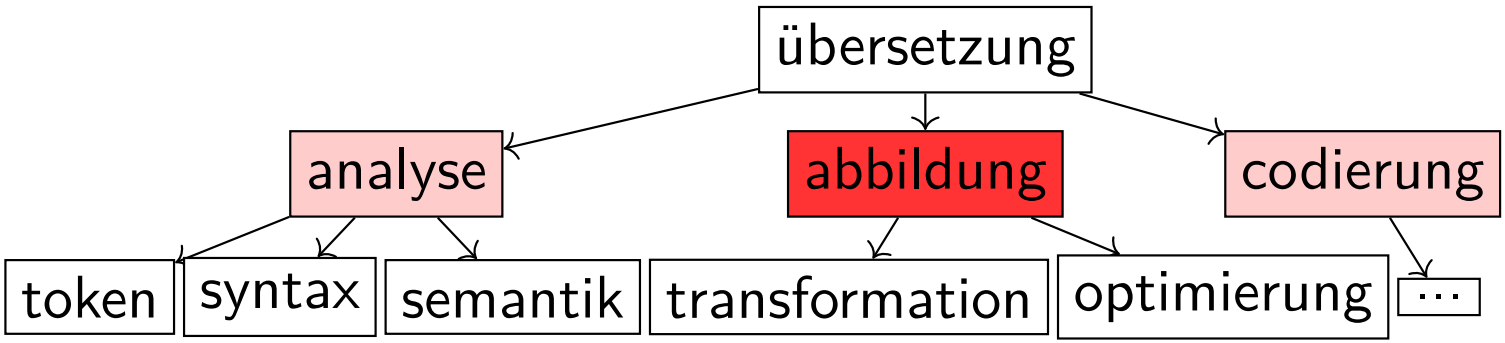
Inhaltsübersicht

Bisher


übersetzung
analyse
abbildung
codierung
token
syntax
semantik
transformation
optimierung
...

6.1 Abbildungsphase

1. Was ist die Ein- und Ausgabe dieser Phase?
   - Eingabe: Vollständig attributierter AST.
   - Ausgabe: Semantisch äquivalenter Code in einer Zwischencodedarstellung (geringeres Abstraktionsniveau)
2. Teilaufgaben der Abbildungsphase
   - Variablen- und Typabbildung
   - Operatorabbildung
   - Ablaufabbildung

6.1 Abbildungsphase

Variablen- und Typabbildung

- Wo werden Variablen platziert?
- Wie groß sind Datentypen?
- Wie stelle ich `true`, `false` oder `null` dar?
- Was mache ich mit sowas wie `int[1][2][3]`?

6.1 Abbildungsphase

Operatorabbildung

- $a := (1 + 3 * a) * 3 - 42/2 \rightarrow$ Liste elementarer Operationen
- Üblich: Zwischencode für Quellsprache X kann alle Operationen von X
- Auswahl der richtigen Operation
  - Numerische Operatoren oft überladen.
  - `+(int, int) -> int`
  - `+(real, real) -> real`
  - Bei `a := 1 + 2;` integer Addition!
- Dank Typanalyse wissen wir welche Operation wir brauchen!
- Aber: JVM z.B. hat `short` aber keine short-Addition?!?

6.1 Abbildungsphase

Ablaufabbildung

- Üblich: Eliminierung von Kontrollstrukturen
  - Schleifen: `do ... while`, `for`, `foreach`, etc.
  - Alternativen: `if ... end if`, `switch ... case`, etc.
  - Abstraktionen: Methoden, Funktionen, Prozeduren, etc.
- Häufig: Wahl zwischen versch. Abbildungsmethoden (vgl. Vorlesung)
- Üblich: Abbildung auf *explizite* Sprunganweisungen

6.2 Zwischencode

Gründe für den Umweg über eine Zielsprache??

Warum nicht direkt von E2 nach ASM??

6.2 Warum Zwischencode?

1. `Semantische Lücke` zwischen Quell- und Zielsprache.
   - Direkte Übersetzung schwer
   - Fast unmöglich *guten* Zielcode zu erzeugen
2. `Portabilität`
   - Üblicherweise unabhängig von Zielsprache (ASM)
   - Interpreter möglich (JVM).
   - Dieser ist einfach portierbar!
   - Programme können im Zielcode ausgeliefert werden (Achtung: Serialisierbarkeit!)
3. $n * m > n + m$
   - { Java, Groovy, ... } $\rightarrow$ JVM Bytecode $\rightarrow$ { x86, ARM, ... }
   - LLVM
4. `Einfachere Optimierungen`
   - Zwischensprache weniger komplex
   - Optimierbare Stellen einfacher zu erkennen!
   - Konstantenfaltung, DCE z.B. wesentlich simpler
5. `Sprachenunabhängige Optimierungen`

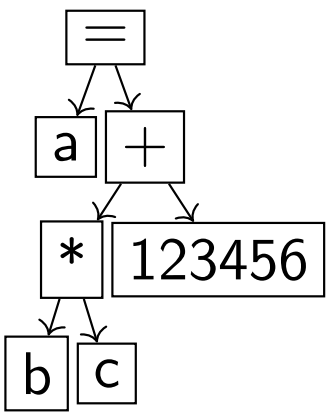
6.2 Eigenschaften von gutem Zwischencode

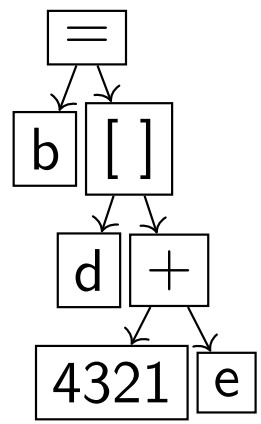
- Hardness(Src $\rightarrow$ IR) $\overset{!}{\approx}$ Hardness(IR $\rightarrow$ Dest)
- Je ausgewogener, desto leichter können beide Schritte “guten” Code erzeugen.
- Zwischensprache unabhängig von Quell- und Zielsprache.
- Ermöglicht einfache Impl. von Transformationen

6.3 JVM-Bytecode — Ausdrücke

$a \to 1,\ b \to 2,\ c \to 3,\ d \to 4,\ e \to 5$

```
a = (b * c) + 123456;
b = d[4321 + e];
```





6.4 JVM-Bytecode — Kontrollstrukturen

$a \to 1, b \to 2$

```
if (a > 13) {
    while (a != 0) {
        b = b + a;
        a = a - 1;
    }
}
```

6.5 Kurzschlusssemantik

```
if ((a > 0 or b > 0) and c < 42) {
    // ...
}
```