Programming Languages and Compilers – Prof. Dr. Uwe Kastens, Zusammenfassung von Florian Schoppmann

Zusammenfassung Programming Languages and Compilers

Terminologie

Continuation Point: Aufsetzpunkt (Fortsetzstelle beim Parsing nach einem Fehler)

Domain Specific Languages (DSL): Gebietsspezifische Sprachen

Identifier: Bezeichner (zur Identifizierung von Programmobjekten)

Kontextfreie Grammatik: $G = (T, N, P, S)$, Vokabular $V = T \cup N$

Parser: Zerteiler

Program Entity: Programmobjekt: Mit individuellen Eigenschaften, u. U. mehrfach im Programm benutzt, z. B. Typ, Funktion, Variable, Label, Modul, Package, etc.

Recursive Descent Parser: Zielgerichteter Verteiler

Scanning: Symbolerkennung

Token: Symbol

1. Spracheigenschaften und Aufgaben eines Compilers

Kompilierung: Transformation von Quell- in Zielsprache

Spracheigenschafen:

statisch: Analyse zur Kompilierzeit, bspw. Definition von Variablen

dynamisch: Überprüfbarkeit erst bei Ausführung (bspw. Array-Grenzen)

Aufgaben eines Compilers:

Lexikalische Analyse: Notation von Symbolen, bspw. Schlüsselwörter, Bezeichner, Literale
Formale Beschreibung durch reguläre Ausdrücke
Einlesen, Isolierung und Kodierung von Symbolen

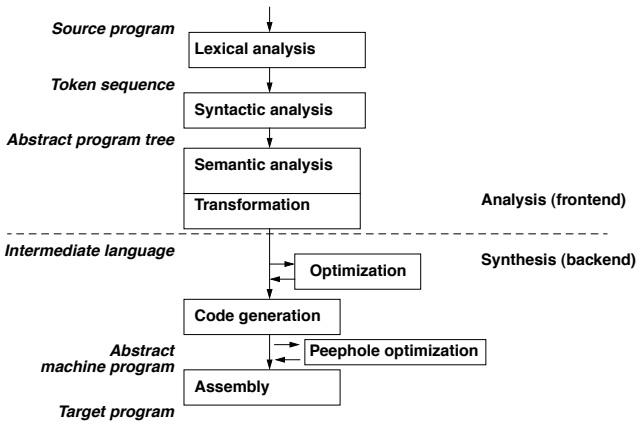
Syntaktische Analyse: Formal beschrieben durch kontextfreie Grammatik
Parsen: Ableitung mit konkreter Syntax bilden, Baum-Konstruktion mit abstrakter Syntax

Semantische Analyse, Transformation: Bindung von Namen an Programmobjekte, Typregeln
Formale Beschreibung durch attributierte Grammatiken
Semantische Analyse: Namens-˜ und Typ-˜, Eigenschaften von Programmobjekten
Transformation: Zuordnung von Daten und Aktionen/Operationen

Transformation, Code-Generierung: Semantik
Formale Beschreibung durch denotationale Semantik
Code-Generierung: Festlegung der Ausführungsreihenfolge, Registerallokierung, Auswahl der Instruktionen

Zusammensetzung (Assembly): Formale Beschreibung durch Zielsprache
Codierung der Instruktionen, Addressierung

Compiler Struktur:



Pipeline-Ansätze, jedoch Schwierigkeiten durch begrenzte Vorschau, benötigte Rückmeldungen (Feeback), bspw. beim `typedef`-Problem in C

Qualitäten eines Compilers: Korrektheit, Effizienz, Code-Effizienz, Benutzerfreundlichkeit, Robustheit

Ziele bei der Compiler-Konstruktion: Einhaltung der Sprachdefinition, Werkzeuge für Generierung (insb. für reguläre Ausdrücke, Kontextfreie Grammatiken, attributierte Grammatiken, Code-Muster), Standardkomponenten und -methoden, Validierung gegen Testsammlung, Komponentenverifizierung

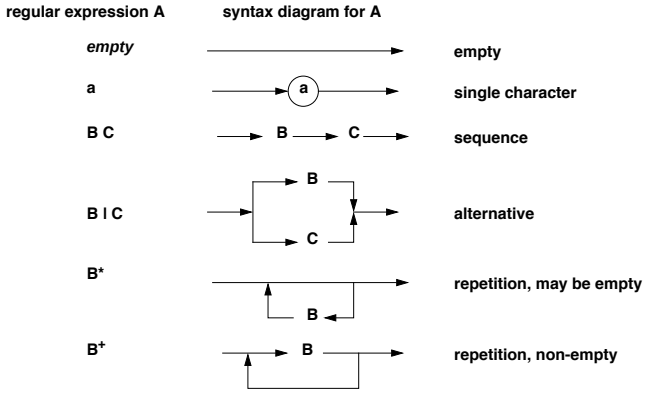
2. Spezifikation von Symbolen und lexikalische Analyse

Ziel: Isolierte Erkennung von Symbolen (ohne dass Informationen der semantischen Analyse benötigt werden)

Kodierung von Symbolen in Tripel: (*Syntaxcode*, *Attribut* (Wert oder Referenz ins Data-Modul), *Position im Quellcode*)

Programming Languages and Compilers – Prof. Dr. Uwe Kastens, Zusammenfassung von Florian Schoppmann

Transformation Reguläre Ausdrücke ↔ Syntaxdiagramme:



Zusammensetzung von endlichen Symbol-Automaten: Abbildung auf einen Anfangszustand, Endzustände der Teilautomaten getrennt halten (sie klassifizieren die Symbole), Teilautomaten für bedeutungslose Zeichen (Kommentare etc.)

Regel des längsten Musters: Automat hält in beliebigem Zustand, falls kein Übergang mit nächstem Zeichen möglich, bei Stopp Zurücksetzung auf zuletzt durchlaufenen Endzustand, Fehler, falls kein Endzustand durchlaufen
$\Longrightarrow$ Manche Symbole müssen explizit getrennt werden.

Implementierung von Scannern: Zur Anzahl von Zeichen proportionale Laufzeit, Automat sollte „hart kodiert sein“ (generierter Code, tabellengesteuert wäre zu langsam)

Bezeichnermodul und Literalmodule: Eingabe: Zeiger auf Symboltext, Ausgabe: Syntaxcode, Attribut

Scanner-Generatoren: GLA (in Eli), Lex, Flex, Rex

3. Kontextfreie Grammatiken und syntaktische Analyse

Gegenüberstellung:

Konkrete Syntax	**Abstrakte Syntax**
Kontextfreie Grammatik	
eindeutig spezifiziert Ableitung und Parser	nicht eindeutig Grundlage für die Übersetzungsphase
Kettenproduktionen	Nicht-Terminale zu Klassen vereinigt (bspw. `Expr`, `Fact`)
Terminale	Nur semnatisch relevante Terminale: Bezeichner, Literale
	Aus konkreter Syntax generierbar

Recursive Descent Parser: rekursiver Abstieg: Transformation von KFG in Menge von Funktionen:
Nichtterminal → Funktion, alternative Produktionen → Zweige im Rumpf, Nichtterminal auf rechter Seite → Aufruf, Terminal auf rechter Seite → akzeptiere Symbol und lies, Entscheidungsmenge zu Produktion → Verzweigungsentscheidung (`switch (currentSymbol)`)

Grammatikbedingung für Recursive Descent: Eine KFG ist start LL(1), wenn für alle Produktionenpaare mit gleicher linker Seite die Entscheidungsmengen disjunkt sind

Die Entscheidungsmenge einer Produktion $A ::= u$ ist definiert als First(u Follow(A)), wobei:
First(u) := $\{t \in T \mid \exists v \in V^*$, Ableitung $u \Rightarrow^* t\,v\}$ und zusätzlich $\varepsilon \in$ First(u), falls $u \Rightarrow^* \varepsilon$ existiert.
Follow(A) := $\{t \in T \mid \exists u, v \in V^*, A \in N$, Ableitung $S \Rightarrow^* u\,A\,v$, so dass $t \in$ First(v)$\}$

Eigenschaften starker LL(1) Grammatiken:

- Keine alternativen Produktionen, die mit dem gleichen Nicht-Terminal beginnen
- Keine direkt oder indirekt links-rekursiven Produktionen

Gegenüberstellung:

zielbezogen (top-down)	**quellbezogen (bottom-up)**
Vorschau (Look-ahead) für Produktionsauswahl benötigt	
Linksableitung vorwärts Produktionen	Rechtsableitung rückwärts Reduktionen

Eine KFG ist LR(k), mit $k \geq 0$, wenn für alle Paare von Rechtsableitungen
$Z \Rightarrow^* u\,A\,w \Rightarrow u\,x\,w$
$Z \Rightarrow^* u'\,B\,w' \Rightarrow u'\,x'\,w'$
mit $u, u', x, x' \in V^*$, $w, w', v \in T^*$ gilt:
Falls es ein v gibt, so dass $u\,x\,v = u'\,x'\,w'$ und die ersten k Zeichen von v und w gleich sind, dann sind auch $A = B$, $x = x'$, $u = u'$ (d. h. der letzte Ableitungsschritt ist gleich)

LR(1) Situationen: Analysezustand *bzgl. einer Produktion*:
$[A ::= u \bullet v\,R]$, wobei $\bullet$ Analyseposition, Rechtskontext R: Menge von Terminalen, die der

Eingabe folgen können, wenn die vollständige Produktion akzeptiert ist
Reduktionssitation: $[A ::= u\,v \bullet R]$, falls nächstes Symbol in R enthalten ist

Ein Zustand eines LR-Automaten besteht aus einer Menge von Sitationen, die beschreiben, wie die Analyse fortfahren kann.

Operationen eines LR(1)-Automaten:
shift (Nächsten Zustand auf den Stapel legen, ausgehend vom aktuellen), reduce (reduzieren, n Zustände vom Stapel entfernen, wenn bei der angewendeten Produktion $p: B ::= u$ die Länge von u genau n ist, und vom neuen Zustand ein shift mit B durchführen), error (Fehler), stop (Halt, Startproduktion reduzieren)

LR Konflikte: Ein LR(1)-Automat mit Konflikten ist nicht deterministisch und seine Grammatik nicht LR(1).

reduce-reduce Konflikt: Ein Zustand mit zwei Reduktionssitationen, deren Rechtskontexte nicht disjunkt sind

shift-reduce Konflikt: Ein Zustand enthält eine Shift-Situation mit Analyseposition vor einem Symbol t und eine Reduktionssituation mit t in ihrem Rechtskontext.

Vereinfachte LR-Klassen

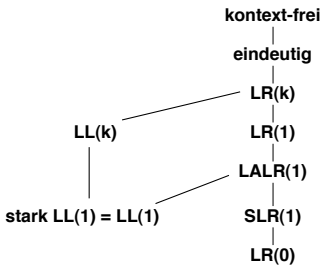
LR(1): Zu viele Zustände

LR(0): Alle Situationen ohne Rechtskontext, folglich: Reduktionssitationen nur einzeln in Zuständen

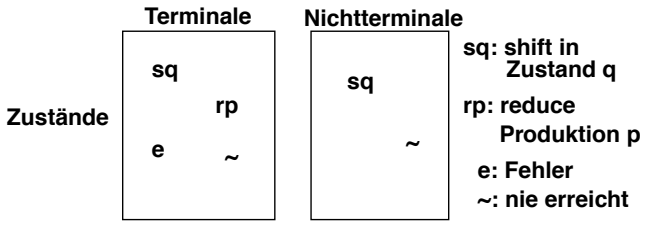
SLR(1): Zustände wie bei LR(0), vergröberter Rechtskontext bei Reduktionssitationen: $[A ::= u \bullet \text{Follow}(A)]$

LALR(1): Wie LR(1), jedoch Zusammenlegung von Zuständen, wenn sich ihre Situationen nur in ihren Rechtskontexten unterscheiden. Liefert LR(0)-Zustände mit „exaktem“ Rechtskontext

Hierarchie:



Tabellengesteuerte Implementierung von LR-Automaten:



Speicherplatz sparen durch: Zusammenlegung von verträglichen Spalten/Zeilen, separater Fehler matrix, kleinsten Wert von Zeilen/Spalten subtrahieren, LR(0)-Reduktionszustände streichen
Direkt programmierte LR-Automaten möglich, jedoch gewöhnlich zu groß

Parser Generatoren: PGS, Cola, Lalr, Yacc, Bison, LLgen, Deer

Behandlung syntaktischer Fehler: so früh wie möglich, Symptom quellbezogen melden, Fortsetzung möglichst kurz nach Fehler, Folgefehler vermeiden, strukturell korrekter Baum, keine Zurücknehmen von Aktionen, möglichst kein zusätzlicher Aufwand für korrekte Programme

Fehlerstelle: t ist die erste Stelle der Eingabe $w\,t\,x \in T^*$, wobei $t \in T$ und $w, x \in T^*$ und w korrekter Präfix in $L(G)$, $w\,t$ jedoch nicht.

Aufsetzstelle: Das Symbol d hinter der Fehlerstelle t, so dass das Parsen der Eingabe mit d fortgesetzt wird.
Fehlerbehebung: Wenn Eingabe $w\,t\,x$ mit Fehlerposition bei t und $w\,t\,x = w\,y\,d\,z$, dann wird $w\,y\,d\,z$ durch $w\,v\,d\,z$ so ersetzt, dass $w\,v\,d$ ein korrekter Präfix in $L(G)$ ist.

Simulierte Fortsetzung: Aus Fehlerstelle und Keller Menge von Symbolen als mögliche Aufsetzpunkte bestimmen.

4. Attributierte Grammatiken und semantische Analyse

Attributiere Grammatiken: spezifiziert deklarativ abhängige Berechnungen im Strukturbaum (→ Ausführungsreihenfolge wird berechnet)
Attributauswerter wird für eine attributierte Grammatik generiert

Anwendungsbeispiel: Berechnung von $Y.a$ vor $X.b$:

```
RULE q: Y ::= w COMPUTE
    X.b = f(Y.a);
END;
RULE p: Y ::= u COMPUTE
    Y.a = g(...);
```

```
END;
```

Abhängigkeit ohne Weiterreichung von Werten:

```
[...]
X.GotType = ResetTypeOf(...);
[...]
Y.Type = GetTypeOf(...) <- X.GotType;
[...]
```

(`ResetTypeOf` wird vor `GetTypeOf` aufgerufen.)

Formale Definition: Eine attributierte Grammatik AG $= (G, A, C)$ ist definiert durch eine KFG G (abstrake Syntax), eine Abbildung A, die jedes Nicht-Terminal X auf eine Menge von Attributen $A(X)$ abbildet (geschrieben $X.a$, falls $a \in A(X)$), sowie für jede Produktion p von G eine Menge von Berechnungen in einer der beiden folgenden Formen: $X.a = f(\dot{Y}.b\dot{)}$ oder $g(\dot{Y}.b\dot{)}$, wobei X und Y in p vorkommen.

Eine **attributierte Grammatik** ist genau dann **konsistent und vollständig**, wenn gilt:

i) Jedes $A(X)$ lässt sich in zwei (disjunkte) Teilmengen zerlegen: Dabei bezeichne
   $AI(X)$: die erworbenen Attribute (inherited attributes), die in Produktionen berechnet werden, in denen X auf der rechten Seite steht und
   $AS(X)$: die abgeleiteten Attribute (synthesized attributes), die in Produktionen berechnet werden, in denen X auf der linken Seite steht
ii) Jede Produktion $p: X ::= \ldots Y \ldots$ hat genau eine Berechnung für jedes Attribut in $AS(X)$ und genau eine Berechnung für jedes Attributs in $AI(Y)$ (für jedes Vorkommen von Y).

Abhängigkeitsanalyse für AGen: Partitionierung von $AI(X)$ und $AS(X)$, so dass $AI(X, i)$ *vor* und $AS(X, i)$ *während* des i-ten Besuchs von X berechnet wird.
Voraussetzung für Zerlegung ist dabei: An keinem X-Knoten in keinem Strukturbaum gibt es direkte oder indirekte Abhängigkeiten entgegen der Folge: $AI(X, 1), AS(X, 1), \ldots, AI(X, k), AS(X, k)$

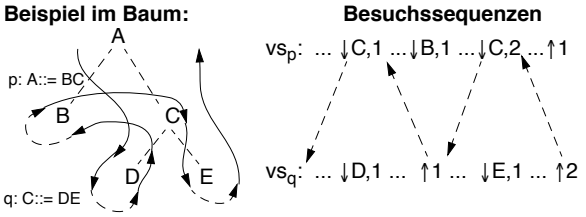
Generierter Attributauswerter ist anwendbar auf jeden Baum zur AG. Mögliche Strategien:
Pass-orientiert:
LAG(k): k mal links-abwärts
AAG(k): k-mal alternierend links/rechts-abwärts
SAG: einmal aufwärts

Nicht Pass-orientiert:
Besuchssequenzen: individueller Plan für jede Produktion der abstrakten Syntax

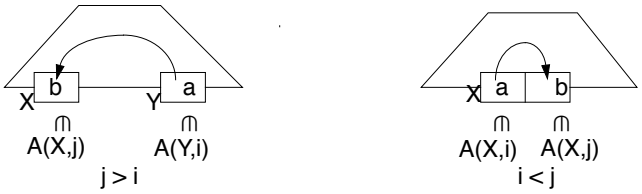
Eine Besuchssequenz vs_p zu einer Produktion $p: X_0 ::= X_1 \ldots X_i \ldots X_n$ der Baumgrammatik ist eine Folge von Operationen: $\downarrow i, j$: j-ter Besuch des i-ten Unterbaums, $\uparrow j$: j-te Rückkehr zum Vaterknoten, eval_c: Auswertung einer p zugeordneten Berechnung c

Anwendungsbeispiele:

i) $vs_p: L ::= B$
   $L.lg = 1; \uparrow 1; B.s = L.s; \downarrow B, 1; L.V = B.V; \uparrow 2$
ii) Zusammenführen von Besuchssequenzen:



Eine AG erfüllt die LAG(k)-Bedingung, wenn für jedes Nicht-Terminal X es eine Zerlegung $A(X, 1), \ldots, A(X, k)$ gibt, so dass die Attribute in $A(X, i)$ im i-ten links-abwärts-Durchgang berechnet werden können.
Notwendige und hinreichende Bedingung über Abhängigkeitsgraphen zu jeder Produktion:



Algorithmus zur Berechnung der $A(i)$, sofern existent:

```
Cand ← alle Attribute
for i ← 1, 2, ... do
    A(i) ← Cand
    Cand ← ∅
    while Es kann noch ein Attribut X.b aus A(i) entfernt werden do
        if X.b ist von einem Attribut aus A(i) nach einem der beiden zuvor beschriebenen Muster abhängig oder X.b hängt von einem Attribut ab, das sich in noch keinem A(j), j ≤ i befindet then
            A(i) ← A(i) \ {X.b}
            Cand ← Cand ∪ {X.b}
        end if
    end while
```

if Cand $= \emptyset$ **then**
Ausgabe $A(j), j = 1, \ldots, i$
Ende
else if $A(i) = \emptyset$ **then**
Ausgabe „LAG(k) für kein k erfüllt."
Ende
end if
end for

Generatoren für AGs: LIGA (OAG), FNC-2 (ANCAG, Oberklasse von OAG), Synthesizer Generator (OAG), CoCo (LAG(1))

5. Namensbindung

Statische Bindung: Bindung zwischen einem Namen und einem Programmobjekt, gültig innerhalb eines Bereichs (Scope) des Programmtexts.

Dynamische Bindung: Bindungen werden erst in der Laufzeit-Umgebung hergestellt

Namensanalyse (Bezeichneridentifikation): Jedem Auftreten eines Bezeichners wird Programmobjekt zugeordnet (konsistente Umbenennung)

Eine Umgebung (Environment) ist eine (lineare) Folge von Mengen von Bindungen $e_1, e_2, \ldots$, für die eine Verdeckungsbeziehung gilt: Eine Bindung (a, k) in e_i verdeckt (a, h) in e_j, wenn $i < j$.

Ein Umgebungsmodul implementiert den abstrakten Datentyp Environment mit den Operationen:

`NewEnv()` Erstellt eine neue Umgebung e als Wurzel.

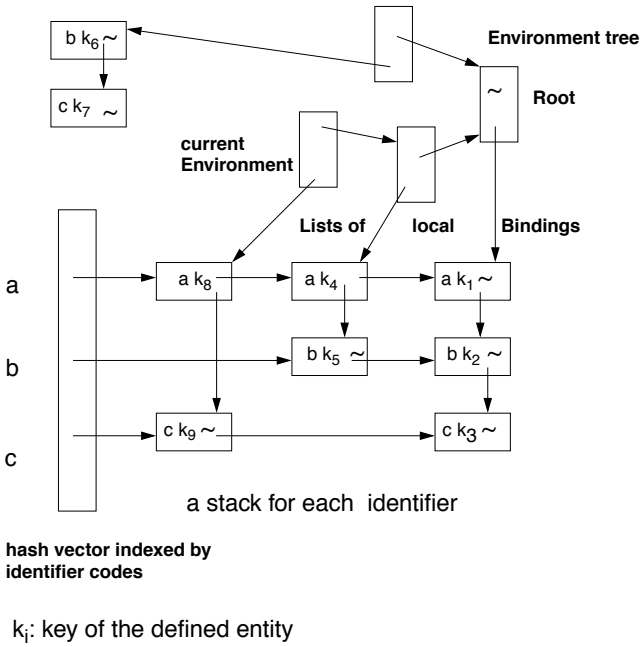
`NewScope(`e_1`)` Erzeugt eine neue Umgebung e_2. Alle Bindungen aus e_1 sind auch in e_2, falls nicht verdeckt.

`BindIdn(`e, id`)` Fügt Bindung (id, k) in e ein, falls sie noch nicht dort existiert, Rückgabe k.

`BindingInEnv(`e, id`)` Liefert das Bindungstripel (id, e, k) sofern vorhanden. Liefert das Tripel (id, e_x, k), falls Bindung nicht in e, aber in einer e umfassenden Umgebung e_k definiert ist und e_k davon die kleinste ist, bzw. „NoBindung", falls keine Bindung existiert.

`BindingInScope(`e, id`)` Liefert das Bindungstripel (id, e, k), „NoBindung", falls Bindung nicht direkt in e existiert.

Effiziente Datenstruktur für ein Umgebundsmodul:



Operationen in Baumkontexten und Reihenfolge ihrer Ausführung **modellieren Gültigkeitsregeln**:

Algol-Regel: Vorbedingung für `BindingInEnv`: Alle `BindIdn` aller umgebenden Umgebungen sind ausgeführt.

C-Regel: `BindingInEnv` und `BindIdn` werden in der Reihenfolge des Programmtextes ausgeführt (links-abwärts).

6. Typspezifikation und -Analyse

Statische Typisierung: Programmobjekte und -konstrukte haben Typeigenschaften, die zur Kompilierzeit bestimmt und auf Konsistenz überprüft werden:

Definierte Objekte: z. B. Variable, Typ als Eigenschaft

Programmkonstrukte: z. B. Ausdruck, Indizierung, Typ als Attribut des Baumknotens

Typen sind selbst Programmobjekte mit Eigenschaften.

Definitionsmodul: Datenstruktur zur Speicherung von Eigenschaften von Programmobjekten. (Wird generiert, im Folgenden ist daher `P` ein Platzhalter für Namen von Eigenschaften.)

`NewKey()` Erstellt einen neuen Schlüssel k

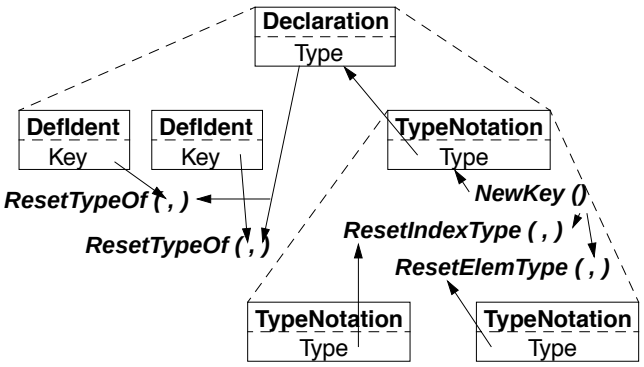
`ResetP(`k, v`)` Setzt für den Schüssel k die Eigenschaft `P` auf den Wert v

`SetP(`k, v, d`)` Wie `ResetP`, Eigenschaft wird jedoch auf d gesetzt, wenn vorher noch nicht nicht gesetzt

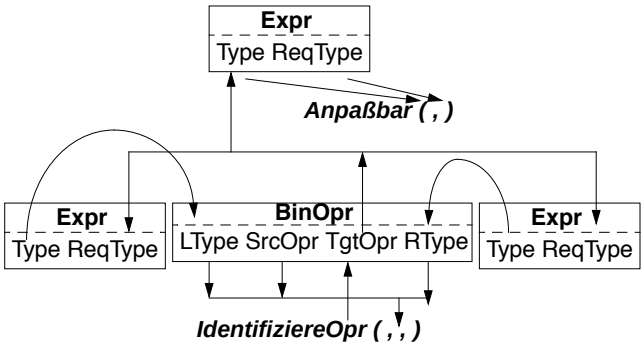
`GetP`(k, d) Rückgabe der Eigenschaft, d wenn nicht gesetzt

Anwendungsbeispiele:

```
a, b: array [1..10] of real;
```



Überladen eines binären Operators:



Ziele bei Fehlerbehandlung: Prüfungen im kleinsten betroffenen Kontext, aussagekräftiger Meldungstext, unterschiedliche Meldungen zu verschiedenen Symptomen, Ausführen aller Operationen im Baum trotz Fehler: Fehlerwerte einführen, z. B. `NoKey`, `NoType`, `NoOpr`, Operationen liefern auch im Fehlerfall Werte zurück, jede Operation akzeptiert auch Fehlerwerte als Parameter, Meldungen zu Folgefehlern vermeiden (z. B. ist jeder Typ ist mit `NoType` verträglich)

7. Dynamische Semantik

Der Effekt des Ausführens eines Programms wird als dynamische Semantik bezeichnet. Informale Spezifikation oft auf abstrakter Maschine beschrieben, z. B. „Jede Variable hat eine Speicherzelle, . . . “

Ein formales Schema zur Beschreibung dynamischer Semantik ist die denotationale Semantik: Die ausführbaren Bestandteile der abstrakten Syntax werden auf Funktionen abgebildet und beschreiben so ihre Auswirkung. Die Zusammensetzung der mit den Knoten eines gegebenen Strukturbaums verbundenen Funktionen ergibt (statisch!) eine semantische Funktion des gesamten Programms.

Denotationale Spezifikation: Spezifikation der semantischen Definitionsbereiche, eine Funktion E, die alle Ausdruckskontrukte sowie eine Funktion C, die alle Anweisungskontrukte auf semantische Funktionen abbildet.

Anwendungsbeispiel: Semantische Wertebereich des Zustands einer sehr einfachen imperativen Sprache:
Programmzustand: $State = Memory \times Input \times Output$

Zuordnung von Ausdrücken: Sei *Expr* die Menge aller Konstrukte der abstrakten Syntax, die einen Ausdruck darstellen. $E : Expr \to (State \to Value)$, wenn Ausdrücke *keine* Nebenwirkungen haben, $E : Expr \to (State \to (State \to Value))$ sonst.

Zuordnung von Anweisungen: Sei *Command* die Menge aller Konstrukte der abstrakten Syntax, die eine Anweisung darstellen. $C : Command \to (State \to State)$. Falls auch Sprünge und Labels modelliert werden sollen, wäre ein zusätzliches Argument nötig.

8. Source-to-Source Übersetzung

Übersetzung von einer Highlevel-Quellsprache in eine -Zielsprache.

Aufgabe der Transformationsphase:
Eingabe: Strutkurbaum und Eigenschaften der Konstrukte (Attribute) und Objekte (Definitionsmodul)
Ausgabe: Aus Zielbaum (Attribute) erzeugte textuelle Repräsentation

Musterbasierter Text-Generator (Pattern-based Text Generator): Einsetzen von Argumenten in (Programm-)Textschablonen