ISA
Institut für
SoftwareArchitektur

Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Kontextanalyse

- Aufgabenstellung der Kontextanalyse
- Formalismen zur Definition und Implementierung der Kontextanalyse
- Organisation von Baumdurchläufen

Kontextsensitive Analyse

Kontextanalyse

Aufgaben eines Compilers

- Quellcode analysieren
- Zielcode erzeugen

Quellcode analysieren

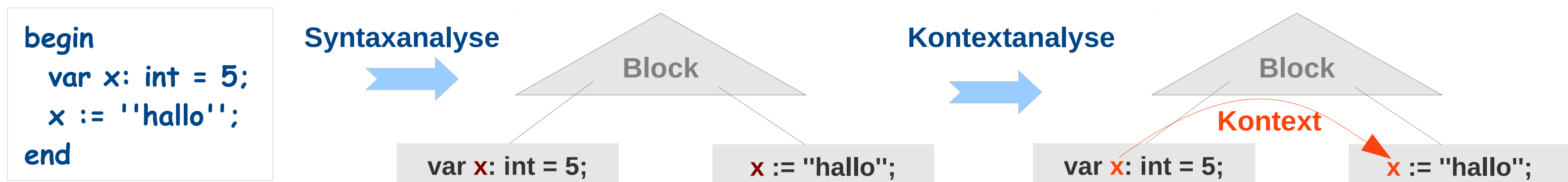
- Syntax-Analyse

  Parser: *Kontextfrei* – Informationen werden rein „bottum-up“ verarbeitet

- Kontext-Analyse

  Viele Informationen müssen „kontext-sensitiv“ verarbeitet werden: sie müssen im Baum verteilt werden:

  - Informationen zur Programmkorrektheit (Typen, Typfehler)
  - Information die bei der Codegenerierung benötigt werden (Größen, Variablen-Positionen, ..)



Kontextanalyse

Andere Bezeichnungen

- Semantische Analyse
- Kontextsensitive Analyse

Basis

- AST – der abstrakte Syntaxbaum
- Das Ergebnis des Parsens der kontext<u>freien</u> Grammatik

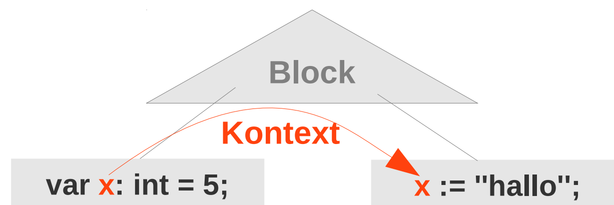
Aktivitäten

- Sammeln, Prüfen, Verarbeiten, Berechnen, Verteilen von Informationen im AST
- Beispiel: „Ist eine Zuweisung an diesen Bezeichner erlaubt?“
  - Grammatik:

    Block ::= Decl$^+$ Smnt$^+$
    Decl ::= … | var Identifier ':' *Type* '=' *Exp* ';' *| …*
    Stmnt ::= … | Identifier ':=' *Exp* ';' *| …*
    Exp ::= … | Exp '+' *Exp | …*

  - AST

Mit einer kontext<u>freien</u> Grammatik können kontext<u>sensitive</u> Beziehungen nicht ausgedrückt werden.



Kontextsensitive Analyse

Kontextanalyse

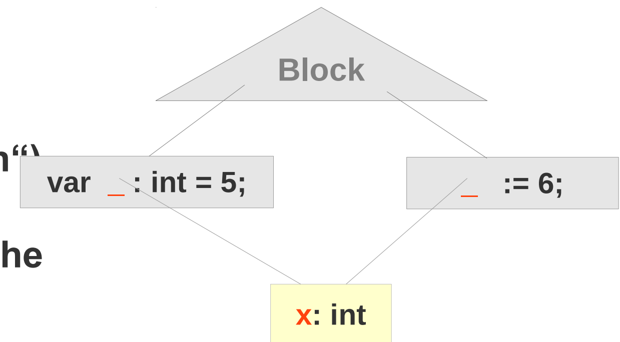
Organisation

Die Kontextanalyse kann auf verschiedene Arten realisiert werden

- **Als Semantische Aktionen des Parsers**

  Der Parser kann bereits einige oder eventuell sogar alle kontextsensitiven Verarbeitungsschritte („semantische Aktionen") ausführen.

  Oft führt der Parser die Bezeichner-Identifikation als „semantische Aktion" des Parsers bezeichnet.



Semantische Aktionen: Der Parser erzeugt einen modifizierten AST (eigentlich einen Graph)

- **Als Modifikation des vom Parser erzeugten ASTs**

  Der vom Parser erzeugte AST wird modifiziert und dabei in einen Graph verwandelt

- **Als Konstruktion einer neuen Datenstruktur aus dem AST**

  Bei der semantischen Analyse wird aus dem AST eine komplett neue Datenstruktur (der analysierte Ast) erzeugt.

- **Als Kombination der Verfahren**

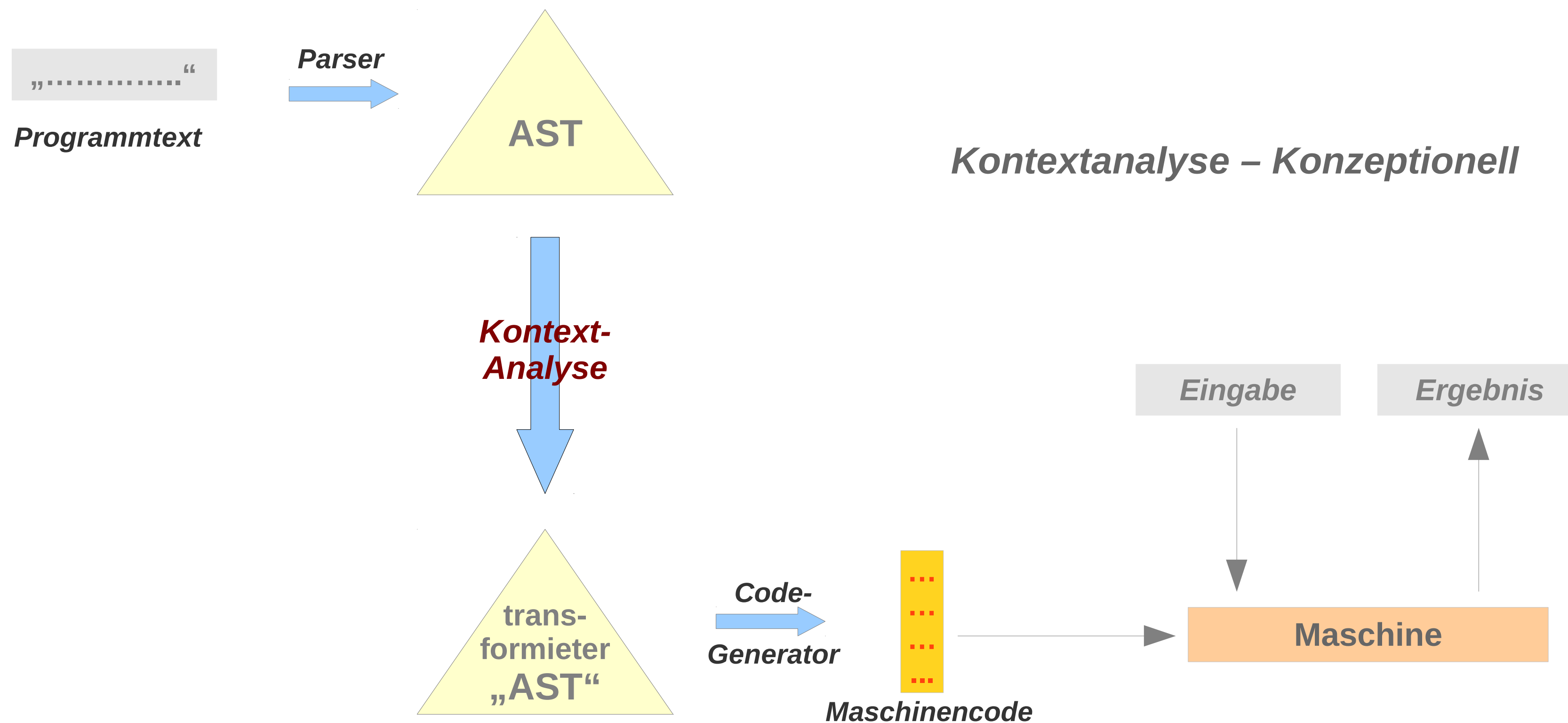
Kontextsensitive Analyse

Kontextanalyse

Allgemeine Aufgabe

- Aufbereiten / Modifizieren / Transformieren des AST zu einer Datenstruktur die für die Codegenerierung geeignet ist
- Dabei werden
  - statische Berechnungen / Prüfungen und
  - Informationsaufbereitungen für die Codegenerierung

  vorgenommen



Kontextanalyse – Konzeptionell

Kontextsensitive Analyse: Problemstellungen

Kontextanalyse: Typische Fragestellungen

Auf was bezieht sich ein Name (Bezeichner)

- Gibt es eine Definition für den Bezeichner?
- Wenn es mehrere Definitionen mit dem gleichen Namen gibt, welche ist hier gemeint?
- Handelt es sich um den Namen eins Typs, einer Variable, einer Prozedur, ….
- ...

Welche Art von Wert enthält eine Variable

- Wie viele Bytes müssen bei einer Zuweisung bewegt werden?
- Ist eine Zuweisung erlaubt?
- Kann der Wert in einer Addition als Operand verwendet werden?
- …

Welches Ergebnis, wie viele und welche Parameter hat eine Funktion

- Wie viele Bytes müssen bei einer Parameterübergabe bewegt werden?
- Ist ein Aufruf erlaubt?
- Kann das Funktionsergebnis in einer Addition als Operand verwendet werden?
- …

Speicherbedarf für Variablen

- Wann müssen wie viele Bytes für eine Variable allokiert werden?
- Wie wird auf die Variable zur Laufzeit zugegriffen
- Wann kann der Speicher wieder frei gegeben werden?
- …

Kontextsensitive Analyse: Problemstellungen

Kontextanalyse: Typische Organisation

Die Kontextanalyse wird üblicherweise in Unterphasen aufgeteilt:

- **Bezeichner- (Namens-) Identifikation**
  Identifikation der Bezeichner, die das Gleiche bezeichnen
- **Typisierung**
  Berechnen und Prüfen von Typiformationen
- **Adressberechnung**
  Berechnen von statisch bekannten Adressen,
  Vorbereiten der (Codegenerierung von) Berechnungen von dynamisch berechneten Adressen

Kontextsensitive Analyse: Problemstellungen

Kontextanalyse: Allgemeine Problemstellungen

Die Kontextanalyse wird meist als Serie von Transformationen des AST organisiert

Thema: Baum- / Graph-Transformation

- **Wie werden diese Transformationen implementiert**

  SWT- / A&D-Problemstellung: Organisation einer Serie von Baumtransformationen

  Besuchermuster, …

- **Wie definiert man sie allgemein (ohne gleich eine Implementierung zu liefern)**

  Baum- / Graph-Transformationen

  Attributierte Grammatiken

  2-stufige Grammatiken, ...

Kontextsensitive Analyse

Kontextanalyse: Spezielle (programmiersprachliche) Problemstellungen

Die Regeln nach denen sich die kontextsensitive Analyse zu richten hat,

- werden von der Sprachdefinition geliefert.
- Sie treten dort in Form
  - von intuitiv verstehbaren
  - aber gelegentlich schwer exakt definierbaren

  Schlüsselkonzepten auf:

Namensbindung

Konzepte wie Gültigkeitsbereiche etc. regeln

- Welcher Name sich auf welche Definition/Deklaration bezieht und
- Wann Speicherplatz angefordert und freigegeben werden muss
- …

Statische Typen / Typ-Systeme

Um Fragen zu klären, wie:

- Ist die Zuweisung, die arithmetische Operation, der Aufruf, …. erlaubt?
- Wie viele Bytes müssen bei einer Zuweisung, Parameterübergabe bewegt werden?
- …

werden Text-Fragmenten (statische) Typen zugeordnet und geprüft

Definition der Kontextanalyse

Kontextanalyse: Allgemeine Problemstellungen

Kontext-sensitiver Aspekt ~ Semantik

Semantik: Alles, was sich nicht mit einer kontextfreien Grammatik ausdrücken lässt

- **Namen müssen vor ihrer Verwendung definiert sein**
- **Bei einer Addition müssen die Operanden kompatible Typen haben**
- **Bei einem Funktionsaufruf werden die Argumente zuerst ausgewertet**
- **….**

Es gibt bisher keinen allgemein anerkannten Formalismus zur Definition kontextsensitiver Sprachaspekte!

Definition kontextsensitiver Aspekte

Wichte Formalismen / Methoden:

- **Definitorischer Interpreter**

  Eine Implementierung der Sprache mit einem Interpreter definiert die Semantik

- **Attributierte Grammatiken**

  Ein allgemeiner Mechanismus zur Definition von Datenflüssen in einem Baum wird zur Definition der Semantik verwendet

- **Kontextsenitive Grammatiken**

  Kontext-sensitive Grammatiken erlauben Produktionen in denen Ausdrücke statt nur Nonterminale links stehen, z.B.:

  $$aAb \rightarrow aaAbb \mid abAba$$

  Damit können (auf extrem umständliche Art) kontextsensitive Aspekte definiert werden

Definition der Kontextanalyse

Zweistufige Grammatik

Was ist das

zweistufige Grammatiken sind ein Formalismus zur Definition kontextsensitiver Aspekte, der äquivalent zu kontextfreien Grammatiken ist, aber leichter handhabbar ist als diese.

Zweistufige Grammatiken werden hier nicht weiter betrachtet

Nur eine von ihnen inspirierte selbsterklärende Notation wird kurz vorgestellt:

Beispiel: Notation inspiriert von Zweistufigen Grammatiken

Expr → *Integer*-Expression
| *Boolean*-Expression

***Integer*-Expression → *Integer*-Expression IntOp *Integer*-Expression**
| Number
| *Integer*-Variable

***Boolean*-Expression → *Integer*-Expression CompareOp *Integer*-Expression**
| *Boolean*-Expression BoolOp *Boolean*-Expression
| 'true' | 'false'
| *Boolean*-Variable

***Integer*-Variable → Identifier** **% the identifier must desginate a declared integer variable**

***Boolean*-Variable → Identifier** **% the identifier must desginate a declared boolean variable**

Die Notation ist durch das Beispiel hinreichend erklärt.

Definition der Kontextanalyse

Attributierte Grammatiken: die Idee

Deklarative Spezifikation von Berechnungen in Bäumen: Wird oft für die Definition der kontextsensitiven Analyse genutzt.

Der Syntaxbaum enthält nur Informationen die auf den kontextfreien Regeln der Grammatik beruhen

Bei der Verarbeitung von kontextsensitiven Informationen wird der Syntaxbaum geprüft, erweitert modifiziert

Attributierte Grammatiken: Die Prüfung, Erweiterung, Modifikation des Baums wird definiert über

- **Berechnungsregeln für**
- **Attribute**

Attribute sind Werte die den Knoten des (abstrakten) Syntaxbaums zugeordnet sind

Die Berechnungsregeln definieren eine rein funktionale Berechnung der Attribute

- **in Form von Gleichungen**
- **ohne Angabe eines Algorithmus', der die Berechnungen ausführt**

Ein Framework für attributierte Grammatiken muss die definierten Berechnungen korrekt ausführen.

Der Sprachdefinierer kann sich auf die Definition der Regeln (Gleichungen) konzentrieren, die Umsetzung der Regeln erfolgt durch das Framework.

Ein Framework für attributierten Grammatiken ist oft in ein Framework zur Generierung von Parsern integriert. (z.B. YACC)

Attributierte Grammatiken: Einsatz

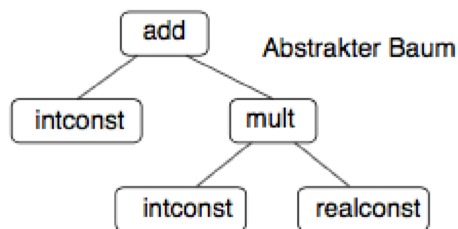
- **Attributierte Grammatiken wurden von D. Knuth 1968 entwickelt***
- **Als Mittel um die Semantik kontextfreier Sprachen zu spezifizieren**
- **Die Idee wurde im Compilerbau aufgenommen und in Frameworks umgesetzt**
- **Der Parser-Generator *yacc* unterstützt eine (sehr) eingeschränkte Form von attributierten Grammatiken. (Siehe Skript M. Jäger)**
- **Wir nutzen in dieser Veranstaltung**
  - **kein Framework für attributierte Grammatiken,**
  - **sondern betrachten nur die Idee der attributierten Grammatiken als Mechanismus zur Spezifikation von Operationen auf Bäumen**
- **Die Operationen auf Bäumen werden statt dessen händisch aus-programmiert**

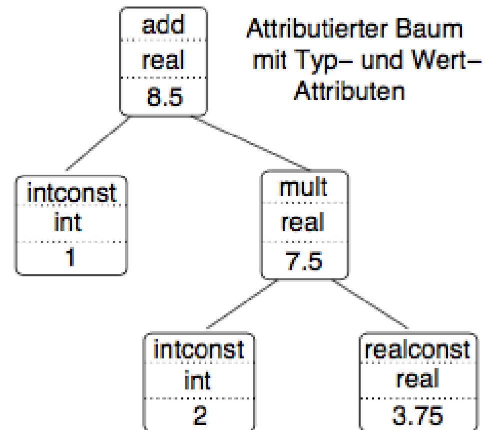
* D. Knuth: *Semantics of Context-Free Languages*. Mathematical Systems Theory Vol. 2, No. 2 (1968)

Attributierte Grammatiken

Beispiel: Attribute

(abstrakter) Syntaxbaum und attributierter Syntaxbaum





Aus: M.Jäger *Compilerbau – Eine Einführung*,
https://homepages.thm.de/~hg52/lv/compiler/skripten/compilerskript/pdf/compilerskript.pdf

Attributierte Grammatik als Datenfluss-System

Mit einer attributierten Grammatik können beliebige Berechnungen „quer durch den Baum“ definiert werden: Jeder Attributwert kann von jedem Attributwert an einer beliebigen anderen Stelle im Baum abhängen.

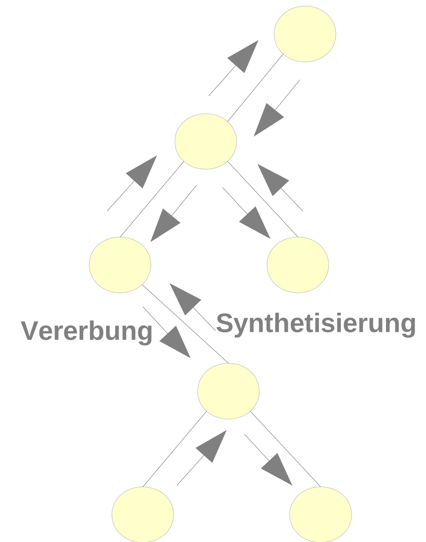
Man kann dabei zwei Arten der Attributweitergabe unterschieden:

Synthetisierung

Der Wert eines Attributs wird von einem Knoten an dessen Elternknoten (nach oben) weitergegeben

Vererbung

Der Wert eines Attributs wird von einem Knoten an dessen Kinder (nach unten) weitergegeben

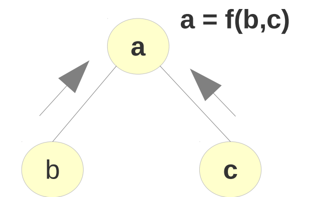


Der Datenfluss kann beliebig sein

dabei können beliebige Abhängigkeiten entstehen, auch solche

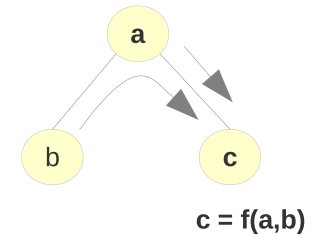
- die nur durch eventuell viele Baumdurchläufe aufgelöst werden können, oder
- die zyklisch sind und so niemals aufgelöst werden können

Synthetisierte und ererbte Attribute



Synthetisiertes Attribut

Der Wert eines synthetisierten Attributs eines Knotens k wird nur von Attritutwerten von Kindknoten von k bestimmt



Ererbtes Attribut

Der Wert eines ererbten Attributs eines Knotens k wird nur von Attributen von Knoten bestimmt, die Kindknoten oder Geschwisterknoten von k sind.

Bemerkungen

- Ein Attribut kann Abhängigkeiten nach oben und nach unten haben. Es muss also nicht entweder synthetisiert oder ererbt sein.
- Auch wenn alle Attribute entweder ererbt oder synthetisiert sind
  - kann die korrekte Auswertungsreihenfolge nur durch eine Datenfluss-Analyse bestimmt werden (Topologische Sortierung!)
  - kann es zyklische Abhängigkeiten geben.

Attributierte Grammatiken

Beispiel Ausdruckssprache: Synthetisierte Attribute

Syntax (AST):

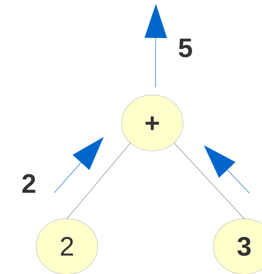
Exp = Const(v: Int)

| Add(e_1: Exp, e_2: Exp)

| Mult(e_1: Exp, e_2: Exp)

Semantik definiert als Auswertungsfunktion eval

```
eval(e: Exp) = e match
    case Const(v)          => v
    case Add(e1, e2)       => eval(e1) + eval(e2)
    case Mult(e1, e2)      => eval(e1) * eval(e2)
```

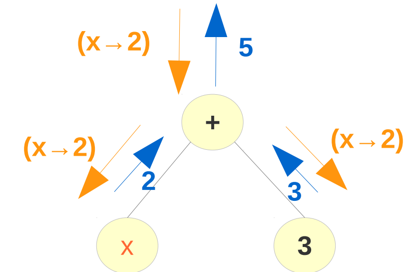


Die Semantik der Ausdruckssprache kann über ein synthetisiertes Attribut beschrieben werden.

Beispiel Ausdruckssprache mit Variablen: Ererbte Attribute

Syntax (AST):

```
Exp = Const(v: Int)
      | Var(x: String)
      | Add(e1: Exp, e2: Exp)
      | Mult(e1: Exp, e2: Exp)
```



Semantik

```
eval(e: Exp, env) =  e match
      case Const(v)        => v
      case Var(x)          => env(x)
      case Add(e1, e2)     => eval(e1, env) + eval(e2, env)
      case Mult(e1, e2)    => eval(e1, env) * eval(e2, env)
```

Die Semantik der Ausdruckssprache mit Variablen kann über ein synthetisiertes Attribut (der Wert) und ein ererbtes Attribut (die Umgebung) beschrieben werden.

Synthetisierte Attribute (rekursiv über den Baum berechnete Werte) und das Umgebungs-Konzept reichen hier also völlig aus, um die Ausdrucksauswertung zu beschreiben.

Attributierung: Bewertung

Attributierte Grammatiken: deklarative, Datenfluss-orientierte Spezifikation von Berechnungen in Bäumen

Erfunden und fast ausschließlich genutzt zur Spezifikation der Semantik von Programmiersprachen

Sehr mächtig und allgemein

Hier (und in der Praxis sonst meist auch) reichen i.d.R. synthetisierte und ererbte Attribute*

Die Attributierung kann in Programmiersprachen konkreter und einfacher spezifiziert werden als:

- **Synthetisierte Attribute: Rekursiv über die Struktur definierte Funktionen**
- **Ererbte Attribute: Umgebungs-Konzept**

Attributierte Grammatiken sind ein Werkzeug, das (nicht nur) im Compilerbau genutzt werden kann. Das wir – wie viele andere auch – aber nicht nutzen wollen, da die zu erledigende (einfache) Aufgabe den Einsatz des komplexen Werkzeugs nicht rechtfertigt.

****** Ein Beispiel für komplexere Abhängigkeiten, die über dieses Schema hinaus gehen, sind rekursive Definitionen (z.B. Prozedur-Definitionen) mit beliebigen gegenseitigen Abhängigkeiten.***

Traversierungen des AST

Reihenfolge der Traversierungen

Der AST muss meist mehrfach und in einer bestimmten Reihenfolge durchlaufen werden.

Die Kontextanlyse muss dazu geplant werden. Bei der Planung ist zu klären

- Welche Aktionen sind bei der Verarbeitung von Typdefinitionen, Variablendefinitionen, Parameterdefinitionen, Prozedurdefinitionen, Ausdrücken und Anweisungen auszuführen.
- Welche Abhängigkeiten bestehen dabei?
- Welche zyklischen Abhängigkeiten gibt es und wie können sie durchbrochen werden?

  Meist ist es hilfreich zuerst alle Prozedurdefinitionen zu verarbeiten bevor Ausdrücke und Anweisungen analysiert werden.
- Welche nicht-zyklischen Abhängigkeiten gibt es und wie kann die dann mögliche topologische Sortierung realisiert werden?

Traversierungen des AST

Kontextanalyse: Allgemeine Problemstellung Baum-Traversierung

Steht der Formalismus der attributierten Grammatiken nicht (oder nicht mit der notwendingen Funktionalität) zur Verfügung, dann müssen die Attribute „zu Fuß“ berechnet werden.

Die zentrale Phase in der die Attribute berechnet werden ist, die Kontextanalyse

Der AST muss bei der Kontextanlyse (und der Codegenerierung) mehrfach durchlaufen werden

Es empfiehlt sich darum die Traversierung als generische Aufgabe von den konkreten Algorithmen zu trennen. (Formalismen wie Attributierte Grammatiken bieten dazu sehr mächtige allgemeine Lösungen.)

Entwurfsmuster wie das Besuchermuster geben Hinweise, wie der Code zur Traversierung einer Datenstruktur gestaltet werden kann. (Siehe Wikipedia: https://en.wikipedia.org/wiki/Visitor_pattern**)**

Das Besuchermuster ist ein OO-Muster.

In funktionalen Sprachen (mit Pattern-Match) organisiert man Traversierungen auf andere Art.

Traversierungen des AST

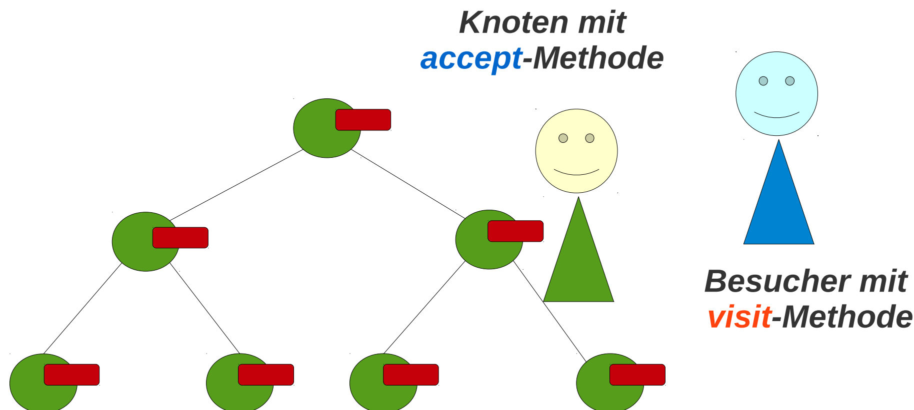
Generische Traversierungsalgorithmen

Besucher-Muster / Visitor-Pattern:

- Besucher, die die speziellen Aktionen auf den Knoten einer Datenstruktur ausführen
- werden zu den Knoten der Datenstruktur geführt

Struktur:

- Jeder Knoten hat eine Methode accept mit der er Besucher akzeptiert
- Ein Besucher hat die Methode visit mit einem Knten als Parameter
- Jeder Knoten führt mit seiner Methode accept folgende Aktionen aus:
  - es lässt den Besucher an seine Daten mit visitor.vist(this)
  - er führt den Besucher zu seinen Nachfolge-Knoten mit unterknoten.accept(visitor)



Traversierungen des AST

Generische Traversierungsalgorithmen

Besucher-Muster / Visitor-Pattern / Beispiel:

```scala
abstract class Visitor {
  def visit(exp: Exp): Unit
}

sealed abstract class Exp {
  def accept(visitor: Visitor)
}

case class Number(v: Int) extends Exp {
  override def accept(visitor: Visitor): Unit =
    visitor.visit(this)
}

case class Add(left: Exp, right: Exp) extends Exp {
  override def accept(visitor: Visitor): Unit = {
    left.accept(visitor)
    right.accept(visitor)
    visitor.visit(this)
  }
}
```

```scala
case object Evaluator extends Visitor {
  var stack = List[Int]()
  override def visit(exp: Exp): Unit = exp match {
    case Number(v) => stack = v :: stack
    case Add(l, r) => stack = stack match {
      case v1 :: v2 :: rest => v1+v2 :: rest
      case _ => throw new IllegalStateException(
      )
    }
  }
}


object Example {
  def main(args: Array[String]): Unit = {
    val e = Add(Number(1), Add(Number(2), Number(3)))

    e.accept(Evaluator)

    println(Evaluator.stack.head)
  }
}
```

Generische Traversierungsalgorithmen

Besucher-Muster / Visitor-Pattern / Beispiel:

```
abstract class Visitor {
  def visit(exp: Exp): Unit
}

sealed abstract class Exp {
  def accept(visitor: Visitor)
}

case class Number(v: Int) extends Exp {
  override def accept(visitor: Visitor): Unit =
    visitor.visit(this)
}

case class Add(left: Exp, right: Exp) extends Exp {
  override def accept(visitor: Visitor): Unit = {
    left.accept(visitor)
    right.accept(visitor)
    visitor.visit(this)
  }
}
```

```
case object Evaluator extends Visitor {
  var stack = List[Int]()
  override def visit(exp: Exp): Unit = exp match {
    case Number(v) => stack = v :: stack
    case Add(l, r) => stack = stack match {
      case v1 :: v2 :: rest => v1+v2 :: rest
      case _ => throw new IllegalStateException(
      )
    }
  }
}


object Example {
  def main(args: Array[String]): Unit = {
    val e = Add(Number(1), Add(Number(2), Number(3)))

    e.accept(Evaluator)

    println(Evaluator.stack.head)
  }
}
```

Traversierungen des AST

Generische Traversierungsalgorithmen

Besucher-Muster / Visitor-Pattern / Bewertung:

- **positiv**
  gute Modularisierung, Trennung generischer von speziellen Aktivitäten
- **negativ**
  Alle Knoten müssen ein voraus bekanntes Interface implementieren
  Wenig flexibel:
  - visit-Methoden müssen sich in gleicher Art verhalten:
    gleiche Reihenfolge (Pre-Order / In-Order / Post-Order)
  - gleicher Ergebnistyp für alle visit-Methoden

Traversierungen des AST

Generische Traversierungsalgorithmen

Alternativen zum Besucher-Muster / Visitor-Pattern

- **Traversable**
  Der AST kann traversierbare Datenstruktur gestaltet werden.
- **Iterable**
  Der AST kann iterierbare Datenstruktur gestaltet werden