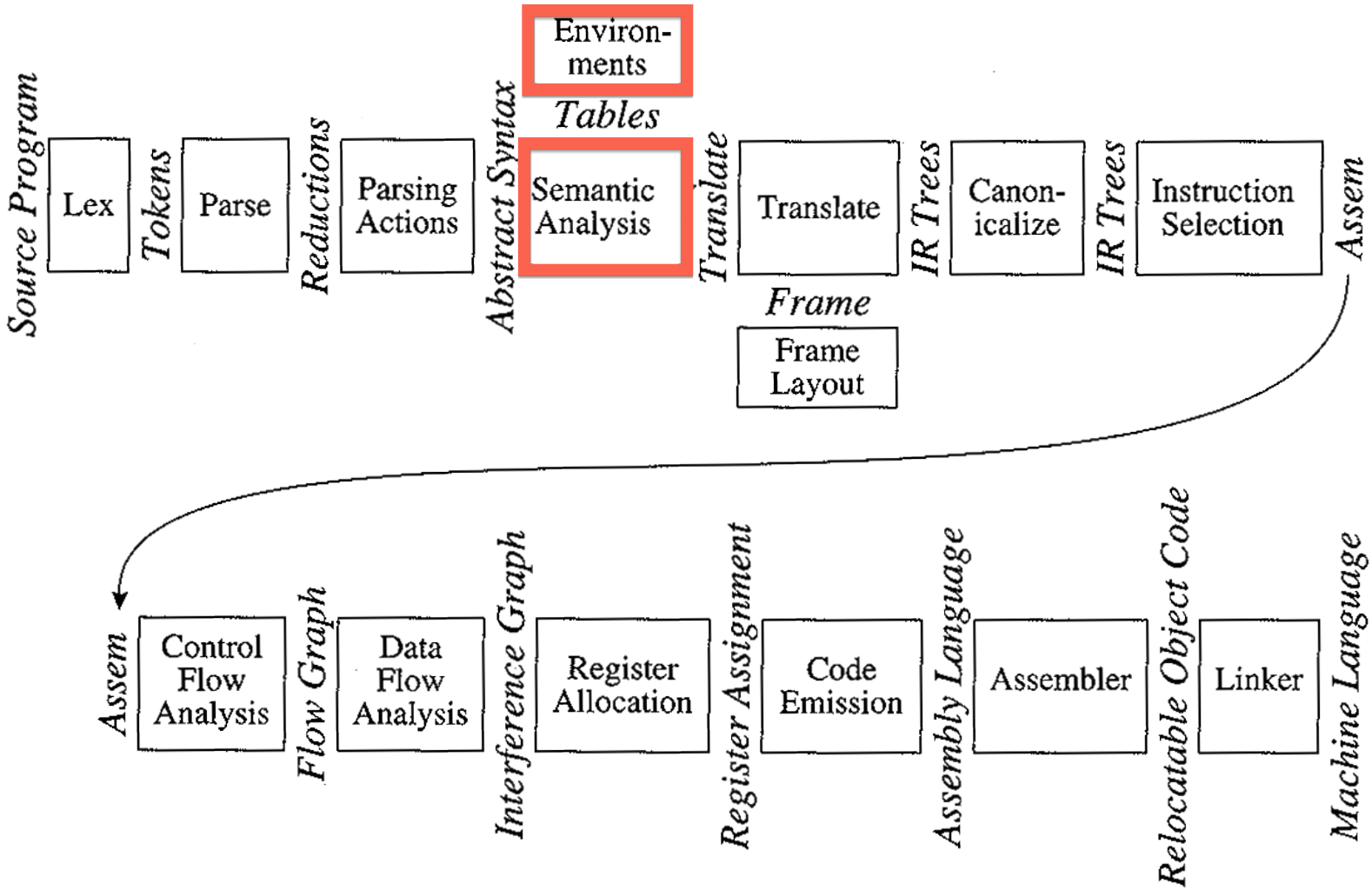
Semantische Analyse

Semantische Analyse



Semantische Analyse

Frond-End muss prüfen, ob die Eingabe ein gültiges Programm der Eingabesprache ist.

Bisher:

- Lexikalische Analyse
  (endlicher Automat)
- Syntaktische Analyse
  (kontextfreie Grammatik)

Endliche Automaten und Kontextfreie Grammatiken sind nicht ausdrucksstark genug, um die Korrektheit von Programmen zu entscheiden. $\Rightarrow$ Semantische Analyse

Semantische Analyse

In der semantischen Analyse im Front-End wird die abstrakte Syntax auf Gültigkeit überprüft:

- Typkorrektheit
- Alle Variablen werden deklariert bevor sie benutzt werden.
- Jede Methode hat `return`-Statements in jedem Lauf.
- ...

Bei möglicherweise ungewolltem Code werden Warnungen ausgegeben.

Semantische Informationen können auch später im Compiler nützlich sein, z.B. Typinformationen.

Semantische Analyse

Im Back-End können weitere semantische Analysen auf der Zwischensprache stattfinden:

- Wertebereiche genauer analysieren als durch die Typen vorgegeben, z.B. $0 < i < 100$

  ```
  for (int i = 1; i < 100; i++) { if (i < 1000) a[i]++; }
  ```

  Test wird von aktuellen C-Compilern entfernt
- Werden Zwischenergebnisse im Endergebnis überhaupt verwendet? Z.B. Entfernung von assert-Statements

  ```
  if (debug) { ... }
  ```
- Wie lange kann auf Variablen zugegriffen werden? Speicherung in Register oder Hauptspeicher?
- Kontrollflussanalyse
- ...

Typüberprüfung

Ein Hauptteil der semantischen Analyse im Front-End ist die Typüberprüfung.

Typsystem

- Teil der Sprachdefinition
- Ziel: Vermeidung von Laufzeitfehlern („well-typed programs don't go wrong“)
- Ausdrucksstärke, z.B. durch Typinferenz

Typinformationen im Compiler

- Generierung von besserem/effizienterem Code
- Ausschluss unsinniger Fälle

Symboltabelle

Für eine effiziente Implementierung der Typüberprüfung ist es nützlich eine *Symboltabelle* anzulegen, in der z.B. Typinformationen für die Identifier im Programm abgelegt werden.

```
class C {
  public int m () {
    D d = new D();
    d.b = 2; }
}
class D {
  boolean b;
}
```

Symboltabellen können auch weitere Informationen enthalten und sind an vielen Stellen des Compilers nützlich.

Symboltabelle für MiniJava

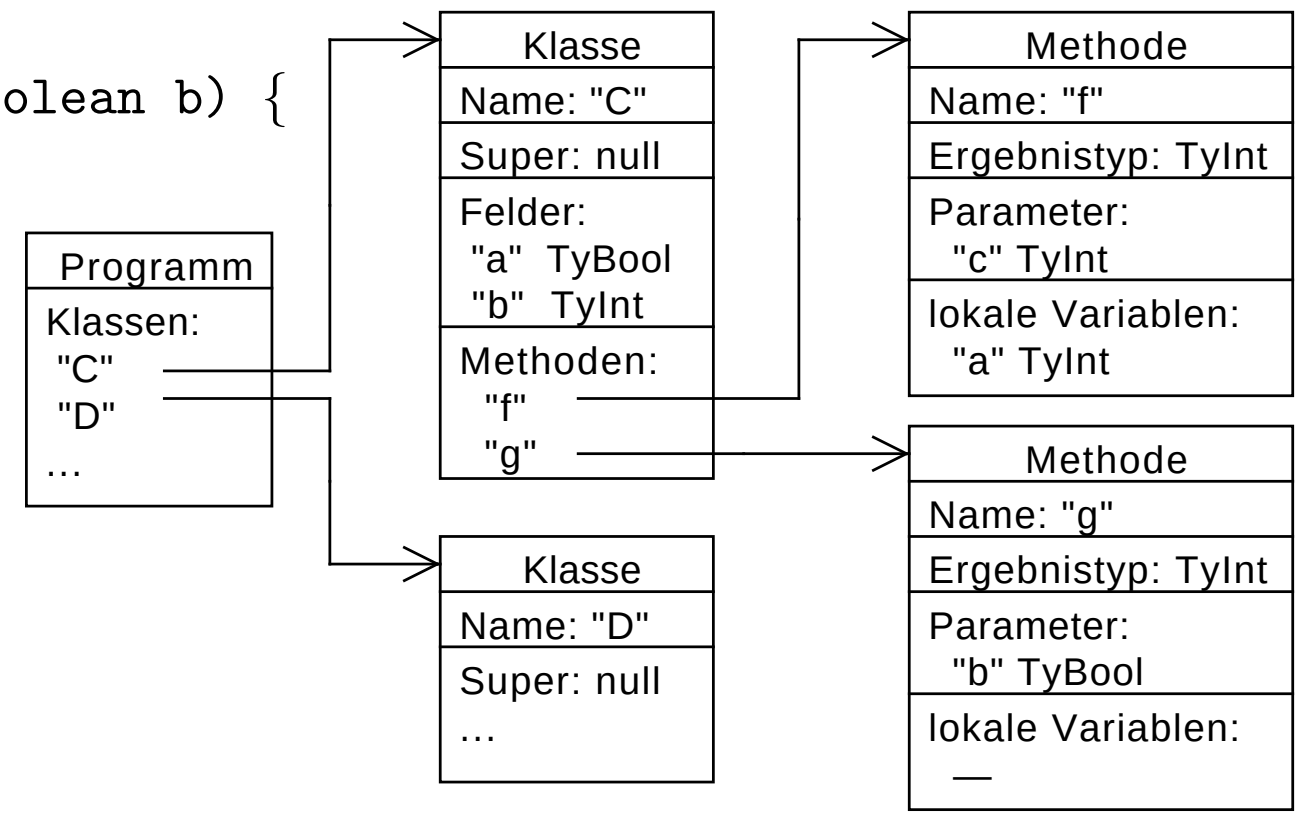
Eine Symboltabelle bildet Bezeichner ab auf „semantische Werte“.

- Semantischer Wert eines **Programms**:
  - Klassennamen mit ihren semantischen Werten
- Semantischer Wert einer **Klasse**:
  - Feldnamen mit ihren semantischen Werten
  - Methodennamen mit ihren semantischen Werten
  - Verweis auf den Eintrag der Elternklasse
- Semantischer Wert eines **Feldes**:
  - Typ
- Semantischer Wert einer **Methode**:
  - Ergebnistyp
  - Vektor der Parameter mit ihren Typen
  - lokale Variablen mit ihren Typen

Symboltabelle — Beispiel

```
class C {
  boolean a;
  int b;
  public int f (int c) {
    int a;
    return a+b+c;
  }
  public int g (boolean b) {
    return b+1;
  }
}

class D {
  ...
```



Typüberprüfung für MiniJava

Die Typüberprüfung selbst besteht nun aus einem Satz rekursiver Methoden, welche in Gegenwart einer Symboltabelle die verschiedenen Programmteile auf Typkorrektheit prüfen:

- Programm
- Klasse
- Methode
- Statement
- Expression

Wir betrachten zunächst MiniJava ohne Vererbung.

Typüberprüfung — Expressions

Die Typen von Ausdrücken werden üblicherweise durch *Typisierungsregeln* festgelegt.

$$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}} \qquad \frac{a : \texttt{int}[] \qquad i : \texttt{int}}{a[i] : \texttt{int}}$$

$$\frac{e : \texttt{int} \qquad x : \texttt{C} \qquad \text{Klasse } \texttt{C} \text{ hat Methode } \texttt{boolean}\ m(\texttt{int})}{x.m(e) : \texttt{boolean}}$$

(analog für restliche Expressions)

Die Typen von Variablen und Methoden werden der Symboltabelle entnommen. Hierbei sind die Gültigkeitsbereiche zu beachten.

Typüberprüfung — Expressions

Für die Typüperprüfung ist es normalerweise nötig, den Typ eines gegebenen Ausdrucks auszurechnen.

$$typeOf(x) = \text{Typ der Variablen } x \text{ nachschlagen:}$$
$$\text{lokale Variablen, Methodenparameter,}$$
$$\text{Felder der Klasse, (Superklasse ...)}$$

$$typeOf(e_1 + e_2) = \begin{cases} \texttt{int} & \text{wenn } typeOf(e_1) = typeOf(e_2) = \texttt{int} \\ \bot & \text{sonst} \end{cases}$$

$$typeOf(x.m(e)) = \begin{cases} t & \text{wenn } typeOf(x) = \texttt{C} \text{ und} \\ & \text{Klasse } \texttt{C} \text{ hat Methode } t\ m(typeOf(e)) \\ \bot & \text{sonst} \end{cases}$$

Implementierung analog zur Interpretation von Straightline-Programmen (Visitor). Der Wert eines Ausdrucks ist nun ein MiniJava-Typ statt einer Zahl.

Typüberprüfung — Statments

Programm-Statements haben selbst keinen Typ, sie können nur wohlgetypt sein oder nicht.

$$\frac{x : t \qquad e : t}{x := e \text{ ok}}$$

$$\frac{s_1 \text{ ok} \qquad s_2 \text{ ok}}{s_1 ; s_2 \text{ ok}}$$

(analog für die restlichen Statements)

Implementierung durch eine Funktion, die entscheidet, ob ein gegebenes Statement wohlgetypt ist:

$$\mathit{typeOk}(x := e) = \begin{cases} \mathit{true} & \text{wenn } \mathit{typeOf}(e) = \mathit{typeOf}(x) \\ \mathit{false} & \text{sonst} \end{cases}$$

...

Typüberprüfung — Methode

```
  public int f (int c) {
    int a;
    ...
    s
    return e;
  }
}
```

Es werden folgende Bedingungen geprüft:

- Die Parameter (bzw. lokalen Variablen) haben paarweise verschiedene Namen.
- Das Statement s ist wohlgetypt.
- Der Typ von e entspricht dem Rückgabetyp entspricht der Funktion.

Typüberprüfung — Klasse, Programm

Klasse

Um eine Klasse zu überprüfen, werden alle Methoden überprüft.
Es wird geprüft, dass keine Methode doppelt definiert ist.
Beachte: Die Methode `main` ist statisch und darf nicht auf `this` zugreifen!

Programm

Um ein Programm zu überprüfen, werden alle Klassen überprüft.
Es wird geprüft, dass keine Klasse mehrfach definiert ist.

Ihre Aufgabe

Implementieren Sie die Typüberprüfung für MiniJava ohne Vererbung.

Es bietet sich an, die Implementierung in zwei Phasen zu gliedern:

1. Erzeugung der Symboltabelle.
   Die abstrakte Syntax des Eingabeprogramms wird einmal durchgegangen und die Symboltabelle wird dabei schrittweise aufgebaut.
2. Typüberprüfung.
   Im zweiten Schritt wird das Programm noch einmal komplett durchgegangen. Nun kann die Symboltabelle benutzt werden, um die Eingabe auf Typkorrektheit zu überprüfen.

Umgang mit Gültigkeitsbereichen

MiniJava hat ein sehr einfaches Typsystem, das die Behandlung von Variablen besonders einfach macht.

Nahezu alle Sprachen erlauben kompliziertere Gültigkeitsbereiche für Variablen.

```
class C {
  int a; int b; int c;
  public void m () {
    System.out.println(a+c);
    int j = a+b;
    String a = "hello";
    System.out.println(a);
    for (int k = 0; k < 23; k++) {
       System.out.println(j+k);
    }
    ...
```

Umgang mit Gültigkeitsbereichen

Man verwendet *Umgebungen*, um die Gültigkeit von Variablen zu verfolgen.

Eine Umgebung Γ ist eine Abbildung von Variablennamen auf semantische Werte, z.B. Typen.

```
class C {                          Γ0 = {g ↦ string, a ↦ int}
int a; int b; int c;               Γ1 = Γ0 + {a ↦ int, b ↦ int, c ↦ int}
public void m () {
  System.out.println(a+c);
  int j = a+b;                                Γ2 = Γ1 + {j ↦ int}
  String a = "hello";                         Γ3 = Γ2 + {a ↦ String}
  System.out.println(a);
  for (int k = 0; k < 23; k++) {
     System.out.println(j+k);                 Γ4 = Γ3 + {k ↦ int}
  }                                           Γ3
  ...
```

Hierbei steht $\Gamma + \Delta$ für die Umgebung, die man erhält, wenn man in Γ alle von Δ definierten Variablen durch ihre Werte in Δ *überschreibt*.

Umgang mit Gültigkeitsbereichen

Umgebungen werden in den Typisierungsregeln benutzt. Die Typurteile sind durch Umgebungen parametrisiert.

Expressions

$$\frac{x \mapsto t \text{ in } \Gamma}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

Statements

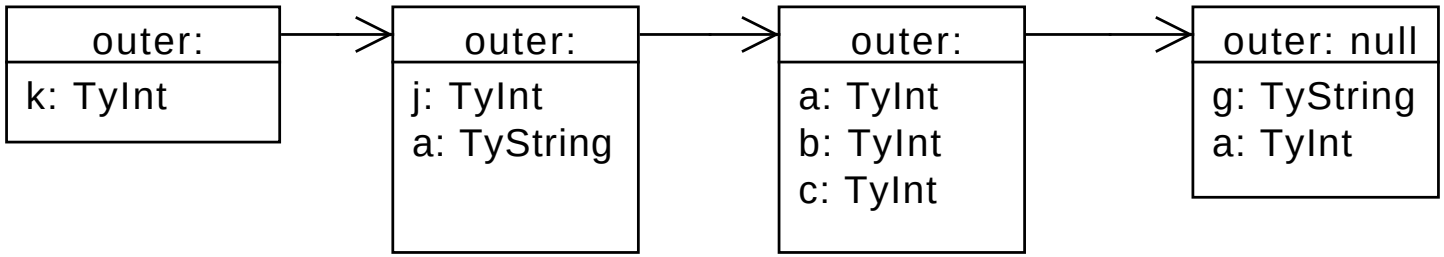
$$\frac{\Gamma \vdash s_1 \text{ ok} \qquad \Gamma \vdash s_2 \text{ ok}}{\Gamma \vdash s_1 ; s_2 \text{ ok}}$$

$$\frac{\Gamma \vdash \texttt{int } x \text{ ok} \qquad \Gamma + \{x \mapsto \texttt{int}\} \vdash s \text{ ok}}{\Gamma \vdash \texttt{int } x ; s \text{ ok}}$$

(N.B. In Sprachen wie Java gibt es noch zusätzliche Regeln, z.B. dass lokale Variablen einander nicht überschatten können)

Implementierung von Umgebungen

Umgebungen können als hierarchische Hash-/TreeMaps implementiert werden:



- Betreten eines Blocks durch Hinzufügen einer neuen Tabelle, deren `outer`-Feld zur Tabelle des aktuellen Güligkeitsbereichs zeigt.
- Verlassen eines Güligkeitsbereichs durch Löschen der aktuellen Tabelle. Die durch `outer` referenzierte Tabelle wird die neue aktuelle Tabelle.
- Um den Typ einer Variablen zu ermitteln, wird zuerst in der aktuellen Tabelle nachgeschaut, dann in der `outer`-Tabelle usw. bis ein Wert gefunden ist.

Umgang mit Vererbung

Bei Einfachvererbung kann die Sichtbarkeit von Methoden und Feldern wie bei den Umgebungen implementiert werden (`super` spielt die Rolle von `outer`).

Die Typisierungsregeln müssen modifiziert werden, um der Unterklassenrelation $\leq$ Rechnung zu tragen.

Beispiele

$$\frac{x\colon \mathtt{C} \qquad \mathtt{C} \text{ hat Methode } t_1\ m(t_2) \qquad e\colon t_2' \leq t_2}{x.m(e)\colon t_1}$$

$$\frac{x\colon t \qquad e\colon t' \qquad t \leq t'}{x := e \text{ ok}}$$

Ihre Aufgabe

Implementieren Sie die Typüberprüfung für MiniJava
ohne geschachtelte Blockstruktur oder Vererbung.

Es bietet sich an, die Implementierung in zwei Phasen zu gliedern:

1. Erzeugung der Symboltabelle.
   Die abstrakte Syntax des Eingabeprogramms wird ein Mal durchgegangen und die Symboltabelle wird dabei schrittweise aufgebaut.
2. Typüberprüfung.
   Im zweiten Schritt wird das Programm noch einmal komplett durchgegangen. Nun kann die Symboltabelle benutzt werden, um die Eingabe auf Typkorrektheit zu überprüfen.