Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Bezeichner, Namen und ihre Identifikation

- Bezeichner, Name,
- Definition, Gültigkeitsbereich, Bindung
- Bezeichner-Identifikation
- Symboltabellen, statische Umgebung
- Bezeichner-Identifikation und getrennte Übersetzung

Übersicht

Bezeichner

Namen für Dinge die im Programmtext definiert werden

Bezeichner-Identifikation

Klären: Welche Bezeichner beziehen sich auf das gleiche Ding (Gültigkeitsbereiche sind relevant)

Kontextanalyse

Bezeichner-Identifikation kann mit einer (kontextfreien) Grammatik nicht ausgedrückt werden.

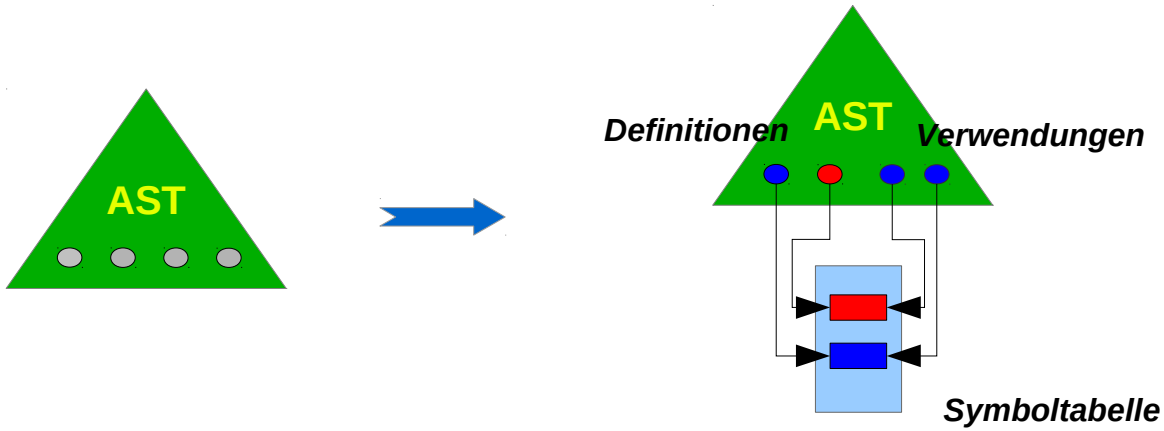
Symboltabelle

Verzeichnis aller Bezeichner (Symbol = Bezeichner)

Ziel der Bezeichner-Identifikation

Baum (AST) in Graph umwandeln – am Besten im Fluge schon durch den Parser

(einige Füße des Baums stecken in gemeinsamen Schuhen der Symboltabelle)



Definitionen: Variablen und Konstante

Begriffe: Literal / Name / Bezeichner / *Identifier*

Literale vs. Bezeichner

- **Literal: Token mit fester Bedeutung an jeder Stelle in jedem Programm einer Sprache.**

  Die Bedeutung der Literale wird vom Sprachdesigner festgelegt

  Beispiel: Zahlen *1, 2.5* , Sting-Literale *"hallo"*

- **Bezeichner: Token mit fester Bedeutung in einem bestimmten Bereich in einem bestimmten Programm**

  Die Bedeutung eines Bezeichners wird im Programm vom Programmierer festgelegt (im Gegensatz zu Schlüsselwörtern, deren Bedeutung fix ist.)

  Beispiel: *i, x_1, f, Person*

Bezeichner

- **Bezeichner bezeichnen etwas (*Identifier* identifizieren etwas),**

  das vom Programmierer mit einer Definition im Programmtext definiert wurde.
- **Bezeichner können sehr unterschiedliche Dinge bezeichnen:**
  - **Konstanten**
  - **Variablen**
  - **Funktionen**
  - **Klassen**
  - **Typen**
  - **.....**

Definitionen: Variablen und Konstante

Bezeichner / Name

Bezeichner vs Name

- **Bezeichner: Lexikalische Einheit:**
  Token mit definierter Bedeutung
- **Name als Sinn-tragende semantische Einheit:**
  Name für etwas, das im Programm benannt wurde

Name

- **Identifiziert etwas, das im Programm definiert wurde**
- **In einfachen Sprachen ist ein Name ein Bezeichner**
- **In komplexeren Sprachen hat ein Name oft auch einen komplizierteren Aufbau**
- **Z.B. Java:**
  ***my_package.test.Hugo* : Ein Name der durch Punkte getrennte Bezeichner enthält**

6.2. Names and Identifiers

A *name* is used to refer to an entity declared in a program.

There are two forms of names: simple names and qualified names.

A *simple name* is a single identifier.

A *qualified name* consists of a name, a "." token, and an identifier.

. . .

Aus *„Java Language Specification“*
http://docs.oracle.com/javase/specs/jls/se8/html/jls-6.html#jls-6.2

Name, Bindung, Gültigkeitsbereich

Name

Ein Name benennt etwas: ein Ding / eine Entität

Bindung: Name ~> benanntes Ding

Eine Bindung verknüpft einen Namen mit dem was er benennt.

Eine Bindung hat zwei Aspekte

- statisch: der textuelle Bereich in dem die Bindung gültig ist
- dynamisch: der Teil der Laufzeit (eines Programms) in dem eine Bindung gültig ist

Gültigkeitsbereich

Der Gültigkeitsbereich ist der textuelle Bereich in einem Programm, in dem eine Bindung gültig ist

Lebensdauer

Die Lebensdauer ist die Zeit in der ein Ding existiert. Solange ein Gültigkeitsbereich sich in Ausführung befindet, sollten alle in ihm gebundenen Dinge existieren. – Sonst haben Dinge die nicht existieren einen Namen.

Gültigkeitsbereich (engl. *Scope*)

Der Gültigkeitsbereich

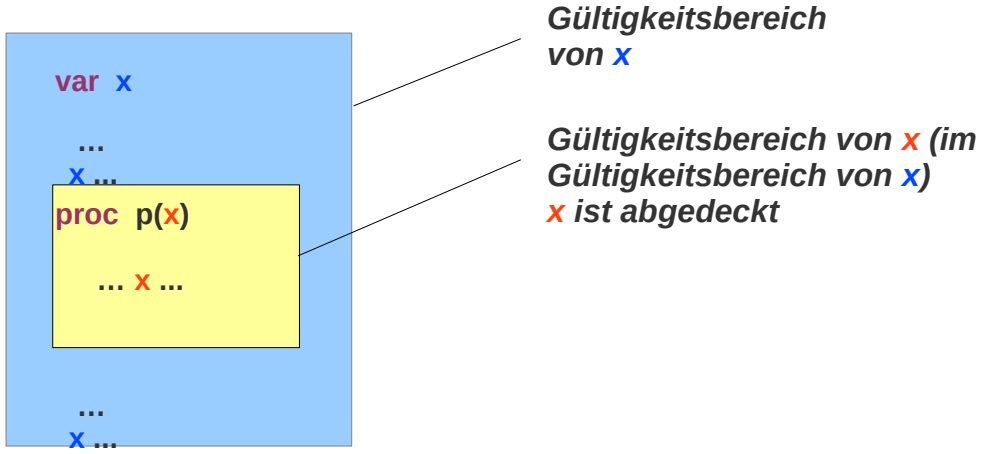
einer Definition ist der ***textuelle Bereich*** in dem die von der Definition / Deklaration erzeugte Bindung gültig ist.

Beispiel: Eine Variablendeklaration in einer Methode gilt nur in dieser Methode.

Sichtbar / Abgedeckt

In den meisten Sprachen gelten Abdeckungsregeln: Eine „innere“ Deklaration deckt eine äussere Deklaration in ihrem Sichtbarkeitsbereich ab.

Beispiel: Eine lokale Variable kann den gleichen Namen wie eine Objektvariable haben. Die Objektvariable ist im Bereich der lokalen Variable dann abgedeckt und nicht sichtbar.



Gültigkeitsbereich

Beispiel

```
object Fac_App {

  def f(n: Int): Int =
    if (n == 0) 1 else f(n-1) * n

  def main(args: Array[String]): Unit = {
    val x = scala.io.StdIn.readInt()
    val y = f(x)
    println(y)
  }

}
```

Gültigkeitsbereich von f

Gültigkeitsbereich von n

n hat viele (dynamische) Bindungen:
Eine für jeden Aufruf von f

Freie und gebundene Namen

Gebundene Namen

Die Namen die in einer Funktion (oder einem anderen Gültigkeitsbereich) lokal definiert werden, nennt man (in der Funktion, ...) gebundene Namen.

Freie Namen

Die Namen die in einer Funktion (oder einem anderen Gültigkeitsbereich) verwendet, aber nicht in ihr lokal definiert werden, nennt man freie Namen.

Freie Namen erhalten ihre Bedeutung aus dem Kontext. Entweder

- Aus dem statischen Kontext: durch den Programmtext bestimmt, oder
- Aus dem dynamischen Kontext: durch die Programmausführung bestimmt.

```
public class Binding {

  static int x = 42;

  static void f() {
     int y = 100;
     System.out.println( x + y);
  }

  public static void main(String[] args) {
     int x = 43;
     f();
  }
}
```

x *ist frei in f, – welches x ist gemeint?*
y *ist gebunden in f*

Gültigkeitsbereiche und Namensbindungen

Namensbindung

Beispiel Java:

```
public class JavaHasStaticNameBinding {

     static int a = 100;

     static int add(int b, int c) {
          return a+b+c;
     }

     public static void main(String[] args) {
          int a = -42;
          System.out.println(add(1,2));
     }

}
```

~> 103

Statische Namesbindung: Das a an der Definitionsstelle zählt

Dieses static hat mit statischer Namensbindung NICHTS zu tun.

Gültigkeitsbereiche und Namensbindungen

Namensbindung

Beispiel Java:

```
public class Binding {

  interface G { void call(); }

  static G f() {
    final int x = 42;
    return new G() {
      public void call() {
         System.out.println("x = " + x);
      }
    };
  }

  public static void main(String[] args) {
    G g = f();
    final int x = 47;
    g.call();
  }

}
```

x ist frei in call, – welches x ist gemeint?

```
public class Binding {

  interface I {
    int x = -100;
    void call();
  }

  public static void main(String[] args) {
    int x = 42;
    (new I(){
       public void call() { System.out.println(x); }
    }).call();
  }

}
```

x ist frei in call, – welches x ist gemeint?

Namensbindung: Verschachtelung / Nesting

Statische Namensbindung

Die Bedeutung eines Namens wird definiert über:

- **Den Gültigkeitsbereichs der Definition und**
- **Die Sichtbarkeitsregeln der Sprache**

Funktionen / Prozeduren und Gültigkeitsbereiche

Gültigkeitsbereiche können in den meisten Sprachen verschachtelt werden

Bei der Verschachtelung gibt es oft besondere Beschränkungen, wenn Funktionen beteiligt sind:

- **In C dürfen Funktionen nur global, nicht in irgendwelchen anderen Gültigkeitsbereichen definiert werden;**
- **In Java dürfen Funktionen (Methoden) nur in Klassen definiert werden (Klassen dürfen aber in Methoden definiert werden), indirekt dürfen also Methoden in Methoden definiert werden;**
- **In manchen Sprachen dürfen Funktionen (Methoden) an jeder Stelle definiert werden, an denen irgendeine Definition stehen darf;**
- **….**

Namensbindung: Sichtbar und gültig

Viele moderne Sprachen erlauben

- verschachtelte Prozedurdefinitionen und
- Prozeduren dürfen sich (gegenseitig) rekursiv aufrufen

Es gelten dabei die folgenden Regeln:

- Jeder Name ist in dem Gültigkeitsbereich **gültig**, in dem er definiert wird,
  Die Verwendungsstelle muss dabei nicht unbedingt textuell vor der Definitionsstelle liegen.
- Ein gültiger Name ist **sichtbar**, wenn er nicht nicht durch eine Definition mit gleichem Namen überdeckt wird.

Ein Name

- darf in einem Gültigkeitsbereich **nicht zweimal definiert** werden,
- es sei denn, die zweite Definition erfolgt in einem weiteren **inneren Gültigkeitsbereich**.
  In dem Fall überdeckt die zweite Definition die erste.

```
(function(){
    hello();
    function hello() { alert("Hello world"); }
  }())
```

Beispiel Javascript: Verwendung einer Funktion im definierenden Gültigkeitsbereich,textuell vor der Definition.

```
object Test extends App {
  var u: Int = v+1;
  var v:Int  = u+1;
  println(s"u =$u v = $v");
}
```

Beispiel Scala: Verwendung einer Variablen im definierenden Gültigkeitsbereich, textuell vor der Definition.

Gültigkeitsbereiche und Namensbindungen

Verschachtelung

Beispiele:

```c
#include <stdio.h>
#include <stdlib.h>

int next(void);  // declaration

int main(void) {
  printf("%d\n", next());
  printf("%d\n", next());
  printf("%d\n", next());
}

int next(void) { // definition
  static int c = 0;
  return c++;
}
```

C: Keine Verschachtelung, define-before-use. Funktionsdefinitionen nur auf oberster Ebene (global) und jeder Verwendung eines Namens muss dessen Definition vorausgehen. Dazu können Prototypen von Funktionen deklariert werden.

```java
import java.util.function.Function;

public class C {
  private int x;
  public C (int x) {this.x = x;}
  class D {
      public Function<Integer, Integer> m(int y) {
        return ( (z) -> x+y+z );
      }
  }

  public static void main(String[] args) {
    System.out.println(new C(1).new D().m(2).apply(3));
  }

}
```

Java: Verschachtelungs-Beispiel.

Definition / Deklaration

Definition / Deklaration

Eine Definition / Deklaration ordnet einem Namen eine Bedeutung zu

Die beide Begriffe werden meist synonym verwendet

Die Java-Spezifikation verwendet (nur) den Begriff „*Declaration*“

Deklaration vs Deklaration

In manchen Programmiersprachen muss man eine Unterscheidung machen

Z.B. in C / C++ wegen dessen Konzepts der getrennten Übersetzung:

- Deklaration: Zuordnung Name ~> Bedeutung des Namens für den Compiler
- Definition: Zuordnung Name ~> komplette Bedeutung des Namens:
  Speicherallokation wenn eine Variable definiert wird,
  Funktionskörper bei Funktionsdefinitionen.

Beispiel:

Modul A:

```
int i=5;  // Variablen-Definition, Compiler reserviert Speicherplatz im
          // zugehörigen Objektmodul
....
```

Modul B:

```
extern int i;  // Variablen-Deklaration, Compiler reserviert keinen Speicherplatz
               // in B wird die Variable aus A verwendet
....
```

Aus M. Jäger: „*Compilerbau*“

Getrennte Übersetzung

Getrennte Übersetzung

Von getrennter Übersetzung spricht man, wenn

- ein „Programm“ (*Executable Image*), der ausgeführte Code, entspricht nicht 1:1 einer
- Übersetzungseinheit (*Compilation Unit*), was der Compiler in einem Lauf übersetzt

Getrennte Übersetzung ist heute üblich

Übersetzungseinheit

Ein Programmiersprache definiert Übersetzungseinheiten

Die (syntaktischen) Einheiten, die der Parser verarbeiten kann

Beispiel Java (Java 8 Language Specification)

CompilationUnit is the goal symbol (§2.1) for the syntactic grammar (§2.3) of Java programs. It is defined by the following productions:

CompilationUnit:
[PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

Getrennte Übersetzung und Gültigkeitsbereiche

Problem

Der Gültigkeitsbereich eines Namens kann größer sein, als eine Übersetzungseinheit

Beispiel C

Der Linker löst alle offenen Bezüge (mit was immer er findet)

Deklarationen werden benutzt, damit der Compiler den richtigen Code erzeugt, wenn das Programm Bezüge auf „Externes“ enthält wie z.B.:

- **getrennt übersetzte Funktion**
- **getrennt übersetzte Variable**

```
extern int x;
void f(int i);

int main () {
    int a = x+1;
    f(a);
}
```

```
Invoking: GCC C Compiler
gcc -O0 -g3 -Wall -c ... "src/MyAnsiCProject.o" "../src/MyAnsiCProject.c"

Invoking: MacOS X C Linker
gcc  -o "MyAnsiCProject"  ./src/MyAnsiCProject.o
Undefined symbols for architecture x86_64:
  "_f", referenced from:
      _main in MyAnsiCProject.o
  "_x", referenced from:
      _main in MyAnsiCProject.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Der Programmcode muss (möglichst korrekte) Info über Definitionen / Namen anderer Übersetzungseinheiten enthalten.

Den Compiler interessiert nur diese Info. Wo und ob die referenzierten Entitäten überhaupt existieren und ob die Deklarationen korrekt sind, interessiert ihn nicht.

Getrennte Übersetzung und Gültigkeitsbereiche

Beispiel Java

Der Compiler löst alle offen Bezüge auf (mit dem was er auf dem Klassenpfad findet), um die für eine korrekte Übersetzung notwendige Information zu finden

Der Linker wird zur Laufzeit aktiviert, wenn eine Klasse geladen wird.

```
public class This_CU {

  public static void main(String[] args) {
    int a = 1;
    Other_CU.f(a);
  }

}
```

Der Programmcode kann Bezüge zu Namen aus anderen Übersetzungseinheiten enthalten.

Der Compiler

- **prüft ob diese (auf dem Klassenpfad) vorhanden sind**
- **übersetzt sie wenn das noch nicht geschehen ist**
- **extrahiert die Info, die er benötigt um diese CU korrekt übersetzen zu können**

Namensräume

Namensraum und Gültigkeitsbereich / Bindung

Der Begriff „Namensraum“ ist eng mit den Begriffen „Gültigkeitsbereich“ und „Bindung“ verwandt.

Namensräume dienen der Organisation von Namen.

Ein Namensraum ist eine Sammlung von Namen

Beispiel Java

Ein *package* definiert einen Namensraum der Übersetzungeinheiten beinhaltet.

Eine import-Anweisung macht die sichtbaren Namen eines *packages* in abgekürzter Form sichtbar

```
package blah;

import static blah.Other_CU.f;

public class This_CU {

  public static void main(String[] args) {
    int a = 1;
    f(a);
  }

}
```

~

```
package blah;

//import static blah.Other_CU.f;

public class This_CU {

  public static void main(String[] args) {
    int a = 1;
    Other_CU.f(a);
  }

}
```

Importe (in Java) importieren nichts, sie öffnen einen Namenraum in einem Gültigkeitsbereich und machen damit (im Wesentlichen) abgekürzte Versionen von Namen sichtbar, die schon sichtbar sind.

Compiler und Namen

Lexikalische Analyse

Für den Scanner sind Namen Bezeichner, die i.d.R. durch reguläre Ausdrücke beschrieben

Syntax-Analyse

Für die Parser sind Bezeichner eine Token-Klasse

Kontext-sensitives Parsen

Sehr oft verwendet der Parser kontextsensitive Information

d.h. Information, die nicht durch eine kontextfreie Grammatik ausgedrückt werden kann.

Parsen mit Bezeichner-Identifikation

Der Parser verarbeitet kontextsensitive Information über Bezeichner

Ein Bezeichner wird zu einem Namen indem der Parser

- entsprechend der Sichtbarkeits- und Scope-Regeln
- jeden Bezeichner der passenden Definition / Deklaration zuordnet

Damit wird der Aufbau des AST erleichtert

Die Namens- / Bezeichner-Identifikation

Symboltabelle

„Symboltabelle“ ist ein historischer Begriff aus der Zeit als Bezeichner und Namen noch „Symbole“ genannt wurden.

Die Identifikation der Bezeichner = Aufbau der Symboltabelle

Namens-Identifiktation / Namens-Auflösung / bezeichner-Identifikation

Die korrekte Zuordnung

- der Verwendung eines Namens zu
- der richtigen Definition

nennt man Namens- / Bezeichner-Identifikation oder Namens-Auflösung

Wird die Namens-Identifikation als *semantische Aktion* im Parser implementiert, dann spricht man i.d.R. von Bezeichner-Identifikation. (*Bezeichner* ist ein syntaktischer Begriff, *Name* ist eher ein semantischer Begriff.)

Namens-Identifikation: Compilezeit- / statische und Laufzeit- / dynamische Aktionen

In Sprachen mit rekursiven Prozeduren

- sind i.A. mehrere Instanzen der gleichen Definition gleichzeitig aktiv

darum muss hier die Namens-Identifikation aufgeteilt werden:

- Zur Compilezeit kann die richtige Definition identifiziert werden
- Zur Laufzeit erst kann die richtige Instanz der Definition,
  d.h. die aktuelle Speicherstelle, identifiziert werden (das ist später ein Thema)

Namens-Identifikation: Wann und Wer

Die statische Namens-Identifikation (Namens-Auflösung) kann auf verschiedne Arten und zu verschiedenen Zeiten während der Compilation erfolgen:

Variante 1: Als Aktion des Parsers

Der Parser führt während der Syntaxanlyse eine Identifikation der Bezeichner aus. Dies gilt als semantische Aktion des Parsers. Hierbei wird eine Symboltabelle gefüllt und abgefragt.

Der Parser kann nur dann Namen identifizieren, wenn die Definition / Deklaration stets vor der Verwendung geparst wird. (Das ist der Grund für die Prototypen in C)

Variante 2: Als Aktion der Kontextanalyse / semantischen Analyse

Die Namen werden nach dem Parsen identifiziert. In einer eigenen Phase des Compilers, der Kontextanalyse, wird der AST durchlaufen und modifiziert. Dabei kann jede Verwendung eines Namens mit der passenden Definition verknüpft werden. Entweder als Verlinkung im AST oder besser als Verlinkung zu einem Eintrag in einer Symboltabelle

Dieses Verfahren ist aufwendiger aber wesentlich flexibler als die Identifikation der Namen durch den Parser.

Gemischte Verfahren

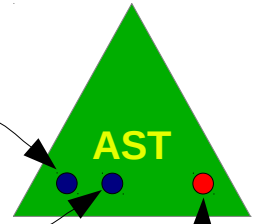
Beide Verfahren können kombiniert werden: Der Parser sammelt nutzt die Informationen, die er zur Verfügung hat. In der Kontextanalyse wird dann der AST durchlaufen und die Identifikation beendet.

Bezeichner- / Namens-Identifikation: Wer, Wann

Definition / Verwendung: Kontextsensitive Abhängigkeiten bei Bezeichnern schon beim Parsen auflösen

```
VAR   s := 0;
```
⇒
```
VarDef("s", NumericConst(0.0))
```

```
CONST c := 5;
```
⇒
```
ConstDef("c", NumericConst(5.0)))
```



Mit einer kontextfreien Grammatik können derartige Abhängigkeiten nicht definiert werden. Ein entsprechender Parser kann sie darum nicht beachten.

```
s := s + c;
```
⇒
```
Assign(Var("s"), Plus(DeRef(Var("s")), Const("c"))))
```

s ist hier eine Variable, c eine Konstante. Der Parser muss dies wissen, um einen korrekten AST für die Ausdrücke „s“ und „c“ in der Zuweisung erstellen zu können!

Dass es sich bei den Bezeichnern „s“ und „c“ um eine Variable bzw. Konstante handelt kann nicht mit einer Grammatik ausgedrückt werden und so auch nicht durch eine Syntaxanalyse festgestellt werden. – Hierzu werden „semantische Aktionen“ benötigt.

```
sealed abstract class ValueExp
...
case class DeRef(lexp: ReferenceExp)        extends ValueExp
case class Const(name: String)              extends ValueExp
...
```

Identifikation von Bezeichnern / Namen: Wer, wann

Die Zuordnung von Verwendungs- und Definitionsstelle kann auf verschiedne Art erfolgen:

- **während der Syntaxanalyse**

  Der Parser verwaltet eine Tabelle mit allen definierten Entitäten und kann im AST die richtigen Knoten erzeugen

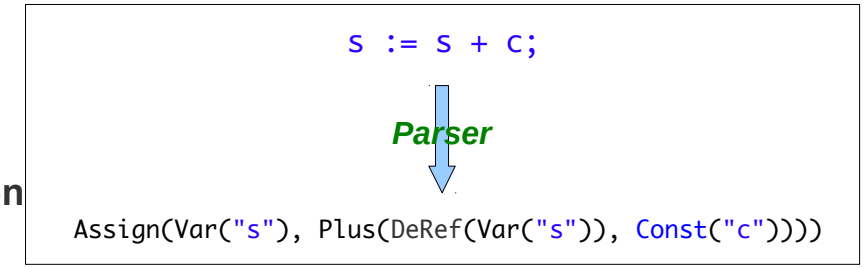
- **während der Kontextanalyse**

  Der Parser erzeugt einen AST in dem „vorläufige“ Bezeichner-Knoten auftreten.

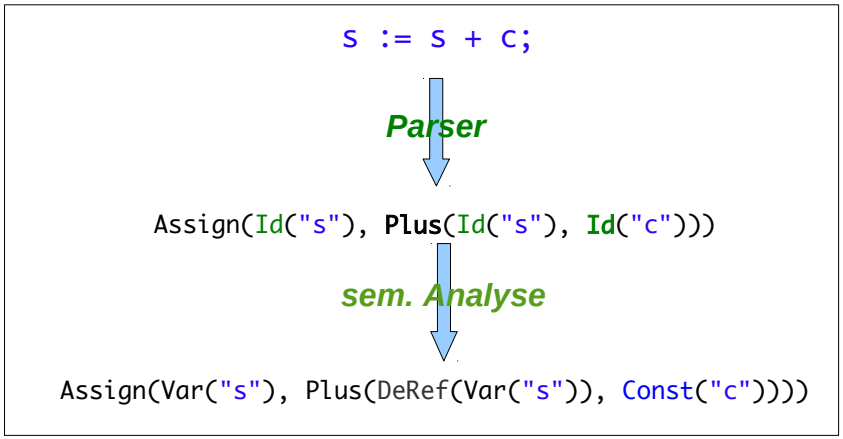
  In semantischen Analyse / Kontextanalyse wird der Ast analysiert und und die Bezeichner-Knoten in Const-Knoten oder DeRef-Knoten transformiert

- **während der Interpretation / Codegenierung**

  Der Parser erzeugt einen AST in dem Bezeichner-Knoten auftreten. Die passende Definition wird zur Laufzeit identifiziert.



Parsen mit Bezeichneridentifikation



Bezeichneridentifikation in der Kontextanalyse

Identifikation von Bezeichnern: Wer, Wann

übliche Verfahren

- **Compiler**
  während der Syntaxanalyse
- **Interpreter**
  während der Syntaxanalyse
  oder bei der Interpretation des AST

```
s := s + c;
```

Parser

```
Assign(Var("s"), Plus(DeRef(Var("s")), Const("c"))))
```

oder

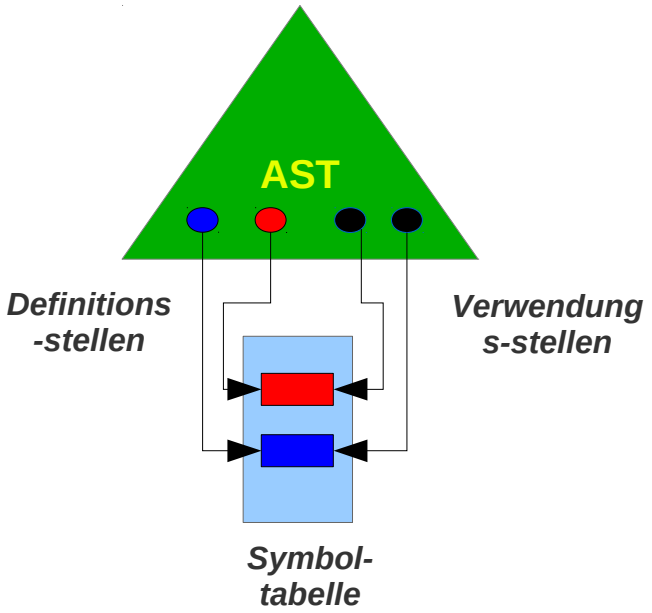
```
Assign(Id("s"), Plus(Id("s"), Id("c")))
```

AST mit Symboltabelle

Ziel der Identifikation der Bezeichner durch den Parser

- Der vom Parser erzeugte AST enthält Verweise auf Einträge der Symboltabelle
- Die Symboltabelle enthält einen Eintrag pro Definition
- Jede Verwendungsstelle verweist auf den Eintrag, der aus der Definitionsstelle erzeugt wurde.

Die Symboltabelle ist eine beliebige Datenstruktur, die Teil des AST ist (nicht zwingend eine Tabelle)



Parsen / AST mit Symboltabelle

Zur Identifikation der Bezeichner wird in jedem Fall eine Abbildung / Tabelle benötigt, in der die Effekte der Definitionen gespeichert werden.

Symboltabelle

Werden Bezeichner durch den Parser identifiziert, dann nennt man die dabei verwendete Abbildung *Name → Info-zum-Name* „Symboltabelle“. („Symbol“ altertümlich für „Bezeichner“)

Semantische Aktion

Aktionen,

- **die der Parser während der Syntaxanalyse ausführt**
- **und die nichts mit dem Parsing direkt zu tun haben**

nennt man semantische Aktionen (des Parsers)

Der

- **Aufbau des AST und das**
- **Manipulieren der Symboltabelle**

sind semantische Aktionen.

Implementierung der Bezeichner-Identifikation

Definition-Vor-Verwendung

Wenn der Parser

- **vor jeder Verwendung eines Bezeichners die zugehörige Definition gesehen hat,**
- **dann kann die Symboltablle recht einfach während des Parsens erstellt werden**

Verwendung-Vor-Definition

Manche Sprachen erlauben es Namen textuell nach ihrer ersten Verwendung zu definieren

Das macht die Bezeichner-Identifikation durch den Parser schwieriger, aber nicht unmöglich.

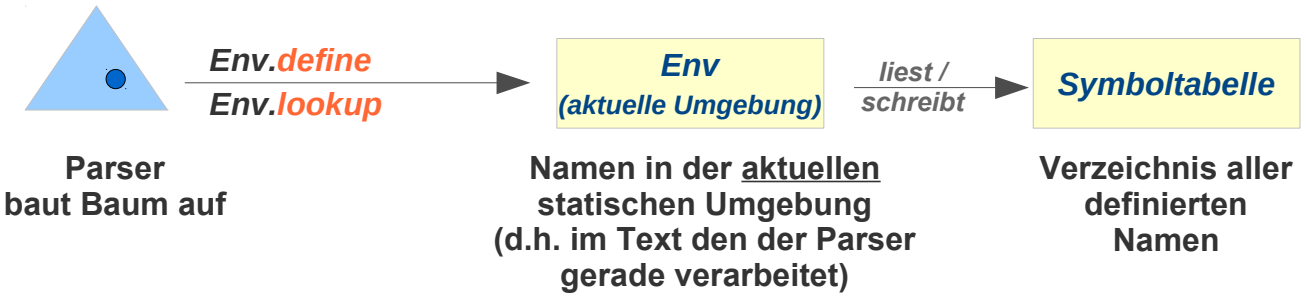
Diese Möglichkeit wird aber hier nicht in Betracht gezogen.

Implementierung der Bezeichner-Identifikation

Implementierung als semantische Aktion des Parsers

Statische Umgebung: Datenstruktur, die Gültigkeitsbereiche repräsentiert.

Pro Scope werden die in diesem Scope gültigen Namen verwaltet



Definition: Namen werden in der Umgebung und der Symboltabelle angelegt, bzw. bei Doppeldefinitionen abgelehnt.

Verwendung: Namen werden unter Beachtung der Gültigkeitsbereiche gesucht

Betreten eines Scopes: Env wird informiert, dass der aktuelle Scope ein neuer Scope ist

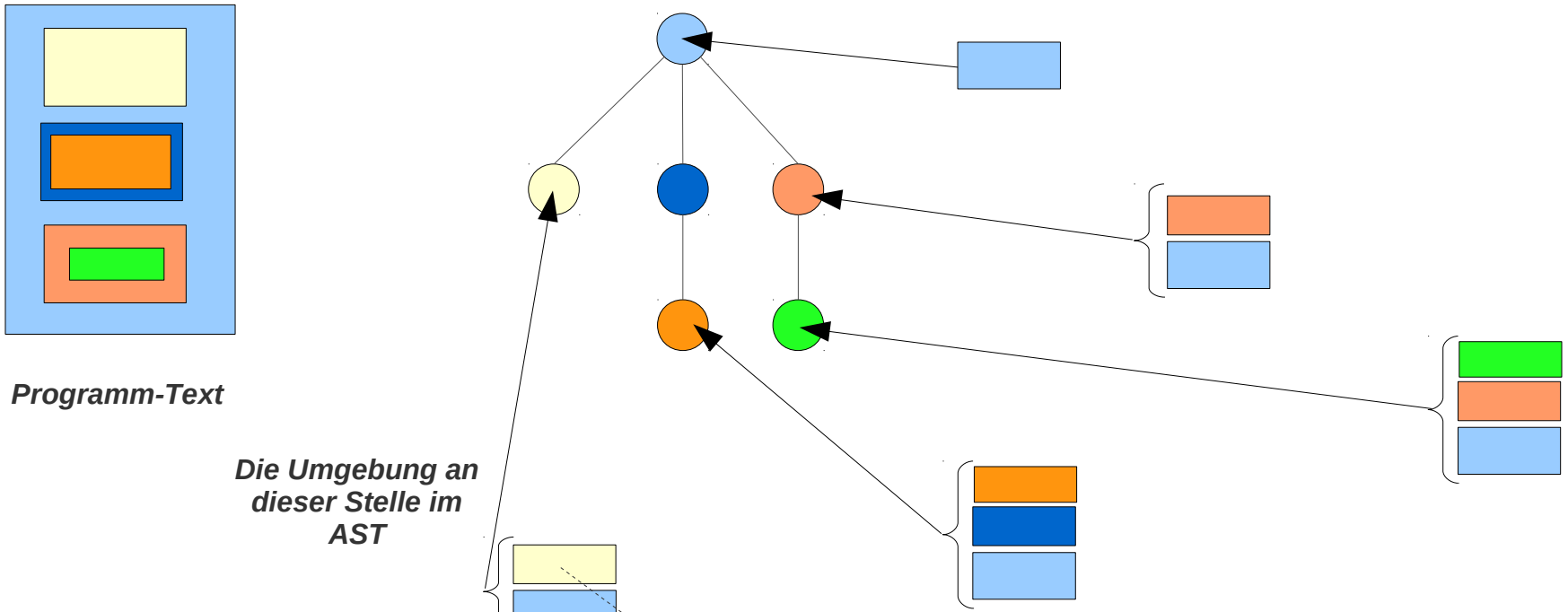
Verlassen eines Scopes: Env wird informiert, dass der aktuelle Scope obsolet ist

Umgebungen – Namens-Kontexte während eines Baumdurchlaufs

Namen treten stets in einer bestimmten Umgebung auf,

diese sind in der Regel ihre verschachtelten der Gültigkeitsbereiche

Eine Umgebungen entsprecht einem Pfad zur Wurzel im AST durch die Gültigkeitsbereiche.



Namens-Identifikation einphasig

Definition stets vor Verwendung: Ein Baumdurchlauf reicht

Die Identifikation der Namen kann komplett in einem Baumdurchlauf mit Hilfe von Umgebungen abgewickelt werden,

vorausgesetzt, dass alle Namen vor ihrer Verwendung definiert werden.

Symboltabelle

Die Symboltabelle kann problemlos während des Baumdurchlaufs aufgebaut werden:

Die in den Umgebungen gesammelten Informationen werden in die Symboltabelle übernommen

Die Symboltabelle muss nicht unbedingt als eigenständige Datenstruktur erzeugt werden, sie kann eventuell durch Menge der Symbole im AST implizit gebildet werden.

einphasige Bezeichner-Identifikation

Umgebungen: Namens-Identifikation mit einem Stapel lokaler Umgebungen

Die Namens-Identifikation kann mit einem Stapel lokaler Umgebungen realisiert werden:

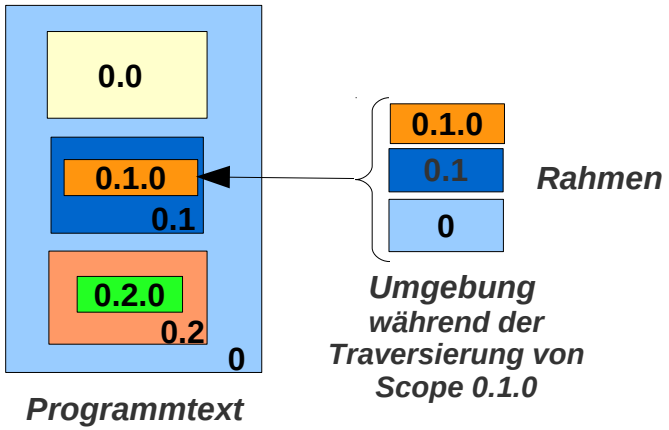
Pro Scope wird

- eine Abbildung (statischer Rahmen / Frame) Name → Definition angelegt
- die alle Definitionen für diesen Scope enthält

Traversierung des Baums

Wird beim Durchlaufen des Programmbaums (durch den Parser oder in einer extra Phase)

- **Ein Scope (Gültigkeitsbereich) betreten**
  Ein neues Stapelelement (Rahmen) wird erzeugt für alle Definitionen des Scopes
- **Ein Scope verlassen**
  Das oberstes Stapelelement (Rahmen) gelöscht
- **Ein Name definiert**
  Ein Eintrag im obersten Rahmen des Stapels wird angelegt
- **Ein Name gesucht**
  Der Stapel wird von oben nach unten durchsucht



Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

AST : Die Blätter eines AST können jetzt Symbole sein

```
object AST {

  import ProgSymbols._

  ...

  // Expression that denote int values
  sealed abstract class ArithExp
  case class Number(d: Int)                  extends ArithExp
  ...
  case class Ref(ref: RefExp)                extends ArithExp

  // Expression that denote storage locations
  sealed abstract class RefExp
  case class VarRef(symb: VarSymbol)   extends RefExp

  ...

  // Definitions
  sealed abstract class Definition {
    val symb: ProgSymbol
  }
  case class VarDef(override val symb: VarSymbol) extends Definition

  ...

}
```

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Symbole : Alles, was definiert werden kann, ist ein Symbol

Definition = statische (*Compile-Time*) Zuordnung: Name ~> Wert

```
object ProgSymbols {

  // all entities defined in a program (i.e. names with a compile time value) are symbols
  sealed abstract class ProgSymbol {
    val name: String
  }

  // Variables
  case class Variable(override val name: String) extends ProgSymbol

}
```

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Parser : Parser erzeugt AST mit Symbolen:

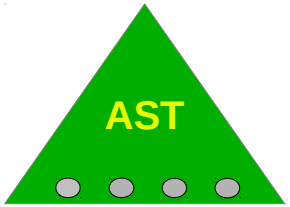
```
object ProgParsers extends TokenParsers {

  import AST._
  import ProgSymbols._

  type Tokens = ProgTokens

  override val lexical : ProgLexical = new ProgLexical

  import lexical._

  val num_any : Parser[Any] =
    elem("number", _.isInstanceOf[NumberToken])

  val id_any : Parser[Any] =
    elem("identifier", _.isInstanceOf[IdentToken])
...
  def factor: Parser[ArithExp] =
    number   |
    LeftPToken("(") ~> arithExp <~ RightPToken(")") |
    refExp

  def number: Parser[Number] =
    num_any   ^^ { case (x: Any) => Number(x.asInstanceOf[NumberToken].chars.toInt)}

  def refExp: Parser[Ref] = lExp ^^ {le => Ref(le) }

  def lExp: Parser[RefExp] = ident

  def ident: Parser[RefExp] =
    id_any   ^^ { case (x: Any) => VarRef(Variable(x.asInstanceOf[IdentToken].chars)) }
```

Implementierung der Bezeichner-Identifikation

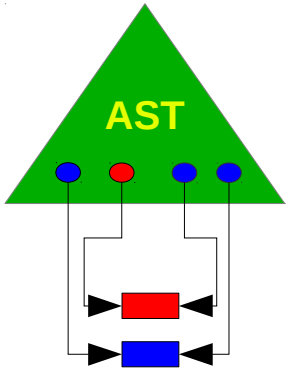
Beispiel Einfache Sprache mit Variablendefinitionen

Parser : Parser erzeugt AST mit Symbolen:

```
object ProgParsers extends TokenParsers {

  import AST._
  import ProgSymbols._

  type Tokens = ProgTokens

  override val lexical : ProgLexical = new ProgLexical

  import lexical._

  val num_any : Parser[Any] =
    elem("number", _.isInstanceOf[NumberToken])

  val id_any : Parser[Any] =
    elem("identifier", _.isInstanceOf[IdentToken])
...
  def factor: Parser[ArithExp] =
    number   |
    LeftPToken("(") ~> arithExp <~ RightPToken(")") |
    refExp

  def number: Parser[Number] =
    num_any   ^^ { case (x: Any) => Number(x.asInstanceOf[NumberToken].chars.toInt)}

  def refExp: Parser[Ref] = lExp ^^ {le => Ref(le) }

  def lExp: Parser[RefExp] = ident

  def ident: Parser[RefExp] =
    id_any   ^^ { case (x: Any) => VarRef(Variable(x.asInstanceOf[IdentToken].chars)) }
```



Die Variablen sind ohne Bezug zueinander: es fehlt die Identifikation



Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Definitionstelle und Verwendungsstelle verknüpfen:

Statische Umgebung zur Verwaltung der Definitionen / Symbole

```
import scala.util.Try

trait StaticEnv {

  /**
    * Enter a new definition context. All names defined in this context have to be unique.
    * All names defined in this context supersede names defined in prevous / outer contexts
    */
  def enterScope(): Unit

  /**
    * Leave the current definition context. All names defined in the current context are invalidated.
    */
  def leaveScope(): Unit

  /**
    * Define a name within the current context
    * @param name the name
    * @param symb the symbol to be associated with the name
    * @return     Success(_) if symbol may be definied in this context
    *             Failure(_) if the symbol may not be defined in this context
    *             (it may already be defined)
    */
  def define(name: String, symb: ProgSymbol): Try[Unit]

  /**
    * Lookup a name within the current context
    * @param name the name to be looked-up
    * @return  Success(symbol) if the symbol that is associated with the name within the current context
    *          Failure(_) if no symbol is associated  with the name within the current context
    */
  def lookup(name: String): Try[ProgSymbol]
}
```

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Bezeichner-Identifikation im Parser – 1: Scopes / Definitionen

```
val env: StaticEnv = new EnvImpl
```

```
def prog: Parser[Prog] =
  progStart ~> body ^^ { case (defList, cmdList) => Prog(defList, cmdList) }


// enter scope when keyword PROGRAM appears
def progStart: Parser[Any] =
  KwToken("PROGRAM") ^^ { x => env.enterScope(); x }


// leave scope after parsing body
def body: Parser[(List[Definition], List[Cmd])] =
  rep(definition) ~ (KwToken("BEGIN") ~> rep(cmd) <~ KwToken("END")) ^^ { case defs ~ cmds =>
    env.leaveScope
    (defs, cmds)
  }


// store symbols in env
def definition: Parser[Definition] =
  (KwToken("VAR") ~> ident) ~ (AssignToken(":=") ~> arithExp <~ SemicolonToken(";"))  ^^ { case name ~ e =>
    val symbol = new Variable(name)
    env.define(name, symbol)
    VarDef(symbol)
  }
```

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Bezeichner-Identifikation im Parser – 2: Verwendungsstellen

```
val env: StaticEnv = new EnvImpl
```

```
def factor: Parser[ArithExp] =
  number   |
  LeftPToken("(") ~> arithExp <~ RightPToken(")") |
  refExp

def refExp: Parser[Ref] =
  lExp ^^ {le => Ref(le) }

// check definition of identifier
def lExp: Parser[RefExp] =
  ident ^?  ({
    case name
     if env.lookup(name).isSuccess &&
        env.lookup(name).get.isInstanceOf[Variable] =>
           VarRef(env.lookup(name).get.asInstanceOf[Variable])
  },
    name => s"undefined name '$name'"
  )

def ident: Parser[String] =
  id_any   ^^ { case (x: Any) => x.asInstanceOf[IdentToken].chars }
```

```
def ^?[U](f: PartialFunction[T, U]): Parser[U]
```

A parser combinator for partial function application.

p ^? f succeeds if p succeeds AND f is defined at the result of p; in that case, it returns f applied to the result of p.

f	a partial function that will be applied to this parser's result (see **mapPartial** in **ParseResult**).
returns	a parser that succeeds if the current parser succeeds *and* **f** is applicable to the result. If so, the result will be transformed by **f**.

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Statische Umgebung / Implementierung – 1

```
import scala.util.{Failure, Success, Try}

class EnvImpl extends StaticEnv {
  import collection.mutable.Map

  private trait Frame {
    def lookup(name: String) : ProgSymbol
    def define(name: String, value: ProgSymbol): Unit
  }

  private case object BottomFrame extends Frame {
    override def lookup(name: String) : ProgSymbol = throw new Throwable(s"$name is not defined")
    override def define(name: String, value: ProgSymbol): Unit =
          throw new Throwable(s"$name can not be defined outside a scope")
  }

  private case class DerivedFrame(val base: Frame) extends Frame {
    val entries : Map[String, ProgSymbol] = Map()

    override def lookup(name: String) : ProgSymbol =
      entries.get(name) match {
        case None => base.lookup(name)
        case Some(x) => x
      }


    override def define(name: String, symb: ProgSymbol): Unit = {
      entries.get(name) match {
        case Some(x) => throw new Throwable("$name already defined")
        case None => entries += (name -> symb)
      }
    }
  }
```

Implementierung der Bezeichner-Identifikation

Beispiel Einfache Sprache mit Variablendefinitionen

Statische Umgebung / Implementierung – 2

```
  def enterScope: Unit = {
     actualFrame = new DerivedFrame(actualFrame)
   }

   def leaveScope: Unit = {
     actualFrame match {
       case DerivedFrame(previousFrame) => {
         actualFrame = previousFrame
       }
       case _ =>
     }
   }

   def define(name: String, symb: ProgSymbol): Try[Unit] = Try {
     actualFrame.define(name, symb)
   } recoverWith {
     case e: Throwable =>
       new Failure(e)
   }

   def lookup(name: String) : Try[ProgSymbol] = Try {
     actualFrame.lookup(name)
   } recoverWith {
         case e: Throwable =>
           new Failure(e)
     }

}
```

Zusammenfassung: Namensbindungen / Bezeichner-Identifikation

Umgebung / Environment / Context

Modellierung der Namensbindung

- Zur Modellierung der Namensbindung in einem Programm verwendet man Abbildungen
- Diese werden Umgebung / *Environment* genannt

Abbildungen: Name ~> Bedeutung des Namens

- Finden sich in praktisch jeder Programmiersprache
- Zuordnungen: Name ~> Bedeutung
  - Sind als Abbildungen zu betrachten
  - Und bei der Auswertung von Ausdrücken (und Anweisungen, …) zu beachten
- Ein Name hat nicht unbedingt an jeder Stelle im Programm die gleiche Bedeutung
  Sie hängt* davon ab, wo der Name im Programmtext auftaucht.

Umgebung

- Namen werden unter Beachtung / in der Umgebung von Definitionen ausgewertet:
  In welchen Gültigkeitsbereichen tritt der Name auf, welche Definitionen gelten dort.
- Die Abb.: Name ~> Bedeutung wird darum allgemein
  Umgebung / engl. Environment (oder auch Context)
  genannt.
- Die Umgebung ist* ein statisches Konzept: Sie kann zur Übersetzungszeit bestimmt werden.

* bei normalen Sprachen, also solchen mit statischer Namensbindung

Zusammenfassung: Namensbindungen / Bezeichner-Identifikation

Namens-Bindung und -Identifikation

Art der Namens-Bindung: Sprachdefinition

Jede Programmiersprache definiert eine Namensbindung.

Diese legt fest, wie die Bedeutung eines Name in einem Programm bestimmt wird

Namens-Identifikation: Compiler

Der Compiler kann zu jedem Namen die zugehörige Definitionen bestimmen:

Er muss nach den Regeln der Sprache „die Namensbindung auflösen“

(Statische) Umgebung: Modellierung der Namensbindung

Umgebungen modellieren die unterschiedlichen Namensbindungen an verschiedenen Stellen im Programm als Abbildungen. Ein Parser der benötigt eine Implementierung der statischen Umgebung zur Bezeichneridentifikation.

Umgebung: Datenstruktur die (nur) während der Bezeichneridentifikation benötigt wird.

Symboltabelle: Compiler

Die Symboltabelle enthält alle Informationen zu allen Namen, die zur Compilerzeit berechnet werden können.

Symboltabelle: Erweiterung / Modifikation des AST; eine Datenstruktur die während der Bezeichnidentifikation aufgebaut wird und nach der Bezeichneridentifikation exisitiert.