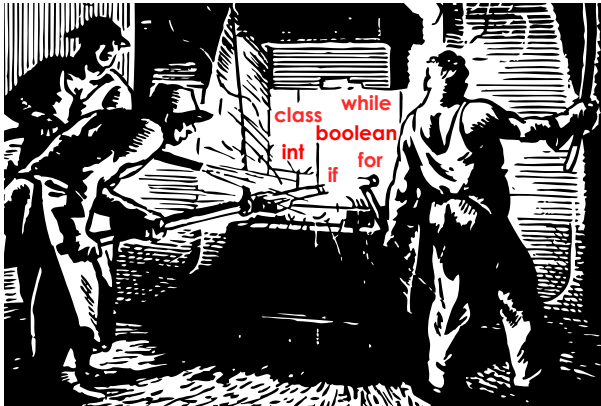


# Fortgeschrittener Compilerbau

## 1. Block: OO-Sprachen am Beispiel von Bantam



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT





# Organisatorisches

- ▶ Jeweils auszugsweise  
**Engineering a Compiler, 2. Auflage**  
von Keith Cooper und Linda Torczon, MKP/Elsevier 2011  
  
**Advanced Compiler Design and Implementation**  
von Steven Muchnick, Morgan-Kaufman 1997
- ▶ Ausgewählte wissenschaftliche Veröffentlichungen (“papers”)
  - ▶ Verlinkt auf der Vorlesungswebseite.



- ▶ Zunächst **zweimal** die Woche
  - ▶ Di 16:15 - 17:45 Uhr
  - ▶ Do 11:40 - 13:10 Uhr
  - ▶ Um möglichst schnell Wissen für Praktikum aufzubauen
- ▶ Nach ca. halber Vorlesungszeit nur noch Dienstags
- ▶ Ziel: ca. 3 SWS über die ganze Vorlesungszeit
- ▶ Diskussion zu Aufzeichnungen



- ▶ Theoretische Übungsblätter zur Vertiefung des Stoffs und Klausurvorbereitung.



- ▶ Theoretische Übungsblätter zur Vertiefung des Stoffs und Klausurvorbereitung.
- ▶ Insgesamt 2 Übungsblätter mit schriftlichen Aufgaben zu Vorlesungsthemen.

- ▶ Theoretische Übungsblätter zur Vertiefung des Stoffs und Klausurvorbereitung.
- ▶ Insgesamt 2 Übungsblätter mit schriftlichen Aufgaben zu Vorlesungsthemen.
- ▶ Bearbeitung freiwillig, keine Abgaben, kein Bonus;
- ▶ Lösungsvorschlag wird nach 2 Wochen im Moodle bereitgestellt.
- ▶ Betreuung in Tutorensprechstunden (Termine werden bekanntgegeben) und im Moodle-Forum.



- ▶ Praktische Anwendung der Algorithmen zur Analyse und Optimierung aus der Vorlesung
- ▶ Separater Moodle-Kurs “Praktikum Compilerbau”
- ▶ Bearbeitung startet erst im Mai.
- ▶ Weitere Informationen in einer der nächsten Vorlesungen.





# Bantam

- ▶ Beispielsprache für Übungsbetrieb: Bantam, eine Untermenge von Java
- ▶ Ausgabe: MIPS Assembler
- ▶ Veranschaulicht Implementierungskonzepte von OO-Sprachen

- ▶ Beispielsprache für Übungsbetrieb: Bantam, eine Untermenge von Java
  - ▶ Ausgabe: MIPS Assembler
  - ▶ Veranschaulicht Implementierungskonzepte von OO-Sprachen
  
  - ▶ Material
    - ▶ ESA-Version des Compiler-Frameworks + JavaDoc
    - ▶ Beispielcode, um damit Bantam-Programme in MIPS-Assembler zu übersetzen
    - ▶ MIPS-Simulator MARS, erweitert um einen Syscall 18: Aktuelle Zeit
- Außerdem, für Interessierte...
- ▶ Umfassendes Handbuch "Lab Manual" (ca. 100 Seiten)
  - ▶ Auszug aus den Original-Quellen: Vollständige IR(s)

- ▶ Beispielsprache für Übungsbetrieb: Bantam, eine Untermenge von Java
  - ▶ Ausgabe: MIPS Assembler
  - ▶ Veranschaulicht Implementierungskonzepte von OO-Sprachen
  
  - ▶ Material
    - ▶ ESA-Version des Compiler-Frameworks + JavaDoc
    - ▶ Beispielcode, um damit Bantam-Programme in MIPS-Assembler zu übersetzen
    - ▶ MIPS-Simulator MARS, erweitert um einen Syscall 18: Aktuelle Zeit
- Außerdem, für Interessierte...
- ▶ Umfassendes Handbuch "Lab Manual" (ca. 100 Seiten)
  - ▶ Auszug aus den Original-Quellen: Vollständige IR(s)
- 
- ▶ Forum zu Fragen rund um Bantam im Moodle-Kurs



- ▶ Java-artige objektorientierte Sprache
- ▶ Stark typisierte Sprache
  - ▶ Statische und dynamische Typprüfung



- ▶ Java-artige objektorientierte Sprache
- ▶ Stark typisierte Sprache
  - ▶ Statische und dynamische Typprüfung
  
- ▶ Eingebaute primitive Typen
  - ▶ `int` und `boolean`

- ▶ Java-artige objektorientierte Sprache
- ▶ Stark typisierte Sprache
  - ▶ Statische und dynamische Typprüfung
  
- ▶ Eingebaute primitive Typen
  - ▶ `int` und `boolean`
  
- ▶ Eingebaute Klassen
  - ▶ `Object`, `String`, `TextIO`, `Sys`

- ▶ Java-artige objektorientierte Sprache
- ▶ Stark typisierte Sprache
  - ▶ Statische und dynamische Typprüfung
- ▶ Eingebaute primitive Typen
  - ▶ `int` und `boolean`
- ▶ Eingebaute Klassen
  - ▶ `Object`, `String`, `TextIO`, `Sys`
- ▶ Programmausführung beginnt in
  - ▶ Klasse `Main`, Methode `main()`
  - ▶ Kurz: `Main.main()`



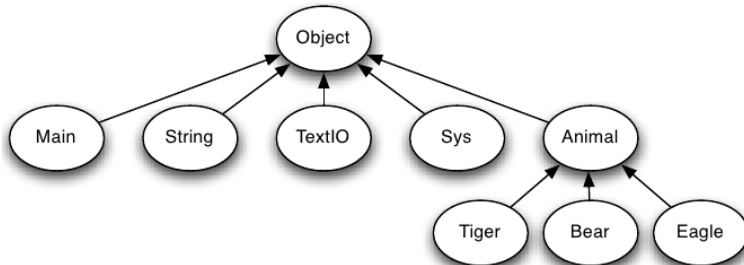


- ▶ Obermenge der einfachen Version Bantam Base, zusätzlich
  - ▶ Arrays
  - ▶ break
  - ▶ ++, --

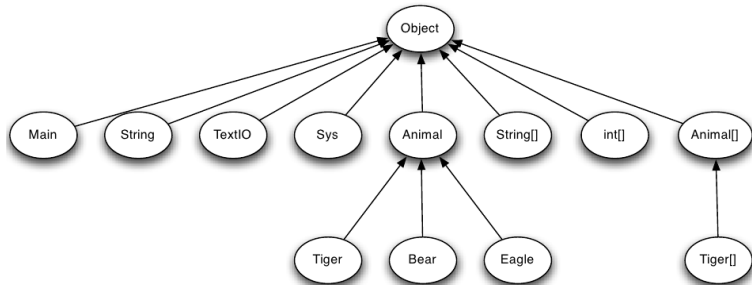


- ▶ Obermenge der einfachen Version Bantam Base, zusätzlich
  - ▶ Arrays
  - ▶ `break`
  - ▶ `++`, `--`
  
- ▶ Bei uns: **Untermenge** von Bantam Extended
  - ▶ Arrays
  - ▶ Kein `break`, `++`, `--`,
    - ▶ Sonst zu komplizierte Optimierung
  - ▶ Garbage Collection deaktiviert
    - ▶ Würde Instruktionszählung zu stark verfälschen

Einfache Vererbung, keine Interfaces



Vererbungs-hierarchie der Basistypen wird auf **verwendete** Arrays angewandt



Array-Typen und ihre Vererbung werden **automatisch** angelegt

- ▶ Falls `Object []` benutzt worden wäre ...
- ▶ ... hätte `Animal []` von `Object []` geerbt

---

```
class <name> [ extends <parent> ] {  
  <members>  
}
```

---

- ▶ Bezeichner beachten Groß/Kleinschreibung
- ▶ Keine Modifikatoren für Klassen
  - ▶ Kein `public`, `package`, `abstract`, `static`, `implements`
- ▶ Falls nicht anders angegeben, erben Klassen von `Object`



---

```
<type> <name> [ = <expression> ] ;
```

---

- ▶ Definieren Typ und Namen der Attribute
- ▶ Optional: Ausdruck zur Initialisierung mit Startwert
  - ▶ Wenn fehlt: Initialisierung je nach Typ auf 0, false, null



```
class Animal {  
    int    numLegs = 4;  
    boolean canFly;           // false  
    Animal mate;             // null, rekursiver Typ  
    int    strength = (numLegs * 10) + 50;  
                // darf vorher definierte Attribute benutzen  
    Object obj = new Object(); // OK  
    Animal prey = new Animal(); // Illegal  
}
```

- ▶ Initialisierungsausdrücke werden nur **einmal** bei Objekterzeugung ausgewertet
  - ▶ In Superklassen vor Subklassen
- ▶ Instanziierung von Objekten der eigenen Klasse oder von Subklassen **illegal**
  - ▶ Endloser Rekursiver Aufruf der Initialisierung
  - ▶ Führt bei Ausführung zum Abbruch wegen Speichermangel

- ▶ Zugriff auf Attribute nur über `this` und `super`
  - ▶ **Nicht** über eine Objektreferenz, also nicht `mate.canFly = true`
- ▶ Keine Zugriffsmodifikatoren wie `public`, `static`, `final`
- ▶ Alle Attribute sind implizit `protected`
  - ▶ Zugreifbar aus Klasse selbst und ihren Unterklassen



# Bestandteile von Klassen: Methoden

## Methods



```
<type> <name> ( [ <parameters> ] ) {  
    <statements>  
    return [ <expression> ];  
}
```

- ▶ Alle Methoden sind implizit `public`
- ▶ **Kein** Überladen von Methodennamen
  - ▶ Alle Methodennamen in einer Klasse müssen eindeutig sein
- ▶ Alle Methoden **müssen** als letzte Anweisung ein `return` haben
  - ▶ Auch solche, die den Rückgabetyt `void` haben!
  - ▶ Es darf nur **genau ein** `return` in jeder Methode geben
  - ▶ Härtere Anforderung als im “normalen” Bantam Java
  - ▶ Grund: Vereinfacht Optimierung (nur ein Ausgang aus Methode)



---

```
class Animal {  
    <member definitions>  
    void fight (int amount, boolean isWinner) {  
        if (isWinner)  
            strength = strength + amount * 5;  
        else  
            strength = 0;  
        return; // zwingend erforderlich  
    }  
}
```

---



```
class Main {
  int x0 = 0;
  void main () {
    int a = x0;
    {b1
      int b = x0;
      int x1 = 1;
      int c = x1;
      {b2
        // this would be illegal:
        // int x2 = 2;
        int d = x1;
      }b2
    }b1
    {b3
      int e = x0;
      int x3 = 3;
      int f = x3;
    }b3
  }b1
  return;
}
```

- ▶ Geltungsbereiche werden in { ... } eingeschlossen
- ▶ Lokale Variablendeklarationen müssen **immer** einen Initialisierungsausdruck haben
- ▶ Lokale Variablen können Attribute gleichen Namens überlagern
  - ▶ Dann Zugriff auf überlagertes Attribut explizit über `this` möglich
- ▶ Lokale Variablen können sich **nicht** gegenseitig überlagern

```
class Main {  
    void main () {  
        ...  
        if (animal.getStrength() > 100) {  
            animal.fight(100, true);  
            strength = animal.getStrength();  
        }  
        ...  
    }  
}
```

- ▶ Wie in Java, `else`-Zweig ist optional
- ▶ `else` bindet immer an innerstes noch offenes `if`
- ▶ Aber keine `switch/case`-Anweisung

```
class Main {  
    void main () {  
        ...  
        while (animal.getStrength() < 100)  
            animal.fight(100, true);  
        ...  
    }  
}
```

- ▶ Wie in Java, aber kein `break/continue`



---

```
for ( [ < initialization > ] ; [ <predicate> ] ; [ <update> ] )  
  <statement>
```

---

- ▶ Ähnlich zu Java, aber kein `break/continue`
- ▶ Aber **keine** Deklaration neuer lokaler Variablen in `for`-Anweisung

---

```
// falsch!  
for ( int i = 0; i < 10; i = i+1 )  
  sum = sum + i;
```

---

- ▶ Bestehen aus Untermenge von **Ausdrücken**
- ▶ Zuweisungen, Methodenaufrufe, instanzieren neuer Objekte mit `new`

---

```
x = y + 2;  
rect.draw(COLOR_GREEN);  
(new Rectangle()).init(20,30);
```

---



*name = expression*

*name [ index ] = expression*

*objref . name = expression*

*objref . name [ index ] = expression*

- ▶ Flexibilität der linken Seite eingeschränkt gegenüber Java
- ▶ Maximal **eine** Objektreferenz, **nicht** `shape.rgbcolor.red = 255`
  - ▶ Gültige Objektreferenzen sind nur `this` und `super`
- ▶ Wert des gesamten Ausdrucks ist *expression* auf rechter Seite
- ▶ Typprüfung
  - ▶ Bei primitiven Typen: Müssen **identisch** sein auf LHS und RHS
  - ▶ Bei Instanzen von Klassen: Klassen müssen **kompatibel** sein
    - ▶ Identisch
    - ▶ oder LHS Superklasse von RHS





*methname* ( *actparameters* )  
*objref* . *methname* ( *actparameters* )

- ▶ Wenn keine Objektreferenz angegeben wird *this* angenommen
  - ▶ Auch möglich: *super*, um Suche nach Methode in Superklasse zu beginnen
  - ▶ Falls Objektreferenz *null*, Exception *\_null\_pointer\_error* auslösen
- ▶ *methname* muß in Klasse/Superklasse der Objektreferenz definiert sein
- ▶ *actparameters* müssen zu formalen Parametern der Methode passen
- ▶ Parameterübergabe
  - ▶ Von links nach rechts
  - ▶ Als *call-by-value*
- ▶ Polymorphismus wird unterstützt
  - ▶ Laufzeittyp von *objref* bestimmt, welche Implementierung von *methname* ausgeführt wird

```
class Animal {  
    <member definitions>  
    // Simuliert einen Konstruktor  
    Animal init (int l, boolean f) {  
        numLegs = l;  
        canFly = f;  
        return this;  
    }  
}  
class Main {  
    void main() {  
        Animal animal = (new Animal()).init (4, false);  
        return;  
    }  
}
```

- ▶ Anlegen immer mit `new`  
*Klassenname* ()
  - ▶ Beachte: Keine Parameter für `new`, da keine Konstruktoren existieren
- ▶ Konstruktorverhalten manuell nachbilden durch Init-Methode
  - ▶ Wird aber nicht automatisch aufgerufen!

`new basetype [ size ]`

- ▶ Legt neues Array für *size* Elemente des Typs *basetype* an
- ▶ Initialisierung je nach Typ auf 0, `false`, `null`
- ▶ Objekte müssen einzeln neu angelegt und ins Array eingetragen werden
- ▶ Einschränkungen in Bantam
  - ▶ Nur eindimensionale Arrays, keine Arrays-von-Arrays
  - ▶ Wenn Garbage Collection aktiviert ist, max 1500 Elemente je Array

## Beispiel

```
int [] numArray = new int [10];
```

( *targetclass* ) ( *expression* )

- ▶ Beachte: *expression* muß immer geklammert sein
- ▶ *targetclass* muß typkompatibel zu dynamischem Typ von *expression* sein
  - ▶ Identisch
  - ▶ Oder Superklasse
- ▶ Beachte: Nur Objektreferenzen können typkonvertiert werden
  - ▶ Keine primitiven Typen wie `boolean` oder `int`

# Beispiel: Typkonvertierung



---

```
// Upcasts, funktionieren immer  
Animal animal = new Tiger();  
Animal animal = (Animal)new Tiger();
```

```
// Downcast, geht nur wenn animal tatsächlich ein Tiger oder eine  
// Subklasse davon ist, oder den Wert null hat  
Tiger tiger = (Tiger)(animal);
```

```
// Gilt auch für Arrays: OK, falls alle Elemente von animals  
// Tiger oder Subklassen davon sind  
Tiger[] tigers = (Tiger [])( animals);
```

---

# Beispiel: Typkonvertierung

---

```
// Upcasts, funktionieren immer  
Animal animal = new Tiger();  
Animal animal = (Animal)(new Tiger());
```

```
// Downcast, geht nur wenn animal tatsächlich ein Tiger oder eine  
// Subklasse davon ist, oder den Wert null hat  
Tiger tiger = (Tiger)(animal);
```

```
// Gilt auch für Arrays: OK, falls alle Elemente von animals  
// Tiger oder Subklassen davon sind  
Tiger[] tigers = (Tiger [])( animals);
```

---

↳ **Dynamische Typprüfung** erforderlich!

Löst Exception `_class_cast_error` bei inkompatiblen Typen aus



---

```
Animal[] animals = (Animal[]) ( tigers );
```

```
// hier Exception _array_store_error  
animal[0] = new Bear();
```

```
// hier aber OK  
animal[0] = (Bear) null;
```

---

- ▶ `animals` speichert Tiger
- ▶ Auch nach Typkonvertierung zu `Animals []`
- ▶ Zuweisung von Bär  $\neq$  `null` an Array fehlerhaft
- ▶ Löst Exception `_array_store_error` aus
- ▶ `null` ist aber zu allen Klassen typkompatibel



*expression instanceof targetclass*

- ▶ *expression* muß Objektreferenz liefern
  - ▶ Keinen primitiven Typ
- ▶ Liefert `true` falls dynamische Klasse der Objektreferenz
  - ▶ ...identisch zu *targetclass* ist, oder
  - ▶ ...eine Subklasse von *targetclass* ist
- ▶ Liefert `false` auch, wenn Objektreferenz = `null`



### Skalare Typen

---

```
Animal animal = getNextAnimal();  
if (animal instanceof Tiger)  
    numTigers = numTigers + 1;  
else if (animal instanceof Bear)  
    numBear = numBear + 1;  
else if (animal instanceof Eagle)  
    numEagles = numEagles + 1;
```

---

### Array Typen

---

```
if (animals instanceof Tiger[]) {  
    Tiger[] tigers = (Tiger [])( animals);  
    ...  
}
```

---

# Ausdrücke: Operatoren

## Im wesentlichen wie in Java



```
int x = 0;
int y = 1;
int z = 2;

x = y + z; // x is set to 3
x = y - z; // x is set to -1
x = y * z; // x is set to 2
x = y / z; // x is set to 0
x = y % z; // x is set to 1
x = -y; // x is set to -1
```

```
boolean b1 = false;
boolean b2 = false;
boolean b3 = true;

b1 = b2 && b3; // b1 is set to false
b1 = b2 || b3; // b1 is set to true
b1 = !b2; // b1 is set to true
```

```
boolean b = false;
int i1 = 0;
int i2 = 1;
int i3 = 0;

Object o1 = new Object();
Object o2 = new Object();
Object o3 = o1;

b = i1 == i2; // b is set to false
b = i1 != i2; // b is set to true
b = o1 == o2; // b is set to false
b = o1 == o3; // b is set to true
b = i1 < i2; // b is set to true
b = i1 >= i3; // b is set to true
```

- ▶ Keine Kurzschreibweisen +=, &&=, ++, -- etc.



## Konstanten

- ▶ `int`: Nur in Dezimaldarstellung,  $-2147483648 \dots 2147483647$  ( $2^{31} - 1$ )
- ▶ `boolean`: `true`, `false`
- ▶ `String`: Nur ASCII Zeichen, übliche Steuerzeichen `\n`, `\t`, etc.



## Konstanten

- ▶ `int`: Nur in Dezimaldarstellung,  $-2147483648 \dots 2147483647$  ( $2^{31} - 1$ )
- ▶ `boolean`: `true`, `false`
- ▶ `String`: Nur ASCII Zeichen, übliche Steuerzeichen `\n`, `\t`, etc.

## Variablen

- ▶ Lokale Variablen: Über Namen
- ▶ Attribute dieser Klasse: Namen, optional mit vorangestelltem `this`
- ▶ Attribute von Superklassen: Namen mit vorangestelltem `super`
- ▶ Größe eines Arrays: Array Name gefolgt von `.length`
- ▶ Ein Array-Element: Array Name gefolgt von `[ index ]`
- ▶ Beachte: **Keine** Zugriffe auf Attribute über sonstige Objektreferenzen

# Eingebaute Klassen

## Implementiert als MIPS-Assembler



Class	Method signature	Method description
Object	<b>Object clone()</b>	copy an object
	<b>boolean equals(Object s)</b>	test if objects are equal ( <i>i.e.</i> , alias)
	<b>String toString()</b>	return string representation of object
Sys	<b>void exit(int status)</b>	exit program with specified status
	<b>int time() *</b>	return UTC time
	<b>int random() *</b>	return random int
String	<b>int length()</b>	return string length
	<b>boolean equals(Object s)</b>	test if strings are equivalent
	<b>String toString()</b>	return itself
	<b>String substring(int beginIndex, int endIndex)</b>	return substring between the indices
	<b>String concat(String s)</b>	return concatenated string
TextIO	<b>void readFile(String filename)</b>	set to read from specified file
	<b>void writeFile(String filename)</b>	set to write to specified file
	<b>void readStdin()</b>	set to read from standard input
	<b>void writeStdout()</b>	set to write to standard output
	<b>void writeStderr()</b>	set to write to standard error
	<b>String getString()</b>	read next string
	<b>int getInt()</b>	read next int
	<b>void putString(String s)</b>	write specified string
	<b>void putInt(int i)</b>	writes specified int

# Beispiel: Benutzung der I/O-Operationen



```
class Main {
    TextIO io = new TextIO();
    String output = "";

    void error () {
        io.writeStderr ();
        io.putString("Bad_input;_exiting\n");
        (new Sys()).exit (1);
        return;
    }

    String getNextLine() {
        String s = io.getString ();
        if (s == null || s.length () < 2)
            error ();
        return s.substring(1, s.length ());
    }
}
```

```
void main() {
    String s = "";
    io.readStdin();
    int n = io.getInt ();
    if (n < 1)
        error ();

    io.readFile("input.txt");
    int i = 0;
    for (i = 0; i < n && !s.equals("quit"); i++) {
        s = getNextLine();
        output = output.concat(s).concat("\n");
    }

    io.writeStdout ();
    io.putString(output);

    io.writeFile ("output.txt ");
    io.putString(output);
    return;
}
}
```



# Benutzung des Systems



- ▶ `bantamc-lib-obf.jar`: Compiler-Framework
- ▶ `ExampleDriver.java`: Einfache Compiler-Steuerung, basierend auf Framework
- ▶ Einfachster Fall: Ausprobieren der Beispielprogramme von der Website  

```
javac -cp bantamc-lib-obf.jar ExampleDriver.java
```

```
java -cp bantamc-lib-obf.jar:commons-lang.jar ExampleDriver X.btm
```

(benötigt aktuelle Apache CommonsLang-Bibliothek)
- ▶ Erzeugt MIPS-Assembler in Datei `out.s`
  - ▶ Kann mit `-o dateiname` in andere Datei geschrieben werden
- ▶ Compiler kann auch mehrere Dateien zusammen als ein Programm übersetzen
  - ▶ ...ging in Triangle ja nicht



- ▶ Mehr Informationen zu MARS

<http://courses.missouristate.edu/kenvollmar/mars/>

- ▶ Verwenden Sie die gepatchte Version von unserer Website

(Mars4\_4+Syscall18.jar)

- ▶ Außerdem erforderlich:

- ▶ Verweis auf zu verwendende Initialisierungsdatei
- ▶ `exceptions.mars.s`: Stellt Laufzeitumgebung bereit
- ▶ **Wichtig**: Diese (Bantam-eigene) `exceptions.mars.s` verwenden
- ▶ Enthält z.B. Implementierung eingebauter Klassen und Garbage Collection

Ausführen des Programmes in `out.s` auf der Kommandozeile:

```
java -jar Mars4_4+Syscall18.jar sm me exceptions.mars.s out.s
```

- ▶ `sm` startet die Ausführung in der vom Compiler erzeugten `main`-Routine
- ▶ `me` konfiguriert zu verwendende Laufzeitumgebung
- ▶ Optionaler Parameter `ic` liefert am Ende Anzahl ausgeführter MIPS-Instruktionen
  - ▶ Enthält auch Instruktionen aus Laufzeitumgebung



# Zielarchitektur und Laufzeitsystem

- ▶ RISC-Prozessor: Nur wenige einfache Befehle
  - ▶ Die aber (hoffentlich) schnell ausgeführt werden
- ▶ Load-Store-Architektur: Speicherzugriffe **nur** mit dedizierten Befehlen
  - ▶ Z.B. keine Speicherzugriffe in arithmetischen Instruktionen
- ▶ 32 Register, jeweils 32 Bit breit
  - ▶ In der Regel allgemein verwendbar
  - ▶ Keine Trennung zwischen Daten- und Adressregistern



---

Registername	Üblicher Verwendungszweck
\$zero	Konstante 0, kann nur gelesen werden
\$at	Temporäre Variable für Pseudoinstruktionen (z.B. <code>b1t</code> )
\$v0...\$v1	Ergebnis von Berechnungen oder Funktionen
\$a0...\$a3	Aufrufparameter für Routinen
\$t0...\$t9	Temporäre Variablen
\$s0...\$s7	Gesicherte Variablen
\$k0...\$k1	Temporäre Variablen für Betriebssystem
\$gp	Zeiger auf globale Variablen im Speicher
\$sp	Stapelzeiger
\$fp	Zeiger auf aktuellen Stack-Frame
\$ra	Rücksprungadresse aus Routine

---

- ▶ `$s0`: In der Regel `this`-Zeiger auf aktuelle Instanz
- ▶ `$v0`: Für Zwischenergebnisse von Rechnungen, Rückgabewert von Methoden
- ▶ `$a0`: Empfängerobjekt bei Methodenaufruf
- ▶ `$a1`: Zeilennummer dieses Methodenaufrufs
- ▶ `$a2`: Quelldateiname dieses Methodenaufrufs
- ▶ `$t0...$t3`: Hilfsregister
- ▶ Stack: Adressiert über `$sp`, wächst von oben nach unten
- ▶ Parameterübergabe für Methoden via Stack
  - ▶ Von links nach rechts mit call-by-value
- ▶ Standard-Compiler hat **keine** intelligente Registerallokation
  - ▶ Register werden bestenfalls **innerhalb** eines Sprachkonstruktes verwendet
  - ▶ In der Regel Datenhaltung in **lokalen** Variablen im Stack-Frame
- ▶ Keine Verwaltung von globalen Variablen via `$gp`



# Dynamic Dispatch

# Polymorphe Methodenaufrufe

## *dynamic dispatch*



Statische Bindung möglich

---

```
Rect rect = (new Rect()).init (0,0,100,200);
```

```
rect.draw();
```

---



# Polymorphe Methodenaufrufe

## *dynamic dispatch*

Statische Bindung möglich

---

```
Rect rect = (new Rect()).init (0,0,100,200);  
  
rect.draw();
```

---

Dynamische Bindung erforderlich

---

```
Rect rect = (new Rect()).init (0,0,100,200);  
  
Shape shape = rect;  
  
shape.draw();
```

---



- ▶ An unterschiedliche Stellen springen
  - ▶ Potentiell beliebig viele
  - ▶ Indirekter Sprung: `jra1 $t0`

- ▶ An unterschiedliche Stellen springen
  - ▶ Potentiell beliebig viele
  - ▶ Indirekter Sprung: `jra1 $t0`
  
- ▶ An welche von diesen Stellen springen?
  - ▶ Methode heißt immer `draw()`
  - ▶ Aber unterschiedliche Implementierungen in verschiedenen Klassen
  - ▶ Auswahl der Implementierung anhand dynamischer Klasse der Instanz



- ▶ An unterschiedliche Stellen springen
  - ▶ Potentiell beliebig viele
  - ▶ Indirekter Sprung: `jr al $t0`
  
- ▶ An welche von diesen Stellen springen?
  - ▶ Methode heißt immer `draw()`
  - ▶ Aber unterschiedliche Implementierungen in verschiedenen Klassen
  - ▶ Auswahl der Implementierung anhand dynamischer Klasse der Instanz
  
- ▶ Idee
  1. Dynamische Klasse von jeder Instanz merken
  2. Je Klasse für jede Methode Startadresse der Routine merken
  3. Damit zur Laufzeit passende Routine aufrufen

# Datenstrukturen für Dynamische Bindung

## Zunächst vereinfacht: Ohne Vererbung



- ▶ In jeder Instanz Typinformationen mitführen
  - ▶ Einfachster Fall: Alle Typen (Klassen) durchnummerieren
  - ▶ **Type-ID** je Instanz speichern

# Datenstrukturen für Dynamische Bindung

## Zunächst vereinfacht: Ohne Vererbung



- ▶ In jeder Instanz Typinformationen mitführen
  - ▶ Einfachster Fall: Alle Typen (Klassen) durchnummerieren
  - ▶ **Type-ID** je Instanz speichern
  
- ▶ Alle Methoden einer Klasse durchnummerieren
  - ▶ **Method-ID**
- ▶ Tabelle ordnet jeder Method-ID die Startadresse der Routine zu
  - ▶ Oft genannt *virtual method table*, *vtable*



- ▶ In jeder Instanz Typinformationen mitführen
  - ▶ Einfachster Fall: Alle Typen (Klassen) durchnummerieren
  - ▶ **Type-ID** je Instanz speichern
  
- ▶ Alle Methoden einer Klasse durchnummerieren
  - ▶ **Method-ID**
- ▶ Tabelle ordnet jeder Method-ID die Startadresse der Routine zu
  - ▶ Oft genannt *virtual method table*, *vtable*

Beispiel: Vtable von Bantam Object-Klasse

---

```
Object_dispatch_table:  
  .word Object.clone      # Method-ID 0  
  .word Object.equals     # Method-ID 1  
  .word Object.toString   # Method-ID 2
```

---

- ▶ Type-ID in jeder Instanz
- ▶ Tabelle enthält zu jeder Type-ID die Vtable
- ▶ Dann Nachschlagen mit Method-ID in Vtable um Routine zu bestimmen



# Beispiel für 1. Implementierungsversuch

```
Object foo = new Object(); foo.toString()
```



```
Object_dispatch_table:
.word Object.clone           # Method-ID 0
.word Object.equals         # Method-ID 1
.word Object.toString       # Method-ID 2
...
class_to_vtables:
.word Object_dispatch_table # Type-ID 0
.word TextIO_dispatch_table # Type-ID 1
.word Sys_dispatch_table   # Type-ID 2
...
Main.main: # Beispiel: Rufe Object.toString auf
...
lw      $t0  0($s0)           # s0: this, Type-ID am Anfang
sll     $t0  $t0  2           # Byte offset bestimmen
li      $t1  class_to_vtables
add     $t1  $t1  $t0
lw      $t0  0($t1)          # Vtable von Object in $t0
lw      $t1  8($t0)          # Routine Object.toString in $t1
jalr   $t1
...
```



## Diskussion 1. Versuch

- ▶ Funktioniert
- ▶ Potentiell aber sehr langsam für jeden Methodenaufruf
- ▶ Insbesondere **drei Speicherzugriffe** je Aufruf

## Diskussion 1. Versuch

- ▶ Funktioniert
- ▶ Potentiell aber sehr langsam für jeden Methodenaufruf
- ▶ Insbesondere **drei Speicherzugriffe** je Aufruf

## Verbesserung

- ▶ Trage Vtable-Adresse direkt in Instanzen ein
- ▶ Kostet 1 Wort mehr Speicher je Instanz ...
  - ▶ Unabhängig von Anzahl der Methoden
- ▶ ... läuft aber deutlich schneller

## Beispiel für 2. Implementierungsversuch

```
Object foo = new Object(); foo.toString()
```



---

Object\_dispatch\_table:

```
.word Object.clone           # Method-ID 0  
.word Object.equals         # Method-ID 1  
.word Object.toString       # Method-ID 2
```

...

Main.main: # Beispiel: Rufe Object.toString auf

...

```
lw    $t0    4($s0)           # s0: this, Vtable-Adresse an 2. Stelle  
lw    $t1    8($t0)          # Routine Object.toString in $t1  
jalr  $t1    $t1              # indirekter Aufruf
```

...

---

## Beispiel für 2. Implementierungsversuch

```
Object foo = new Object(); foo.toString()
```



---

Object\_dispatch\_table:

```
.word Object.clone           # Method-ID 0  
.word Object.equals          # Method-ID 1  
.word Object.toString        # Method-ID 2
```

...

Main.main: # Beispiel: Rufe Object.toString auf

...

```
lw    $t0    4($s0)           # s0: this, Vtable-Adresse an 2. Stelle  
lw    $t1    8($t0)           # Routine Object.toString in $t1  
jalr  $t1    $t1              # indirekter Aufruf
```

...

---

Nur noch zwei Speicherzugriffe je Methodenaufruf.  
Noch weiter verbesserbar?

Idee: Vtable **direkt** in Instanzen speichern

---

```
# Beispiel: Instanz im Speicher
# this zeigt auf den Anfang
.word 0 # 0. Wort: Type-ID
.word Object.clone # 1. Wort: Method clone
.word Object.equals # 2. Wort: Method equals
.word Object.toString # 3. Wort: Method toString
...
Main.main: # Beispiel: Rufe Object.toString auf
...
lw $t0 12($s0) # s0: this, Vtable beginnt im 1. Wort
jalr $t0 # indirekter Aufruf
...
```

---

## 3. Versuch

- ▶ Theoretisch noch schneller
  - ▶ Nur noch **einen** Speicherzugriff pro Methodenaufruf
- ▶ Braucht potentiell deutlich mehr Speicher
  - ▶ Kompletter Vtable **je Instanz**
  - ▶ Extrem teuer bei vielen Instanzen und vielen Methoden
- ▶ Ausblick: Problematisch auch bei Vererbung
  - ▶ Ändert Speicher-Layout der nachfolgenden Attribute

## 3. Versuch

- ▶ Theoretisch noch schneller
  - ▶ Nur noch **einen** Speicherzugriff pro Methodenaufruf
- ▶ Braucht potentiell deutlich mehr Speicher
  - ▶ Kompletter Vtable **je Instanz**
  - ▶ Extrem teuer bei vielen Instanzen und vielen Methoden
- ▶ Ausblick: Problematisch auch bei Vererbung
  - ▶ Ändert Speicher-Layout der nachfolgenden Attribute

## ↳ Weit verbreitet: 2. Implementierung

- ▶ **Gemeinsamer** Vtable für alle Instanzen einer Klasse
- ▶ Jede Instanz hat **eigenen** Verweis auf Vtable ihrer Klasse
- ▶ Auch in Bantam verwendet





Ziel: Ersetze dynamische durch statische Bindung

- ▶ Analysiere Programm
- ▶ Erkenne Fälle, in denen Klasse **statisch** exakt bestimmbar ist
  - ▶ Superklasse reicht nicht!
- ▶ Ersetze dann indirekten Aufruf durch direkten Aufruf

Ziel: Ersetze dynamische durch statische Bindung

- ▶ Analysiere Programm
- ▶ Erkenne Fälle, in denen Klasse **statisch** exakt bestimmbar ist
  - ▶ Superklasse reicht nicht!
- ▶ Ersetze dann indirekten Aufruf durch direkten Aufruf

---

```
Object foo = new Object();
```

```
foo.toString ();    // hier Typ genau bestimmbar
```

---

Main.main: # Beispiel: Rufe Object.toString auf

...

```
    jal    Object.toString          # direkter Aufruf
```

...

---

## Auswirkungen von Vererbung

- ▶ **Erweitere** Funktionalität relativ zur Oberklasse
  - ▶ Füge neue Methoden hinzu
- ▶ **Verändere** Funktion relativ zur Oberklasse
  - ▶ Durch Überschreiben
  - ▶ Methode der Oberklasse muß aufrufbar bleiben (mit `super`)
- ▶ Nicht möglich: Entfernen von Funktionalität
- ▶ In Bantam: Nur einfache Vererbung

## Auswirkungen von Vererbung

- ▶ **Erweitere** Funktionalität relativ zur Oberklasse
  - ▶ Füge neue Methoden hinzu
- ▶ **Verändere** Funktion relativ zur Oberklasse
  - ▶ Durch Überschreiben
  - ▶ Methode der Oberklasse muß aufrufbar bleiben (mit `super`)
- ▶ Nicht möglich: Entfernen von Funktionalität
- ▶ In Bantam: Nur einfache Vererbung

➡ Idee: Realisiere in Vtable der Unterklasse



## 1. Keine Änderungen relativ zur Oberklasse



1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)



1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse



1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
  - ▶ Füge entsprechende Methoden **am Ende** von Vtable(Unter) an





1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
  - ▶ Füge entsprechende Methoden **am Ende** von Vtable(Unter) an
  - ▶ Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand



1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
  - ▶ Füge entsprechende Methoden **am Ende** von Vtable(Unter) an
  - ▶ Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse

1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
  - ▶ Füge entsprechende Methoden **am Ende** von Vtable(Unter) an
  - ▶ Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse
  - ▶ Trage neue Routinen **unter alter Method-ID** in Vtable(Unter) ein



1. Keine Änderungen relativ zur Oberklasse
  - ▶ Vtable(Unter) ist Kopie von Vtable(Ober)
2. Zusätzliche Methoden relativ zur Oberklasse
  - ▶ Füge entsprechende Methoden **am Ende** von Vtable(Unter) an
  - ▶ Alle Method-IDs der Oberklasse haben auch in Unterklasse Bestand
3. Ändere Methoden relativ zur Oberklasse
  - ▶ Trage neue Routinen **unter alter Method-ID** in Vtable(Unter) ein
  - ▶ Alte Implementierungen stehen noch in Vtable(Ober) zur Verfügung

# Beispiel: 1. Keine Änderungen zur Oberklasse

Bantam

---

```
class X {  
    int i;  
}
```

---

Vtable von X

---

```
X_dispatch_table:  
    .word Object.clone    # Method-ID 0  
    .word Object.equals  # Method-ID 1  
    .word Object.toString # Method-ID 2
```

---

## Beispiel: 2. Zusätzliche Methode in Unterklasse



Bantam

---

```
class X {  
    int i;  
    void setI(int i) {  
        this.i = i;  
        return;  
    }  
}
```

---

Vtable von X

---

```
X_dispatch_table:  
.word Object.clone    # Method-ID 0  
.word Object.equals  # Method-ID 1  
.word Object.toString # Method-ID 2  
.word X.setI         # Method-ID 3
```

---

# Beispiel: 2. Zusätzliche Methode in Unterklasse

## Geerbte Methoden bleiben unverändert



### Bantam

```
X x = new X();  
Object obj = x;  
  
obj.toString (); // ruft Method-ID 2 von Object auf  
                // ist auch Method-ID 2 von X
```

### MIPS-Implementierung

```
X_dispatch_table:  
.word Object.clone # Method-ID 0  
.word Object.equals # Method-ID 1  
.word Object.toString # Method-ID 2  
.word X.setl # Method-ID 3  
...  
Main.main: # Beispiel: Rufe Object.toString von x auf  
...  
lw $t0 4($s0) # s0: this = x, Vtable-Adresse an 2. Stelle  
lw $t1 8($t0) # Routine Object.toString in $t1  
jalr $t1 # indirekter Aufruf  
...
```

## Beispiel: 3. Veränderte Methode in Unterklasse

Bantam

---

```
class X {  
    ...  
    String toString () {  
        return "foo".concat(super.toString());  
    }  
}
```

---

Vtable von X

---

X\_dispatch\_table:

<b>.word</b> Object.clone	# Method-ID 0
<b>.word</b> Object.equals	# Method-ID 1
<b>.word</b> X.toString	# Method-ID 2
<b>.word</b> X.setl	# Method-ID 3

---



# Beispiel: 3. Veränderte Methode in Unterklasse

## Geerbte Methode wird überschrieben

```
X x = new X();  
Object obj = x;
```

```
obj.toString (); // ruft Method-ID 2 von X via obj auf
```

```
X_dispatch_table:
```

```
.word Object.clone # Method-ID 0  
.word Object.equals # Method-ID 1  
.word X.toString # Method-ID 2  
.word X.setl # Method-ID 3
```

```
Main.main: # Beispiel: Rufe X.toString von obj auf
```

```
...
```

```
lw $t0 4($s0) # s0: this = obj, Vtable-Adresse an 2. Stelle  
lw $t1 8($t0) # Routine X.toString in $t1  
jalr $t1 # indirekter Aufruf
```

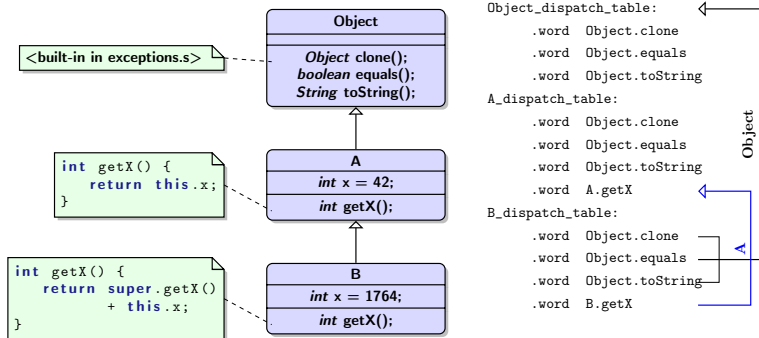
```
...
```

```
X.toString: # Aufruf von super.toString() in X.toString()
```

```
...
```

```
la $t0 Object_dispatch_table # expliziter Zugriff auf Oberklasse  
lw $t1 8($t0) # Routine Object.toString in $t1  
jalr $t1 # indirekter Aufruf
```

# Beispiel: Vererbung mit zwei Unterklassen



# Dynamische Prüfungen beim Methodenaufruf

... bisher weggelassen

Empfängerobjekt muß ungleich `null` sein

```
Object_dispatch_table:
  .word Object.clone   # Method-ID 0
  .word Object.equals  # Method-ID 1
  .word Object.toString # Method-ID 2
Main.main: # Beispiel: Rufe Object.toString von obj auf
...
  move $a0 $s0          # s0: this = obj
  bne  $a0 $zero label45 # this != null
  jal  _null_pointer_error # definiert in Bantam exceptions.s
                                # kehrt nicht mehr zurück (Programmende)

label45:
  lw   $t0 4($a0)       # a0: Empfängerobj, Vtable-Adresse an 2. Stelle
  lw   $t1 8($t0)       # Routine X.toString in $t1
  jalr $t1          # indirekter Aufruf
...
```



# Erzeugen von Objekten

- ▶ Jede neue Instanz benötigt Speicher
  - ▶ Kann vom Betriebssystem mit `syscall` angefordert werden
- ▶ Muß aber richtig initialisiert werden
  - ▶ Type-ID und Zeiger auf Vtable von Klasse
  - ▶ Attribute
    - ▶ Standardwerte `0`, `false`, `null`
    - ▶ Eigene (explizite) Initialisierung
    - ▶ Initialisiere auch Attribute der Oberklasse(n) korrekt

- ▶ Jede neue Instanz benötigt Speicher
  - ▶ Kann vom Betriebssystem mit `syscall` angefordert werden
- ▶ Muß aber richtig initialisiert werden
  - ▶ Type-ID und Zeiger auf Vtable von Klasse
  - ▶ Attribute
    - ▶ Standardwerte `0`, `false`, `null`
    - ▶ Eigene (explizite) Initialisierung
    - ▶ Initialisiere auch Attribute der Oberklasse(n) korrekt

➡ Idee: Lege **Vorlagen** von Objekten an und kopiere diese bei Instanziierung



- ▶ Templates enthalten Type-ID, Vtable und Standardwerte für alle Attribute
  - ▶ In Bantam: *Klasse\_template*
- ▶ Neue Instanzen werden mittels `Object.clone` von Vorlage erzeugt
- ▶ Dann Aufruf einer Hilfsroutine für explizite Initialisierung
  - ▶ In Bantam: *Klasse\_init*

# Beispiel: Objekterzeugung 1



Bantam

```
class X {  
    int i;  
    int j = 42;  
    void setI(int i) { ... }  
    String toString() { ... }  
}
```

MIPS-Implementierung: Vorlage

X\_template:

```
.word 2           # Type-ID  
.word 20          # Gesamtgröße der Template in Bytes  
.word X_dispatch_table # Verweis auf Vtable der Klasse X  
.word 0           # Attribut i, Standardwert  
.word 0           # Attribut j, Standardwert
```



## Beispiel: Objekterzeugung 2

### MIPS-Implementierung: Initialisierung

---

```
X_init :  
...  
jal Object_init      # rufe Init der Oberklasse auf  
li $v0 42  
sw $v0 16($s0)      # initialisiere Attribut j auf 42  
...
```

---

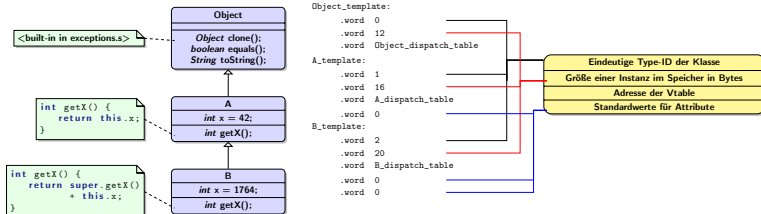
### MIPS-Implementierung: Realisierung von `new X()`

---

```
Main.main:  
...  
la $a0 X_template # Zielobjekt (this) ist X_template  
jal Object.clone  # Kopie erzeugen, Zeiger auf Kopie zurück in $v0  
move $a0 $v0     # Zielobjekt (this) ist neue Instanz von X  
jal X_init       # Explizite Initialisierung der Attribute von neuer Instanz  
...
```

---

# Beispiel: Objektvorlagen mit zwei Unterklassen



- ▶ Werden als **Singletons** realisiert
  - ▶ Für jede im Programm vorkommende Zeichenkette existiert **genau** ein String-Objekt
- ▶ Die String-Vorlagen enthalten bereits die einzelnen Zeichen des Strings
  - ▶ Müssten sonst einzeln initialisiert werden
- ▶ Haben variable Größe
  - ▶ "Foo" (3 Zeichen) vs. "Blahfase1" (9 Zeichen)
  - ▶ ASCII NUL (Wert 0x00) wird als Marker für String-Ende verwendet
  - ▶ String-Objekte müssen aber auf **Wortgrenzen** enden
    - ▶ Sonst schlägt Zugriff auf nachfolgende Daten mit `lw` fehlerhaft
    - ▶ Auffüllen auf Vielfache von vier Bytes

# Beispiel: Objektvorlagen für String-Literale

## Vorlage für "foo"



---

```
String_const_5:  
  .word 11           # Type-ID von Klasse String  
  .word 20          # Gesamtgröße in Bytes  
  .word String_dispatch_table # Vtable für Klasse String  
  .word 3           # String-Länge (ohne NUL)  
  .ascii "foo"     # Zeichen  
  .byte 0          # Endemarker NUL  
  .align 2         # Fülle auf Vielfache von 2^2 auf
```

---

# Beispiel: Objektvorlagen für String-Literale

## Vorlage für "foo"



---

```
String_const_5:  
  .word 11           # Type-ID von Klasse String  
  .word 20           # Gesamtgröße in Bytes  
  .word String_dispatch_table # Vtable für Klasse String  
  .word 3            # String-Länge (ohne NUL)  
  .ascii "foo"      # Zeichen  
  .byte 0            # Endemarker NUL  
  .align 2           # Fülle auf Vielfache von 2^2 auf
```

---

- ▶ Sonderbehandlung **nur** für String-Literale
  - ▶ Dynamisch erzeugte String-Objekte verwenden Vorlage `String_template`
- ▶ Bantam Compiler legt automatisch String-Literale an, z.B. für
  - ▶ Namen jeder Klasse
  - ▶ Dateinamen der Bantam-Quelldateien
  - ▶ ➡ werden für Exception-Fehlermeldungen verwendet



# Methodenprotokoll



- ▶ Übergabe von Parametern
- ▶ Rückgabe von Ergebnis
- ▶ Aufräumen auf Stack
- ▶ **Ausnahmebehandlung** (gab es in Triangle nicht!)



- ▶ Parameter werden auf Stack übergeben
  - ▶ Von links nach rechts
  - ▶ Aufgerufene Methode räumt Stack auf
- ▶ `$a0`: `this`-Zeiger von Empfängerobjekt





- ▶ Parameter werden auf Stack übergeben
  - ▶ Von links nach rechts
  - ▶ Aufgerufene Methode räumt Stack auf
- ▶ `$a0`: `this`-Zeiger von Empfängerobjekt
- ▶ Für Ausnahmebehandlung bei ungültigem Empfängerobjekt
  - ▶ `$a1`: Zeilennummer dieses Aufrufs in Quelldatei
  - ▶ `$a2`: Zeigt auf Vorlage von String-Literal mit Quelldateiname
  - ▶ Werden verwendet zur Ausgabe einer Fehlermeldung in `_null_pointer_error`
    - ▶ Siehe Bantam-spezifisches `exceptions.s`

# Beispiel: Methodenprotokoll

Bantam



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

---

```
class T {  
    ...  
    int bar(int a, int b) { // Aufgerufene Methode  
        return a+b;  
    }  
}  
  
...  
class Main {  
    void main() {  
        int temp;  
        T t = new T();  
        temp = t.bar(42, 23); // Relevante Aufrufstelle  
        ...  
    }  
}
```

---

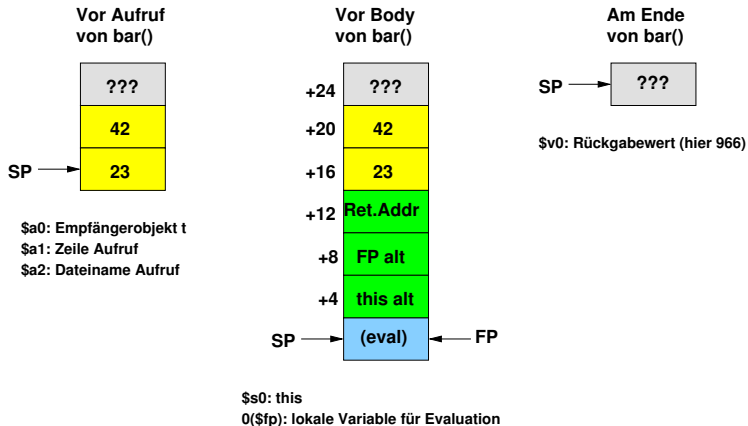
# Beispiel: Methodenprotokoll beim Aufrufer

## MIPS-Implementierung

Main.main:

```
...
li    $v0 42           # \
sw    $v0 -4($sp)     # | Push 42
sub   $sp $sp 4       # /
li    $v0 23          # \
sw    $v0 -4($sp)     # | Push 23
sub   $sp $sp 4       # /
li    $a1 85           # Zeile des Aufrufes in Quelldatei
la    $a2 String_const_3 # String-Literal für Dateiname "TestInherit.btm"
lw    $a0 12($fp)     # Hole Zeiger auf Instanz aus lokaler Variable t
bne   $a0 $zero label34 # Gültiges Empfängerobjekt?
jal   _null_pointer_error # nein, -> Null Pointer Exception: Fehlermeldung
label34:
lw    $t0 8($a0)      # Hole Vtable von Klasse T aus Instanz t
lw    $t0 20($t0)     # Hole Startadresse von Routine bar
jalr  $t0             # indirekter Sprung
sw    $v0 0($fp)     # speichere Rückgabewert in lokale Variable temp
...
```

# Stackorganisation innerhalb einer Methode



# Beispiel: Methodenprotokoll beim Aufgerufenen



T.bar:

```
add $sp $sp -16 # Platz für 3 Worte Verwaltungsdaten und eine lokale Variable
sw  $ra 12($sp) # Sichere Rücksprungadresse auf Stack
sw  $fp 8($sp)  # Sichere alten Framepointer auf Stack (dynamic link)
sw  $s0 4($sp)  # sichere this-Zeiger des Aufrufers auf Stack
move $fp $sp    # aktueller FP zeigt nun auf erste lokale Variable
move $s0 $a0    # this-Zeiger dieser Instanz wurde in $a0 übergeben
lw  $v0 20($fp) # Hole 1. formalen Parameter
sw  $v0 0($fp)  # Lege ab als Zwischenergebnis der aktuellen Rechnung
lw  $v0 16($fp) # Hole 2. formalen Parameter
lw  $t0 0($fp)  # Hole bisheriges Zwischenergebnis der aktuellen Rechnung
mul $v0 $t0 $v0 # Multipliziere 2. Parameter und Zwischenergebnis
lw  $s0 4($sp)  # stelle this-Zeiger des Aufrufers wieder her
lw  $fp 8($sp)  # stelle Framepointer des Aufrufers wieder her
lw  $ra 12($sp) # Hole Rücksprungadresse
add $sp $sp 24  # gib aktuellen Frame _und_ aktuelle Parameter wieder frei
jr  $ra        # Rücksprung zum Aufrufer
```

# Methodenprotokoll: Prolog und Epilog

## Anfang und Ende von Methoden haben immer gleiches Muster



### Prolog

---

```
AnyClass.anyMethod: # #Locals ist Anzahl der lokalen Variablen
  add $sp $sp N      # N = (3 + #Locals) * 4 Bytes
  sw  $ra R($sp)    # R = (2 + #Locals) * 4 Bytes
  sw  $fp F($sp)    # F = (1 + #Locals) * 4 Bytes
  sw  $s0 T($sp)    # T = (0 + #Locals) * 4 Bytes
  move $fp $sp      # aktueller SP zeigt nun auf neuen Framepointer
  move $s0 $a0      # this-Zeiger dieser Instanz wurde in $a0 übergeben
  ...
```

---

# Methodenprotokoll: Prolog und Epilog

Anfang und Ende von Methoden haben immer gleiches Muster



## Prolog

---

```
AnyClass.anyMethod: # #Locals ist Anzahl der lokalen Variablen
add $sp $sp N      # N = (3 + #Locals) * 4 Bytes
sw  $ra R($sp)    # R = (2 + #Locals) * 4 Bytes
sw  $fp F($sp)    # F = (1 + #Locals) * 4 Bytes
sw  $s0 T($sp)    # T = (0 + #Locals) * 4 Bytes
move $fp $sp      # aktueller SP zeigt nun auf neuen Framepointer
move $s0 $a0      # this-Zeiger dieser Instanz wurde in $a0 übergeben
...
```

---

## Epilog

---

```
... # erwarte Rückgabewert in $v0
lw  $s0 T($sp)    # stelle this-Zeiger des Aufrufers wieder her
lw  $fp F($sp)    # stelle Framepointer des Aufrufers wieder her
lw  $ra R($sp)    # Hole Rücksprungadresse
add $sp $sp M     # M = N + #Params * 4 Bytes
jr  $ra           # Rücksprung zum Aufrufer
```

---



# Dynamische Typprüfung





- ▶ Statische Typprüfung in Bantam nicht ausreichend
  - ▶ Down-Cast bei Typkonvertierung
  - ▶ `instanceof` Operatoren
  - ▶ Zuweisung an Array-Element
- ▶ Gute Nachricht
  - ▶ Objekte kennen ihre Klasse zur Laufzeit durch Type-ID
    - ▶ 1. Feld eines Objektes im Speicher

- ▶ Statische Typprüfung in Bantam nicht ausreichend
  - ▶ Down-Cast bei Typkonvertierung
  - ▶ `instanceof` Operatoren
  - ▶ Zuweisung an Array-Element
- ▶ Gute Nachricht
  - ▶ Objekte kennen ihre Klasse zur Laufzeit durch Type-ID
    - ▶ 1. Feld eines Objektes im Speicher

↳ Gesucht: Verfahren, um

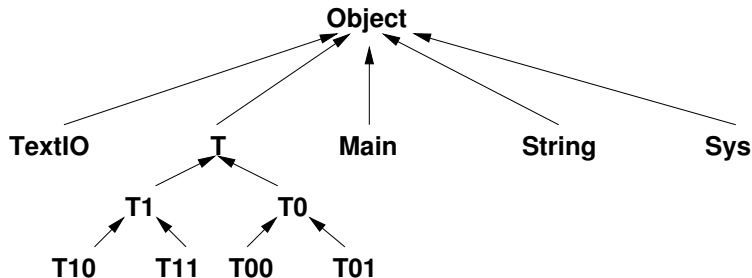
**schnell** zu bestimmen, ob Type-ID  $t_1$  kompatibel zu Type-ID  $t_2$  ist?

Kompatibel:  $t_1 = t_2$  oder  $t_1$  ist Unterklasse von  $t_2$

Auch genannt: " $t_1$  ist **konform** zu  $t_2$ "

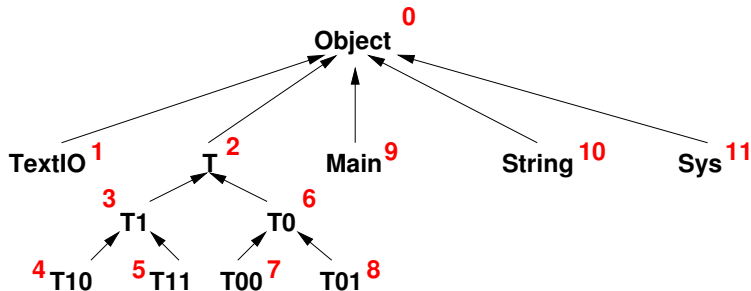
# Idee: Geschickte Zuordnung von Klassen an Type-IDs

## Beispiel



# Idee: Geschickte Zuordnung von Klassen an Type-IDs

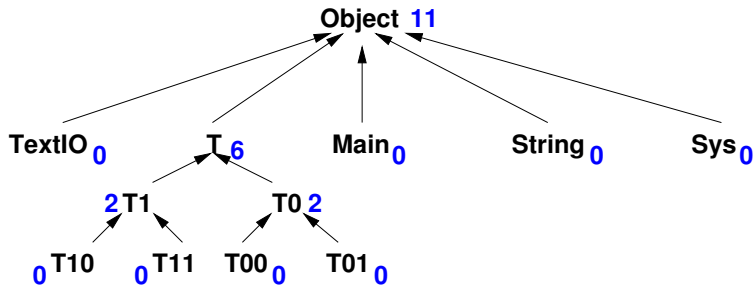
Pre-Order DFS



**typeid: Numeriert in Pre-Order DFS**

# Idee: Geschickte Zuordnung von Klassen an Type-IDs

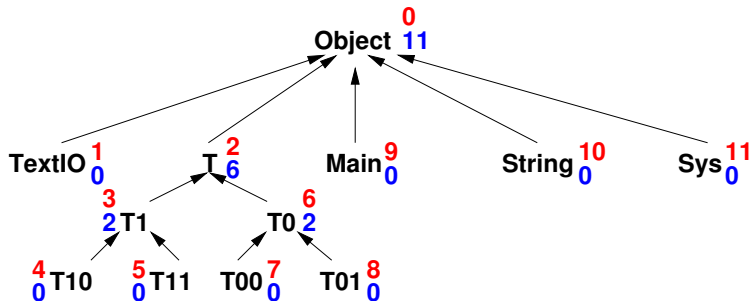
## Größe der Unterbäume



**k: Anzahl von Elementen im Unterbaum**

# Idee: Geschickte Zuordnung von Klassen an Type-IDs

## Kombination



**typeid: Numeriert in Pre-Order DFS**

**k: Anzahl von Elementen im Unterbaum**



**Beobachtung:** Für alle transitiven Unterklassen  $u_i$ ,  $1 \leq i \leq k$  einer Oberklasse  $O$  gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

**Beobachtung:** Für alle transitiven Unterklassen  $u_i$ ,  $1 \leq i \leq k$  einer Oberklasse  $O$  gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

Damit formulierbar

“Klasse  $K_1$  ist konform zu Klasse  $K_2$ ”

$$\Leftrightarrow \text{typeid}(K_2) \leq \text{typeid}(K_1) \leq \text{typeid}(K_2) + k$$



**Beobachtung:** Für alle transitiven Unterklassen  $u_i$ ,  $1 \leq i \leq k$  einer Oberklasse  $O$  gilt

$$\text{typeid}(O) \leq \text{typeid}(u_i) \leq \text{typeid}(O) + k$$

Damit formulierbar

“Klasse  $K_1$  ist konform zu Klasse  $K_2$ ”

$$\Leftrightarrow \text{typeid}(K_2) \leq \text{typeid}(K_1) \leq \text{typeid}(K_2) + k$$

➡ Nutzen für dynamische Typprüfungen

# Typumwandlung mit Down-Cast



```
class T { // Type-ID 2, k=1
    void foo() { (new TextIO()).putString("blah"); return; }
}

class T0 extends T { // Type-ID 3, k = 0
    void bar() { (new TextIO()).putString("fasel"); return; }
}

class Main {
    void main() {
        T0 t0 = new T0();
        Object obj = t0;
        ((T) (obj)).foo(); // Typkonvertierung mit Down-Cast
        t0 = (T0) (null); // ist immer legal
        return;
    }
}
```

```
((T) (obj)).foo();  
t0 = (T0) (null);
```

- ▶ Zielklasse  $K_2$  der Konvertierung ist **bekannt** (hier T)
- ▶ Klasse  $K_1$  der Instanz `obj` ist unbekannt
- ▶ `null` ist zu jedem Typ konform
- ▶ Wenn  $K_1$  nicht zu  $K_2$  konform, Class Cast Exception auslösen

# Dynamische Typprüfung für Down-Cast

## MIPS-Assembler



```
lw $v0 4($fp)           # obj ist in 4($fp)
beq $v0 $zero label11  # falls t == null, Down Cast immer OK
lw $t0 0($v0)           # hole typeid(obj) nach $t0
li $t1 3                # hole typeid(T)+k = 2+1
bgt $t0 $t1 label10    # wenn typeid(obj) > typeid(T)+k, Fehler
li $t1 2                # hole typeid(T) = 2
blt $t0 $t1 label10    # wenn typeid(obj) < typeid(T), Fehler
b label11              # alles soweit OK, weitermachen
label10:               # HIER: Class Cast Exception auslösen -----
li $a1 13              # Zeilennummer dieses Down Casts
la $a2 String_const_3 # Dateiname dieses Down Casts
jal _class_cast_error  # Fehlermeldung ausgeben und Programm beenden
label11:               # HIER: Down-Cast OK -----
sw $v0 8($fp)         # Speichere Zwischenergebnis von Down-Cast
li $a1 13              # Zeile des Aufrufes von foo()
la $a2 String_const_3 # Dateiname des Aufrufes von foo()
lw $a0 8($fp)         # Empfängerobjekt (this) ist Ergebnis von Down-Cast
bne $a0 $zero label8  # Empfängerobjekt gültig?
jal _null_pointer_error # Nein, Null Pointer Exception
label8:               # HIER: Aufruf von foo() ausführen -----
lw $t0 8($a0)         # Vtable von Instanz holen
lw $t0 12($t0)        # Methode foo() hat Method-ID 3, offset=12 Bytes
jalr $t0               # foo() anspringen
```

`obj instanceof T`

- ▶ Vergleichbar zur Typrprüfung bei Down-Cast
- ▶ Kann aber keine Exception auslösen
- ▶ Liefert entweder `true` oder `false`
  - ▶ `false` wenn `obj == null` ist

# Dynamische Typprüfung für `instanceof`

## MIPS-Assembler

---

```
...
lw $v0 8($fp)           # lade obj nach $v0
beq $v0 $zero label14  # falls obj == null, instanceof -> false
lw $t0 0($v0)           # hole typeid(obj)
li $t1 3                # hole typeid(T)+k = 2+1
bgt $t0 $t1 label14    # wenn typeid(obj) > typeid(T)+k -> false
li $t1 2                # hole typeid(T) = 2
blt $t0 $t1 label14    # wenn typeid(obj) < typeid(T) -> false
b label13               # obj ist != null und konform zu T -> true
label14:               # HIER: gebe Ergebnis false zurück -----
li $v0 0                # setze Wert für false in Ergebnisregister
b label15               # zum Ende der Berechnung
label13:               # HIER: gebe Ergebnis true zurück -----
li $v0 -1              # setze Wert für true in Ergebnisregister
label15:               # Ende der Berechnung von instanceof
...                    # Boolesches Ergebnis in Ergebnisregister $v0
```

---

# Zuweisung von Array-Elementen



```
class Animal { ... } // Type-ID 3

class Tiger extends Animal { ... } // Type-ID 5

class Bear extends Animal { ... } // Type-ID 4

// implizit deklariert, werden automatisch erzeugt
// class Animal[] extends Object {} // Type-ID 11
// class Tiger[] extends Animal[] {} // Type-ID 13
// Array Type-IDs sind Type-IDs der Basistypen
// + Anzahl der nicht-Array-Klassen (hier =8)

class Main {
    void main() {
        Animal[] animals = new Tiger[10];
        animals[0] = new Bear(); // dynamische Typprüfung erforderlich
        return;
    }
}
```

```
Animal[] animals = new Tiger[10];  
    animals[0] = new Bear();
```

- ▶ Ähnlich wie Typprüfung für Down-Cast
- ▶ Aber: Hier  $K_1$  (Typ von RHS) **und**  $K_2$  (Basistyp des Arrays) variabel



```
Animal[] animals = new Tiger[10];  
    animals[0] = new Bear();
```

- ▶ Ähnlich wie Typprüfung für Down-Cast
- ▶ Aber: Hier  $K_1$  (Typ von RHS) **und**  $K_2$  (Basistyp des Arrays) variabel
- ▶ Vorgehen
  - ▶ Zur Laufzeit Typen von LHS und RHS bestimmen
  - ▶ **Zur Laufzeit** Intervall der zu  $K_2$  konformen Type-IDs bestimmen
  - ▶ Benötigt zusätzliche Datenstruktur
    - ▶ Zu jeder Klasse Anzahl der transitiven Unterklassen

# Dynamische Typprüfung für Zuweisung an Array Element

## Infrastruktur



---

subclass\_num\_table:

```
.word 17 # Unterklassen von Object (Type-ID 0)
.word 0  # Unterklassen von String (Type-ID 1)
.word 0  # Unterklassen von Sys (Type-ID 2)
.word 2  # Unterklassen von Animal (Type-ID 3)
.word 0  # Unterklassen von Bear (Type-ID 4)
.word 0  # Unterklassen von Tiger (Type-ID 5)
.word 0  # Unterklassen von Main (Type-ID 6)
.word 0  # Unterklassen von TextIO (Type-ID 7)
```

---

- ▶ Bildet Type-IDs auf Anzahl der transitiven Unterklassen ab
- ▶ Wird nur benötigt für Basistypen, nicht für die automatisch angelegten Array-Typen

# Dynamische Typprüfung für Zuweisung an Array Element

## MIPS-Assembler

```
...
lw $t2 0($t0)           # hole typeid(LHS[]), hier Tiger[], nach $t2. Type-ID 13
sub $t2 $t2 8           # berechne typeid(LHS)=Tiger=Type-ID 5, den Basistyp von Tiger[]
lw $t1 4($fp)          # hole RHS nach $t1 (neue Bear-Instanz)
beq $t1 $zero label8   # wenn null -> Zuweisung immer OK
lw $t1 0($t1)          # hole typeid(RHS) = Bear, Type-ID 4
blt $t1 $t2 label9     # wenn typeid(RHS) < typeid(LHS), nicht konform -> Fehler
mul $t2 $t2 4          # Berechne Offset in subclass_num_table als Type-ID*4 Bytes
la $t3 subclass_num_table # Adressberechnung für Array-Zugriff
add $t3 $t2 $t3        # --- " ---
lw $t3 0($t3)          # Lese Anzahl transitiver Unterklassen zu typeid(LHS), hier 0
lw $t2 0($t0)          # hole typeid(LHS[]), hier Tiger[], nach $t2. Type-ID 13
sub $t2 $t2 8           # berechne typeid(LHS)=Tiger=Type-ID 5, den Basistyp von Tiger[]
add $t2 $t2 $t3        # Bestimme Obergrenze der konformen Type-IDs in $t3, hier Type-ID 5
bgt $t1 $t2 label9     # wenn typeid(RHS) > konforme Obergrenze -> Fehler
b label8              # sonst Typprüfung der Zuweisung OK
label9:               # HIER: Inkompatible Typen, Fehlermeldung ausgeben -----
lw $t0 0($t0)          # Für Fehlermeldung: $t0 = typeid(LHS[]), $t1 = typeid(RHS)
jal _array_store_error # Fehlermeldung und Programmende
label8:               # HIER: Zuweisung ist legal -----
...

```



# Hilfsinfrastruktur

# Bilde Type-IDs auf Namen ab

## Für Fehlermeldungen



```
class_name_table:
  .word String_const_1 # "Object"
  .word String_const_2 # "String"
  .word String_const_3 # "Sys"
  .word String_const_5 # "Animal"
  .word String_const_6 # "Bear"
  .word String_const_7 # "Tiger"
  .word String_const_8 # "Main"
  .word String_const_11 # "TextIO"
  .word 0 # nicht benutzt, wäre Object[]
  .word 0 # nicht benutzt, wäre String[]
  .word 0 # nicht benutzt, wäre Sys[]
  .word String_const_9 # "Animal[]"
  .word 0 # nicht benutzt, wäre Bear[]
  .word String_const_10 # "Tiger[]"
  .word 0 # nicht benutzt, wäre Main[]
  .word 0 # nicht benutzt, wäre TextIO[]
  .word 0 # nicht benutzt, wäre int[] (Sonderfall!)
  .word 0 # nicht benutzt, wäre boolean[] (Sonderfall!)
```

- ▶ Für skalare und Array-Klassen
  - ▶ Enthält Einträge für alle **potentiellen** Array-Klassen
  - ▶ Bleiben leer, wenn Array-Klassen nicht benutzt wurden



# Exkurs: Kontrollflussgraphen als IR

## Basisblock (BB)

Längste Folge von Anweisungen **ohne** Kontrollfluß.

## Basisblock (BB)

Längste Folge von Anweisungen **ohne** Kontrollfluß.

Beispiel:

```
a := b + 42;  
if (a > 23) then  
  c := a - 46;  
  d := b * 15;  
else  
  c := a + 46;  
  d := 0  
  q := false;  
endif
```



## Basisblock (BB)

Längste Folge von Anweisungen **ohne** Kontrollfluß.

Beispiel:

```
a := b + 42;  
if (a > 23) then  
  c := a - 46;  
  d := b * 15;  
else  
  c := a + 46;  
  d := 0  
  q := false;  
endif
```

Basisblöcke:

```
a := b + 42;
```

```
c := a - 46;  
d := b * 15;
```

```
c := a + 46;  
d := 0  
q := false;
```



- ▶ Basisblöcke alleine **nicht** ausreichend als allgemeine Zwischendarstellung
  - ▶ Kontrollfluss fehlt völlig

- ▶ Basisblöcke alleine **nicht** ausreichend als allgemeine Zwischendarstellung
  - ▶ Kontrollfluss fehlt völlig
- ▶ Erweiterung auf **Graph** von Basisblöcken
  - ▶ Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
  - ▶ Kanten symbolisieren **Kontrollfluß**

- ▶ Basisblöcke alleine **nicht** ausreichend als allgemeine Zwischendarstellung
  - ▶ Kontrollfluss fehlt völlig
- ▶ Erweiterung auf **Graph** von Basisblöcken
  - ▶ Am Ende jedes Basisblockes (bedingter) Sprung zum nächsten Block
  - ▶ Kanten symbolisieren **Kontrollfluß**
- ▶ Sehr gut für viele Optimierungen brauchbar
- ▶ Häufig verwendete Zwischendarstellung im Optimierer



Engl. *control flow graph* (CFG)

Engl. *control flow graph* (CFG)

- ▶ **Knoten** sind Basisblöcke
- ▶ **Kanten** sind Sprünge zwischen den Blöcken



Engl. *control flow graph* (CFG)

- ▶ **Knoten** sind Basisblöcke
- ▶ **Kanten** sind Sprünge zwischen den Blöcken
- ▶ Sprünge treten also nur am **Ende** eines Blocks auf!
- ▶ Sprungziel ist immer ein **Blockanfang**
  - ▶ In Triangle: if/then/else, while/do
    - ▶ Strukturierte Programmierung

Engl. *control flow graph* (CFG)

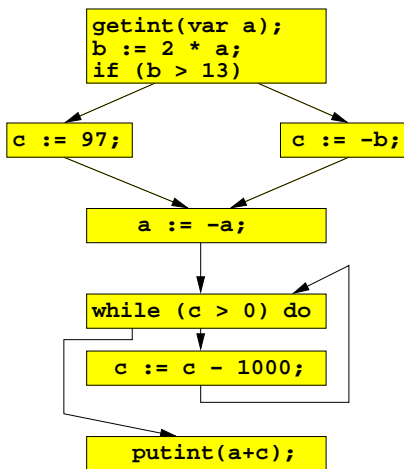
- ▶ **Knoten** sind Basisblöcke
- ▶ **Kanten** sind Sprünge zwischen den Blöcken
- ▶ Sprünge treten also nur am **Ende** eines Blocks auf!
- ▶ Sprungziel ist immer ein **Blockanfang**
  - ▶ In Triangle: if/then/else, while/do
    - ▶ Strukturierte Programmierung
  - ▶ Allgemeiner Fall deutlich komplizierter
    - ▶ goto
    - ▶ setjmp()/longjmp()
    - ▶ Exceptions



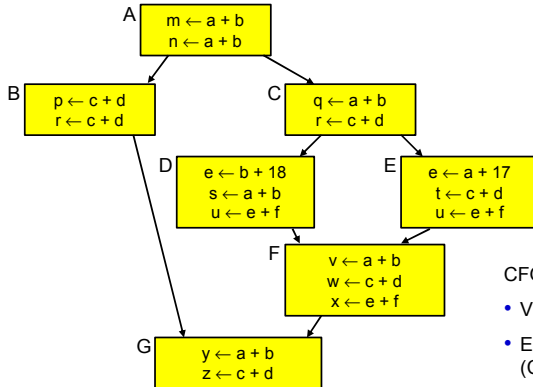
# Beispiel Kontrollflußgraph 1



```
getint(var a);  
b := 2 * a;  
if (b > 13) then  
  c := 97;  
else  
  c := -b;  
a := -a;  
while (c > 0) do  
  c := c - 1000;  
putint(a+c);
```



## Beispiel Kontrollflußgraph 2



CFG  $G = (V, E)$

- $V = \{A, B, C, D, E, F, G\}$
- $E = \{(A, B), (A, C), (B, G), (C, D), (C, E), (D, F), (E, F), (F, E)\}$
- $|V| = 7, |E| = 8$



# Organisation des Bantam-Compilers

Bekommt als Eingabedaten

- ▶ Eingangs-Basisblock des CFGs
- ▶ Name der Methode
- ▶ Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll



Bekommt als Eingabedaten

- ▶ Eingangs-Basisblock des CFGs
- ▶ Name der Methode
- ▶ Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll

Realisiert in Form des Interfaces `opt.Optimization`

- ▶ Wird automatisch auf alle Methoden in allen Klassen angewandt
- ▶ Einschließlich der internen `_init` Methoden
- ▶ Fehlerbehandlung durch Auslösen von `Exception`

Bekommt als Eingabedaten

- ▶ Eingangs-Basisblock des CFGs
- ▶ Name der Methode
- ▶ Ob die Operation mit Debug-Ausgaben (*verbose*) erfolgen soll

Realisiert in Form des Interfaces `opt.Optimization`

- ▶ Wird automatisch auf alle Methoden in allen Klassen angewandt
- ▶ Einschließlich der internen `_init` Methoden
- ▶ Fehlerbehandlung durch Auslösen von `Exception`

JavaDoc zur ESA-Version des Compilers liegt auf Web-Seite als `bantam-api.zip`

# Allgemeine Schnittstelle im Middle-End

## Parameterübergabe



```
public interface Optimization {  
    /**  
     * Sets the current root of a CFG.  
     * @param block BasicBlock  
     */  
    public void setEntryBlock(final BasicBlock block);  
  
    /**  
     * Sets the current CFGs method name.  
     * @param name Name  
     */  
    public void setMethodName(final String name);  
  
    /**  
     * Sets verbose mode.  
     * @param verbose on/off.  
     */  
    public void setVerbose(final boolean verbose);  
  
    /**  
     * Returns true, if in verbose mode.  
     * @return verbose?  
     */  
    public boolean isVerbose();  
}
```

# Allgemeine Schnittstelle im Middle-End

## Aktionen



---

```
/**
 * Defines a generic interface for Optimizations:
 * Two phase operation, analysis and transformation.
 */
public interface Optimization {

    ...

    /**
     * Perform analysis for optimization.
     */
    public void analyze();

    /**
     * Perform actual optimization on CFG.
     * @return new start basic block of CFG
     */
    public BasicBlock transform();
}
```

---



# Anwenden auf CFG

Beispiel aus `example_driver`



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
import opt.Optimization;
...
// Vorher Lexing, Parsing, AST-Erzeugung, Semantische Analyse und CFG-Erzeugung

// Erzeuge Instanz von neuem Optimierungspass
// muß Interface Optimization implementieren
Optimization myOpt = new MyNewOptimizationPass(MyNewOptimizationPass.DOEVERYTHING, "logfile.txt");

// Weise bestehenden Optimierungsverwalter an, den neuen Pass auf
// auf alle Methoden aller Klassen anzuwenden
optimizer.optimize(myOpt);

// Nun Code-Erzeugung aus CFG
...
```

# Konvertieren von CFG nach SSA-CFG

Bereits im Framework realisiert

Ebenfalls als Optimierungspass realisiert

```
import opt.ssa.CFG2SSA;  
...  
// Vorher normale CFG-Erzeugung  
  
// Optimierungspass für SSA-Konvertierung anlegen und konfigurieren  
CFG2SSA cfg2ssa = new CFG2SSA();  
cfg2ssa.setVerbose(false);  
  
// Pass auf alle CFGs aller Methoden aller Klassen anwenden  
optimizer.optimize(cfg2ssa);  
  
// Ab jetzt CFGs aller Methoden aller Klassen in SSA-CFG Form  
...  
...
```

# Konvertieren von SSA-CFG nach CFG

Bereits im Framework realisiert

Ebenfalls als Optimierungspass realisiert

```
import opt.ssa.PhiRemover;
...
// Bisher SSA-CFG Form

// Optimierungspass für SSA-Konvertierung anlegen und konfigurieren
PhiRemover phiremover = new PhiRemover();
phiremover.setVerbose(false);

// Pass auf alle CFGs aller Methoden aller Klassen anwenden
optimizer.optimize(phiremover);

// Ab jetzt CFGs aller Methoden aller Klassen in CFG Form
...

```



# Zwischendarstellung für Optimierung

- ▶ Beschrieben durch `cfg.BasicBlock`
  - ▶ Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)



- ▶ Beschrieben durch `cfg.BasicBlock`
  - ▶ Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- ▶ Kanten
  - ▶ Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - ▶ Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - ▶ Beispiel `int getNumOutEdges(), BasicBlock getNthOutEdge(int n)`



- ▶ Beschrieben durch `cfg.BasicBlock`
  - ▶ Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- ▶ Kanten
  - ▶ Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - ▶ Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - ▶ Beispiel `int getNumOutEdges(), BasicBlock getNthOutEdge(int n)`
- ▶ Hilfsfunktionen
  - ▶ Konsistenzprüfung: `void check()`
  - ▶ Ausgabe eines Blockes: `void print()`
  - ▶ Des ganzen CFGs: `void printAll()` auf Startblock

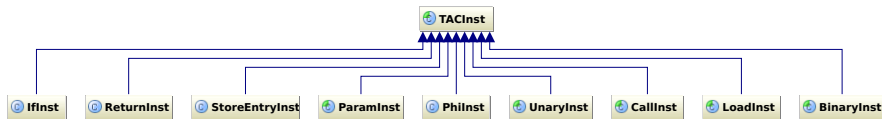


- ▶ Beschrieben durch `cfg.BasicBlock`
  - ▶ Doku in Abschnitt des Bantam Extended Lab Manuals (p. 90ff)
- ▶ Kanten
  - ▶ Nicht explizit modelliert, aber maximal zwei Ausgangskanten
  - ▶ Dargestellt als Mengen von Vorgänger/Nachfolger-Blöcken
  - ▶ Beispiel `int getNumOutEdges(), BasicBlock getNthOutEdge(int n)`
- ▶ Hilfsfunktionen
  - ▶ Konsistenzprüfung: `void check()`
  - ▶ Ausgabe eines Blockes: `void print()`
  - ▶ Des ganzen CFGs: `void printAll()` auf Startblock
- ▶ Anweisungen
  - ▶ Dargestellt als Listen von Drei-Address-Instruktionen (TAC)
  - ▶ `List<TACInst> getInstructions(), int getNumInsn()`



# Drei-Address-Code

*three-address code (TAC)*



Schon vorbereitet für Benutzung durch Visitor

```
<ReturnType, ArgumentType>
```

```
ReturnType accept(cfg.TACInstVisitor<ReturnType, ArgumentType> visitor,  
ArgumentType o);
```

- ▶ Variablen, noch **keine** Register

- ▶ Variablen, noch **keine** Register
- ▶ Bisher: alle identifiziert durch **Strings**
  - ▶ Namenskonventionen in Lab Manual Abschnitt 6.2 (p. 91ff)
  - ▶ Suffixe beschreiben Geltungsbereiche
    - ▶ @l: Lokale Variable (z.B. foo@l)
    - ▶ @p: Formaler Parameter (z.B. bar@p)
    - ▶ @f\_*Klassenname*: Attribut von *Klassenname* (z.B. color@f\_Shape)
- ➡ Hier noch zur textuellen Darstellung verwendet



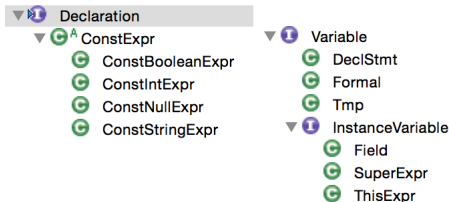
- ▶ Variablen, noch **keine** Register
- ▶ Bisher: alle identifiziert durch **Strings**
  - ▶ Namenskonventionen in Lab Manual Abschnitt 6.2 (p. 91ff)
  - ▶ Suffixe beschreiben Geltungsbereiche
    - ▶ @l: Lokale Variable (z.B. foo@l)
    - ▶ @p: Formaler Parameter (z.B. bar@p)
    - ▶ @f\_Klassenname: Attribut von *Klassenname* (z.B. color@f\_Shape)
  - ➡ Hier noch zur textuellen Darstellung verwendet
- ▶ Nun: Verweis auf ein entsprechendes Declaration-Objekt

- ▶ Verwendung einer Variablen/Konstanten ➔ Referenz zur Declaration

- ▶ Declaration-Interface:

Location getLocation(), void setLocation(Location loc)

Location abstrahiert konkreten Speicherort, der bei der Codeerzeugung vergeben wird





- ▶ Bedingung: EQ, GE, GT, LE, LT, NE
  - ▶ `getType()`, `setType()`
- ▶ Linker/Rechter Operand des Vergleichs
  - ▶ `Declaration getLeftSource()`, `Declaration getRightSource()`
  - ▶ `setLeftSource(Declaration l)`, `setRightSource(Declaration r)`
- ▶ Sprungziele: Basisblocks
  - ▶ `BasicBlock getFalseTarg()`, `BasicBlock getTrueTarg()`
  - ▶ `setFalseTarg(BasicBlock falseTarg)`, `setTrueTarg(BasicBlock trueTarg)`

- ▶ Bedingung: EQ, GE, GT, LE, LT, NE
  - ▶ `getType()`, `setType()`
- ▶ Linker/Rechter Operand des Vergleichs
  - ▶ Declaration `getLeftSource()`, Declaration `getRightSource()`
  - ▶ `setLeftSource(Declaration l)`, `setRightSource(Declaration r)`
- ▶ Sprungziele: Basisblocks
  - ▶ `BasicBlock getFalseTarg()`, `BasicBlock getTrueTarg()`
  - ▶ `setFalseTarg(BasicBlock falseTarg)`, `setTrueTarg(BasicBlock trueTarg)`

Beispiel:

```
if (@t13 != null) goto bb18;  
goto bb17;
```

# Unterprogrammaufruf: CallInst

DirCallInst, IndirCallInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Anzahl Parameter
  - ▶ `setNumParam(int i), int getNumParam()`
- ▶ Optional: Zielvariable für Rückgabewert (`null` wenn keine)
  - ▶ `Variable getDestVar(), setDestVar(Variable dstVar)`
- ▶ Sprungziel
  - ▶ Direkter Sprung `DirCallInst: setTarget(Label target)`  
Label enthält Routinename gemäß MIPS-Konventionen als String
  - ▶ Indirekter Sprung `IndirCallInst: setTarget(Variable target)`  
Referenz auf Variablendeklaration mit Sprungzieladresse



# Unterprogrammaufruf: CallInst

DirCallInst, IndirCallInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Anzahl Parameter
  - ▶ `setNumParam(int i), int getNumParam()`
- ▶ Optional: Zielvariable für Rückgabewert (null wenn keine)
  - ▶ `Variable getDestVar(), setDestVar(Variable dstVar)`
- ▶ Sprungziel
  - ▶ Direkter Sprung `DirCallInst: setTarget(Label target)`  
Label enthält Routinename gemäß MIPS-Konventionen als String
  - ▶ Indirekter Sprung `IndirCallInst: setTarget(Variable target)`  
Referenz auf Variablendeklaration mit Sprungzieladresse

Beispiel:

```
// direkter Sprung
dircall Object_init, 0;           // keine Parameter, kein Ergebnis

// indirekter Sprung
@t4 = Object_dispatch_table;
@t5 = @t4[2];
@t3 = indircall @t5, 1;          // ein Parameter, mit Ergebnis
```

# Parameterübergabe: ParamInst

StdParamInst, RefParamInst, ErrParamInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Quelle des übergebenen Wertes als *Declaration*
  - ▶ `Declaration getSource(), setSource(Declaration actual)`
- ▶ Zielobjekt mit `RefParamInst` (genau eines bei Methodenaufruf)
- ▶ Aktuelle Parameter mit `StdParamInst`
- ▶ Fehlerparameter für Exception-Handler mit `ErrParamInst`
  - ▶ Braucht auch noch Art des Fehlerparameters
  - ▶ `FILENAME, LINENUM, OBJECTID, TARGETID, ARRAYIDXID, ...`
- ▶ Als erstes Empfängerobjekt, dann Aufrufparameter von links nach rechts

## Beispiel: Parameterübergabe / Methodenaufruf

```
tmp = t.bar(42, 23)
```

```
refparam t@1; // Empfängerobjekt
stdparam 42; // aktuelle Parameter
stdparam 23;
errparam filename, "TestInherit.btm";
errparam linenum, 78;
if (t@1 != null) goto bb50;
goto bb49;

# basic block near source line 78
bb50:
    @t41 = t@1[2]; // vtable von t
    @t42 = @t41[5]; // Method-ID 5
    tmp@1 = indircall @t42, 3;
    ...

# basic block near source line 78
bb49:
    dircall _null_pointer_error, 0;
    goto bb50;
```



- ▶ Rückkehr von Unterprogramm
- ▶ Optional mit Rückgabewert
  - ▶ Declaration `getSource(), setSource(Declaration retval)`
  - ▶ `null` wenn kein Rückgabewert

```
// int bar(int a, int b) { return a * b; }  
T.bar:  
# basic block near source line 12  
bb20:  
    @t17 = a@p * b@p;  
    retn @t17;
```

## Werte in Variablen laden: LoadInst

LoadAddrInst, LoadConstInst, LoadEntryInst, LoadVarInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Zielvariable: `Variable getDestination(), setDestination(Variable var)`
- ▶ Lade von Variable `LoadVarInst`
  - ▶ Quellvariable: `Variable getSource(), setSource(Variable var)`

# Werte in Variablen laden: LoadInst

LoadAddrInst, LoadConstInst, LoadEntryInst, LoadVarInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Zielvariable: `Variable getDestination(), setDestination(Variable var)`
- ▶ Lade von Variable `LoadVarInst`
  - ▶ Quellvariable: `Variable getSource(), setSource(Variable var)`
- ▶ Lade Konstante `LoadConstInst`
  - ▶ Typ der Konstante: `Integer, Boolean, String, null`
  - ▶ Wert der Konstante `Declaration getSource(), setSource(Declaration literal)`

# Werte in Variablen laden: LoadInst

LoadAddrInst, LoadConstInst, LoadEntryInst, LoadVarInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Zielvariable: `Variable getDestination(), setDestination(Variable var)`
- ▶ Lade von Variable `LoadVarInst`
  - ▶ Quellvariable: `Variable getSource(), setSource(Variable var)`
- ▶ Lade Konstante `LoadConstInst`
  - ▶ Typ der Konstante: `Integer, Boolean, String, null`
  - ▶ Wert der Konstante `Declaration getSource(), setSource(Declaration literal)`
- ▶ Lade Adresse, gegeben durch Label im Programm `LoadAddrInst`
  - ▶ Label `Label getSource(), setSource(Label label)`

# Werte in Variablen laden: LoadInst

LoadAddrInst, LoadConstInst, LoadEntryInst, LoadVarInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Zielvariable: Variable `getDestination()`, `setDestination(Variable var)`
- ▶ Lade von Variable `LoadVarInst`
  - ▶ Quellvariable: Variable `getSource()`, `setSource(Variable var)`
- ▶ Lade Konstante `LoadConstInst`
  - ▶ Typ der Konstante: Integer, Boolean, String, null
  - ▶ Wert der Konstante Declaration `getSource()`, `setSource(Declaration literal)`
- ▶ Lade Adresse, gegeben durch Label im Programm `LoadAddrInst`
  - ▶ Label `Label` `getSource()`, `setSource(Label label)`
- ▶ Lade Speicherwort `LoadEntryInst`
  - ▶ Basisadresse ist Declaration
    - ▶ Declaration `getSource()`, `setSource(Declaration base)`
  - ▶ Offset ist Variable oder Integer-Literal  $> 0$ 
    - ▶ Declaration `getIndex()`, `setIndex(Declaration idx)`



# Beispiel: Laden von Werten



```
// Load variable
canswim@f_Animal = canswim@p;
...

// Load Constant (integer)
numwheels@f_Car = 4;
...

// Load Address
@t11 = TextIO_template;
refparam @t11;
errparam filename, "TestInherit.btm";
errparam linenum, 9;
@t12 = dircall Object.clone, 1;
...

// Load Entry
@t41 = t@1[2]; // vtable von t
@t42 = @t41[5]; // Method-ID 5
```

- ▶ Ziel: Basisadresse plus Offset (Variable, Integer-Literal > 0)
  - ▶ Basisadresse Variable `getBaseAddress()`, `setBaseAddress(Variable base)`
  - ▶ Offset Declaration `getIndex()`, `setIndex(Declaration idx)`
- ▶ Quelle (= Wert, der geschrieben wird): Declaration

Beispiel:

```
// int[] a;  
// a[i] = 42;  
a@1[i@1] = 42;
```

# Arithmetische und Logische Operatoren: UnaryInst und BinaryInst



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- ▶ Ziel für evaluierten Ausdruck: Variable
  - ▶ `Variable getDestination(), setDestination(Variable var)`

# Arithmetische und Logische Operatoren: UnaryInst und BinaryInst



- ▶ Ziel für evaluierten Ausdruck: Variable
  - ▶ Variable `getDestination()`, `setDestination(Variable var)`
- ▶ Unäre Operatoren `UnaryNegInst`, `UnaryNotInst`
  - ▶ Operand `Neg Declaration` `getSource()`, `setSource(Declaration varorint)`
  - ▶ Operand `Not Declaration` `getSource()`, `setSource(Declaration varorbool)`

# Arithmetische und Logische Operatoren: UnaryInst und BinaryInst

- ▶ Ziel für evaluierten Ausdruck: Variable
  - ▶ `Variable getDestination(), setDestination(Variable var)`
- ▶ Unäre Operatoren `UnaryNegInst, UnaryNotInst`
  - ▶ Operand `Neg Declaration` `getSource(), setSource(Declaration varorint)`
  - ▶ Operand `Not Declaration` `getSource(), setSource(Declaration varorbool)`
- ▶ Binäre Operatoren
  - ▶ Operanden Arithmetische `BinaryAddInst, BinarySubInst, ...`
    - ▶ `Declaration getLeftSource(), Declaration getRightSource()`
    - ▶ `setLeftSource(Declaration varorint), setRightSource(Declaration varorint)`
  - ▶ Operanden Logische `BinaryAndInst, BinaryOrInst`
    - ▶ `Declaration getLeftSource(), Declaration getRightSource()`
    - ▶ `setLeftSource(Declaration varorbool), setRightSource(Declaration varorbool)`

# Beispiel: Binäre und unäre Operatoren



```
@t0 = @t1 * @t2;  
@t3 = @t0 * 4;  
@t4 = 4 + @t3;
```

```
isflyingfish@l = canswim@f_Animal && canfly@f_Animal;  
@t8 = isflyingfish@l && true;
```

- ▶ Realisiert Phi-Funktion in Join-Knoten der SSA-Form
- ▶ Ziel: `Variable getDestination(), setDestination(Variable var)`
- ▶ Parameter: Einer für jede eingehende Kante, in gleicher Reihenfolge
  - ▶ `int getNumSources()`
  - ▶ Declaration `getNthSource(final int n)`
  - ▶ `setNthSource(final int n, final Declaration value)`

- ▶ Realisiert Phi-Funktion in Join-Knoten der SSA-Form
- ▶ Ziel: `Variable getDestination(), setDestination(Variable var)`
- ▶ Parameter: Einer für jede eingehende Kante, in gleicher Reihenfolge
  - ▶ `int getNumSources()`
  - ▶ Declaration `getNthSource(final int n)`
  - ▶ `setNthSource(final int n, final Declaration value)`

Beispiel:

```
@t100003 = phi (@t100001, @t100002);
```



- ▶ Keine unbedingten Sprünge
  - ▶ Werden implizit durch Kanten zwischen Blöcken realisiert
- ▶ Wenn Block zwei ausgehende Kanten hat, muss er mit `IfInst` enden
- ▶ Blöcke ohne ausgehende Kanten müssen enden mit
  - ▶ `ReturnInst`
  - ▶ Aufruf einer Exception Handler-Routine
- ▶ Alle Blöcke mit Ausnahme des CFG-Startblockes müssen mindestens eine Eingangskante haben
- ▶ Alle TAC-Instruktionen sind auf Visitor vorbereitet.

# Beispiel: CFG/TAC-Visitor

LoadAddrInst und Visitor-Interface TACInstVisitor



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
public class LoadAddrInst extends LoadInst {
    ...
    public Object accept(TACInstVisitor<ReturnType, ArgumentType> visitor, Object o) {
        return visitor.visitInst(this, o);
    }
}
```

```
public interface TACInstVisitor<ReturnType, ArgumentType> {
    abstract ReturnType visitInst(BinaryAddInst inst, ArgumentType o);
    ...
    abstract ReturnType visitInst(LoadAddrInst inst, ArgumentType o);
    ...
    abstract ReturnType visitInst(UnaryNegInst inst, ArgumentType o);
}
```

# Beispiel: CFG/TAC-Visitor

## Code-Generierung



```
...  
public Void visitInst (LoadAddrInst inst, Void o) {  
    final Label src = inst.getSource();  
    final Variable dest = inst.getDestination();  
    final String dreg = MipsSupport.isRegister(dest)  
        ? dest.getLocation().getReg()  
        : assemblySupport.getResultReg();  
  
    // la dreg src  
    assemblySupport.genLoadAddr(dreg, src);  
  
    return null;  
}  
...
```