Script generated by TTT

Title: Simon: Compilerbau (30.05.2012)

Date: Wed May 30 14:05:28 CEST 2012

Duration: 77:46 min

Pages: 67

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Compilerbau I

Dr. Michael Petter, Dr. Axel Simon

SoSe 2012

Themengebiet:

Die semantische Analyse

Semantische Analyse

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn

Compiler:

- Syntax
- Semantik ←
- Pragmatik

Semantische Analyse

Compiler:

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn

Syntax

- Semantik ←
- Pragmatik

Semantische Analyse

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn
- der Compiler kann einige dieser sinnlosen Programme erkennen
  - diese Programme werden als fehlerhaft zurückgewiesen
  - Sprachdefinition definiert was fehlerhaft bedeutet

Semantische Analyse

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn
- der Compiler kann einige dieser sinnlosen Programme erkennen
  - diese Programme werden als fehlerhaft zurückgewiesen
  - Sprachdefinition definiert was fehlerhaft bedeutet
- semantische Analysen sind hierzu erforderlich, die
  - Bezeichner eindeutig machen;
  - die Typen von Variablen ermitteln;

Semantische Analyse

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn
- der Compiler kann einige dieser sinnlosen Programme erkennen
  - diese Programme werden als fehlerhaft zurückgewiesen
  - Sprachdefinition definiert was fehlerhaft bedeutet
- semantische Analysen sind hierzu erforderlich, die
  - Bezeichner eindeutig machen;
  - die Typen von Variablen ermitteln;
- semantische Analysen dienen auch dazu
  - Möglichkeiten zur Programm-Optimierung zu finden;
  - über möglicherweise fehlerhafte Programme zu warnen

Semantische Analyse

Scanner und Parser akzeptieren Programme mit korrekter Syntax.

- nicht alle lexikalisch und syntaktisch korrekten Programme machen Sinn
- der Compiler kann einige dieser sinnlosen Programme erkennen
  - diese Programme werden als fehlerhaft zurückgewiesen
  - Sprachdefinition definiert was fehlerhaft bedeutet
- semantische Analysen sind hierzu erforderlich, die
  - Bezeichner eindeutig machen;
  - die Typen von Variablen ermitteln;
- semantische Analysen dienen auch dazu
  - Möglichkeiten zur Programm-Optimierung zu finden;
  - über möglicherweise fehlerhafte Programme zu warnen

$\rightsquigarrow$ semantische Analysen attributieren den Syntaxbaum

3 / 184

Kapitel 1:
Attributierte Grammatiken

Attributierte Grammatiken

- viele Berechnungen der semantischen Analyse als auch der Code-Generierung arbeiten auf dem Syntaxbaum.
- was an einem (i.d.R. Nicht-Terminal) Knoten berechnet wird, hängt nur von dem *Typ* des Knotens ab
- eine *lokale* Berechnung
  - greift auf bereits berechnete Informationen von Nachbarknoten zu und
  - berechnen daraus neue Informationen für den lokalen Knoten und andere Nachbarknoten

Attributierte Grammatiken

- viele Berechnungen der semantischen Analyse als auch der Code-Generierung arbeiten auf dem Syntaxbaum.
- was an einem (i.d.R. Nicht-Terminal) Knoten berechnet wird, hängt nur von dem *Typ* des Knotens ab
- eine *lokale* Berechnung
  - greift auf bereits berechnete Informationen von Nachbarknoten zu und
  - berechnen daraus neue Informationen für den lokalen Knoten und andere Nachbarknoten

Definition

Eine attributierte Grammatik ist eine CFG erweitert um:

- Attribute für jedes Nichtterminal;
- lokale Attribut-Gleichungen.

Attributierte Grammatiken

- viele Berechnungen der semantischen Analyse als auch der Code-Generierung arbeiten auf dem Syntaxbaum.
- was an einem (i.d.R. Nicht-Terminal) Knoten berechnet wird, hängt nur von dem *Typ* des Knotens ab
- eine *lokale* Berechnung
  - greift auf bereits berechnete Informationen von Nachbarknoten zu und
  - berechnen daraus neue Informationen für den lokalen Knoten und andere Nachbarknoten

Definition

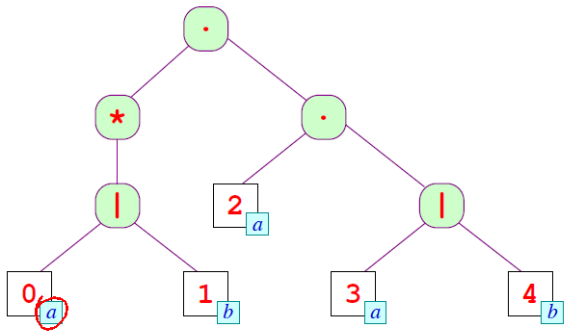
Eine attributierte Grammatik ist eine CFG erweitert um:

- Attribute für jedes Nichtterminal;
- lokale Attribut-Gleichungen.

- damit die Eingabe-Werte eines Knotens bei der Berechnung bereits vorliegen, müssen die Knoten des Syntaxbaums in einer bestimmten *Reihenfolge* durchlaufen werden

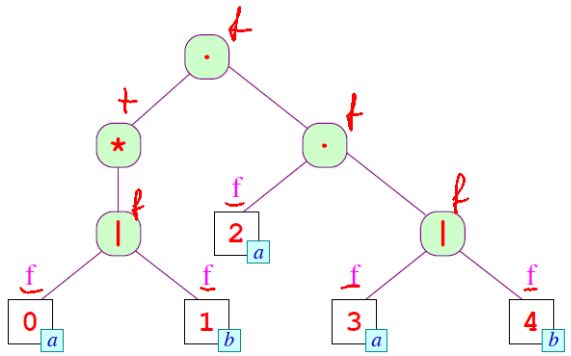
Beispiel: Berechnung des Prädikats empty[r]

Betrachte den Syntaxbaum des regulären Ausdrucks (a|b)*a(a|b):



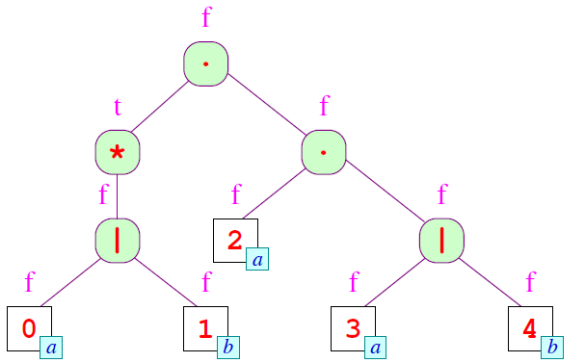
Beispiel: Berechnung des Prädikats empty[r]

Betrachte den Syntaxbaum des regulären Ausdrucks (a|b)*a(a|b):



Beispiel: Berechnung des Prädikats empty$[r]$

Betrachte den Syntaxbaum des regulären Ausdrucks (a|b)*a(a|b):



$\rightsquigarrow$ Werte von empty$[r]$ werden von unten nach oben berechnet.

Idee zur Implementierung

- für jeden Knoten führen wir ein Attribut empty ein.
- die Attribute werden in einer *depth-first* Traversierung berechnet:
  - an einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln
  - das Attribut an einem inneren Knoten hängt nur von den Attributen der Nachfolger ab
- das empty ist ein *synthetisches* Attribut
- kann durch pre- oder post-order Traversierung berechnet werden

Idee zur Implementierung

- für jeden Knoten führen wir ein Attribut empty ein.
- die Attribute werden in einer *depth-first* Traversierung berechnet:
  - an einem Blatt lässt sich der Wert des Attributs unmittelbar ermitteln
  - das Attribut an einem inneren Knoten hängt nur von den Attributen der Nachfolger ab
- das empty ist ein *synthetisches* Attribut
- kann durch pre- oder post-order Traversierung berechnet werden

Im Allgemeinen:

Definition

Ein Attribut ist

- synthetisch: Wert wird von den Blättern zur Wurzel propagiert
- inherit: Wert wird von der Wurzel zu den Blättern propagiert

Berechnungsregeln für empty

Wie das Attribut *lokal* zu berechnen ist, ergibt sich aus dem *Typ* des Knotens:

für Blätter: $r \equiv \boxed{i \mid x}$ definieren wir $\text{empty}[r] = (x \equiv \epsilon)$.

andernfalls:

$$\begin{aligned}
\text{empty}[r_1 \mid r_2] &= \text{empty}[r_1] \vee \text{empty}[r_2] \\
\text{empty}[r_1 \cdot r_2] &= \text{empty}[r_1] \wedge \text{empty}[r_2] \\
\text{empty}[r_1^*] &= t \\
\text{empty}[r_1?] &= t
\end{aligned}$$

Spezifizierung von allgemeinen Attributsystemen

Das empty Attributs ist rein *synthetisch*, daher kann es durch einfache *strukturelle Induktion* definiert werden.

- wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über allgemeine Attribute an einem Knoten und seinen Nachfolgern reden können.
- der Einfachheit halber benutzen wir fortlaufende Indizes:

$\text{empty}[0]$: das Attribut des aktuellen Knotens
$\text{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

Spezifizierung von allgemeinen Attributsystemen

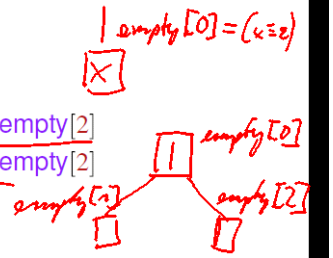
Das empty Attributs ist rein *synthetisch*, daher kann es durch einfache *strukturelle Induktion* definiert werden.

- wir benötigen einen einfachen und flexiblen Mechanismus, mit dem wir über allgemeine Attribute an einem Knoten und seinen Nachfolgern reden können.
- der Einfachheit halber benutzen wir fortlaufende Indizes:

$\text{empty}[0]$: das Attribut des aktuellen Knotens
$\text{empty}[i]$: das Attribut des i-ten Sohns $(i > 0)$

... im Beispiel:

x	:	$\text{empty}[0]$	:=	$(x \equiv \epsilon)$
$\mid$	:	$\text{empty}[0]$	:=	$\text{empty}[1] \vee \text{empty}[2]$
$\cdot$	:	$\text{empty}[0]$	:=	$\text{empty}[1] \wedge \text{empty}[2]$
$*$	:	$\text{empty}[0]$	:=	t
$?$	:	$\text{empty}[0]$	:=	t



9 / 184

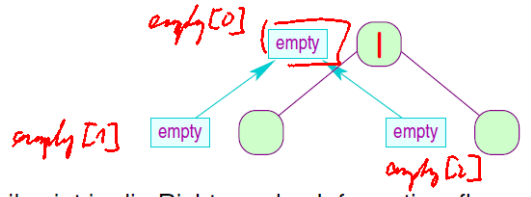
Beobachtungen:

- Die lokalen Berechnungen der Attributwerte müssen von einem *globalen* Algorithmus zusammen gesetzt werden
- Dazu benötigen wir:
  1. eine Besuchsreihenfolge der Knoten des Baums;
  2. eine lokale Berechnungsreihenfolgen
- Die Auswertungsstrategie muss mit den *Attribut-Abhängigkeiten* kompatibel sein

Beobachtungen:

- Die lokalen Berechnungen der Attributwerte müssen von einem *globalen* Algorithmus zusammen gesetzt werden
- Dazu benötigen wir:
  1. eine Besuchsreihenfolge der Knoten des Baums;
  2. eine lokale Berechnungsreihenfolgen
- Die Auswertungsstrategie muss mit den *Attribut-Abhängigkeiten* kompatibel sein

Wir geben Abhängigkeiten als gerichtete Kanten an:



$\rightsquigarrow$ Pfeil zeigt in die Richtung des Informationsflusses

10 / 184

Beobachtung:

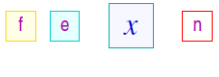
- Zur Ermittlung einer Auswertungsstrategie reicht es nicht, sich die *lokalen* Attribut-Abhängigkeiten anzusehen.
- Es kommt auch darauf an, wie sie sich *global* zu einem Abhängigkeitsgraphen zusammen setzen.
- Im Beispiel sind die Abhängigkeiten stets von den Attributen der Söhne zu den Attributen des Vaters gerichtet.
  $\rightsquigarrow$ post-order depth-first Traversierung möglich
- Die Variablen-Abhängigkeiten können aber auch *komplizierter* sein.

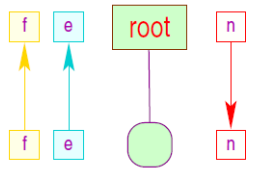
Simultane Berechnung von mehreren Attributen

Berechne empty, first, next von regulären Ausdrücken:

x :
$\quad \text{empty}[0] := (x \equiv \epsilon)$
$\quad \text{first}[0] := \{x \mid x \neq \epsilon\}$
$\quad$ // (keine Gleichung für next)

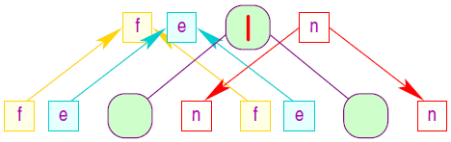
root: :
$\quad \text{empty}[0] := \text{empty}[1]$
$\quad \text{first}[0] := \text{first}[1]$
$\quad \text{next}[0] := \emptyset$
$\quad \text{next}[1] := \text{next}[0]$





RE Auswertung: Regeln für Alternative

| :
$\quad \text{empty}[0] :=$
$\quad \text{first}[0] :=$
$\quad \text{next}[1] :=$
$\quad \text{next}[2] :=$



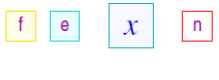
Simultane Berechnung von mehreren Attributen

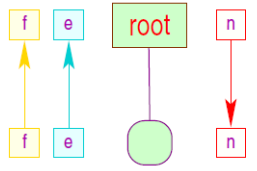
Berechne empty, first, next von regulären Ausdrücken:

x :
$\quad \text{empty}[0] := (x \equiv \epsilon)$
$\quad \text{first}[0] := \{x \mid x \neq \epsilon\}$
$\quad$ // (keine Gleichung für next)

root: :
$\quad \text{empty}[0] := \text{empty}[1]$
$\quad \text{first}[0] := \text{first}[1]$
$\quad \text{next}[0] := \emptyset$
$\quad \text{next}[1] := \text{next}[0]$

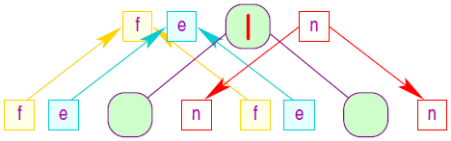
ROOT → S
S → &IS
| S·S
| S*
| a | b | c





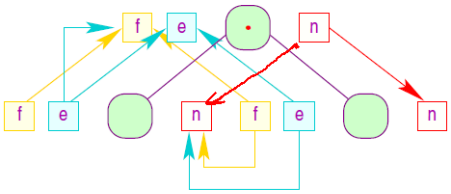
RE Auswertung: Regeln für Alternative

$$
\begin{array}{lll}
| & : & \text{empty}[0] := \text{empty}[1] \vee \text{empty}[2] \\
 & & \text{first}[0] := \text{first}[1] \cup (\text{empty}[2] \,?\, \text{first}[2] : \emptyset) \\
 & & \text{next}[1] := \text{next}[0] \\
 & & \text{next}[2] := \text{next}[0]
\end{array}
$$



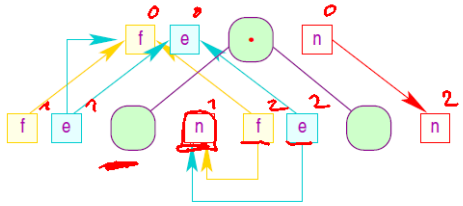
RE Auswertung: Regeln für Konkatenation

$$
\begin{array}{lll}
\cdot & : & \text{empty}[0] := \text{empty}[1] \wedge \text{empty}[2] \\
 & & \text{first}[0] := \text{first}[1] \cup (\text{empty}[1] \,?\, \text{first}[2] : \emptyset) \\
 & & \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset) \\
 & & \text{next}[2] := \text{next}[0]
\end{array}
$$



RE Auswertung: Regeln für Konkatenation

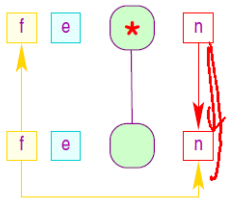
$$
\begin{array}{lll}
\cdot & : & \text{empty}[0] := \text{empty}[1] \wedge \text{empty}[2] \\
 & & \text{first}[0] := \text{first}[1] \cup (\text{empty}[1] \,?\, \text{first}[2] : \emptyset) \\
 & & \text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset) \\
 & & \text{next}[2] := \text{next}[0]
\end{array}
$$

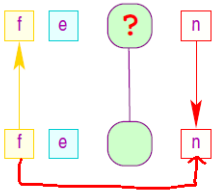


RE Auswertung: Regeln für Kleene-Stern und '?'

$$
\begin{array}{lll}
* & : & \text{empty}[0] := \text{empty}[1] \vee \text{empty}[2] \\
 & & \text{first}[0] := \text{first}[1] \cup \text{next}[1] \\
 & & \text{next}[1] := \text{next}[0]
\end{array}
$$

$$
\begin{array}{lll}
? & : & \text{empty}[0] := \text{true} \\
 & & \text{first}[0] := \text{first}[1] \cup \text{next}[1] \\
 & & \text{next}[1] := \text{next}[0]
\end{array}
$$

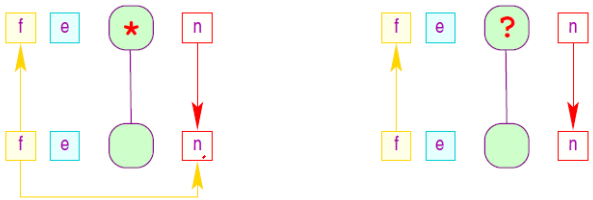




RE Auswertung: Regeln für Kleene-Stern und '?'

$*$:
$$\begin{aligned} \text{empty}[0] &:= t \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[1] &:= \text{first}[1] \cup \text{next}[0] \end{aligned}$$

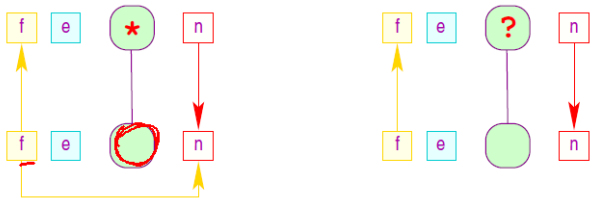
$?$:
$$\begin{aligned} \text{empty}[0] &:= t \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[1] &:= \text{next}[0] \end{aligned}$$



RE Auswertung: Regeln für Kleene-Stern und '?'

$*$:
$$\begin{aligned} \text{empty}[0] &:= t \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[1] &:= \text{first}[1] \cup \text{next}[0] \end{aligned}$$

$?$:
$$\begin{aligned} \text{empty}[0] &:= t \\ \text{first}[0] &:= \text{first}[1] \\ \text{next}[1] &:= \text{next}[0] \end{aligned}$$

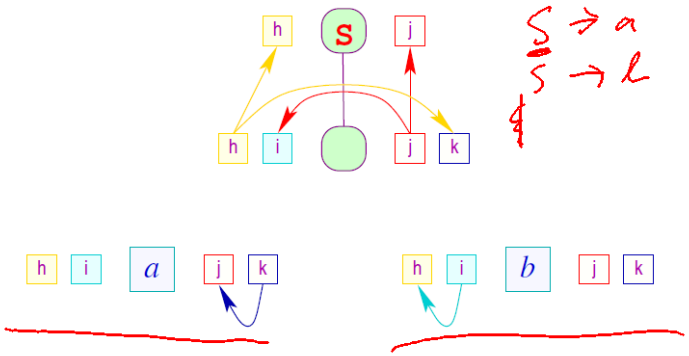


15 / 184

Problem bei allgemeinen Attributen

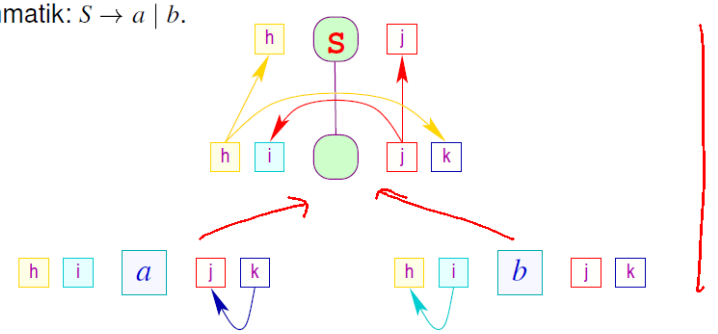
- Eine Auswertungsstrategie kann es nur dann geben, wenn die Variablen-Abhängigkeiten in jedem attributierten Baum azyklisch sind
- Es ist DEXPTIME-vollständig, herauszufinden, ob keine zyklischen Variablenabhängigkeiten vorkommen können [Jazayeri, Odgen, Rounds, 1975]

Beispiel: Grammatik $S \to a \mid b$; Attributenabhängigkeiten:

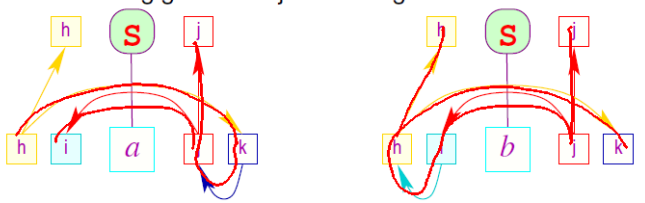


Berechnung der Abhängigkeiten

Grammatik: $S \to a \mid b$.

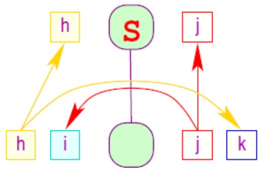


Betrachte Abhängigkeiten in jedem möglichen Baum:

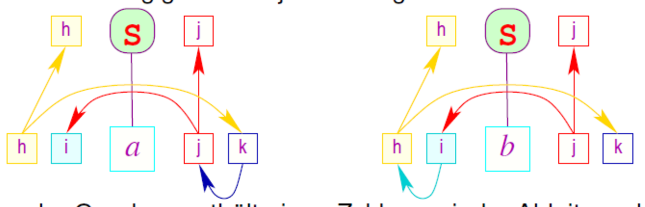


Berechnung der Abhängigkeiten

Grammatik: $S \to a \mid b$.



Betrachte Abhängigkeiten in jedem möglichen Baum:



Keiner der Graphen enthält einen Zyklus $\leadsto$ jeder Ableitungsbaum kann berechnet werden.

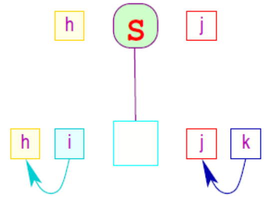
Stark azyklische Attributierung

Ziel: Berechne *hinreichende* Bedingung, dass Attributierung azyklisch ist.

Idee: Berechne *einen* Abhängigkeitsgraphen für jedes Symbol N.

- Wir starten mit der lokalen Ordung $\sqsubseteq_N \;=\; \to$
- Betrachte jede Produktion $N \to S_1 \cdots S_n$ von N
- Für jeden möglichen Sohn-Knoten S_i an i-ter Stelle
  - finde Abhängigkeiten zwischen $a[0] \ldots z[0]$ von S_i
  - füge diese in $\sqsubseteq_N$ ein zwischen $a[i] \ldots z[i]$
- Hat $\sqsubseteq_N$ einen Zykel, brechen wir erfolglos ab.
- Lässt sich $\sqsubseteq_N$ für kein N mehr vergrößern, hören wir auf.

Im Beispiel:



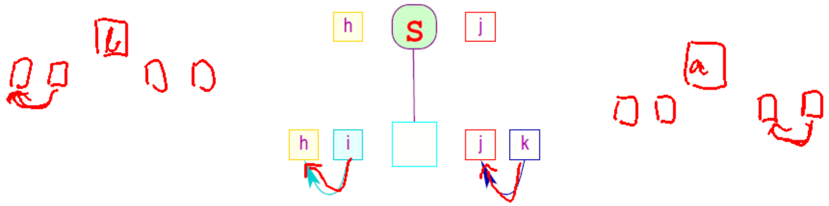
Stark azyklische Attributierung

Ziel: Berechne *hinreichende* Bedingung, dass Attributierung azyklisch ist.

Idee: Berechne *einen* Abhängigkeitsgraphen für jedes Symbol N.

- Wir starten mit der lokalen Ordung $\sqsubseteq_N \;=\; \to$
- Betrachte jede Produktion $N \to S_1 \cdots S_n$ von N
- Für jeden möglichen Sohn-Knoten S_i an i-ter Stelle
  - finde Abhängigkeiten zwischen $a[0] \ldots z[0]$ von S_i
  - füge diese in $\sqsubseteq_N$ ein zwischen $a[i] \ldots z[i]$
- Hat $\sqsubseteq_N$ einen Zykel, brechen wir erfolglos ab.
- Lässt sich $\sqsubseteq_N$ für kein N mehr vergrößern, hören wir auf.

Im Beispiel:



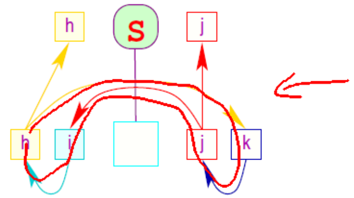
Stark azyklische Attributierung

Ziel: Berechne *hinreichende* Bedingung, dass Attributierung azyklisch ist.

Idee: Berechne *einen* Abhängigkeitsgraphen für jedes Symbol N.

- Wir starten mit der lokalen Ordung $\sqsubseteq_N \;=\; \to$
- Betrachte jede Produktion $N \to S_1 \cdots S_n$ von N
- Für jeden möglichen Sohn-Knoten S_i an i-ter Stelle
  - finde Abhängigkeiten zwischen $a[0] \ldots z[0]$ von S_i
  - füge diese in $\sqsubseteq_N$ ein zwischen $a[i] \ldots z[i]$
- Hat $\sqsubseteq_N$ einen Zykel, brechen wir erfolglos ab.
- Lässt sich $\sqsubseteq_N$ für kein N mehr vergrößern, hören wir auf.

Im Beispiel:



Von den Abhängigkeiten zur Auswertungsstrategie

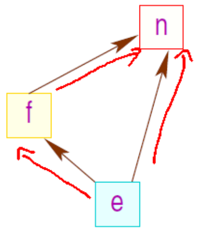
Mögliche Strategien:

1. Der Benutzer soll die Strategie spezifizieren

Von den Abhängigkeiten zur Auswertungsstrategie

Mögliche Strategien:

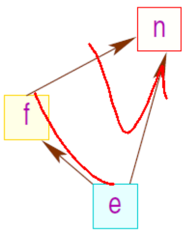
1. Der Benutzer soll die Strategie spezifizieren
2. Berechne eine Strategie anhand der Abhängigkeiten
   - Berechne eine linear Ordnung aus der partielle Ordnung $\rightarrow$ bzw. $\sqsubseteq$.
   - Werte die Attribute anhand dieser linearen Ordnung aus.
   - Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie.



Von den Abhängigkeiten zur Auswertungsstrategie

Mögliche Strategien:

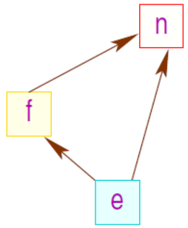
1. Der Benutzer soll die Strategie spezifizieren
2. Berechne eine Strategie anhand der Abhängigkeiten
   - Berechne eine linear Ordnung aus der partielle Ordnung $\rightarrow$ bzw. $\sqsubseteq$.
   - Werte die Attribute anhand dieser linearen Ordnung aus.
   - Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie.



Von den Abhängigkeiten zur Auswertungsstrategie

Mögliche Strategien:

1. Der Benutzer soll die Strategie spezifizieren
2. Berechne eine Strategie anhand der Abhängigkeiten
   - Berechne eine linear Ordnung aus der partielle Ordnung $\rightarrow$ bzw. $\sqsubseteq$.
   - Werte die Attribute anhand dieser linearen Ordnung aus.
   - Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie.



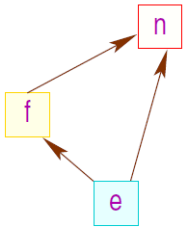
3. Betrachte fixe Strategie und erlaube nur entsprechende Attribute

Frage: Wie berechnet man eine sinnvolle Linearisierung?

Von den Abhängigkeiten zur Auswertungsstrategie

Mögliche Strategien:

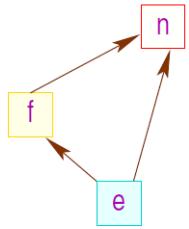
1. Der Benutzer soll die Strategie spezifizieren
2. Berechne eine Strategie anhand der Abhängigkeiten
   - Berechne eine linear Ordnung aus der partielle Ordnung $\rightarrow$ bzw. $\sqsubseteq$.
   - Werte die Attribute anhand dieser linearen Ordnung aus.
   - Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie.



Von den Abhängigkeiten zur Auswertungsstrategie

Mögliche Strategien:

1. Der Benutzer soll die Strategie spezifizieren
2. Berechne eine Strategie anhand der Abhängigkeiten
   - Berechne eine linear Ordnung aus der partielle Ordnung $\rightarrow$ bzw. $\sqsubseteq$.
   - Werte die Attribute anhand dieser linearen Ordnung aus.
   - Die lokalen Abhängigkeitsgraphen zusammen mit der linearen Ordnung erlauben die Berechnung einer Strategie.



3. Betrachte fixe Strategie und erlaube nur entsprechende Attribute

Frage: Wie berechnet man eine sinnvolle Linearisierung?

19 / 184

Linearisierung der Abhängigkeitsordnung

Mögliche automatische Strategien:

1. Bedarfsgetriebene Auswertung
   - Beginne mit der Berechnung eines Attributs.
   - Sind die Argument-Attribute noch nicht berechnet, berechne rekursiv deren Werte
   - Besuche die Knoten des Baum *nach Bedarf*

Linearisierung der Abhängigkeitsordnung

Mögliche automatische Strategien:

1. Bedarfsgetriebene Auswertung
   - Beginne mit der Berechnung eines Attributs.
   - Sind die Argument-Attribute noch nicht berechnet, berechne rekursiv deren Werte
   - Besuche die Knoten des Baum *nach Bedarf*

2. Auswertung in Pässen:
   - *Minimiere* die Anzahl der *Besuche* an jedem Knoten.
   - Organisiere die Auswertung in Durchläufen durch den Baum.
   - Berechne für jeden Durchlauf eine Besuchsstrategie für die Knoten zusammen mit einer lokalen Strategie für jeden Knoten-Typ

Betrachte Beispiel für bedarfsgetriebene Auswertung

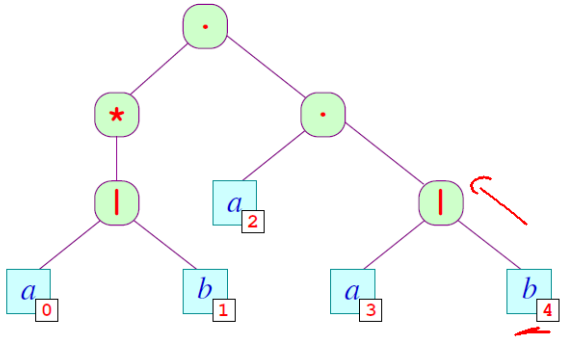
20 / 184

Beispiel bedarfsgetriebene Auswertung

Berechne next der Blätter von $((a|b)^*a(a|b))$:

$|$: $\text{next}[1] := \text{next}[0]$
$\text{next}[2] := \text{next}[0]$

$\cdot$: $\text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset)$
$\text{next}[2] := \text{next}[0]$

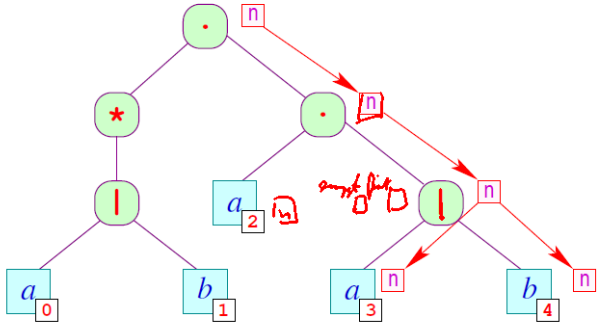


Beispiel bedarfsgetriebene Auswertung

Berechne next der Blätter von $((a|b)^*a(a|b))$:

$|$: $\text{next}[1] := \text{next}[0]$
$\text{next}[2] := \text{next}[0]$

$\cdot$: $\text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset)$
$\text{next}[2] := \text{next}[0]$

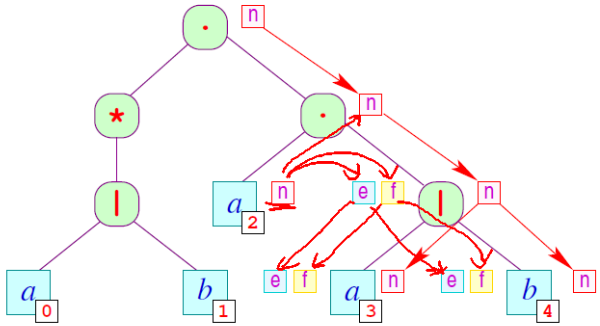


Beispiel bedarfsgetriebene Auswertung

Berechne next der Blätter von $((a|b)^*a(a|b))$:

$|$: $\text{next}[1] := \text{next}[0]$
$\text{next}[2] := \text{next}[0]$

$\cdot$: $\text{next}[1] := \text{first}[2] \cup (\text{empty}[2] \,?\, \text{next}[0] : \emptyset)$
$\text{next}[2] := \text{next}[0]$



Bedarfsgetriebene Auswertung

Diskussion:

- Die Reihenfolge hängt i.a. vom zu attributierenden Baum ab.
- Der Algorithmus muss sich merken, welche Attribute er bereits berechnete
- Der Algorithmus besucht manche Knoten unnötig oft.
- Der Algorithmus ist nicht-lokal

Bedarfsgetriebene Auswertung

Diskussion:

- Die Reihenfolge hängt i.a. vom zu attributierenden Baum ab.
- Der Algorithmus muss sich merken, welche Attribute er bereits berechnete
- Der Algorithmus besucht manche Knoten unnötig oft.
- Der Algorithmus ist nicht-lokal

Ansatz nur im Prinzip günstig:

- Berechnungsstrategie ist dynamisch: Fehlersuche schwierig
- Berechnung aller Attribute ist oft billiger
- Meistens werden alle Attribute in allen Knoten benötigt

Diskussion

Dann gilt:

- in jedem Durchlauf wird mindestens ein Attribut in einer stark azyklischen Attributierung berechnet
- man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt
- hat man einen Baum-Durchlauf zur Berechnung eines Attributes, kann man überprüfen, ob er geeignet ist, gleichzeitig weitere Attribute auszuwerten $\leadsto$ Optimierungsproblem

Diskussion

Dann gilt:

- in jedem Durchlauf wird mindestens ein Attribut in einer stark azyklischen Attributierung berechnet
- man braucht folglich für stark azyklische Attributierungen maximal so viele Pässe, wie es Attribute gibt
- hat man einen Baum-Durchlauf zur Berechnung eines Attributes, kann man überprüfen, ob er geeignet ist, gleichzeitig weitere Attribute auszuwerten $\leadsto$ Optimierungsproblem

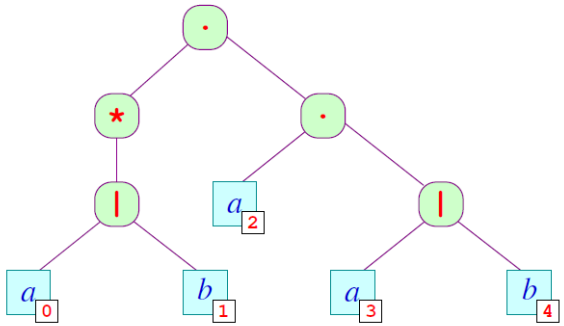
... im Beispiel:

- empty und first lassen sich gemeinsam berechnen.
- next muss in einem weiteren Pass berechnet werden

$\leadsto$ lasse den Benutzer festlegen, wie Attribute ausgewertet werden

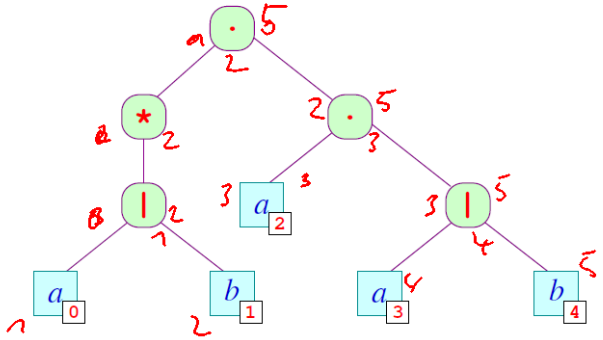
Implementierung der lokalen Auswertung: Pre-Order vs. Post-Order

Betrachte Beispiel: Nummerierung der Blätter eines Baums:



Implementierung der lokalen Auswertung: Pre-Order vs. Post-Order

Betrachte Beispiel: Nummerierung der Blätter eines Baums:



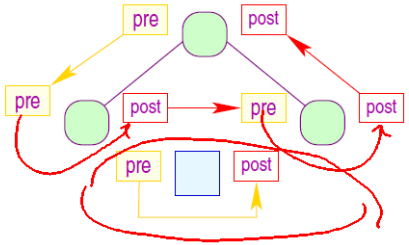
Praktische Implementierung der Nummerierung

Idee:

- Führe Hilfsattribute pre und post ein
- mit pre reichen wir einen Zählerstand nach unten (inherites Attribut)
- mit post reichen wir einen Zählerstand wieder nach oben (synthetisches Attribut)

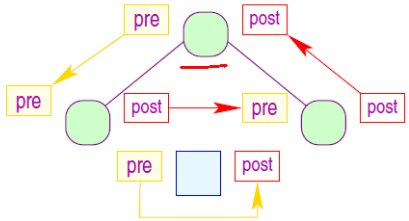
Root:	$pre[0]$	$:=$	0
	$pre[1]$	$:=$	$pre[0]$
	$post[0]$	$:=$	$post[1]$
Node:	$pre[1]$	$:=$	$pre[0]$
	$pre[2]$	$:=$	$post[1]$
	$post[0]$	$:=$	$post[2]$
Leaf:	$post[0]$	$:=$	$pre[0] + 1$

Die lokalen Attribut-Abhängigkeiten:



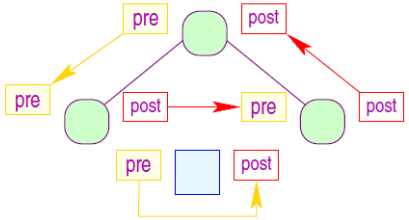
- die Attributierung ist offenbar stark azyklisch

Die lokalen Attribut-Abhängigkeiten:



- die Attributierung ist offenbar stark azyklisch
- ein innerer Knoten berechnet
  - inherite Attribute bevor in einen Kindknoten abgestiegen wird (pre-order traversal)
  - synthetische Attribute nach der Rückkehr von einem Kindknoten (post-order traversal)

Die lokalen Attribut-Abhängigkeiten:

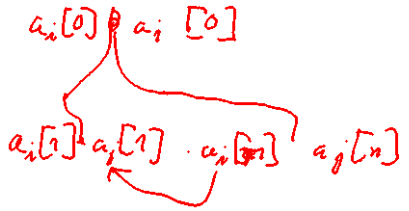


- die Attributierung ist offenbar stark azyklisch
- ein innerer Knoten berechnet
  - inherite Attribute bevor in einen Kindknoten abgestiegen wird (pre-order traversal)
  - synthetische Attribute nach der Rückkehr von einem Kindknoten (post-order traversal)
- man kann alle Attribute in *einer* depth-first Traversierung von links nach rechts auswerten (mit pre- und post-order Berechnung)
- So etwas nennen wir *L-Attributierung*

L-Attributierung

Definition

Ein Knoten mit Attributen A ist L-attributiert wenn jedes inherite Attribut $a_i[n]$ nur von $a_j[m]$ abhängt wobei $m < n$ und $a_i, a_j \in A$.



L-Attributierung

Definition

Ein Knoten mit Attributen A ist L-attributiert wenn jedes inherite Attribut $a_i[n]$ nur von $a_j[m]$ abhängt wobei $m < n$ und $a_i, a_j \in A$.

Ursprung:

- eine L-attributierte Grammatik kann während des Parsens ausgewertet werden

L-Attributierung

Definition

Ein Knoten mit Attributen A ist L-attributiert wenn jedes inherite Attribut $a_i[n]$ nur von $a_j[m]$ abhängt wobei $m < n$ und $a_i, a_j \in A$.

Ursprung:

- eine L-attributierte Grammatik kann während des Parsens ausgewertet werden

Idee:

- partitioniere alle Attribute $\mathcal{A} = A_1 \cup \ldots \cup A_n$ so dass für alle Attribute in A_i jeder Knoten L-attributiert ist
- für jede Attributmenge A_i führe eine depth-first Traversierung durch

$\rightsquigarrow$ Bestimme Attribute mit Hinblick auf wenige Traversierungen

Praktische Benutzung

- Symboltabellen, Typ-Überprüfung / Inferenz und (einfache) Codegenerierung können durch Attributierung berechnet werden
- In diesen Anwendungen werden stets *Syntaxbäume* annotiert.
- Die Knotenbeschriftungen entsprechen den Regeln einer kontextfreien Grammatik
- Knoten können in *Typen* eingeteilt werden — entsprechend den *Nichtterminalen* auf der linken Seite
- Unterschiedliche Nichtterminale benötigen evt. *unterschiedliche Mengen von Attributen*.

Praktische Implementierung

In objekt-orientierten Sprachen benutze visitor pattern:

- Klasse mit Methode für jedes Nicht-Terminal der Grammatik

```
public abstract class Regex {
  public abstract void accept(Visitor v);
}
```

- Durch Überschreiben der folgenden Methoden wird eine Attribut-spezifische Auswertung implementiert

```
public interface Visitor {
  public void visit(Dot re) { re.children(this); }
  public void visit(Bar re) { re.children(this); }
  ...
  public void visit(Token tok) {}
}
```

- Vordefiniert ist die Traversierung des Ableitungsbaumes

```
public class OrEx extends Regex {
  RegEx l,r;
  public void accept(Visitor v) { v.visit(this); }
  public void children(Visitor v) {
    l.accept(v); r.accept(v);
}}
```


Praktische Implementierung

In objekt-orientierten Sprachen benutze visitor pattern:

- Klasse mit Methode für jedes Nicht-Terminal der Grammatik

```
public abstract class Regex {
  public abstract void accept(Visitor v);
}
```

- Durch Überschreiben der folgenden Methoden wird eine Attribut-spezifische Auswertung implementiert

```
public interface Visitor {
  public void visit(Dot re) { re.children(this); }
  public void visit(Bar re) { re.children(this); }
  ...
  public void visit(Token tok) {}
}
```

- Vordefiniert ist die Traversierung des Ableitungsbaumes

```
public class OrEx extends Regex {
  RegEx l,r;
  public void accept(Visitor v) { v.visit(this); }
  public void children(Visitor v) {
    l.accept(v); r.accept(v);
}}
```


Kapitel 2:

Symboltabellen