Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Typen und statische Typisierung

- Statische und dynamische Typen
- Typsysteme
- Typprüfung und Typisierung

Dynamische Typen = Typen von Werten

Werte haben Typen / gehören zu einem Typ

Der Typ bestimmt

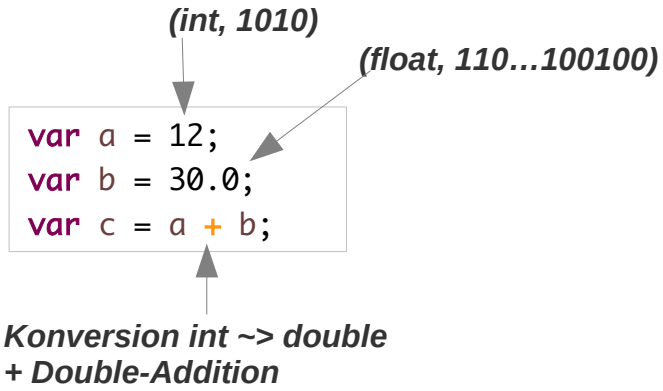
- Welche Werte es gibt
- Welche Operationen auf den Werten
  - möglich sind und
  - welche Art von Ergebnis sie haben

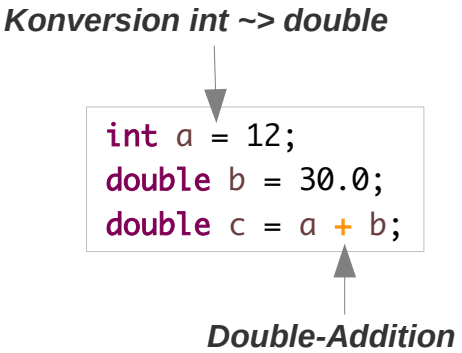
Typ-Markierungen oder statische Typen

Im Rechner gibt es nur Bits

Um Werte korrekt zu verarbeiten muss

- entweder jeder Repräsentant eines Wertes (eine Bitsequenz) eine Typ-Markierung tragen
- oder es muss zur Übersetzungszeit (statisch) klar sein, welchen Typ ein Wert hat





Statische Typen = Typen von Ausdrücken

Typen für Ausdrücke, d.h. für Programm-Texte

- jedem Ausdruck kann ein Typ zugeordnet werden: der statische Typ des Ausdrucks

Statische Typen – Typen von Ausdrücken im Quellcode

- Aktionen, die In Zusammenhang mit Typen stehen
- Dieses Konzept erfasst Dinge „die man dem Programmtext ansieht, ohne ihn ausführen zu müssen“

Konversion int ~> double

```
int a = 12;
double b = 30.0;
double c = a + b;
```

Double-Addition

Das Literal 12 *(Text!) hat den (statischen) Typ* int. *Die Namen* a *und* b *haben den (statischen) Typ* double*!*

```
double a = 12.0;
String b = "30.0";
double c = a + b;
```

Das wird garantiert schiefgehen, das Programm muss gar nicht erst gestartet werden.

Der Name a *hat den (statischen) Typ* double*, b hat den (statischen) Typ* String*.*

Statische Typen = Typen von Ausdrücken

Typen statisch verarbeiten

- Typ-Informationen können oft schon vor der Laufzeit – vom Compiler – ausgeführt werden (auf Typ-Fehler prüfen, Auswahl der richtigen Maschinen-Operationen, z.B. für '+')
- Das spart (eventuell viele) Aktionen und Speicherplatz (Typmarkierungen) zur Laufzeit

Statisch typisierte Sprachen – auch einfach typisierte Sprachen

- Verarbeiten Typ-Informationen zur Übersetzungszeit
- Vermeiden manchen Laufzeitfehler
- Machen aber gelegentlich einen korrekten Programmlauf unmöglich
- Erfordert vom Programmierer das Verständnis oft komplexer Typ-Systeme

Statische Typen und dynamische Typen

Dynamische Typen sind die Typen von **Werten**

Statische Typen sind die Typen von **Ausdrücken**

In einfachen Sprachen
- hat ein **Ausdruck**
  - zur Laufzeit einen **Wert** mit einem **Typ**
- der exakt gleich dem statischen **Typ** des **Ausdrucks** ist.

In anspruchsvolleren Sprachen
- hat ein **Ausdruck**
  - zur Laufzeit einen **Wert** mit einem **Typ**
- der zum statischen **Typ** des **Ausdrucks** kompatibel („kleiner-gleich“) ist.

```
int a = 12;
double b = 30.0;
double c = a + b;
```

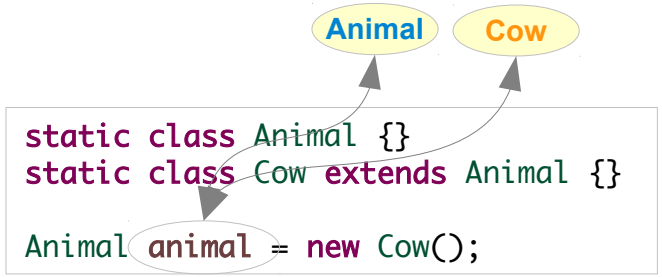
„a + b“ Wert : 42.0 mit dynamischem Typ double

„a + b“ statischer Typ: double =

```
int a = 12;
double b = 30.0;
double c = a + b;
```

12 Wert : 12 mit dynamischem Typ int

a statischer Typ: double ≤



In Sprachen mit flexiblem statischen Typsystem wird die Sache richtig interessant

Zweck des Typsystems / der statischen Typisierung

Sicherheit

Vermeidung von Laufzeitfehlern durch Typisierung.

- **Der Compiler oder das Laufzeitsystem lehnen Konstrukte ab, die zur Laufzeit (eventuell) zu Typfehlern führen**
- **Eine Sprache ist typsicher, wenn ein Programm niemals wegen eines Typfehlers ein unerwünschtes oder fehlerhaftes Verhalten zeigen kann**
- **C ist nicht typsicher**

  Casts / Pointer-Arithmetik setzen das Typsystem ausser Kraft, alles kann passieren
- **Java ist (im Wesentlichen) typsicher**

  Nur in komplexen Situationen und in Zusammenhang mit mehreren Klassenladern kann es u.U. zu einem unerwünschten Verhalten kommen

Zweck des Typsystems / der statischen Typisierung

Sicherheit

Beispiel

```
class A {}
class B extends A {
   public int val = 42;
}

A a = new B();      // upcast OK
((B)a).val = 43;    // downcast erlaubt (a ist ein B) und OK

B b = (B)(new A()); // downcast erlaubt aber Exception
```

Manche Typfehler können erst zur Laufzeit entdeckt werden. Der Compiler erzeugt dann (hoffentlich) die notwendigen Prüfungen.

Die geworfene Exception ist weder unerwünscht noch fehlerhaft: Der reale Typfehler wird als Fehler zur Laufzeit entdeckt und mit einer Exception behandelt: Java ist typsicher!

```
union {
     long i;
     char * cp;
} u;

u.i = 123456;

printf("%d\n", (int)*u.cp);
```

Der (reale) Typfehler wird weder vom C-Compiler noch zur Laufzeit entdeckt. Das Programm stürzt ab, weil es einen unentdeckten Typfehler enthält: C ist nicht typsicher

Zweck des Typsystems / der statischen Typisierung

Effizienz

Typsierung ermöglicht die Erzeugung von effizienterem Code.

- **Typprüfungen zur Laufzeit können (weitgehend) entfallen**
- **Typmarkierungen der Werte können (weitgehend) entfallen**
- **Erzeugung von Code, der an die Typen der Werte direkt angepasst ist, wird ermöglicht**
  - **Beispiel: Zuweisung / Parameterübergabe als ein Maschinenbefehl (richtige Anzahl von Bytes verschieben)**
  - **Beispiel: Arithmetische Operation als ein Maschinenbefehl**
    (Bitmuster und Maschinenbefehle für int- und double-Werte sind komplett unterschiedlich)

Typsysteme: Basistypen und Typkonstruktoren

Typen

Basistypen und Typkonstruktoren

- **Basistyp**
  Typ der von der Sprache vorgegeben ist
  Beispiel Java: int, double, char, boolean
- **Typkonstruktor**
  Mechanismus mit dem Programmierer neue Typen definieren können
  Beispiel Java: array, interface, class

Typen, Werte, Literale

- Literale
  Für alle Werte der Basistypen definieren Programmiersprachen i.R. Literale
  Für Werte die zu Typen gehören,
  - die mit Hilfe von Typkonstruktoren definiert wurden,
  - gibt es i.d.R. keine Literale

  Ausnahme: String-Literale in Java / String ist ein (in der API) definierter Typ

Typkonstruktoren

Array

- Array-Werte bestehen aus einer Folge von Werten mit dem gleichen Typ
- Zugriff über einen Index
- Länge unveränderlich
- „gleicher Typ“: gleich entsprechend den Regeln der Sprache
  Java: Arrays sind **kovariant**, d.h.
  Die Subtyp-Eigenschaft der Elemente überträgt sich auf die Arrays.

```
class Tier {}
class Kuh extends Tier {}

Tier[] tiere = new Tier[1];
Kuh[] kuehe = new Kuh[1];

tiere[0] = new Kuh();          // OK
//kuehe[0] = new Tier();       Compilerfehler
kuehe[0] = (Kuh) new Tier();   // Laufzeitfehler java.lang.ClassCastException
```

Typkonstruktoren

Verbund / Structure / Record / Produkt-Typ

- Verbund-Werte bestehen aus einer Folge von Werten mit beliebigem Typ
- Zugriff über Namen
- Größe unveränderlich

Beispiel struct in C:

```c
struct MyStruct {                       // Typdefinition
      int  i;
      char c;
};

struct MyStruct structVar = {0, 42};  // Variablendefinition

structVar.i++;                          // Zugriff auf Komponente

printf("%i, %c", structVar.i, structVar.c);
```

Typkonstruktoren

Vereinigung / Union / Summen-Typ

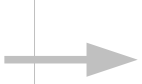
- Werte mit einem Vereinigungs-Typ bestehen aus einer von mehreren Alternativen von beliebigem Typ
- Zugriff über Namen
- Größe unveränderlich

Beispiel union in C:

```c
union MyUnion { // Typdefinition. Hier
      int i;     // Entweder ein int
      char c;    // Oder ein char
};

union MyUnion unionVar = { 42 }; // Variablendefinition

// Zugriffe auf die selben Bits
printf("%i, %c\n", unionVar.i, unionVar.c);
unionVar.c++;
printf("%i, %c\n", unionVar.i, unionVar.c);
```



```
42, *
43, +
```

40	0x28	050	(
41	0x29	051	)
42	0x2A	052	*
43	0x2B	053	+

Ausschnitt ASCII-Tabelle

Das ist kein Bug sondern ein Feature: Die Sprache C wurde für die Systemprogrammierung konzipiert, nicht für die Anwendungsentwicklung.

Typsysteme: Basistypen und Typkonstruktoren

Typkonstruktoren

Vereinigung / Union / Summen-Typ

- **Statt Unions (ganz und gar nicht nicht typsicher!) verwendet man heute i.d.R. Klassen mit gemeinsamer Basis**

C

```c
union MyUnion { // Typdefinition. Hier
      int i;     // Entweder ein int
      char c;    // Oder ein char
};

union MyUnion unionVar = { 42 }; // Variablendefinition

// Zugriffe auf die selben Bits
printf("%i, %c\n", unionVar.i, unionVar.c);
unionVar.c++;
printf("%i, %c\n", unionVar.i, unionVar.c);
```

→

```
42, *
43, +
```

Java

```java
interface MyUnion {};

static class Variante1 implements MyUnion {
      Variante1(int i) { this.i = i; }
      int i;
}

static class Variante2 implements MyUnion {
      Variante2(char c) { this.c = c; }
      char c;
}

public static void main(String[] args) {
      MyUnion unionVar = new Variante1(42);
      System.out.println(((Variante1)unionVar).i);
      System.out.println(((Variante2)unionVar).c);
      ((Variante1)unionVar).i++;
      System.out.println(((Variante1)unionVar).i);
      System.out.println(((Variante2)unionVar).c);
}
```

→

```
42
Exception in thread "main"
java.lang.ClassCastException
```

Typkonstruktoren

Klasse, Interface

– **Erweiterungen von Struct-Typen um Methoden und Vererbung**

Beispiel:

```
interface IntTree {}

class Leaf implements IntTree {
  Leaf(int value) {
    this.value = value;
  }
  int value;
}

class Node implements IntTree {
  Node(IntTree left, IntTree right) {
    this.left = left;
    this.right = right;
  }
  IntTree left, right;
}


public class Int_Trees {

  public static int sum(IntTree tree) {
    if (tree instanceof Leaf) {
      return ((Leaf)tree).value;
    } else {
      Node node = (Node) tree;
      return sum(node.left) + sum(node.right);
    }
  }

}
```

Kompatibilität von Typen

Subtyp-Relation

- **Liskov'sches Substitution-Prinzip**:
  - Ein Typ S ist kompatibel zu einem Typ T
  - Wenn jeder Wert vom Typ S an Stelle eines Werts vom Typ T treten kann

  Intuitiv klar und ausführlich in OOP und PIS behandelt.

- **Subtyp-Relation**

  S ≤ T: S ist ein Subtyp von T

  ein **Wert** vom Typ S kann in **jeder Situation** verwendet werden, in der einen Wert vom Typ T verlangt wird

  Beispiel: **Kuh-Objekt ≤ Tier-Objekt**

Subtyp-Relation auf statischen Typen

- Das Liskov'sches Substitution-Prinzip und die Subtyp-Relation ist auf Werten und dynamischen Typen definiert
- Kann aber natürlich auf statische Typen übertragen werden:

  S ≤ T: S ist ein Subtyp von T

  ein **Ausdruck** vom Typ S kann in jedem **textuellen Kontext** verwendet werden, in dem ein **Ausdruck** vom Typ T verlangt wird

- Beispiel: **Kuh-Ausdruck ≤ Tier-Ausdruck**

https://de.wikipedia.org/wiki/Liskovsches_Substitutionsprinzip

Kompatibilität von Typen: Namens- / Struktur-Äquivalenz

Äquivalenz von Typen auf Basis ihres Namens oder ihrer Struktur

- **Namens-Äquivalenz**
  - Es ist wichtig, welchen Namen ein Typ hat
- **Struktur-Äquivalenz**
  - Es ist nicht wichtig, welchen Namen ein Typ hat, es kommt nur auf seine innere Struktur an

```c
typedef double Kg;
typedef double Liter;

Kg gewicht = 10.0;
Liter volumen = 15.0;

volumen = gewicht;  // OK: Kg und Liter sind kompatibel (!) trotz unterschiedlicher Namen

struct KG {
     double value;
};

struct LITER {
     double value;
};

struct KG gewicht1 = { 10.0 };
struct LITER volumen1 = { 15.0 };

volumen1 = gewicht1; // error: assigning to 'struct LITER' from incompatible type 'struct KG'
                     // LITER und GEWICHT sind nicht kompatibel trotz gleicher Struktur
```

Beispiel C: Struktur-Äquivalenz bei unstrukturierten Typen. Namens-Äquivalenz bei strukturierten Typen.

Kompatibilität von Typen: Varianz

Ko-Varianz bei strukturierten Typen

- Ein Typ-Konstruktor X ist dann ko-variant, wenn aus
  - S ≤ T (Typ S ist Kompatibel mit T)
- folgt, dass
  - X[S] ≤ X[T]

Kontra-Varianz bei strukturierten Typen (Unüblich)

- Ein Typ-Konstruktor X ist dann kontra-variant, wenn aus
  - T ≤ S (Typ T ist Kompatibel mit S)
- folgt, dass
  - X[S] ≤ X[T]

Arrays in Java: ko-variant

```
class Tier {}
class Kuh extends Tier {}

Tier[] tiere = new Tier[5];
Kuh[] kuhe = new Kuh[5];

tiere = kuhe; // OK ko-variant
kuhe = tiere; Fehler: nicht kontra-variant
```

Listen in Java: Invariant: Weder ko- noch kontra-variant.

```
class Tier {}
class Kuh extends Tier {}

List<Tier> tiere = new ArrayList<>();
List<Kuh> kuhe = new ArrayList<>();

tiere = kuhe;  Fehler: nicht ko-variant
kuhe = tiere;  Fehler: nicht kontra-variant
```

Kompatibilität von Typen: Varianz

Varianz in Scala kann definiert werden

– **Beispiel**

```
class Tier;
class Kuh extends Tier

class Stall[T](aT: T);

var stall_kuh: Stall[Kuh]  = new Stall[Kuh](new Kuh)
var stall_tier: Stall[Tier] = new Stall[Tier](new Tier)

stall_tier = stall_kuh  Fehler Stall ist nicht ko-variant
stall_kuh = stall_tier  Fehler Stall ist nicht kontra-variant
```

Invariant als „Default-Einstellung“

```
class Tier;
class Kuh extends Tier

class Stall[+T](aT: T);

var stall_kuh: Stall[Kuh]  = new Stall[Kuh](new Kuh)
var stall_tier: Stall[Tier] = new Stall[Tier](new Tier)

stall_tier = stall_kuh // OK Stall ist ko-variant
stall_kuh = stall_tier  Fehler Stall ist nicht kontra-variant
```

***+**: Stall wird als ko-variant deklariert*

Typisierung: Statische Typ-Berechnung und Typ-Prüfung

Ziele der Typprüfung

- Vermeide (unnötige) Typprüfungen zur Laufzeit
- Identifiziere Programme mit (potentiellen) Typfehlern
- Sammle / verarbeite Typ-relevante Informationen in Vorbereitung auf die Codegenerierung

Basis: Statisches Typsystem

- Das statische Typsystem ist Teil der Sprachdefinition
- Das (statische) Typsystem legt fest:
  - Welche Typen gibt es
  - Die Zuordnung von Typen zu Ausdrücken
  - Die Kompatibilität von Typen (z.B. Subtyp-Relation, Namens- oder Struktur-Äquivalenz)
  - Die Beziehung von Typen und Werten

Definition des Typsystems

Informale Definition

- Üblich ist eine mehr oder weniger längliche ausführliche textuelle Beschreibung des Typsystems in der Sprachdefinition
- Beispiel, (sehr kurzer) Auszug aus der Java-Spezifikation:

An expression is *potentially compatible* with a target type according to the following rules:

- A lambda expression (§15.27) is potentially compatible with a functional interface type (§9.8) if all of the following are true:
  - The arity of the target type's function type is the same as the arity of the lambda expression.
  - If the target type's function type has a `void` return, then the lambda body is either a statement expression (§14.8) or a void-compatible block (§15.27.2).
  - If the target type's function type has a (non-`void`) return type, then the lambda body is either an expression or a value-compatible block (§15.27.2).
- A method reference expression (§15.13) is potentially compatible with a functional interface type if, where the type's function type arity is *n*, there exists at least one potentially applicable method for the method reference expression with arity *n* (§15.13.1), and one of the following is true:
  - The method reference expression has the form *ReferenceType* `::` *[TypeArguments] Identifier* and at least one potentially applicable method is i) `static` and supports arity *n*, or ii) not `static` and supports arity *n*-1.
  - The method reference expression has some other form and at least one potentially applicable method is not `static`.
- A lambda expression or a method reference expression is potentially compatible with a type variable if the type variable is a type parameter of the candidate method.
- A parenthesized expression (§15.8.5) is potentially compatible with a type if its contained expression is potentially compatible with that type.
- A conditional expression (§15.25) is potentially compatible with a type if each of its second and third operand expressions are potentially compatible with that type.
- A class instance creation expression, a method invocation expression, or an expression of a standalone form (§15.2) is potentially compatible with any type.

The definition of potential applicability goes beyond a basic arity check to also take into account the presence and "shape" of functional interface target types. In some cases involving type argument inference, a lambda expression appearing as a method invocation argument cannot be properly typed until after overload resolution. These rules allow the form of the lambda expression to still be taken into account, discarding obviously incorrect target types that might otherwise cause ambiguity errors.

Definition des Typsystems

Kontext-sensitive Grammatik

Das Typsystem kann mit einer *informalen „kontext-sensitiven“* Grammatik definiert werden.

Beispiel:

***int*-exp ::= *int*-const | *int*-var | *int*-exp + *int*-exp | …**
***int*-var ::= identifier**

Syntaktische Kategorien (Nonterminale) enthalten „kontextsensitive Bestandteile“:

***int*-exp statt exp**

Bei der Implementierung muss

- **der Parser den kontextfreien Kern der Grammatik parsen.**
- **die Bezeichner-Identifikation Definitionen und Verwendung von Bezeichnern zusammen bringen,**
- **die semantische Analyse den kontextsensitiven Teil verarbeiten.**

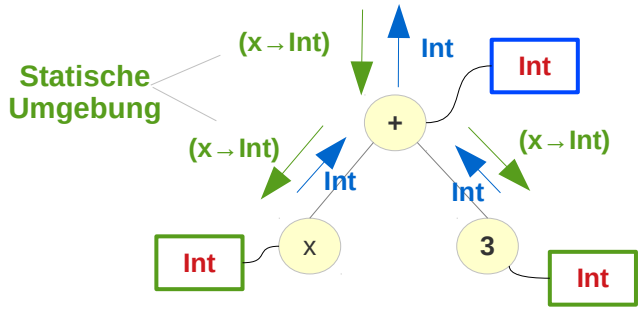
Definition des Typsystems

Prozedurale Definition mit einfacher Attributierung

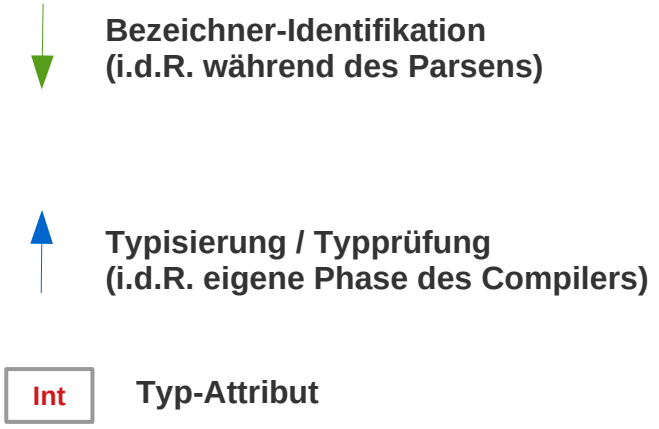
Attribut: Zusatz-Info an den Knoten des AST

Die Typsierung wird in einfacher Version definiert:

- **Attribute der Blätter im Baum (Bezeichner / Namen):**
  erhalten Typinfo durch Bezeichner-Identifikation
  Compilerphase: Parser
- **Attribute der Knoten:**
  erhalten Typinfo durch Rekursion über die Baumstruktur
  Compiler-Phase: Typprüfung / Typisierung



Umgebungskonzept und Typisierung



Umgebung zur Übersetzungszeit = Statische Umgebung

Statische Umgebung

- Enthält die statisch bekannten Informationen zu jedem Namen
- Kann zur Typisierung verwendet werden

Beispiel

Minisprache mit 2 Typen: Int / String

```
val x: int = 5,
val y: string = "5";
x+y
```


Typen der Operanden passen nicht

Beispiel mit Typfehler

Typisierung / Beispiel Minisprache mit 2 Typen

Abstrakte Syntax

Variablen-Definitionen

```
case class VarDef(override val symb: Variable, t: TypeExp, e: Exp) extends Definition
```

Ausdrücke: können Int- oder String-Ausdrücke sein

```
// Expression that denote int or string values
sealed abstract class Exp

case class Number(d: Int)         extends Exp
case class StrLit(str: String)    extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Div(e1: Exp, e2: Exp) extends Exp
case class Mul(e1: Exp, e2: Exp) extends Exp
case class Ref(ref: RefExp)       extends Exp
```

Die Sprache hat Typausdrücke.

Sehr einfache hier: die Konstanten
„int“ und
„string“

```
VAL x: INT    = 5,
VAL y: STRING = "5";
 x+y
```

Exp TypeExp Exp

Typ-Ausdrücke

```
// Expressions that denote type values
sealed abstract class TypeExp
case object IntTypeExp extends TypeExp
case object StringTypeExp extends TypeExp
```

Typisierung / Beispiel Minisprache mit 2 Typen

Konkrete Syntax

Scanner und Parser müssen angepasst / erweitert werden

Beispiel Parser:

```
// parser for string literals
val str_any : Parser[Any] =
  elem("string", _.isInstanceOf[StringToken])

def str: Parser[StrLit] =
  str_any   ^^ { case (x: Any) => StrLit(x.asInstanceOf[StringToken].chars) }
```

```
// parse type expressions
def typeExp: Parser[TypeExp] =
  KwToken("INT")    ^^^ IntTypeExp |
  KwToken("STRING") ^^^ StringTypeExp
```

```
// parse definitions
def definition: Parser[Definition] =
  (KwToken("VAR") ~> ident <~ ColonToken(":")) ~ typeExp ~ (AssignToken(":=") ~>
  arithExp <~ SemicolonToken(";"))  ^^ { case name ~ t ~e =>
    val symbol = Variable(name)      // create Symbol for the defined name
    env.define(name, symbol)         // store definition in static environment
    VarDef(symbol, t, e)             // return Ast containing a reference to the symbol
  }
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typen: Typ-Ausdrücke und Typ-Werte

Bei den Typen unterscheiden wir, wie auch sonst immer, Syntax und Semantik

- **Typ-Syntax**

  Typenausdrücke in der „Objektsprache“, also der Sprache, deren Programme analysiert werden

- **Typ-Semantik**

  Das was die Typ-Ausdrücke in der Meta-Sprache bedeuten sollen, Metasprache ist die Sprache in der über die Objektsprache geredet wird. Hier ist es die Sprache in der der Compiler geschrieben ist.

```
// Expressions that denote type values
sealed abstract class TypeExp
case object IntTypeExp extends TypeExp
case object StringTypeExp extends TypeExp
```

abstrakte Syntax der Typen

```
object StaticTypes {
  sealed abstract class StaticType
  object IntStaticType extends StaticType
  object StringStaticType extends StaticType
}
```

Semantik der Typen

In diesem einfachen Beispiel sind Syntax und Semantik der Typen völlig äquivalent und die Unterscheidung scheint trivial und überflüssig zu sein. In komplexeren Beispielen ist die Unterscheidung aber weder trivial noch überflüssig, sondern essentiell.

Typisierung / Beispiel Minisprache mit 2 Typen

Kompatibilität von Typen

Typen können miteinander vertraglich sein oder nicht.

Das ist eine Eigenschaft der Typ-Werte

```
object StaticTypes {

  sealed abstract class StaticType {
    def isCompatibleWith(other: StaticType): Boolean
  }

  object IntStaticType extends StaticType {
    override def isCompatibleWith(other: StaticType): Boolean = other match {
      case IntStaticType => true
      case _ => false
    }
  }

  object StringStaticType extends StaticType {
    override def isCompatibleWith(other: StaticType): Boolean = other match {
      case StringStaticType => true
      case _ => false
    }
  }

}
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typen von Variablen

- Variablen haben Typen
- Die entsprechende Info wird im Symbol gespeichert
- Symbole werden vom Parser erzeugt
- Der Parser weiß nichts von Typen (darum None)
  (es sei denn wir lassen ihn auch die Typen berechnen, das wäre in diesem Beispiel leicht möglich, ist i.A. aber nicht zu empfehlen)

```
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,
                       var staticType: Option[StaticType] = None // will be set by typifier
                     ) extends ProgSymbol

}
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typisierung von Ausdrücken

Die Typisierung kann analog zur Auswertung von Ausdrücken organisiert werden.

Vorgehen: Aktionen

- **Datenfluss Top-Down:**
  Wurde durch die Bezeichner-Identifikation weitgehend erledigt
  Es fehlt noch die Berechnung der Typen der Bezeichner
- **Datenfluss Bottom-Up: Rekursion über die Struktur**

Vorgehen: SWT

- **Funktional oder Imperativ?**
- **Imperativ / Baumtransformation**
  Die berechneten (Typ-) Informationen werden im AST gespeichert
- **Funktional / neu generierter Baum**
  Die berechneten Informationen werden in einem neuen AST integriert

Typisierung / Beispiel Minisprache mit 2 Typen

Typisierung / Vorgehen: Baumtransformation oder neuer Baum

Compilerbau: Imperatives Vorgehen (Baumtransformation) üblich.

Der AST wird in den Compilerphasen modifiziert

Im Ast muss dazu Platz geschaffen werden:

```
// Expression that denote int or string values
sealed abstract class Exp {
  var staticType: Option[StaticType] = None // will be set by typifier
}

case class Number(d: Int)         extends Exp
case class StrLit(str: String)    extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Div(e1: Exp, e2: Exp) extends Exp
case class Mul(e1: Exp, e2: Exp) extends Exp
case class Ref(ref: RefExp)       extends Exp


// Expression that denote references to storage locations
sealed abstract class RefExp {
  var staticType: Option[StaticType] = None // will be set by typifier
}

case class VarRef(symb: Variable)   extends RefExp
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typisierung / Vorgehen: Baumtransformation oder neuer Baum

Implementierung / Beispiel

```
object Typifier {

  def typify(prog: Prog): Unit = {
    typifyDefs(prog.defs)
    typifyCmds(prog.cmds)
  }

  private def typifyDefs(defs: List[Definition]) : Unit = {
    defs.foreach {
      case VarDef(symb@Variable(name, None), typeExp, e) =>
        symb.staticType = Some(evalTypeExp(typeExp))
      case VarDef(symb@Variable(name, _), _, _) =>
        throw new Exception(s"internal error: type of $name is already set")
    }
    defs.foreach {
      case VarDef(Variable(name, Some(varType)), typeExp, e) =>
        typifyExp(e)
        (e.staticType, varType) match {
          case (None, _) => throw new Exception(s"internal error: type of var not set")
          case (Some(IntStaticType), IntStaticType) => // OK
          case (Some(StringStaticType), StringStaticType) => // OK
          case (Some(t1), t2) => throw new Throwable("Type mismatch in init statement")
        }
      case VarDef(Variable(name, None), _, _) =>
        throw new Exception(s"internal error: type of var not set")
    }
  }

   ...
}
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typisierung / Vorgehen: Baumtransformation oder neuer Baum

Implementierung / Beispiel

```
object Typifier {

  ...

  private def evalTypeExp(te: TypeExp): StaticType = te match {
    case IntTypeExp     => IntStaticType
    case StringTypeExp  => StringStaticType
  }

  private def typifyExp(e: Exp): Unit = e match {
    case Number(d: Int)        =>
      e.staticType= Some(IntStaticType)
    case StrLit(str: String)   =>
      e.staticType = Some(StringStaticType)
    case Add(e1: Exp, e2: Exp) =>
      typifyExp(e1)
      typifyExp(e2)
      (e1.staticType, e2.staticType) match {
        case (None, _) => throw new Exception(s"internal error: type not set")
        case (_, None) => throw new Exception(s"internal error: type not set")
        case (Some(st1), Some(st2)) =>
          if (st1.isCompatibleWith(st2))
            e.staticType = Some(st1)
          else throw new Throwable("Type error")
      }
      ...
   }
   ...
}
```

Typisierung / Beispiel Minisprache mit 2 Typen

Typisierung / Vorgehen: Baumtransformation oder neuer Baum

Implementierung / Beispiel

```
object Typifier {

  ...

  private def typifyCmds(cmds: List[Cmd]): Unit = {
    cmds.foreach {
      case Assign(left, right)       =>
        typifyRef(left)
        typifyExp(right)
        (left.staticType, right.staticType) match {
          case (None, _) => throw new Exception(s"internal error: type not set")
          case (_, None) => throw new Exception(s"internal error: type not set")
          case (Some(IntStaticType), Some(IntStaticType)) => // OK
          case (Some(StringStaticType), Some(StringStaticType)) => // OK
          case (_, _) => throw new Throwable("Type error in assignment")
        }
      ...
  }

  ...

}
```