rightarrow a \cdot A]$	$i.0 = 0$								
$[A \rightarrow \cdot c]$	$i.0 = 1$								
$\vdash b$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr><tr><td>$[A \rightarrow a \cdot A]$</td><td>$i.0 = 0$</td></tr><tr><td>$[A \rightarrow c \cdot]$</td><td>$i.0 = 1$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—	$[A \rightarrow a \cdot A]$	$i.0 = 0$	$[A \rightarrow c \cdot]$	$i.0 = 1$
$[\rightarrow \cdot S]$	—								
$[S \rightarrow \cdot AB]$	—								
$[A \rightarrow a \cdot A]$	$i.0 = 0$								
$[A \rightarrow c \cdot]$	$i.0 = 1$								

$\vdash b$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow \cdot AB]$</td><td>—</td></tr><tr><td>$[A \rightarrow aA \cdot]$</td><td>$i.0 = 0, s.2 = 2$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow \cdot AB]$	—	$[A \rightarrow aA \cdot]$	$i.0 = 0, s.2 = 2$
$[\rightarrow \cdot S]$	—						
$[S \rightarrow \cdot AB]$	—						
$[A \rightarrow aA \cdot]$	$i.0 = 0, s.2 = 2$						
$\vdash b$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow A \cdot B]$</td><td>$s.1 = 3$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow A \cdot B]$	$s.1 = 3$		
$[\rightarrow \cdot S]$	—						
$[S \rightarrow A \cdot B]$	$s.1 = 3$						
$\vdash b$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow A \cdot B]$</td><td>$s.1 = 3$</td></tr><tr><td>$[B \rightarrow \cdot b]$</td><td>$i.0 = 4$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow A \cdot B]$	$s.1 = 3$	$[B \rightarrow \cdot b]$	$i.0 = 4$
$[\rightarrow \cdot S]$	—						
$[S \rightarrow A \cdot B]$	$s.1 = 3$						
$[B \rightarrow \cdot b]$	$i.0 = 4$						
$\vdash \epsilon$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow A \cdot B]$</td><td>$s.1 = 3$</td></tr><tr><td>$[B \rightarrow b \cdot]$</td><td>$i.0 = 4$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow A \cdot B]$	$s.1 = 3$	$[B \rightarrow b \cdot]$	$i.0 = 4$
$[\rightarrow \cdot S]$	—						
$[S \rightarrow A \cdot B]$	$s.1 = 3$						
$[B \rightarrow b \cdot]$	$i.0 = 4$						
$\vdash \epsilon$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow \cdot S]$</td><td>—</td></tr><tr><td>$[S \rightarrow AB \cdot]$</td><td>$s.1 = 3, s.2 = 5$</td></tr></table>	$[\rightarrow \cdot S]$	—	$[S \rightarrow AB \cdot]$	$s.1 = 3, s.2 = 5$		
$[\rightarrow \cdot S]$	—						
$[S \rightarrow AB \cdot]$	$s.1 = 3, s.2 = 5$						
$\vdash \epsilon$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>$[\rightarrow S \cdot]$</td><td>$s.1 = 6$</td></tr></table>	$[\rightarrow S \cdot]$	$s.1 = 6$				
$[\rightarrow S \cdot]$	$s.1 = 6$						

5 Code Generierung

5.1 Vorlesung 19

Modularization of Code Generation **Spaltung** von Code-Generierung für programming language PL:

$$PL \xrightarrow{trans} IC \xrightarrow{code} MC$$

Frontend *trans* generiert maschinenunabhängigen Zwischencode (IC) für abstrakte (stack) Maschinen

Backend *code* generiert wirklichen Maschinencode (MC)

Vorteile: IC ist Maschinenunabhängig \Rightarrow Portabel, schnelle Compiler-Implementierung, Codelänge (kurz), Code-Optimierung.

Struktur imperativer Programmiersprachen

- Grundlegende Datentypen und Operationen
- Statische und dynamische Datenstrukturen
- Ausdrücke und Zuweisungen
- Kontrollstrukturen (Sequenzen, verzweigte Statements, Schleifen, ...)
- Prozeduren und Funktionen
- Modularität: Blöcke, Module und Klassen

Struktur in Maschinensprachen (von Neumann/SISD)

Speicherhierarchie: zentrales Rechenregister, Register, Cache, Hauptspeicher, Hintergrundspeicher

Instruktionstypen: arithm./Boolean/... Operation, test/jump Instruktion, Transfer-Instruktion, I/O Instruktion, ...

Adressmodi: direkt/indirekt, absolut/relativ

Architekturen: RISC (wenige Instruktionen, viele Register), CISC (viele Instr., wenige Register)

Struktur in Zwischencode

- Typen und Operationen wie PL
- Daten-Stack mit Basisoperationen
- Sprunganweisungen für Kontrollstrukturen
- Laufzeit-Stack für Blöcke, Prozeduren und statische Datenstrukturen
- Heap für dynamische Datenstrukturen

The Example Programming Language EPL (BPS) Nur Integer- und Boolean-Werte, Arithmetische und Boolean-Ausdrücke (strikte/nicht strikte Semantik), Kontrollstrukturen: Sequenzen, Verzweigungen, Iterationen; Verzweigte Blöcke und rekur. Prozeduren (lokale/globale Variablen \Rightarrow dynamisches Speichermanagement). Prozedurparameter und Datenstrukturen später!

Definition: Syntax von EPL

\mathbb{Z} : z (* z ist ein Integer *)
 Ide : I (* I ist ein Identifier *)
 $AExp$: $A ::= z | I | A_1 + A_2 | \dots$
 $BExp$: $B ::= A_1 < A_2 | \text{not } B | B_1 \text{ and } B_2 | B_1 \text{ or } B_2$
 Cmd : $C ::= I := A | C_1 ; C_2 | \text{if } B \text{ then } C_1 \text{ else } C_2 | \text{while } B \text{ do } C | I()$
 Dcl : $D ::= D_C D_V D_P$
 D_C : $\epsilon | \text{const } I_1 := z_1, \dots, I_n := z_n;$
 D_V : $\epsilon | \text{var } I_1, \dots, I_n;$
 D_P : $\epsilon | \text{proc } I_1; K_1; \dots, I_n; K_n;$
 $Block$: $K ::= DC$
 Pgm : $P ::= \text{in/out } I_1, \dots, I_n; K.$

Static Semantics of EPL

- Alle Bezeichner in einer Deklaration D müssen unterschiedlich sein
- Jeder Bezeichner, der im Kommando C eines Blockes D vorkommt, muss deklariert sein in D oder in der Deklarationsliste des umgebenden Blocks
- *Multiple Deklarationen* eines Bezeichners in verschiedenen Blöcken ist möglich. Jede Benutzung in einem Kommando C bezieht sich auf die innerste Deklaration
- *Static scoping*: Die Benutzung eines Bezeichners im Body einer aufgerufenen Prozedur bezieht sich auf seine Deklarationsumgebung (und nicht auf seine aufrufende Umgebung)

Example: Static Semantics of EPL

```
in/ out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [...z := 1; P() ...]  
  [...P() ... R() ...]  
proc R;  
  [...P() ...]  
[...x := 0; P() ...].
```

- „Innermost“-Prinzip
- Static scoping: Body von P kann auf x, y, z verweisen
- Späte Deklaration: Aufruf von R in P gefolgt von einer Deklaration (in Pascal: *forward* Deklarationen für eine one-pass Compilierung)

Definition: Semantic Domains of EPL Speicheradressen (locations) speichern der Integer-Werte (\Rightarrow Zustände), Variablenbezeichner verweisen auf locations, Konstante Bezeichner verweisen auf Integers, Kommandos transformieren Zustände, Prozedur-Bezeichner verweisen auf Zustandstransformationen, Bezeichner-Referenzen (environments) werden durch Deklarationen ermittelt. Die semantischen Bereiche von EPL sind wie folgt gegeben:

$\mathbb{Z} := \{0, 1, -1, \dots\}$ Integer-Zahlen
 $\mathbb{B} := \{true, false\}$ Boolesche Werte
 $Loc := \{\alpha_1, \alpha_2, \dots\}$ Locations
 $Stt := \{\sigma | \sigma : Loc \rightarrow \mathbb{Z}\}$ States
 $Trn := \{\tau | \tau : Stt \rightarrow Stt\}$ State-Transformationen
 $Env := \{\rho | \rho : Ide \rightarrow \mathbb{Z} \cup Loc \cup Trn\}$ Environments

Definition: Semantics of Arithmetic Expression Die Semantik eines AExp ist sein Integerwert (oder undefiniert)

$$\mathfrak{A} : AExp \times Env \times Stt \rightarrow \mathbb{Z}$$

$$\mathfrak{A}[\![z]\!] \rho \sigma := z$$

$$\mathfrak{A}[\![I]\!] \rho \sigma := \begin{cases} z & \text{if } \rho(I) = z \in \mathbb{Z} \\ \sigma(\alpha) & \text{if } \rho(I) = \alpha \in Loc \end{cases}$$

$$\mathfrak{A}[\![A_1 + A_2]\!] \rho \sigma := \mathfrak{A}[\![A_1]\!] \rho \sigma + \mathfrak{A}[\![A_2]\!] \rho \sigma$$

Bemerkung: $\mathfrak{A}[\![I]\!] \rho \sigma$ ist undefiniert (Notation: $\mathfrak{A}[\![I]\!] \rho \sigma = \perp$) wenn I ein Prozedur-Bezeichner ist, zB $\rho(I) \in Trn$

Definition: Semantics of Boolean Expression Die Semantik eines BExp ist sein Wahrheitswert (oder undefiniert)

$$\mathfrak{B} : BExp \times Env \times Stt \rightarrow \mathbb{B}$$

$$\mathfrak{B}[\![A_1 < A_2]\!] \rho \sigma := \mathfrak{A}[\![A_1]\!] \rho \sigma < \mathfrak{A}[\![A_2]\!] \rho \sigma$$

$$\mathfrak{B}[\![not B]\!] \rho \sigma := \neg \mathfrak{B}[\![B]\!] \rho \sigma$$

$$\mathfrak{B}[\![B_1 \text{ and } B_2]\!] \rho \sigma := \mathfrak{B}[\![B_1]\!] \rho \sigma \wedge \mathfrak{B}[\![B_2]\!] \rho \sigma$$

$$\mathfrak{B}[\![B_1 \text{ or } B_2]\!] \rho \sigma := \mathfrak{B}[\![B_1]\!] \rho \sigma \vee \mathfrak{B}[\![B_2]\!] \rho \sigma$$

Bemerkungen: $\mathfrak{B}[\![B]\!] \rho \sigma$ ist nur dann undefiniert, wenn der Wert einer arithmetischen Subexpression von B undefiniert ist.

	Strict:	Sequential:	Non-strict:
Mögl. Interpretationen von binären Operationen:	$a \hat{\vee} \perp = \perp$	$\text{false} \wedge \perp = \text{false}$	$\text{false} \wedge \perp = \perp \wedge \text{false} = \text{false}$
	$\perp \hat{\vee} b = \perp$	$\text{true} \vee \perp = \text{true}$	$\text{true} \vee \perp = \perp \vee \text{true} = \text{true}$
		$\perp \hat{\vee} b = \perp$	

Definition: Semantics of Commands Cmd's modifizieren den Wert von Variablen, zB transform states.

$$\mathfrak{C} : Cmd \times Env \times Stt \rightarrow Stt$$

$$\mathfrak{C}[\![I := A]\!] \rho \sigma := \sigma[\alpha \mapsto z]$$

$$\text{if } \rho(I) = \alpha \in Loc \text{ and } \mathfrak{A}[\![A]\!] \rho \sigma = z \in \mathbb{Z}$$

$$\text{where } \rho[\alpha \mapsto z](\beta) := \begin{cases} z & \text{if } \beta = \alpha \\ \sigma(\alpha) & \text{otherwise} \end{cases}$$

$$\mathfrak{C}[\![C_1; C_2]\!] \rho \sigma := \mathfrak{C}[\![C_2]\!] \rho(\mathfrak{C}[\![C_1]\!] \rho \sigma)$$

$$\mathfrak{C}[\![\text{if } B \text{ then } C_1 \text{ else } C_2]\!] \rho \sigma := \begin{cases} \mathfrak{C}[\![C_1]\!] \rho \sigma & \text{if } \mathfrak{B}[\![B]\!] \rho \sigma \\ \mathfrak{C}[\![C_2]\!] \rho \sigma & \text{if } \neg \mathfrak{B}[\![B]\!] \rho \sigma \end{cases}$$

$$\mathfrak{C}[\![\text{while } B \text{ do } C =: C']]\!] \rho \sigma := \begin{cases} \mathfrak{C}[\![C']]\!] \rho(\mathfrak{C}[\![C]\!] \rho \sigma) & \text{if } \mathfrak{B}[\![B]\!] \rho \sigma \\ \sigma & \text{if } \neg \mathfrak{B}[\![B]\!] \rho \sigma \end{cases}$$

$$\mathfrak{C}[\![I()]\!] \rho \sigma := \tau(\rho) \text{ if } \rho(I) = \tau \in Trn$$

Bemerkung: In der letzten Klausel wird $\rho(I) = \tau$ unter Berücksichtigung der Deklarationsumgebung von I berechnet

5.2 Vorlesung 20

Definition: Semantic Declarations

$$\begin{aligned} \mathfrak{D} &: Dcl \times Env \times Stt \rightarrow Env \times Sit \\ \mathfrak{D}[[D_C D_V D_P]]\rho\sigma &:= \mathfrak{D}[[D_P]](\mathfrak{D}[[D_V]](\mathfrak{D}[[D_C]]\rho\sigma)) \\ &\text{wenn alle Bezeichner in } D_C D_V D_P \text{ unterschiedlich sind} \\ \mathfrak{D}[[\epsilon]]\rho\sigma &:= \rho\sigma \\ \mathfrak{D}[[\text{const } I_1 := z_1, \dots, I_n := z_n]]\rho\sigma &:= \rho[I_1 \mapsto z_1, \dots, I_n \mapsto z_n]\sigma \\ \mathfrak{D}[[\text{var } I_1, \dots, I_n;]] &:= \rho[I_1 \mapsto \alpha_{l+1}, \dots, I_n \mapsto \alpha_{l+n}]\sigma[\alpha_{l+1} \mapsto 0, \dots, \alpha_{l+n} \mapsto 0] \\ &\text{mit } l := \max \{m \geq 1 \mid \sigma(\alpha_m) \neq \perp\} \\ \mathfrak{D}[[\text{proc } I_1; K_1; \dots; I_n; K_n;]]\rho\sigma &:= \rho[I_1 \mapsto \tau_1, \dots, I_n \mapsto \tau_n]\sigma \\ &\text{mit } \tau_i(\sigma) = \mathfrak{K}[[K_i]]\rho[I_1 \mapsto \tau_1, \dots, I_n \mapsto \tau_n]\sigma \quad \forall i \in [n] \end{aligned}$$

Definition: Sematic of Blocks Blocksemantik: $K = D C$; Erweiterung der aktuellen Umgebung in Bezug auf Deklaration D , Ausführen von Kommando C in der erweiterten Umgebung, Freigeben der von D belegten Speicheradressen

$$\begin{aligned} \mathfrak{K} &: Block \times Env \times Stt \rightarrow Stt \\ \mathfrak{K}[[DC]]\rho\sigma &:= \mathfrak{C}[[C]](\mathfrak{D}[[D]]\rho\sigma)|_{\text{dom}(\sigma)} \quad \text{mit } \text{dom}(\sigma) := \{\alpha \in Loc \mid \sigma(\alpha) \neq \perp\} \end{aligned}$$

Definition: Programmsemantik

$$\begin{aligned} \mathfrak{M} &: Pgm \times \mathbb{Z}^* \rightarrow \mathbb{Z}^* \\ \mathfrak{M}[[\text{in/out } I_1, \dots, I_n; K.]](z_1, \dots, z_n) &:= (\sigma(\alpha_1), \dots, \sigma(\alpha_n)) \\ \text{mit } \sigma &:= \mathfrak{K}[[K]]\rho_{\emptyset}[I_1 \mapsto \alpha_1, \dots, I_n \mapsto \alpha_n]\sigma_{\emptyset}[\alpha_1 \mapsto z_1, \dots, \alpha_n \mapsto z_n] \\ \text{und } \rho_{\emptyset}(I) &:= \sigma_{\emptyset}(\alpha) := \perp \quad \forall I \in Ide, \alpha \in Loc \end{aligned}$$

EPL: Abstract Machine (AM) Die AM für EPL ist def. durch den Zustandsraum: $S := PC \times DS \times PS$ mit: *Programmzähler* (program counter) $PC := \mathbb{N}$, *Datenstack* (data stack) $DS := \mathbb{Z}^*$ (Spitze des Stacks nach rechts!), *Prozedurstack* (procedure stack or runtime stack) $PS := \mathbb{Z}^*$ (Spitze des Stacks nach links!)

Ein Zustand $s = (l, d, p) \in S$ ist gegeben durch: Programmlabel $l \in PC$, Datenstack $d = d.r : \dots : d.1 \in DS$, Prozedurstack $p = p.1 : \dots : p.t \in PS$

Folgende AM Instruktionen gibt es: *Arithmetische* ADD, MULT, ..., *Boolean* NOT, AND, OR, LT, ..., *Sprung* (jump) JMP(ca), JFALSE(ca) ($ca \in PC$), *Prozedur* CALL(ca, dif, loc) ($ca \in PC, dif, loc \in \mathbb{N}$), RET, *Transfer* LOAD(dif, off), STORE(dif, off) ($dif, off \in \mathbb{N}$), LIT(z) ($z \in \mathbb{Z}$)

EPL: Instruktionssymantik Semantik einer Instruktion O

$$\begin{aligned} [[O]] &: S \rightarrow S \\ [[\text{ADD}]](l, d : z_1 : z_2, p) &:= (l + 1, d : z_1 + z_2, p) \\ [[\text{NOT}]](l, d : b, p) &:= (l + 1, d : \neg b, p) \text{ if } b \in \{0, 1\} \\ [[\text{AND}]](l, d : b_1 : b_2, p) &:= (l + 1, d : b_1 \wedge b_2, p) \text{ if } b_i \in \{0, 1\} \\ [[\text{OR}]](l, d : b_1 : b_2, p) &:= (l + 1, d : b_1 \vee b_2, p) \text{ if } b_i \in \{0, 1\} \\ [[\text{LT}]](l, d : z_1 : z_2, p) &:= \begin{cases} (l + 1, d : 1, p) & \text{if } z_1 < z_2 \\ (l + 1, d : 0, p) & \text{if } z_1 \geq z_2 \end{cases} \\ [[\text{JMP}(ca)]](l, d, p) &:= (ca, d, p) \\ [[\text{JFALSE}(ca)]](l, d : b, p) &:= \begin{cases} (ca, d, p) & \text{if } b = 0 \\ (l + 1, d, p) & \text{if } b = 1 \end{cases} \end{aligned}$$

EPL: Struktur des Prozedurstacks Die Semantik von Prozeduren und Transfer Instruktionen benötigt eine spezielle Struktur des Prozedurstacks $p \in PS$. Dieser muß aus *Frames* (oder: *activation records*) der folgenden Form $sl : dl : ra : v_1 : \dots : v_k$ bestehen. Mit:

static link (sl): zeigt auf das Frame der umgebenen Deklarationsumgebung
 → benutzt für Zugriff auf nichtlokale Variablen

dynamic link (dn): zeigt auf das vorherige Frame (z.B. aufrufende Prozedur)
 → benutzt um das oberste Frame nach Prozedurende zu entfernen

return address (ra): Programmlabel nach Beendigung des Prozeduraufrufes
 → benutzt um Programmausführung nach Prozedurende fortzusetzen

local variables (v_i) Werte der lokal deklarierten Variablen

Frames werden immer erzeugt wenn ein Prozeduraufruf ausgeführt wird. Und es gibt zwei spezielle Frames:

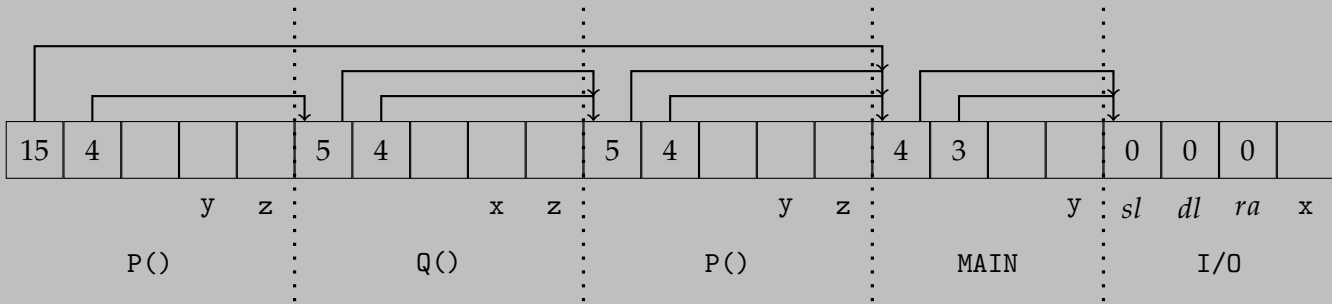
I/O frame: um Werte von Ein-/Ausgabevariablen zu halten ($sl = dl = ra = 0$)

MAIN frame: um Werte des obersten Blocks zu halten ($sl = dl = I/O \text{ frame}$)

```

in/out x;
const c = 10;
var y;
proc P;
  var y, z;
  proc Q;
    var x, z;
    [... P() ...]
    [... Q() ...]
  proc R;
    [... P() ...]
  [... P() ...].
    
```

Prozedurstack nach dem zweiten Aufruf von P.



Beobachtung: Die Benutzung einer Variable in einer Prozedur bezieht sich auf die innerste Deklaration. Wenn der Levelunterschied zwischen der Benutzung und der Deklaration *dif* ist, dann muß einer Kette von *dif* statischen Links (sl) gefolgt werden um das entsprechende Frame zu betreten.

5.3 Vorlesung 21

EPL: base Funktion Beim Prozeduraufruf muß der sl berechnet werden, dies geschieht mithilfe der base Hilfsfunktion, welche mit Prozedurstack und Differenz den Beginn des entsprechenden Frames zurück gibt.

$$\begin{aligned}
 \text{base} &: PS \times \mathbb{N} \rightarrow \mathbb{N} \\
 \text{base}(p, 0) &:= 1 \\
 \text{base}(p, dif + 1) &:= \text{base}(p, dif) + p.\text{base}(p, dif)
 \end{aligned}$$

base-Funktion Im zweiten Aufruf von P (aus Q): $dif = 2$

$$\begin{aligned} \text{base}(p, 0) &= 1 \\ \Rightarrow \text{base}(p, 1) &= 1 + p.1 = 6 \\ \Rightarrow \text{base}(p, 2) &= 6 + p.6 = 11 \\ \Rightarrow sl &= \text{base}(p, 2) + \underbrace{2}_{y, z} + \underbrace{2}_{ra, dl} = 15 \end{aligned}$$

EPL: Prozedursemantik

- $\text{CALL}(ca, dif, loc)$ erzeugt das neue Frame und springt zur Startadresse. Mit *code address* $ca \in PC$, *level difference* $dif \in \mathbb{N}$, *Anzahl lokaler Variablen* $loc \in \mathbb{N}$.
- RET entfernt das oberste Frame und kehrt zur aufrufenden Stelle zurück.

Die Semantik einer Prozedurinstruktion: $\llbracket O \rrbracket : S \rightarrow S$ ist wie folgt definiert:

$$\begin{aligned} \llbracket \text{CALL}(ca, dif, loc) \rrbracket(l, d, p) &:= (ca, d, \underbrace{(\text{base}(p, dif) + loc + 2)}_{sl}) : \underbrace{(loc + 2)}_d l : \underbrace{(l + 1)}_{ra} : \underbrace{0 : \dots : 0}_{loc \text{ times}} : p) \\ \llbracket \text{RET} \rrbracket(l, d, p.1 : \dots : p.t) &:= (\underbrace{p.3}_{ra}, d, p.(p.2 + 2) : \dots : p.t) \text{ wenn } t \geq p.2 + 2 \end{aligned}$$

EPL: Transfersemantik

- $\text{LOAD}(dif, off)$ und $\text{STORE}(dif, off)$ lädt und speichert Variablenwerte zwischen Daten- und Prozedurstack (Kette von dif static links) mit *Levelunterschied* $dif \in \mathbb{N}$ und *Variablen Offset* $off \in \mathbb{N}$
- $\text{LIT}(z)$ lädt die Literal-Konstante $z \in \mathbb{Z}$

Die Semantik einer Transferinstruktion: $\llbracket O \rrbracket : S \rightarrow S$ ist wie folgt definiert:

$$\begin{aligned} \llbracket \text{LOAD}(dif, off) \rrbracket(l, d, p) &:= (l + 1, d : p.(\text{base}(p, dif) + off + 2), p) \\ \llbracket \text{STORE}(dif, off) \rrbracket(l, d : z, p) &:= (l + 1, d, p[\text{base}(p, dif) + off + 2 \mapsto z]) \\ \llbracket \text{LIT}(z) \rrbracket(l, d, p) &:= (l + 1, d : z, p) \end{aligned}$$

EPL: AM Programmsymantik Ein AM Programm ist Folge von $k \geq 1$ AM Instruktionen: $P = a_1 : O_1; \dots; a_k : O_k$ wobei $a_i \in PC | a_i = a_1 + i - 1 \forall i \in [k]$. AM die Menge aller AM Programme ist.

Die Semantik eines AM Programms ist gegeben durch:

$$\begin{aligned} \llbracket \cdot \rrbracket &: AM \times S \rightarrow S \\ \llbracket P \rrbracket(l, d, p) &:= \begin{cases} \llbracket P \rrbracket(\llbracket O_{l-a_1+1} \rrbracket(l, d, p)) & \text{if } l \in \{a_1, \dots, a_k\} \\ (l, d, p) & \text{sonst} \end{cases} \end{aligned}$$

Übersetzen von EPL nach AM Ziel ist die Definition von Übersetzungsabbildung: $\text{trans} : Pgm \rightarrow AM$ mithilfe einer *Symboltabelle* dessen Einträge von Deklarationen erzeugt werden und zur Kommandoübersetzung benutzt werden:

$$\begin{aligned} Tab &:= \{st | st : Ide \rightarrow (\{\text{const} \times \mathbb{Z}\} \\ &\quad \cup (\{\text{var}\} \times Lev \times Off) \\ &\quad \cup (\{\text{proc}\} \times PC \times Lev \times Size)\} \end{aligned}$$

variablen Deklarationen *Deklarationslevel* lev (unterschied zwischen Benutzung und Deklarationslevel), *Offset* off (Position im aktuellen Frame)

Prozedurdeklarationen ca, lev, loc mit $loc \in Size := \mathbb{N}$ Anzahl lokaler Variablen.

Warten der Symboltabelle Durch Funktion: $update(D, st, a, l)$ welches das Update von Symboltabelle st in Bezug zur Deklaration D unter Beachtung der nächsten freien Codeadresse a und aktuellem Level l spezifiziert.

$$update : Dcl \times Tab \times PC \times Lev \rightarrow Tab$$

$$update(D_C D_V D_P, st, a, l) := update(D_P, update(D_V, update(D_C, st, a, l), a, l), a, l)$$

wenn alle Bezeichner $D_C D_V D_P$ unterschiedlich sind.

$$update(\epsilon, st, a, l) := st$$

$$update(const I_1 := z_1, \dots, I_n := z_n; st, a, l) := st[I_1 \mapsto (const, z_1), \dots, I_n \mapsto (const, z_n)]$$

$$update(var I_1, \dots, I_n; st, a, l) := st[I_1 \mapsto (var, l, 1), \dots, I_n \mapsto (var, l, n)]$$

$$update(proc I_1; K_1 : \dots ; I_n; K_n; st, a, l) := st[I_1 \mapsto (proc, a_1, l, size(K_1)), \dots, I_n \mapsto (proc, a_n, l, size(K_n))]$$

mit frischen Adressen a_1, \dots, a_n und $size(D_C var I_1, \dots, I_n; D_P C) := n$

Anfangssymboltabelle Mit gegebenem Startzustand $s := (1, \epsilon, 0 : 0 : 0 : \underbrace{z_1 : \dots : z_n}_{I/Oframe}) \in S = PC \times DS \times PS$. Also hat

die zugehörige Anfangssymboltabelle n Einträge: $st_{I/O}(I_j) := (var, 0, j) \forall j \in [n]$

Übersetzung von Programmen Übersetzung von $in/out I_1, \dots, I_n; D C$: Erstens erzeugen von MAIN Frame zur Ausführung von C . Zweitens Beenden der Programmausführung nach Rücksprung (return).

$$trans : Pgm \rightarrow AM$$

$$trans(in/out I_1, \dots, I_n; K.) := 1 : CALL(a, 0, size(K));$$

$$2 : JMP(0);$$

$$kt(K, st_{I/O}, a, 1)$$

Übersetzung von Blöcken Übersetzung von $D C$: Erstens Aktualisieren der Symboltabelle nach D . Zweitens Erzeugen des Codes für die Prozedurdeklaration in D (mit aktualisierter Symboltabelle \rightarrow Rekursion). Drittens Code für C erzeugen (mit aktualisierter Symboltabelle).

$$kt : Block \times Tab \times PC \times Lev \rightarrow AM$$

$$kt(D C, st, a, l) := dt(D, update(D, st, a_1, l), l)$$

$$ct(C, update(D, st, a_1, l), a, l)$$

$$a' : RET;$$

Übersetzung von Deklarationen Übersetzung von D : Erzeuge Code für die in D deklarierten Prozeduren

$$dt : Decl \times Tab \times Lev \rightarrow AM$$

$$dt(D_C D_V D_P, st, l) := dt(D_P, st, l)$$

$$dt(\epsilon, st, l) := \epsilon$$

$$dt(proc I_1; K_1; \dots ; I_n; K_n; st, l) := kt(K_1, st, a_1, l + 1)$$

$$\vdots$$

$$kt(K_n, st, a_n, l + 1) \text{ mit } st(I_j) = (proc, a_j, \dots, \dots) \forall j \in [n]$$

Übersetzung von Kommandos

$$\begin{aligned} \text{ct} &: \text{Cmd} \times \text{Tab} \times \text{PC} \times \text{Lev} \rightarrow \text{AM} \\ \text{ct}(I := A, \text{st}, a, l) &:= \text{at}(A, \text{st}, a, l) \\ &\quad a' : \text{STORE}(l - \text{lev}, \text{loc}); \text{ if } \text{st}(I) = (\text{var}, \text{lev}, \text{off}) \\ \text{ct}(I(), \text{st}, a, l) &:= a : \text{CALL}(ca, l - \text{lev}, \text{loc}); \text{ if } \text{st}(I) = (\text{proc}, ca, \text{lev}, \text{loc}) \\ \text{ct}(C_1; C_2, \text{st}, a, l) &:= \text{ct}(C_1, \text{st}, a, l) \\ &\quad \text{ct}(C_2, \text{st}, a', l) \\ \text{ct}(\text{if } B \text{ then } C_1 \text{ else } C_2, \text{st}, a, l) &:= \text{bt}(B, \text{st}, a, l) \\ &\quad a' : \text{JFALSE}(a''); \\ &\quad \text{ct}(C_1, \text{st}, a' + 1, l) \\ &\quad a'' - 1 : \text{JMP}(a'''); \\ &\quad \text{ct}(C_2, \text{st}, a'', l) \\ &\quad a''' : \\ \text{ct}(\text{while } B \text{ do } C, \text{st}, a, l) &:= \text{bt}(B, \text{st}, a, l) \\ &\quad a' : \text{JFALSE}(a'' + 1); \\ &\quad \text{ct}(C, \text{st}, a' + 1, l) \\ &\quad a'' : \text{JMP}(a); \end{aligned}$$

Übersetzung von boolean Ausdrücken

$$\begin{aligned} \text{bt} &: \text{BExp} \times \text{Tab} \times \text{PC} \times \text{Lev} \rightarrow \text{AM} \\ \text{bt}(A_1 < A_2, \text{st}, a, l) &:= \text{at}(A_1, \text{st}, a, l) \\ &\quad \text{at}(A_2, \text{st}, a', l) \\ &\quad a'' : \text{LT}; \\ \text{bt}(\text{not } B, \text{st}, a, l) &:= \text{bt}(B, \text{st}, a, l) \\ &\quad a' : \text{NOT}; \\ \text{bt}(B_1 \text{ and } B_2, \text{st}, a, l) &:= \text{bt}(B_1, \text{st}, a, l) \\ &\quad \text{bt}(B_2, \text{st}, a', l) \\ &\quad a'' : \text{AND}; \\ \text{bt}(B_1 \text{ or } B_2, \text{st}, a, l) &:= \text{bt}(B_1, \text{st}, a, l) \\ &\quad \text{bt}(B_2, \text{st}, a', l) \\ &\quad a'' : \text{OR}; \end{aligned}$$

Übersetzung von arithmetischen Ausdrücken

$$\begin{aligned} \text{at} &: \text{AExp} \times \text{Tab} \times \text{PC} \times \text{Lev} \rightarrow \text{AM} \\ \text{at}(z, \text{st}, a, l) &:= a : \text{LIT}(z); \\ \text{at}(I, \text{st}, a, l) &:= \begin{cases} a : \text{LIT}(z); & \text{if } \text{st}(I) = (\text{const}, z) \\ a : \text{LOAD}(l - \text{lev}, \text{off}); & \text{if } \text{st}(I) = (\text{var}, \text{lev}, \text{off}) \end{cases} \\ \text{at}(A_1 + A_2, \text{st}, a, l) &:= \text{at}(A_1, \text{st}, a, l) \\ &\quad \text{at}(A_2, \text{st}, a', l) \\ &\quad a'' : \text{ADD}; \end{aligned}$$

5.4 Vorlesung 22

Fakultät (nur letzter Schritt)

Source code:

in/out x;

var y;

proc F;

if x > 1 then

 y := y * x;

 x := x - 1;

 F()

y := 1;

F();

x := y.

st' = [x ↦ (var, 0, 1),
 y ↦ (var, 1, 1)
 F ↦ (proc, a₁, a, 0)]

Intermediate code:
 (symbolische Adressen)

```

1 : CALL(a0, 0, 1);
2 : JMP(0);
a1 : LOAD(2, 1);
      LIT(1);
      GT;
a4 : JFALSE(a3);
      LOAD(1, 1);
      LOAD(2, 1);
      MULT;
      STORE(1, 1);
      LOAD(2, 1);
      LIT(1);
      SUB;
      STORE(2, 1);
      CALL(a1, 1, 0);
a3 : RET;
a0 : LIT(1);
      STORE(0, 1);
      CALL(a1, 0, 0);
      LOAD(0, 1);
      STORE(1, 1);
a2 : RET;
  
```

Intermediate code: (linearisiert)

a₀ = 17, a₁ = 3, a₂ = 22, a₃ = 16, a₄ = 6

```

1 : CALL(17, 0, 1);
2 : JMP(0);
3 : LOAD(2, 1);
4 : LIT(1);
5 : GT;
6 : JFALSE(16);
7 : LOAD(1, 1);
8 : LOAD(2, 1);
9 : MULT;
10 : STORE(1, 1);
11 : LOAD(2, 1);
12 : LIT(1);
13 : SUB;
14 : STORE(2, 1);
15 : CALL(3, 1, 0);
16 : RET;
17 : LIT(1);
18 : STORE(0, 1);
19 : CALL(3, 0, 0);
20 : LOAD(0, 1);
21 : STORE(1, 1);
22 : RET;
  
```

Berechnung für x = 2:

PC	DS	PS
1	ε	0 : 0 : 0 : 2
17	ε	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
18	1	4 : 3 : 2 : 0 : 0 : 0 : 0 : 2
19	ε	4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
3	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
4	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
5	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
6	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
7	ε	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
8	1	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
9	1 : 2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
10	2	3 : 2 : 20 : 4 : 3 : 2 : 1 : 0 : 0 : 0 : 2
11	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
12	2	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
13	2 : 1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
14	1	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
15	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
3	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
4	1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
5	1 : 1	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
6	0	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	6 : 2 : 16 : 3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
16	ε	3 : 2 : 20 : 4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
20	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
21	2	4 : 3 : 2 : 2 : 0 : 0 : 0 : 1
22	ε	4 : 3 : 2 : 2 : 0 : 0 : 0 : 2
2	ε	0 : 0 : 0 : 2
0	ε	0 : 0 : 0 : 2

5.5 Vorlesung 23

Boolische Ausdrücke mit sequenzieller Semantik

Bis jetzt immer strikte Semantik:

$$b_1 \diamond \perp = \perp$$

$$\perp \diamond b_2 = \perp$$

nun sequentielle Semantik:

$$\text{false} \wedge \perp = \text{false}$$

$$\text{true} \vee \perp = \text{true}$$

$$\perp \wedge b = \perp$$

Idee: Sprünge boolischen Ausdrücken vorziehen (jumping code) und bt und ct mit zwei zusätzlichen Adressparametern ausstatten: a_t : target address for true und a_f : target address for false

sbt-Funktion

$$\text{sbt} : BExp \times Tab \times PC^3 \times Lev \rightarrow AM$$

$$\text{abt}(A_1 < A_2, st, a, a_t, a_f, l) := \text{at}(A_1, st, a, l)$$

$$\text{at}(A_2, st, a', l)$$

$$a'' : LT;$$

$$a'' + 1 : \text{JFALSE}(a_f);$$

$$a'' + 2 : \text{JMP}(a_t);$$

$$\text{sbt}(\text{not } B, st, a, a_t, a_f, l) := \text{sbt}(B, st, a, a_f, a_t, l)$$

$$\text{sbt}(B_1 \text{ and } B_2, st, a, a_t, a_f, l) := \text{sbt}(B_1, st, a, a', a_f, l)$$

$$\text{sbt}(B_2, st, a', a_t, a_f, l)$$

$$\text{sbt}(B_1 \text{ or } B_2, st, a, a_t, a_f, l) := \text{sbt}(B_1, st, a, a_t, a', l)$$

$$\text{sbt}(B_2, st, a', a_t, a_f, l)$$

sct-Funktion

$$\text{sct} : Cmd \times Tab \times PC \times Lev \rightarrow AM$$

$$\text{sct}(\text{if } B \text{ then } C_1 \text{ else } C_2, st, a, l) := \text{sbt}(B, st, a, a_t, a_f, l)$$

$$\text{sct}(C_1, st, a_t, l)$$

$$a_f - 1 : \text{JMP}(a');$$

$$\text{ct}(C_2, st, a_f, l)$$

$$a' :$$

$$\text{sct}(\text{while } B \text{ do } C, st, a, l) := \text{bt}(B, st, a, a_t, a_f, l)$$

$$\text{sct}(C, st, a_t, l)$$

$$a_f - 1 : \text{JMP}(a);$$

$$a_f :$$

übrigen Fälle analog.

Übersetzung von while not $(x < 1)$ and $(x < y)$ do C

Sequential:

Strict:		
		1 :LOAD(x);
1 :LOAD(x);		LIT(1);
LIT(1);		LT;
LT;		JFALSE(6);
NOT;		JMP(a);
LOAD(x);	6 :LOAD(x);	
LOAD(y);	LOAD(y);	
LT;	LT;	
AND;	JFALSE(a);	
JFALSE(a);	JMP(11);	
ct(C,...)	11 :sct(C,...)	
JMP(1);	JMP(1);	
a :...	a :...	

Wenn $x = 0$ werden 9 Instruktionen ausgeführt.

Wenn $x = 0$ werden 5 Instruktionen ausgeführt.

⇒ Im Allgemeinen längerer Code aber kürzere Ausführung.

Erweiterte EPL Syntax

$Cmd : C ::= I := A | C_1; C_2 | \text{if } B \text{ then } C_1 \text{ else } C_2 |$
 $\text{while } B \text{ do } C | I(\underbrace{A_1, \dots, A_p; V_1, \dots, V_q}_{\text{aktueller Wert und ref. Parameter}})$

$Dcl : D ::= D_C D_V D_P$

$D_C ::= \epsilon | \text{const } I_1 := z_1, \dots, I_n := z_n;$

$D_V ::= \epsilon | \text{var } I_1, \dots, I_n;$

$D_P ::= \epsilon | \text{proc } I(\underbrace{I_1, \dots, I_p; \text{var } J_1, \dots, J_q}_{\text{formaler Wert und ref. Parameter}}); K : \text{proc } \dots$

mit $I_k, J_k, V_k \in Ide$

Prozedurdeklarationen formale Parameter unterschiedlich und disjunkt von lokalen Variablen; werden wie Variablen behandelt; entsprechende Benutzung in Prozedur

Prozeduraufruf formale Werte-Parameter implementiert wie lokale Variablen (neue Speicherstelle); nur Variablen als formale ref. Parameter erlaubt; implementiert als Zeiger auf zugehörige aktuelle Adresse

Erweiterte Abstract Machine (AM) für EPL

ERWEITERTE AM

Zustandsraum: $S := PC \times SP \times FP \times IR \times RS$ mit: Programmzähler $PC := \mathbb{N}$; Stackpointer $SP := \mathbb{N}$; Framepointer $FP := \mathbb{N}$; Indexregister $IR := \mathbb{N}$; Laufzeitstack $RS := (\mathbb{N} \rightarrow \mathbb{Z})$

Charakteristiken: näher an "realen" Maschinen (weniger mächtige Instruktionen); RS kombiniert DS und PS ; absolute Adressierung der Stackinträge; SP zeigt auf spitze (top) des Stacks; FP zeigt auf (dynamisches Linkfeld); des obersten Frames; IR implementiert dereferenzierung von statischen links

Framestruktur

	⋮	↓ increasing stack addresses
	par_1	
	⋮	actual value and ref. parameters
	par_r	
	sl	static link to declaration environment
	ra	return address
$FP \rightarrow$	dl	dynamic link to previous frame
	loc_1	
	⋮	local variables
$SP \rightarrow$	loc_k	
	⋮	

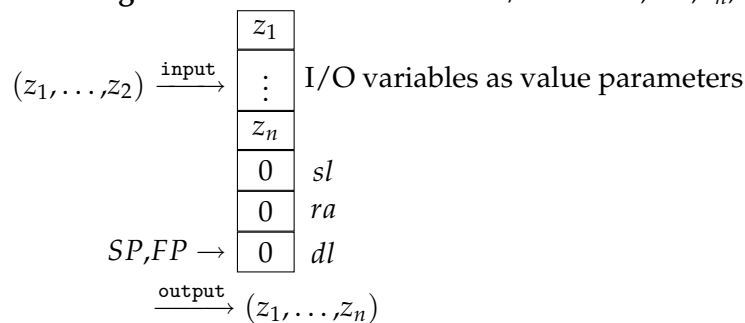
Fortlaufende Konstruktion:

- durch aufrufende Prozedur:
 1. Berechnung des aktuellen Parameters par_i
 2. Berechnung des statischen Links sl mittels IR
 3. Sprung zur aufgerufenen Prozedur und setzen von ra
- durch aufgerufene Prozedur:
 1. Speichern von Zeiger auf vorheriges Frame als dynamischer Link dl
 2. Speicher für lokale Variablen loc_j reservieren

Neue AM Instruktionen $CALL(ca, dif, loc), RET, LOAD(dif, off), STORE(dif, off)$ werden durch folgende Instruktionen mit den entsprechenden Semantiken: $\llbracket O \rrbracket : S \rightarrow S$ ersetzt:

$$\begin{aligned} \llbracket CALL\ ca \rrbracket (a, sp, fp, ir, p) &:= (ca, sp + 1, fp, ir, p[sp + 1 \mapsto a + 1]) \\ \llbracket RET\ k \rrbracket (a, sp, fp, ir, p) &:= (p(sp), sp - k - 1, fp, ir, p) \\ \llbracket PUSH\ z \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp + 1, fp, ir, p[sp + 1 \mapsto z]) \\ \llbracket PUSH\ FP \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp + 1, fp, ir, p[sp + 1 \mapsto fp]) \\ \llbracket PUSH\ \langle FP \rangle \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp + 1, fp, ir, p[sp + 1 \mapsto p(fp)]) \\ \llbracket PUSH\ \langle IR - 2 \rangle \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp + 1, fp, ir, p[sp + 1 \mapsto p(ir - 2)]) \\ \llbracket POP\ FP \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp - 1, p(sp), ir, p) \\ \llbracket POP\ \langle n \rangle \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp - 1, fp, ir, p[n \mapsto p(sp)]) \\ \llbracket ADD\ SP, n \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp + n, fp, ir, p) \\ \llbracket LOAD\ FP, SP \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp, sp, ir, p) \\ \llbracket LOAD\ SP, FP \rrbracket (a, sp, fp, ir, p) &:= (a + 1, fp, fp, ir, p) \\ \llbracket LOAD\ IR, \langle n \rangle \rrbracket (a, sp, fp, ir, p) &:= (a + 1, sp, fp, p(n), p) \end{aligned}$$

Ein-/Ausgabe verarbeiten Für $P = in/out\ I - 1, \dots, I_n; K. \in Pgm$



Ändern der Übersetzungsfunktion Ziel ist es $trans : Pgm \rightarrow AM$ zu modifizieren um Prozeduren mit Parametern und den modifizierten Instruktionen gerecht zu werden.

Modifizierung der Symboltabelle

$$\begin{aligned}
 Tab := & \{st \mid st : Ide \rightarrow (\{const\} \times \mathbb{Z}) \\
 & \cup \underbrace{(\{var\} \times Lev \times Off)}_{\text{lok. Var's und Wert Parameter}} \\
 & \cup (\{proc\} \times PC \times Lev \times Size) \\
 & \cup \underbrace{(\{rpar\} \times Lev \times Off)}_{\text{ref.Parameter}}\}
 \end{aligned}$$

Position von $FP \Rightarrow$ negativer Offset möglich $\Rightarrow Off := \mathbb{Z}$

Anfangssymboltabelle für $P = in/out I_1, \dots, I_n; K. \in Pgm$ $st_{I/O}(I_j) := (var, 0, j - n - 3) \forall j \in [n]$

update Funktion wie zuvor (Verarbeitung von Prozedurparametern durch dt)

Übersetzung von Programmen Übersetzung von $in/out I_1, \dots, I_n; D C.$: Erzeugen von Anfangsteil vom MAIN Frame zur Ausführung von C ; Beenden der Programmausführung nach Rücksprung

```

trans : Pgm → AM
trans(in/out I1, ..., In; K.) := 1 : PUSH FP; % create static link
                                2 : CALL a; % create return address
                                3 : JMP 0; % STOP
                                kt(K, stI/O, a, 1, 0)
                                                no. of parameters

```

Übersetzung von Blöcken

```

kt : Block × Tab × PC × Lev × ℕ → AM
kt(D C.st, l, r) := dt(D, update(D, st, a1, l), l)
                    a : PUSH FP; % entry code:
                    a + 1 : LOAD FP, SP; % create dynamic link and
                    a + 2 : ADD SP, size(D C); % allocate local variables
                            ct(C, update(D, st, a1, l), a + 3, l)
                    a' : LOAD SP, FP; % exit code:
                    a' + 1 : POP FP; % reset frame pointer and
                    a' + 2 : RET r + 1; % jump back

```

Übersetzen von Deklarationen

$$\begin{aligned} dt &: Dcl \times Tab \times Lev - \rightarrow AM \\ dt(D_C D_V D_P, st, l) &:= dt(D_P, st, l) \\ dt(\epsilon, st, l) &:= \epsilon \\ dt(\text{proc } I(I_1, \dots, I_p; \text{var } J_1, \dots, J_q); K; D'_P, st, l) &:= kt(K, st', a_1, l + 1, p + q) \\ &\quad dt(D'_P, st, l) \\ &\quad \text{mit } st(I) = (\text{proc}, a_I, \dots, \dots) \\ &\quad \text{und} \\ &\quad st' := st[I_1 \mapsto (\text{var}, l + 1, -p - q - 2), \\ &\quad \quad \vdots \\ &\quad \quad I_p \mapsto (\text{var}, l + 1, -q - 3), \\ &\quad \quad J_1 \mapsto (\text{rpar}, l + 1, -q - 2), \\ &\quad \quad \vdots \\ &\quad \quad J_q \mapsto (\text{rpar}, l + 1, -3)] \end{aligned}$$

5.6 Kapitel 24

Übersetzung von Kommandos Kommandos müssen nur für Zuweisungen und Prozeduraufrufe modifiziert werden (sonst ändert sich ct nicht).

$$ct : Cmd \times Tab \times PC \times Lev - \rightarrow AM$$

Übersetzungen von Prozeduraufrufen

$ct(I(A_1, \dots, A_p; V_1, \dots, V_q), st, a, l) :=$
 $at(A_1, st, a, l) \dots at(A_p, st, a, l) \text{ ref}(V_1, l) \dots \text{ref}(V_q, l) \text{ slink}(l) \text{ CALL } ca;$

mit:

$st(I) = (\text{proc}, ca, lev, loc)$

$\text{ref}(V, l) := \begin{cases} \text{PUSH FP} + off; & \text{if } st(V) = (\text{var}, lev, off) \text{ and } l = lev \\ \left. \begin{array}{l} \text{LOAD IR}, \langle \text{FP} - 2 \rangle; \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \\ \dots \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \end{array} \right\} k \text{ times} & \text{if } st(V) = (\text{var}, lev, off) \text{ and } l - lev = k + 1 > 0 \\ \text{PUSH IR} + off; \\ \text{PUSH } \langle \text{FP} + off \rangle; & \text{if } st(V) = (\text{rpar}, lev, off) \text{ and } l = lev \\ \left. \begin{array}{l} \text{LOAD IR}, \langle \text{FP} - 2 \rangle; \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \\ \dots \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \end{array} \right\} k \text{ times} & \text{if } st(V) = (\text{rpar}, lev, off) \text{ and } l - lev = k + 1 > 0 \\ \text{PUSH } \langle \text{IR} + off \rangle; \end{cases}$

$\text{slink}(l) := \begin{cases} \text{PUSH FP}; & \text{if } st(I) = (\text{proc}, ca, lev, loc) \text{ and } l = lev \\ \left. \begin{array}{l} \text{LOAD IR}, \langle \text{FP} - 2 \rangle; \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \\ \dots \\ \text{LOAD IR}, \langle \text{IR} - 2 \rangle; \end{array} \right\} k \text{ times} & \text{if } st(I) = (\text{proc}, ca, lev, loc) \text{ and } l - lev = k + 1 > 0 \\ \text{PUSH IR}; \end{cases}$

Übersetzungen arithmetischer Ausdrücke In $at : AExp \times Tab \times PC \times Lev \rightarrow AM$ muß nur die Behandlung von Bezeichner angepasst werden.

$$at(I, st, a, l) := \left\{ \begin{array}{ll}
 \text{PUSH } z; & \text{if } st(I) = (\text{const}, z) \\
 \text{PUSH } \langle \text{FP} + \text{off} \rangle; & \text{if } st(I) = (\text{var}, lev, \text{off}) \text{ and } l = lev \\
 \text{LOAD IR, } \langle \text{FP} - 2 \rangle; & \text{if } st(I) = (\text{var}, lev, \text{off}) \text{ and } l - lev = k + 1 > 0 \\
 \text{LOAD IR, } \langle \text{IR} - 2 \rangle; & \\
 \dots & \left. \vphantom{\begin{array}{l} \text{LOAD IR, } \langle \text{FP} - 2 \rangle; \\ \text{LOAD IR, } \langle \text{IR} - 2 \rangle; \\ \dots \end{array}} \right\} k \text{ times} \\
 \text{LOAD IR, } \langle \text{IR} - 2 \rangle; & \\
 \text{PUSH } \langle \text{IR} + \text{off} \rangle; & \\
 \text{LOAD IR, } \langle \text{FP} + \text{off} \rangle; & \text{if } st(I) = (\text{rpar}, lev, \text{off}) \text{ and } l = lev \\
 \text{PUSH } \langle \text{IR} \rangle; & \\
 \text{LOAD IR, } \langle \text{FP} - 2 \rangle; & \text{if } st(I) = (\text{rpar}, lev, \text{off}) \text{ and } l - lev = k + 1 > 0 \\
 \text{LOAD IR, } \langle \text{IR} - 2 \rangle; & \\
 \dots & \left. \vphantom{\begin{array}{l} \text{LOAD IR, } \langle \text{FP} - 2 \rangle; \\ \text{LOAD IR, } \langle \text{IR} - 2 \rangle; \\ \dots \end{array}} \right\} k \text{ times} \\
 \text{LOAD IR, } \langle \text{IR} - 2 \rangle; & \\
 \text{LOAD IR, } \langle \text{IR} + \text{off} \rangle; & \\
 \text{PUSH } \langle \text{IR} \rangle; &
 \end{array} \right.$$

Beispiel I

$P = \text{in/out } x, y;$
 $\left. \begin{array}{l} \text{proc } F(x; \text{var } y); \\ \quad \text{if } x > 1 \text{ then} \\ \quad \quad y := y * x; \\ \quad \quad F(x-1; y); \end{array} \right\} C_F \left. \vphantom{\begin{array}{l} \text{proc } F(x; \text{var } y); \\ \quad \text{if } x > 1 \text{ then} \\ \quad \quad y := y * x; \\ \quad \quad F(x-1; y); \end{array}} \right\} D \left. \vphantom{\begin{array}{l} \text{proc } F(x; \text{var } y); \\ \quad \text{if } x > 1 \text{ then} \\ \quad \quad y := y * x; \\ \quad \quad F(x-1; y); \\ \quad y := 1; \\ \quad F(x, y); \end{array}} \right\} K$
 $\left. \begin{array}{l} y := 1; \\ F(x, y); \end{array} \right\} C$

$\text{trans}(P) = 1 : \text{PUSH FP};$

2 : CALL a_0 ;

3 : JMP 0;

$\text{kt}(K, \text{st}_{I/0}, a_0, 1, 0)$

mit $\text{st}_{I/0} = [x \mapsto (\text{var}, 0, -4), y \mapsto (\text{var}, 0, -3)]$

$\text{kt}(K, \text{st}_{I/0}, a_0, 1, 0) = \text{dt}(D, \text{st}', 1)$

$a_0 : \text{PUSH FP};$

LOAD FP, SP;

ADD SP, $\underbrace{\text{size}(K)}_0$;

$\text{ct}(C, \text{st}', a_0 + 3, 1)$

LOAD SP, FP;

POP FP;

RET 1;

mit $\text{st}' = \text{update}(D, \text{st}_{I/0}, a_1, 1)$

$= \text{st}_{I/0}[F \mapsto (\text{proc}, a_1, 1, 0)]$

$\text{dt}(D, \text{st}', 1) = \text{kt}(C_P, \text{st}'', a_1, 2, 2)$

$a_1 : \text{PUSH FP};$

LOAD FP, SP;

ADD SP, 0;

$\text{ct}(C_F, \text{st}'', a_1 + 3, 2)$

LOAD SP, FP;

POP FP;

RET 3;

mit $\text{st}'' =$

$\text{st}'[x \mapsto (\text{var}, 2, -4), y \mapsto (\text{rpar}, 2, -3)]$

$\text{ct}(C_F, \text{st}'', a_1 + 3, 2)$

$= \text{bt}(x > 1, \text{st}'', a_1 + 3, 2)$

JFALSE a_2 ;

$\text{ct}(y := y * x; F(x-1; y), \text{st}'', a', 2)$

$a_2 :$

$\text{bt}(x > 1, \text{st}'', a_1 + 3, 2) = \text{PUSH } \langle \text{FP} - 4 \rangle;$

PUSH 1;

GT;

$\text{ct}(y := y * x; F(x-1; y), \text{st}'', a', 2) = \text{ct}(y := y * x, \text{st}'', a', 2)$

$\text{ct}(F(x-1; y), \text{st}'', a'', 2)$

$\text{ct}(y := y * x, \text{st}'', a', 2) = \text{at}(y * x, \text{st}'', a', 2)$

LOAD IR, $\langle \text{FP} - 3 \rangle;$

POP $\langle \text{IR} \rangle;$

$\text{at}(y * x, \text{st}'', a', 2) = \text{LOAD IR}, \langle \text{FP} - 3 \rangle;$

PUSH $\langle \text{IR} \rangle;$

PUSH $\langle \text{FP} - 4 \rangle;$

MULT;

$\text{ct}(F(x-1; y), \text{st}'', a'', 2) = \text{at}(x-1, \text{st}'', a'', 2)$

PUSH $\langle \text{FP} - 3 \rangle;$

LOAD IR, $\langle \text{FP} - 2 \rangle;$

PUSH IR;

CALL a_1 ;

$\text{at}(x-1, \text{st}'', a'', 2) = \text{PUSH } \langle \text{FP} - 4 \rangle;$

PUSH 1;

SUB;

$\text{ct}(C, \text{st}', a_0 + 3, 1) = \text{ct}(y := 1, \text{st}', a_0 + 3, 1)$

$\text{ct}(F(x; y), \text{st}', a', 1)$

$\text{ct}(y := 1, \text{st}', a_0 + 3, 1) = \text{PUSH } 1;$

LOAD IR, $\langle \text{FP} - 2 \rangle;$

POP $\langle \text{IR} - 3 \rangle;$

$\text{ct}(F(x; y), \text{st}', a', 1) = \text{at}(x, \text{st}', a', 1)$

LOAD IR, $\langle \text{FP} - 2 \rangle;$

PUSH IR - 3;

PUSH FP;

CALL a_1 ;

$\text{at}(x, \text{st}', a', 1) = \text{LOAD IR}, \langle \text{FP} - 2 \rangle;$

PUSH $\langle \text{IR} - 4 \rangle;$

Beispiel II

Für $x = 1, y = 0$ (bottom = $p(1)$, top = SP, FP, IR)

	PC	RS
1 : PUSH FP;	1	<u>1 : 0 : 0 : 0 : 0</u>
2 : CALL 27;	2	<u>1 : 0 : 0 : 0 : 0 : 5</u>
3 : JMP 0;	27	<u>1 : 0 : 0 : 0 : 0 : 5 : 3</u>
4 : PUSH FP; % entry F	28	<u>1 : 0 : 0 : 0 : 0 : 5 : 3 : 5</u>
5 : LOAD FP, SP;	29	<u>1 : 0 : 0 : 0 : 0 : 5 : 3 : 5</u>
6 : ADD SP, 0;	30	<u>1 : 0 : 0 : 0 : 0 : 5 : 3 : 5 : 1</u>
7 : PUSH <FP - 4>; % x>1	31	<u>1 : 0 : 0 : 0 : 0 : 5 : 3 : 5 : 1</u>
8 : PUSH 1;	32	<u>1 : 0 : 0 : 0 : 0 : 5 : 3 : 5</u>
9 : GT;	33	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5</u>
10 : JFALSE 24;	34	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5</u>
11 : LOAD IR, <FP - 3>; % y:=y*x	35	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1</u>
12 : PUSH <IR>;	36	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1</u>
13 : PUSH <FP - 4>;	37	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2</u>
14 : MULT;	38	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8</u>
15 : LOAD IR, <FP - 3>;	4	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39</u>
16 : POP <IR>;	5	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8</u>
17 : PUSH <FP - 4>; % F(x-1;y)	6	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8</u>
18 : PUSH 1;	7	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8</u>
19 : SUB;	8	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 1</u>
20 : PUSH <FP - 3>;	9	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 1 : 1</u>
21 : LOAD IR, <FP - 2>;	10	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8 : 0</u>
22 : PUSH IR;	24	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8</u>
23 : CALL 4;	25	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39 : 8</u>
24 : LOAD SP, FP; % exit F	26	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5 : 1 : 2 : 8 : 39</u>
25 : POP FP;	39	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5</u>
26 : RET 3;	40	<u>1 : 1 : 0 : 0 : 0 : 5 : 3 : 5</u>
27 : PUSH FP; % entry MAIN	41	<u>1 : 1 : 0 : 0 : 0 : 5 : 3</u>
28 : LOAD FP, SP;	3	<u>1 : 1 : 0 : 0 : 0</u>
29 : ADD SP, 0;	0	<u>1 : 1 : 0 : 0 : 0</u>
30 : PUSH 1; % y:=1		
31 : LOAD IR, <FP - 2>;		
32 : POP <IR - 3>;		
33 : LOAD IR, <FP - 2>; % F(x;y)		
34 : PUSH <IR - 4>;		
35 : LOAD IR, <FP - 2>;		
36 : PUSH IR - 3;		
37 : PUSH FP;		
38 : CALL 4;		
39 : LOAD SP, FP; % exit MAIN		
40 : POP FP;		
41 : RET 1;		

Alternative Implementierungen von statischen Links

- alternative impl. von statischer Link deref.: *display technique*
- display gibt direkt den Framepointer für gegebene Leveldifferenz zurück.
- meist als Array organisiert
- Mögliche Implementierungen:
 - Local Displays** aufgerufene Prozedur wartet Displayarray im Frame
 - Global Displays** Framepointer wird in globaler Static Link Array (SLA) gespeichert.

5.7 Kapitel 25

Sourcecode: Datenstrukturen = Arrays, Records, Listen, Bäume,...

Abstract machine: lineare Speicherstruktur, in den Zellen werden atomare Daten gespeichert

Übersetzung: Übersetzung des strukturierten Zustandsraums = Adressbereich

statische Datenstrukturen: Speicherbedarf bei der Compilierzeit bekannt

dynamische Datenstrukturen: Speicherbedarf ist Laufzeitabhängig

Definition: Modifizierte Syntax von EPL

z (* z ist ein Integer *)

$b ::= true|false$

r (* r ist ein Real *)

$Con : c ::= z|b|r$

$Ide : I$ (* I ist ein Identifier *)

$Type : T ::= bool|int|real|I|array[z_1 \dots z_2]ofT|record I_1 : T_1; \dots; I_n : T_n end$

$Var : V ::= I|V[E]|V.I$

$Exp : E ::= c|V|E_1 + E_2|E_1 < E_2|E_1 and E_2$

$Cmd : C ::= V := E|C_1; C_2|if E then C_1 else C_2|while E do C$

$Dcl : D ::= D_C D_T D_V$

$D_C ::= \epsilon|const I_1 := c_1, \dots, I_n := c_n;$

$D_T ::= \epsilon|type I_1 := T_1, \dots, I_n := T_n;$

$D_V ::= \epsilon|var I_1 : T_1; \dots, I_n : T_n;$

$Pgm : P ::= DC$

Statische Semantik

- Alle Bezeichner in einer Deklaration müssen unterschiedlich sein
- Typdefinitionen dürfen nicht rekursiv sein
- Alles, was in Deklarationen und Kommandos benutzt wird, muss vorher deklariert worden sein
- Variablen in Ausdrücken und Zuweisungen haben einen Basistypen (*bool, int, ...*)

- Arrayindizes müssen vom Typ *int* sein, Schleifen und Bedingungen müssen vom Typ *bool* sein, Typen auf linken und rechten Seiten von Zuweisungen müssen die selben sein
- Typcasting: *weak* = Implizit vom Compiler, mögliche Fehlerquelle und *strong* = Explizites Casting vom Programmierer

Neue AM Anweisungen

Prozedurinstruktionen werden nicht mehr benötigt, Transferfunktionen ($\text{LOAD}(dif, off)$, $\text{STORE}(dif, off)$) werden durch die folgenden ersetzt:

- $\text{LOAD}(a, d : n, \sigma) := (a + 1, d : \sigma(n), \sigma)$ if $n \in$
- $\text{STORE}(a, d : r : n, \sigma) := (a + 1, d, \sigma[n \rightarrow r])$ if $n \in$

Zusätzlich wird folgende Anweisung eingeführt um Arraygrenzen zu checken:

$$\text{CAB}(z_1, z_2)(a, d : z, \sigma) := \begin{cases} (a + 1, d : z, \sigma) & \text{if } z \in \{z_1, \dots, z_2\} \\ (0, d : \text{RTE}, \sigma) & \text{otherwise} \end{cases}$$

Modifizieren der Symboltabelle

$$\begin{aligned} \text{Tab} := \{st \mid st : \text{Ide} & \quad (\{\text{const}\} \times (\cup \cup)) \\ & \cup (\{\text{var}\} \times \text{Ide} \times) \\ & \cup (\{\text{type}\} \times \{\text{bool}, \text{int}, \text{real}\} \times \{1\}) \\ & \cup (\{\text{type}\} \times \{\text{array}\} \times {}^2 \times \text{Ide} \times) \\ & \cup (\{\text{type}\} \times \{\text{record}\} \times (\text{Ide}^2 \times)^* \times) \end{aligned}$$

- Variablenbezeichner (var, I, n): *type* I , Speicheradresse n
- Letzte Komponente vom *type*-Eintrag: Speicherbedarf
- Arraybezeichner: Grenzen z_1, z_2 , Typ I
- ...

Wie man die Symboltabelle erhält

Wieder durch die Funktion $\text{update}(D, st)$, die die Symboltabelle st bezüglich der Deklaration D updated.

Der Einfachheit halber nehmen wir an, dass $D = D_C D_T D_V \in Dcl$ abgeflacht ist. Das heißt, dass jeder Untertyp durch einen Identifier bestimmt wird:

Wenn $D_T = \text{type } I_1 := T_1; \dots; I_n := T_n;$, dann gilt für jedes $k \in [n]$:

$$\begin{aligned} T_k \in \{\text{bool}, \text{int}, \text{real}\} \quad \vee \quad T_k \in \{I_1, \dots, I_{k-1}\} \quad \vee \\ T_k = \text{array}[z_1 \dots z_2] \text{ of } I_j \text{ mit } j \in [k-1] \quad \vee \quad T_k = \text{record } J_1 : I_{j_1}; \dots; J_l : I_{j_l} \text{ end mit } j_1, \dots, j_l \in [k-1] \end{aligned}$$

Mit D_T von oben, D_V muss von der folgenden Form sein: $D_V = \text{var } J_1 : I_{j_1}; \dots; J_k : I_{j_k};$ mit $j_1, \dots, j_k \in [n]$

Modifizierte update-Funktion $update: Dcl \times - \rightarrow Tab$ wird definiert durch:

$$\begin{aligned}
update(D_C D_T D_V, st) &:= update(D_V, update(D_T, update(D_C, st))) \\
update(\epsilon, st) &:= st \\
update(const I_1 := c_1; \dots; I_n := c_n; st) &:= st[I_1 \rightarrow (const, c_1), \dots, I_n \rightarrow (const, c_n)] \\
update(type I := bool; D'_T, st) &:= update(type D'_T, st[I \rightarrow (type, bool, 1)]) \\
update(type I := int; D'_T, st) &:= update(type D'_T, st[I \rightarrow (type, int, 1)]) \\
update(type I := real; D'_T, st) &:= update(type D'_T, st[I \rightarrow (type, real, 1)]) \\
update(type I := J; D'_T, st) &:= update(type D'_T, st[I \rightarrow st(J)]) \\
update(type I := array[z_1 \dots z_2] of J; D'_T, st) &:= update(type D'_T, st[I \rightarrow (type, array, z_1, z_2, J, k \cdot n)]) \\
&\quad \text{if } st(J) = (type, \dots, n) \text{ and } k = z_2 - z_1 + 1 \\
update(type I := record I_1 : J_1; \dots; I_l : J_l \text{ end}; D'_T, st) &:= update(type D'_T, st[I \rightarrow (type, record, I_1, J_1, 0, I_2, J_2, n_1, \dots, I_l, J_l, \\
&\quad \sum_{i=1}^{l-1} n_i, \sum_{i=1}^l n_i)]) \text{ if } st(J_i) = (type, \dots, n_i) \text{ for } i \in [l] \\
update(type, st) &:= st \\
update(var I_1 : J_1; \dots; I_n : J_n; st) &:= st[I_1 \rightarrow (var, J_1, 0), I_2 \rightarrow (var, J_2, n_1), \dots, I_n \rightarrow (var, J_n, \sum_{i=1}^{n-1} n_i)] \\
&\quad \text{if } st(J_i) = (type, \dots, n_i) \text{ for } i \in [l]
\end{aligned}$$

Beispiel Sei $D := type, Bool=bool; Int=int; Array=array[1..20] of Bool; Record=record S:Array; T:Int end;$
 $var x:Int; y:Array; z:Record$
Dann

$$\begin{aligned}
update(D, st) &= st[Bool \rightarrow (type, bool, 1), & (1) \\
&\quad Int \rightarrow (type, int, 1), & (2) \\
&\quad Array \rightarrow (type, array, 1, 20, Bool, 20), & (3) \\
&\quad Record \rightarrow (type, record, S, Array, 0, T, Int, 20, 21), & (4) \\
&\quad x \rightarrow (var, Int, 0), & (5) \\
&\quad y \rightarrow (var, Array, 1), & (6) \\
&\quad z \rightarrow (var, Record, 21)] & (7)
\end{aligned}$$