1 Einführung

Nutzen ComBau
- Programmiersprachen und Sprachkonzepte besser verstehen
- Sprachfeatures beurteilen können
- Konzepte in verwandten Bereichen einsetzen

1.1 Begriffe

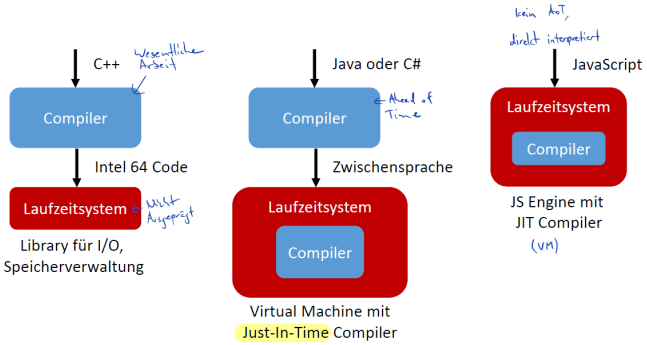
Compiler
- Transformiert Quellcode in Maschinencode

Runtime System
- Unterstützt die Programmausführung mit Software und Hardware Mechanismen

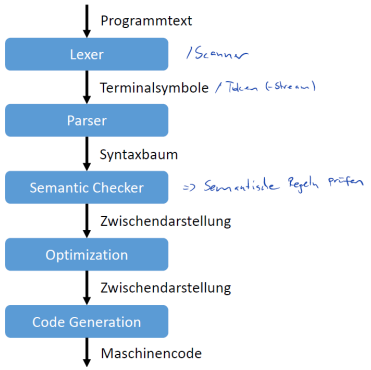
Syntax
- Definiert Struktur des Programms
- Bewährte Formalismen für Syntax

Semantik
- Definiert Bedeutung des Programms
- Meist in Prosa beschrieben

1.2 Architekturen



1.3 Aufbau Compiler



1.3.1 Lexer

Lexikalische Analyse, Scanner
- Zerlegt Programmtext in Terminalsymbole (Tokens)
- keine Tiefenstruktur

1.3.2 Parser

Syntaktische Analyse
- Erzeugt Syntaxbaum gemäss Programmstruktur
- Kontextfreie Sprache

1.3.3 Semantic Checker

Semantische Analyse
- Löst Symbole auf
- Prüft Typen und semantische Regeln

1.3.4 Optimization

- Wandelt Zwischendarstellung in effizientere um

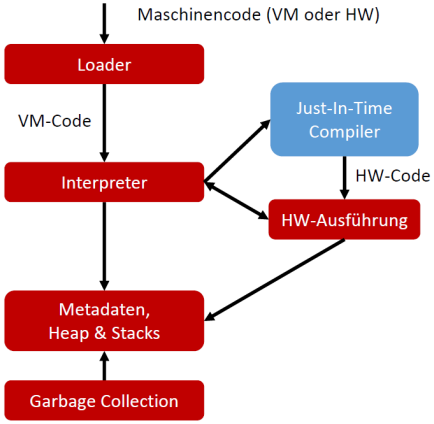
1.3.5 Code Generation

- Erzeugt ausführbarer Maschinencode

1.3.6 Zwischenarstellung

Intermediate Representation
- Beschreibt Programm als Datenstruktur (diverse Varianten)

1.4 Aufbau Laufzeitsystem



1.4.1 Loader

- Lädt Maschinencode in Speicher
- Veranlasst Ausführung

1.4.2 Interpreter

- Liest Instruktionen und emuliert diese in Software

1.4.3 JIT (Just-In-Time) Compiler

- Übersetzt Code-Teile in Hardware-Instruktionscode

1.4.4 HW-Ausführung (nativ)

- Lässt Instruktionscode direkt auf HW-Prozessor laufen

1.4.5 Metadaten, Heap + Stacks

- Merken Programminfos, Objekte und Prozeduraufrufe

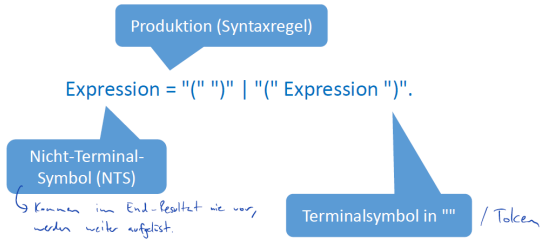
1.4.6 Garbage Collection

- Räumt nicht erreichbare Objecte ab

1.5 Syntax

1.5.1 EBNF

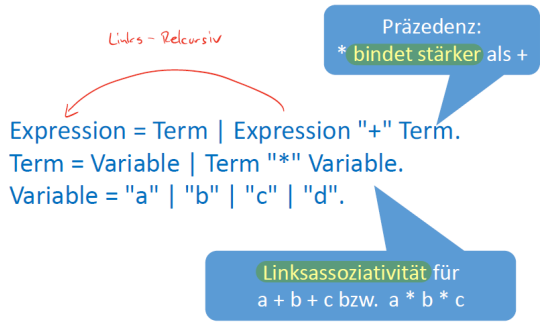
Extended Backus-Naur Form



	Beispiel	Sätze
Konkatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" oder "B"
Option	["A"]	leer oder "A"
Wiederholung	{ "A" }	leer, "A", "AA", "AAA", etc.

Runde Klammern für stärkere Bindung.
| bindet schwächer als andere Konstrukte.

1.5.2 Arithmetische Ausdrücke



2 Lexikalische Analyse

2.1 Lexer / Scanner

Endlicher Automat (DEA)
- Kümmert sich um die lexikalische Analyse
- Input: Zeichenfolge (Programmtext)
- Output: Folge von Terminalsymbolen (Tokens)

2.1.1 Aufgaben

- Fasst Textzeichen zu tokens zusammen
- Eliminiert Whitespaces
- Eliminiert Kommentare
- Merkt Positionen in Programmcode

2.1.2 Nutzen

Abstraktion
- Parser muss sich nicht um Textzeichen kümmern

Einfachheit
- Parser braucht Lookahead pro Symbol, nicht Textzeichen

Effizienz
- Lexer benötigt keinen Stack im Gegensatz zu Parser

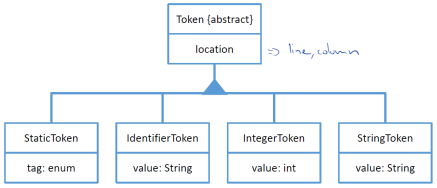
2.2 Tokens

Statisch (Keywords, Operationen, Interpunktion)
- *if*
- *else*
- *while*
- *
- *&&*
- *;*

Identifiers
- MyClass
- readFile
- name2

Zahlen
- 123
- 0xfe12
- 1.2e-3

Strings
- ”Hello”
- ””
- ”\n”



2.2.1 Lexem

- Spezifische Zeichenfolge, die einen Token darstellt
- z.B. *MyClass* ist ein Lexem des Tokens Identifier

2.2.2 Maximum Munch

- Lexer absorbiert möglichst viel in einem Token

2.3 Reguläre Sprachen

- Lexer unterstützt nur reguläre Sprachen
- **Regulär:** Als EBNF ohne Rekursion ausdrückbar

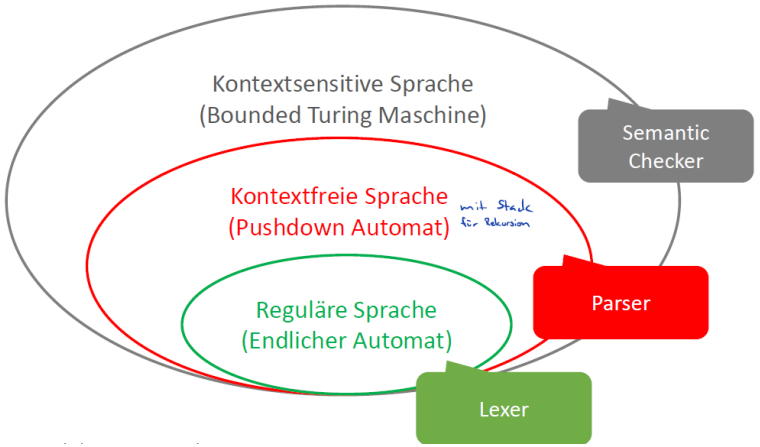
```
Integer = Digit { Digit }.
Digit = "0" | ... | "9".
```

Regulär
=> nicht rekursiv

```
Ausdruck = [ "(" Ausdruck ")" ].
```

Nicht regulär
=> rekursiv nötig

2.4 Chomsky Hierarchie



2.5 Lexer Gerüst

```
class Lexer {
    private final Reader reader;
    private char current; // one Character Lookahead
    private boolean end; // EOF

    private Lexer(Reader reader) {
        this.reader = reader;
    }

    public static Iterable<Token> scan(Reader reader) {
        return new Lexer(reader).readTokenStream();
    }

    // ...
}
```

2.6 Token Stream lesen

```
Iterable<Token> readTokenStream() {
    var stream = new ArrayList<Token>();
    readNext(); // One Character Lookahead
    skipBlanks();
    while (!end) {
        stream.add(readToken());
        skipBlanks();
    }
    return stream;
}
```

2.7 Lexer Kernlogik

```
Token readToken() {
    if(isDigit(current)) {
        return readInteger();
    }
    if(isLetter(current)) {
        return readName(); // Identifier / Keyword
    }
    return switch(current) {
        case '"': readString();
        case '+': readStaticToken(Tag.Plus);
        case '-': readStaticToken(Tag.Minus);
        case '/': readPotentialSlash();
    }
}
```

2.7.1 Static Token scannen

```
StaticToken readStaticToken(Tag tag) {
    readNext();
    return new StaticToken(tag);
}
```

2.7.2 Zahlen scannen

Beachten:
- Range Check (32 bit): Integer Overflow
- $Integer.MIN = Integer.MAX + 1$

```
IntegerToken readInteger() {
    int value = 0;
    while (!_end && isDigit(current)) {
        int digit = current - '0'; // char to int
        value = value * 10 + digit; // create decimal number
        readNext();
    }
    return new IntegerToken(value);
}
```

2.7.3 Identifier und Keywords scannen

```
Token readName() {
    String name = Character.toString(current);
    readNext();
    while(!end & (isLetter(current) || isDigit(current))) {
        name += current;
        readNext();
    }
    if(KEYWORDS.containsKey(name)) {
        return new StaticToken(KEYWORDS.get(name));
    }
    return new IdentifierToken(name);
}
```

2.7.4 String scannen

Beachten:
- Kein \t
- Kein \"
- Kein \n
- Keine mehrzeiligen Strings

```
StringToken readString() {
    readNext(); // Skip leading double Quote
    String value = "";
    while (!end && current != '"') {
        value += current;
        readNext();
    }
    if(end) {
        // Error: String not closed
    }
    readNext(); // Skip trailing double Quote
    return new StringToken(value);
}
```

2.7.5 Kommentare erkennen

```
StaticToken readPotentialSlash() {
    readNext();
    if(current == '/') {
        skipLineComment();
        // move on to next token
    } else if (current == '*') {
        skipCommentBlock();
        // move on to next token
    } else {
        return new StaticToken(Tag.Divide);
    }
}
```

3 Parser

Kontextfreie Sprache

- Kümmert sich um die syntaktische Analyse
- Input: Tokens (Terminalsymbole)
- Output: Syntaxbaum / Parse Tree

3.1 Aufgabe

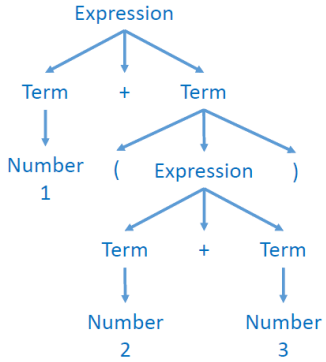
- Finde eindeutige Ableitung der Syntaxregeln, um einen gegebenen Input herzuleiten
- Analysiert die gesamte Syntaxdefinition (mit rekursiven Regeln)
- Erkennt, ob Eingabetext Syntax erfüllt
- Erzeugt Syntaxbaum

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

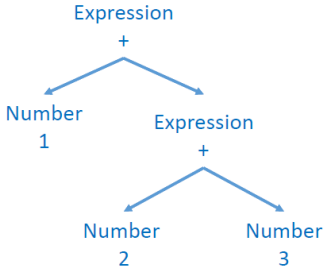
3.2 Parse Tree

- Concrete Syntax Tree
- Ableitung der Syntaxregeln als Baum wiedergespiegelt
- Kann generiert werden



3.2.1 Abstract Syntax Tree

- Unwichtige Details auslassen
- Struktur vereinfacht
- Für Weiterverarbeitung massgeschneidert
- Eigendesign nach Gusto des Compiler-Entwicklers
- Nur mit Selbstimplementation möglich



3.3 Parser Strategien

3.3.1 Top-Down

- Beginne mit Start-Symbol
- Wende Produktionen an
- Expandiere Start-Symbol auf Eingabetext
- $Expr- > Term + Term- > ...- > 1 + (2 - 3)$

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

Top-Down

linksseitig expandieren

3.3.2 Bottom-Up

- Beginne mit Eingabetext
- Wende Produktionen an
- Reduziere Eingabetext auf Start-Symbol
- $Expr < -Term + Term < -... < -1 + (2 - 3)$

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Term "+" "(" Expression ")"
Term "+" "(" Term "-" Term ")"
Term "+" "(" Term "-" Number ")"
Term "+" "(" Number "-" Number ")"
Number "+" "(" Number "-" Number ")"

Bottom-Up

rechtsseitig reduzieren

3.4 Recursive Descent

- Pro Nicht-Terminalsymbol eine Methode
  - Implementiert die Erkennung gemäss EBNF-Produktion
- Vorkommen eines Nicht-Terminalsymbols in Syntax
  - Aufruf der entsprechenden Methode
- Funktioniert bei rekursiven und nicht-rekursiven Produktionen

```
void parseExpression() {
    parseTerm();
    // ...
}
void parseTerm() {
    parseExpression();
    // ...
}
```

3.5 Parser Gerüst

```
public class Parser {
    private final Iterator<Token> tokenStream;
    private Token current; // One Token Lookahead

    private Parser(Iterable<Token> tokenStream) {
        this.tokenStream = tokenStream.iterator();
    }

    public static ProgramNode parse(Iterable<Token> stream) {
        return new Parser(stream).parseProgram(); // Aufbasierte Klasse
    }
}
```

3.5.1 Parser-Einstieg

$Program = Expression$

```
private ProgramNode parseProgram() {
    var classes = new ArrayList<ClassNode>();
    parseExpression();
    while (!isEnd()) {
        next();
        classes.add(parseClass());
    }
    return new ProgramNode(classes);
```

```
}
```

3.5.2 Expression

$Expression = Term(" + "|" - ")Term$

```
Expression parseExpression() {
    var left = parseTerm();
    while(is(Tag.PLUS) || is(Tag.MINUS)) {
        var op = is(Tag.PLUS) ? Operator.PLUS : Operator.MINUS;
        next();
        var right = parseTerm();
        var left = new BinaryExpression(op, left, right);
    }
    return left;
}
```

3.5.3 Term

$Term = Number|"("Expression")"$

```
Expression parseTerm() {
    if(isInteger()) {
        int value = readInteger();
        next();
        return new IntegerLiteral(value);
    } else if (is(Tag.OPEN_PARENTHESIS)) {
        next();
        var expression = parseExpression();
        if(is(Tag.CLOSE_PARENTHESIS)) {
            next();
        } else {
            error(); // missing closed parenthesis
        }
        return expression();
    } else {
        error(); // missing open parenthesis
    }
}
```

3.6 One Symbol Lookahead

Statement
$Assignment|IfStatement$

- Bestimme mögliche Terminalsymbole, die mit einer Produktion ableitbar sind (FIRST-Menge)
- Benutze FIRST zur Entscheidung der Alternative beim zielorientierten Parsen

```
void parseStatement() {
    if(isIdentifier()) { // FIRST(Assignment)
        parseAssignment();
    } else if(is(Tag.IF)) { // FIRST(IfStatement)
        parseIfStatement();
    } else {
        error();
    }
}
```

3.7 Technische Syntax-Umformung

- Falls 1 Lookahead nicht reicht

```
Statement = Assignment | Invocation
Assignment = Identifier "=" Expression
Invocation = Identifier "(" ")"
```

↓

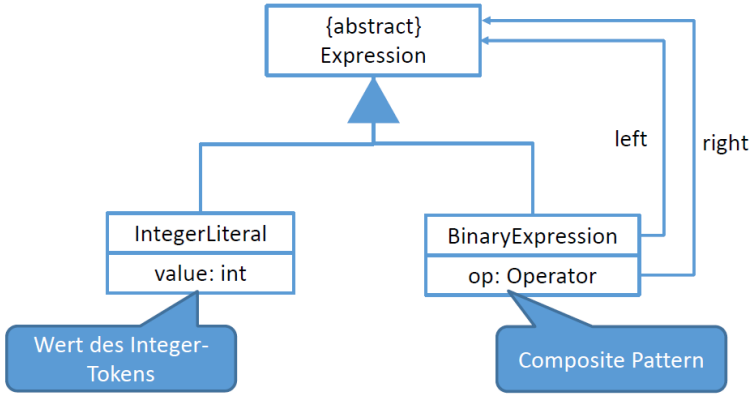
```
Statement = Identifier (AssignmentRest | InvocationRest)
AssignmentRest = "=" Expression
InvocationRest = "(" ")"
// Lookahead 1 reicht wieder
```

3.7.1 Code Beispiel

```
var parseStatement() {
    var identifier = readIdentifier();
    next();
    if (is(Tag.ASSIGN)) {
        parseAssignmentRest(identifier);
    } else if (is(Tag.OPEN_PARENTHESIS)) {
        parseInvocationRest(identifier);
    } else {
        error();
    }
}
```

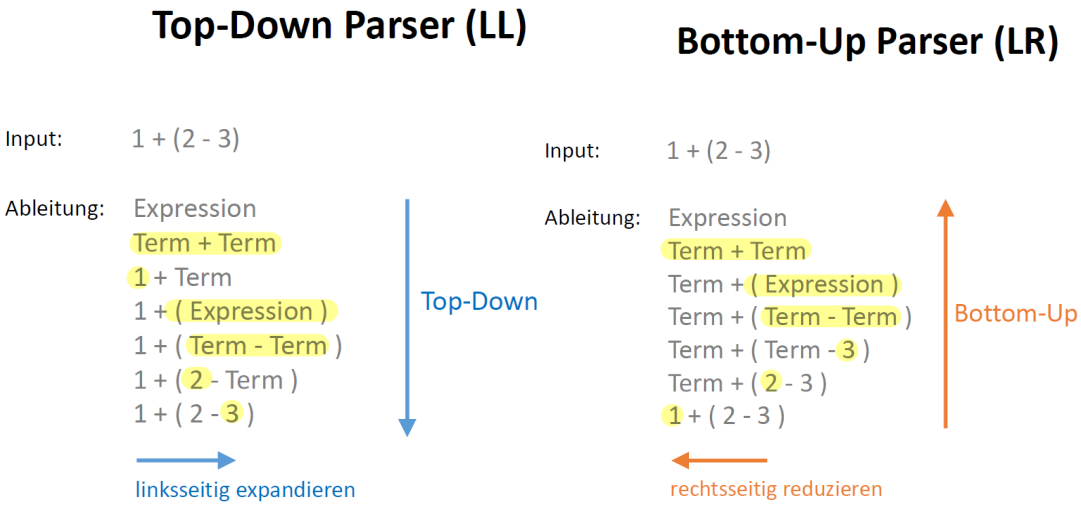
4 Parser Vertiefung

4.1 Abstrakter Syntaxbaum Design



4.2 Bottom-Up Parsing

- Mächtiger als LL (Top-Down) Parser
- Kann Linksrekursion behandeln



4.2.1 Ansatz

- Lese Symbole im Text ohne fixes Ziel
- Prüfe nach jedem Schritt, ob gelesene Folge Produktion entspricht
  - Wenn ja: Reduziere auf Syntaxkonstrukt (REDUCE)
  - Wenn nein: Lese weiteres Symbol im Text (SHIFT)
- Am Schluss bleibt Startsymbol übrig, sonst Syntaxfehler

4.2.2 Beispielablauf

Schritt	Erkannte Konstrukte	Rest der Eingabe
		1 + (2 - 3)
SHIFT	1	+ (2 - 3)
REDUCE	Term	+ (2 - 3)
SHIFT	Term +	(2 - 3)
SHIFT	Term + (	2 - 3)
SHIFT	Term + (2	- 3)
REDUCE	Term + (Term	- 3)
SHIFT	Term + (Term -	- 3)
SHIFT	Term + (Term - 3	)
REDUCE	Term + (Term - Term	)
REDUCE	Term + (Expression	)
SHIFT	Term + (Expression)	
REDUCE	Term + Term	
REDUCE	Expression	

4.2.3 Parser Tabelle - Vereinfacht

Erkannte Konstrukte	Regel
... Number	REDUCE Term
... Term + Term	REDUCE Expression
... "(" Expression ")"	REDUCE Term
Sonst	SHIFT

Suffix des Zustands entscheidet (Stack-Prinzip)

4.3 LR-Parser (Bottom-Up) Varianten

LR(0)
- Parse Tabelle ohne Lookahead erstellen
- Zustand reicht, um zu entscheiden

SLR(k) (Simple LR)
- Lookahead bei REDUCE, um Konflikt zu lösen
- Keine neuen Zustände

LALR(k) (Look-Ahead LR)
- Analysiert Sprache auf LR(0)-Konflikte
- Benutzt Lookahead bei Konfliktstellen mit neuen Zuständen

LR(k)
- Pro Grammatikschritt + Lookahead ein Zustand
- Nicht praxistauglich, zu viele Zustände

5 Semantic Checker

- Kümmert sich um die semantische Analyse
- Input: Syntaxbaum
- Output: Zwischendarstellung (Syntaxbaum + Symboltabelle)

5.1 Semantische Prüfung

Deklarationen
- Jeder Identifier ist eindeutig deklariert

Typen
- Typregeln sind erfüllt

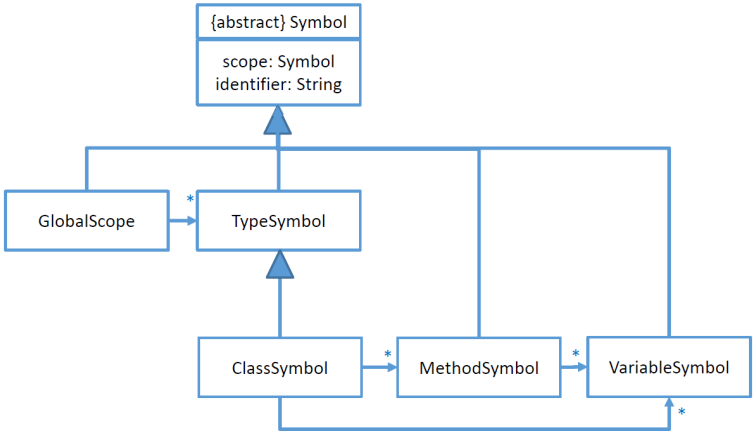
Methodenaufrufe
- Argumente und Parameter sind kompatibel

Weitere Regeln
- z.B. keine zyklischen Vererbung
- nur eine main() Methode

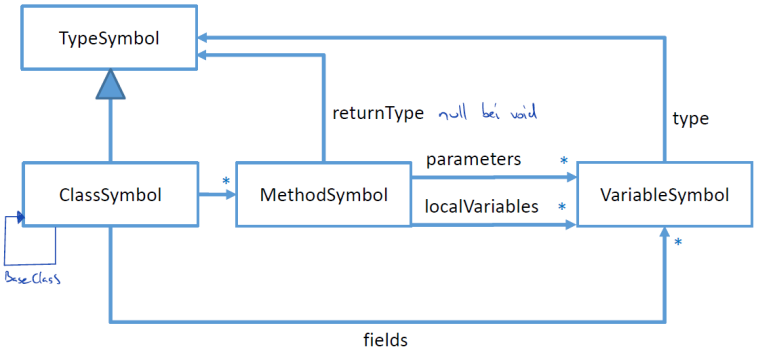
5.2 Symboltabelle

- Datenstruktur zur Verwaltung der Deklarationen
- Wiederspiegelt hierarchische Bereiche im Programm

5.2.1 Design



5.2.2 Detailliertere Beziehungen



5.2.3 Design Aspekte

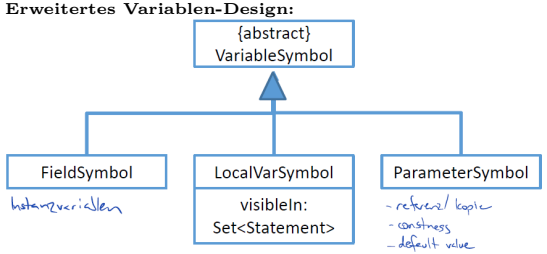
Typinfo für Variable-Symbol
- Zuerst unaufgelöst (Identifier)

Weitere Infos
- Klassen: Basisklasse

Lokale Variablen
- Deklarationsbereich merken (Statements)

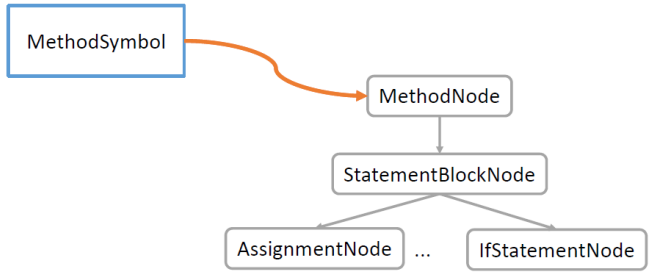
Erweitertes Typ-Design
- Klassen
- Basistypen (int, boolean, string)
- Arrays

Erweitertes Variablen-Design:



5.2.4 AST Verknüpfung

- Symboltabelle enthält Mapping Symbol → AST
- Für alle Deklarationen




```
node = symbolTable.getDeclarationNode(symbol)
```

5.3 Global Scope

- Mehrere Klassen im Programm

5.4 Shadowing

- Deklarationen in inneren Bereichen verdecken gleichnamige von äusseren Bereichen
- Hiding: Bei gleicher Member-Name bei Vererbung

5.5 Vorgehen

1. Konstruktion der Symboltabelle
2. Typen in Tabelle auflösen
3. Deklaration in AST auflösen
4. Typen in AST auflösen

5.5.1 1. Konstruktion der Symboltabelle

AST traversieren
- Beginne mit Global Scope
- Pro Klasse, Methode, Parameter, Variable: Symbol in übergeordnetem Scope einfügen
- Explizit und/oder mit Visitor

Forward-Referenzen → Typ-Namen und Designatoren noch nicht auflösen!
- Da vlt noch nicht alle Klassen in der Symboltabelle sind

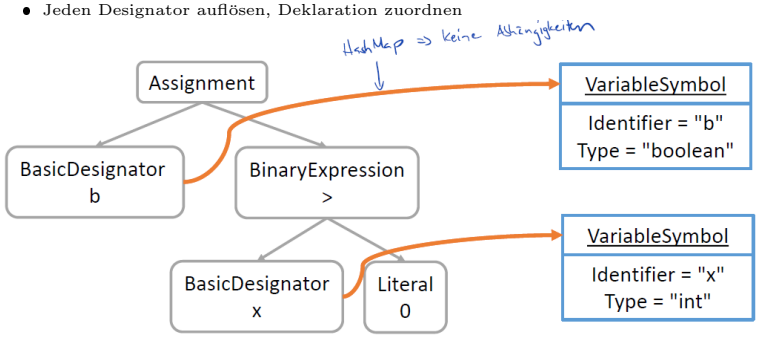
5.5.2 2. Typen in Tabelle auflösen

- Für Variablentype, Parametertyp, Rückgabetyp etc.
- Brauche Suche für Identifier auf Symboltabelle
  - Starte mit innerstem Scope
  - Suche stetig nach aussen ausbreiten
  - Zuletzt in Global Scope suchen, ansonsten nicht vorhanden

```
Symbol find(Symbol scope, String identifier) {
    if(scope == null) {
        return null; // nicht im global scope
    }
    for (Symbol declaration : scope.allDeclarations()) {
        if(declaration.getIdentifier().equals(identifier)) {
            return declaration;
        }
    }
    return find(scope.getScope(), identifier); // rekursiv in nächst höheren Bereich
}
```

5.5.3 3. Deklaration in AST auflösen

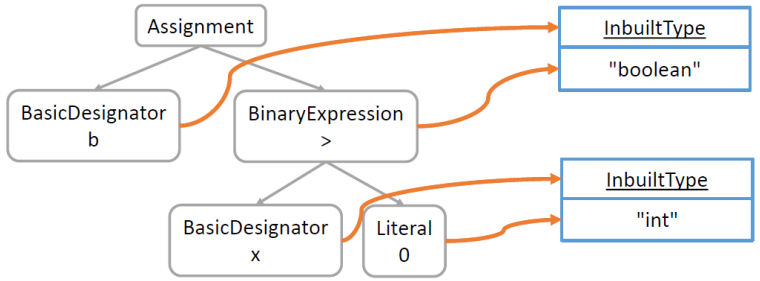
- Traversiere Ausführungscode in AST
- Jeden Designator auflösen, Deklaration zuordnen



5.5.4 4. Typen in AST bestimmen

Typ zu jeder Expression zuordnen

- Literal: definierter Typ
- Designator: Typ der Deklaration
- Unary/BinaryExpression: Resultat des Operators



Ablauf der Typenbestimmung:

- Post-Order-Traversierung
- AST am besten nicht erweitern sondern Maps in Symboltabelle verwenden

5.5.5 Typauflösung per Visitor

```
@Override
public void visit(BinaryExpressionNode node) {
    Visitor.super.visit(node); // post-order travers
    var leftType = symboltable.findType(node.getLeft());
    var rightType = symboltable.findType(node.getRight());
    // ...
    switch(node.getOperator()) {
        case PLUS -> {
            checkType(leftType, globalScope.getIntType());
            checkType(rightType, globalScope.getIntType());
            symboltable.fixType(node, globalScope.getIntType());
        }
        // ...
    }
}
```

5.6 Semantic Checks

- Alle Designatoren beziehen sich auf Variablen/Methoden
- Typen stimmen bei Operatoren
- Kompatible Typen bei Zuweisungen
- Argumentliste passt auf Parameterliste
- Bedingungen in if, while sind boolean
- Return Ausdruck passt
- Keine Mehrfachdeklaration
- Kein Identifier ist reserviertes Keyword
- Exakt eine main() Methode
- Array length is read-only
- Kein Exit ohne Return (ausser void)
- Lesen von unitialisierten Variablen
- Null-Dereferenzierung
- Ungültiger Array-Index
- Division by Zero
- Out of Memory bei new()

6 Code Generator

6.1 Aufgabe

Erzeugung von ausführbarem Maschinencode

- Input: Zwischendarstellung (Symboltabelle + AST)
- Output: Maschinencode

Mögliche Zielmaschinen

- Reale Maschine, z.B. intel 64, ARM Prozessor
- VM, z.B. JVM, .NET CLI

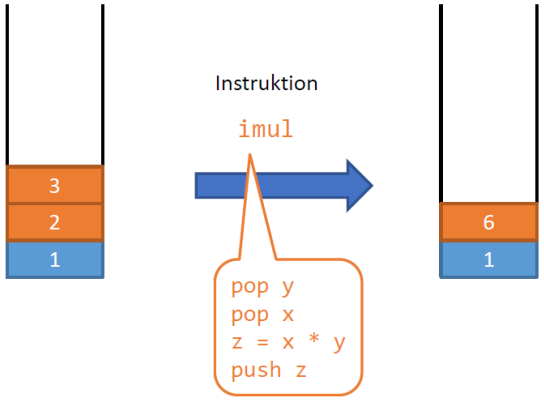
6.2 Unsere Zielmaschine

Kernkonzepte

- Virtueller Stack-Prozessor: keine Register
- Branch Instructions (Goto): Programmfluss steuern
- Metadaten

6.3 Stack Prozessor

- Instruktionen benutzen Auswertungs-Stack
- Keine Register wie auf echten Prozessoren



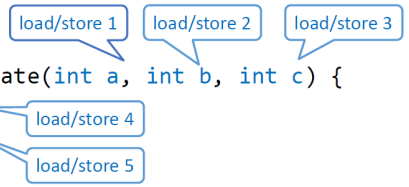
6.4 Auswertungs-Stack

- Jede Instruktion hat definierte Anzahl von Pop- und Push-Aufrufen
- Eigener Stack pro Methodenaufruf
- Stack hat unbeschränkte Kapazität

Instruktion	Bedeutung	Auswertungs-Stack
`ldc <const>`	Lade Konstante (int, boolean, string)	1 Push
`iadd`	Integer Addition	2 Pop, 1 Push
`isub`	Subtraktion	2 Pop, 1 Push
`imul`	Multiplikation	2 Pop, 1 Push
`idiv`	Integer Division	2 Pop, 1 Push
`irem`	Modulo	2 Pop, 1 Push
`ineg`	Integer Negation	1 Pop, 1 Push
`load <num>`	Lade parameter oder lokale Variable	1 Push
`store <num>`	Speichere Parameter oder lokale Variable	1 Pop

6.4.1 Load/Store Nummerierungen

- *this* Referenz: Index 0
- Danach, n **Parameters**: Index $1..n$
- Danach, m **lokale Variablen**: Index $n+1...n+m$



6.4.2 Compare-Instruktionen

Instruktion	Bedeutung
`cmpeq`	Compare Equal (verschiedene Typen)
`cmpne`	Compare Not Equal (verschiedene Typen)
`icmpgt`	Integer Compare Greater Than
`icmpge`	Integer Compare Greater Equal
`icmplt`	Integer Compare Less Than
`icmple`	Integer Compare Less Equal

Pop right, Pop left, Push boolean

6.4.3 Branch-Instruktionen

Instruktion	Bedeutung	Auswertungs-Stack
`goto <label>`	Branch (bedingungslos)	-
`if_true <label>`	Branch falls true	1 pop
`if_false <label>`	Branch falls false	1 pop

6.4.4 Metadaten

- Zwischensprache kennt alle Informationen zu
  - Klassen (Namen, Typen der Fields und Methoden)
  - Methoden (Namen, Parametertypen und Rückgabetyp)
  - Lokale Variablen (Typen)
- Kein direktes Speicherlayout festgelegt
- Nicht enthalten
  - Namen von lokalen Variablen und Parameter
  - Diese sind nur nummeriert

Verwendung:

- Speicherplatz-Allozierung
- Fehlermeldungen
- Funktionsaufrufe

6.5 Code Generierung

1. Traversiere Symboltabelle: Erzeuge Bytecode Metadaten
2. Traversiere AST pro Methode (Visitor): Erzeuge Instruktionen via Bytecode Assembler
3. Serialisiere in Output Format

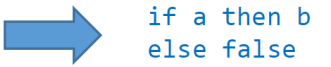
6.5.1 Backpatching

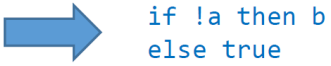
- Branch Offsets auflösen
- Label: relativer Instruktions-Offset ab Ende der aktuellen Branch-Instruktion

6.5.2 Template-Basierte Generierung

- Postorder-Traversierung: Kinder zuerst besuchen
- Jeweils Template für erkanntes Teilbaum-Muster anwenden

6.5.3 Short-Circuit Semantik

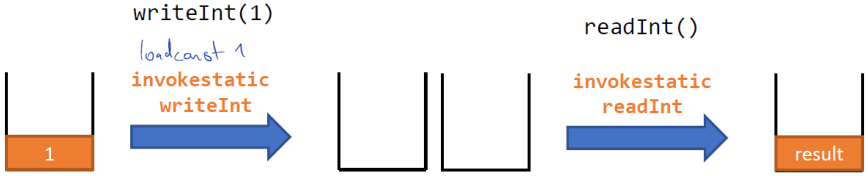




6.5.4 Methodenaufruf

Statisch

- Vordefinierte Methoden: *readInt*() etc.

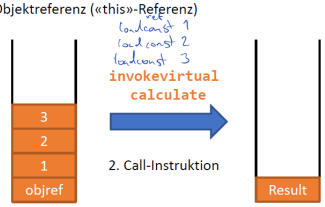


Virtuell

- Alle anderen Methoden

```
objref.calculate(1, 2, 3);
```

1. Argumente sind auf Stack (letzter zuoberst), zuunterst Objektreferenz («this»-Referenz)



2. Call-Instruktion

3. Call entfernt Argumente & Objektreferenz, legt Rückgabewert auf Stack (falls nicht void)

6.5.5 Parameter & Rückgabe

```
int sum(int x, int y) {
    return x + y;
}

load 1 // load param x
load 2 // load param y
iadd // x + y
ret // return from method (auch bei void, max 1 Wert auf Stack)
```

7 Virtual Machine

Hypotetische Maschine mit virtuellem Prozessor
- Eigener Instruktionssatz: Intermediate Language
- Hülle um den realen Prozessor

Nutzen:
- Mehrplattformen
- Mehrsprachigkeit
- Sicherheit

7.1 Loader

- Lädt Zwischencode (File) in Speicher
- Alloziert Speicher (Metadaten für Klassen, Methoden, Variablen, Code)
- Definiert Layouts (Speicherbereiche für Fields/Variablen/Parameter)
- Address Relocation
  - Löst Verweise auf zu Methoden, Typen, andere Assemblies
- Initiiert Programmausführung
- Optional: Verifier

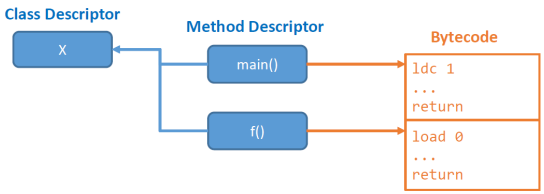
7.1.1 Verifier

Erkenne und verhindere falscher IL-Code
- Statische Analyse zur Ladezeit
- Fehler in Compuler, böswillige Manipulationen

Denkbare Fehler
- Typfehler
- Stack-Überlauf oder Unterlauf
- Nicht definierte Variablen/Methoden/Klassen
- Illegale Sprünge

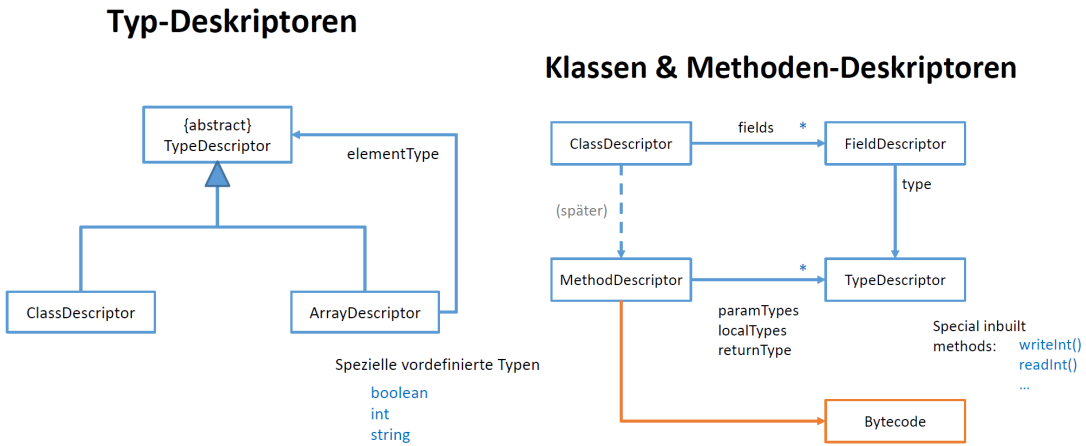
Alternative: Überprüfung zur Laufzeit

7.1.2 Metadaten



7.1.3 Deskriptoren

Laufzeitinfo für Typen & Methoden
- Typen: Klassen, Arrays oder Basistypen
- Klassen: Field-Typen
- Methoden: Typen von Parameter & Locals, Rückgabetyp, Bytecode
- Zusätzlich bei Klassen: Parent-Klasse, virtual Method Table

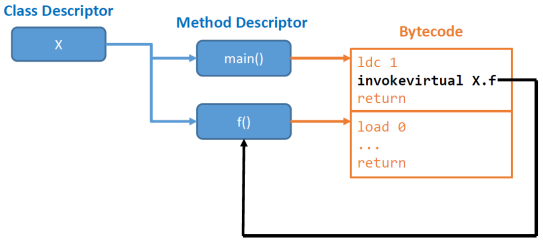


7.1.4 Bytecode Loading

- Von File direkt in Speicher geladen
- **Patching/Fixup:** Argumente in Instruktion anpassen
- Referenzen auf entsprechende Metadaten

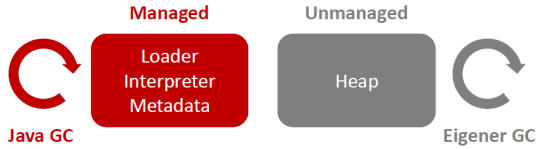
Original	Patched
invokevirtual MyMethod	invokevirtual <method_desc>
new MyClass	new <class_desc>
newarr MyType	newarr <type_desc>
getfield MyField	getfield <field_desc>
putfield MyField	putfield <field_desc>
...	

7.1.5 Patching

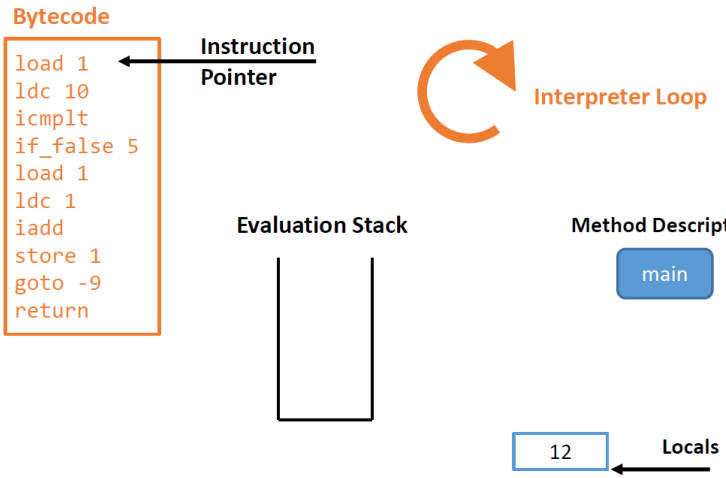


7.1.6 VM: Managed & Unmanaged

Kleine Unmanaged Teile neben der Java VM
- Heap und HW-Excution (JIT)



7.2 Interpreter



7.2.1 Bestandteile

Interpreter Loop
- Emuliert Instruktion nach der anderen

Instruction Pointer (IP)
- Adresse der nächsten Instruktion

Evaluation Stack
- Für virtuellen Stack Prozessor

Locals & Parameter
- Für aktive Methoden

Method Deskriptor
- Für aktive Methode

7.2.2 Implementation

```
private void execute(Instruction instruction) {
    var operand = instruction.getOperand();
    var frame = activeFrame();
    switch (instruction.getOpCode()) {
    case LDC -> push(operand);
    case ACONST_NULL -> push(null);
    case IADD -> {
        int right = checkInt(pop());
        int left = checkInt(pop());
        push(left + right);
    }
    // ISUB, IMUL, IDIV, IREM, INEG, BNEG
    case CMPEQ -> {
        var right = pop();
        var left = pop();
        if(left != null) {
            push(left.equals(right));
        } else {
            push(left == right);
        }
    }
    // CMPNE
    case ICMPLT -> {
        int right = checkInt(pop());
        int left = checkInt(pop());
        push(left < right);
    }
    // ICMPLE, ICMPGT, ICMPGE
    case IF_TRUE -> {
        if(checkBoolean(pop())) {
            frame.setInstructionPointer(frame.getInstructionPointer() + checkInt(operand))
                ;
        }
    }
    // IF_FALSE
    case GOTO -> {
        frame.setInstructionPointer(frame.getInstructionPointer() + checkInt(operand));
    }
    case LOAD -> {
        push(getParamOrLocal(checkInt(operand)));
    }
    case STORE -> {
        setParamOrLocal(checkInt(operand), pop());
    }
    case ALOAD -> {
        var arrayIndex = checkInt(pop());
        var pointer = checkPointer(pop());
        if(pointer == null) {
            throw new InvalidBytecodeException("Null pointer");
        }
        var length = heap.getArrayLength(pointer);
        if(arrayIndex < 0 || arrayIndex >= length) {
            throw new InvalidBytecodeException("Index out of bound");
        }
        var element = heap.readElement(pointer, arrayIndex);
        push(element);
    }
    // ASTORE
    case ARRAYLENGTH -> {
        var pointer = checkPointer(pop());
        if(pointer == null) {
            throw new InvalidBytecodeException("Null pointer");
        }
        push(heap.getArrayLength(pointer));

    }
    // GETFIELD
    case PUTFIELD -> {
        var field = checkFieldDescriptor(operand);
        var index = field.getIndex();
        var value = pop();
        checkType(value, field.getType());
        var instance = checkPointer(pop());
        if(instance == null) {
            throw new InvalidBytecodeException("Accessing uninitialized object");
        }
        heap.writeField(instance, index, value);
    }
    case NEW -> {
        var type = checkClassDescriptor(operand);
        var instance = newObject(type);
        push(instance);
    }
    case NEWARRAY -> {
        var length = checkInt(pop());
        if(length < 0) {
            throw new InvalidBytecodeException("Negative array size");
        }
        var descriptor = checkArrayDescriptor(operand);
        var pointer = heap.allocateArray(descriptor, length);
        for (int i = 0; i < length; i++) {
            heap.writeElement(pointer, i, defaultValue(descriptor.getElementType()));
        }
        push(pointer);
    }
    case INSTANCEOF -> instanceofTest(operand);
    case CHECKCAST -> checkCast(operand);
    case INVOKESTATIC -> invokeStatic(operand);
    case INVOKEVIRTUAL -> invokeVirtual(operand);
    case RETURN -> returnCall();
    default -> throw new InvalidBytecodeException("Unsupported instruction opcode");
    }
}
```

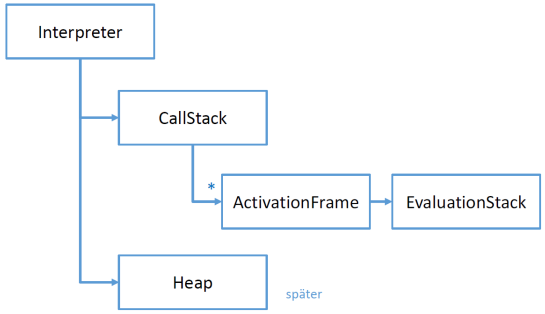
7.2.3 Prozedurale Unterstützung

Methodenaufrufe
- invokevirtual: Aufruf neuer Methode
- return: Rücksprung aus Methode

Activation Frame
- Datenraum einer Methode
- Parameter, lokale Variablen, temporäre Auswertungen

Call Stack
- Stack der Activation Frames gemäss Aufrufreihenfolge
- Design:
  - Managed im Interpreter: OO Darstellung für Komfort
  - Unmanaged bei HW Execution: Kontinuierlicher Speicherblock für Effizienz

7.2.4 Gesamtbild



7.2.5 Verifikation im Interpreter

Korrekte Benutzung der Instruktionen
- Typen stimmen (e.g. *checkInt()*)
- Methodenaufrufe stimmen (Argumente, Rückgabe etc.)
- Sprünge sind gültig
- Op-Codes stimmen

7.2.6 Sicherheitsmassnahmen

- Korrekter Bytecode und Typkonsistenz prüfen
- Variablen immer initialisieren (auch lokale)
- Checks durchführen (Null, Array-Index etc.)
- Stack Overflow und Underflow Detection

7.2.7 Interpretation vs. Kompilation

Interpreter ist ineffizient
- Dafür flexibel und einfach zu entwickeln
- Akzeptabel für selten ausgeführten Code

Kompilierter HW-Prozessor Code ist schneller
- Just-In-Time (JIT) Compilation für Hot Spots
- Kompilation kostet, Laufzeit macht es (allenfalls) wett

8 Object-Orientierung

8.1 Objekte zur Laufzeit

- Ablage erzeugter Objekte auf Heap
- Heap: Linearer Adressraum

Wiso werden Objekte nicht auf dem Stack augelegt?

- Sie würden Methoden-Ende nicht überleben, sondern mit Activation Frame zerstört werden.

Objekt Lebenszeit

- Nicht an Methodeninkarnatiion gebunden

8.1.1 Deallokation

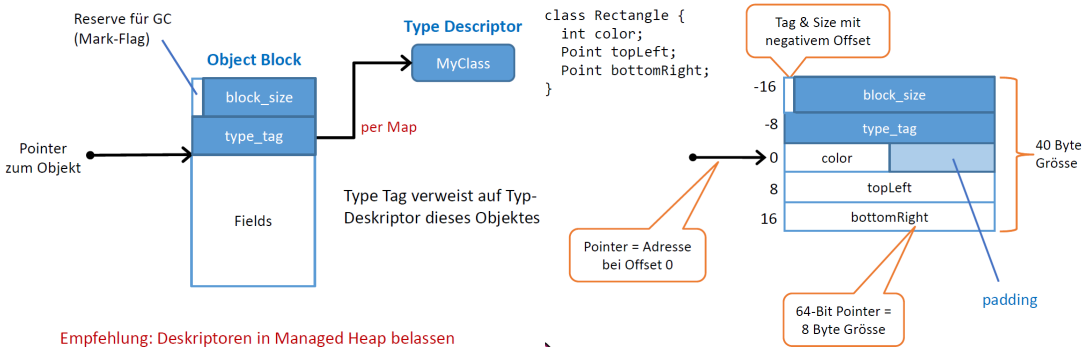
- Keine Hierarchie unter Objekten
- Keine hierarchische Lebensdauer
- Deallokation verursacht Lücken

8.2 Unmanaged Memory in Java

- Speicher über Java Native Access allozieren
- Raw Heap kann keine Java Referenzen speichern! Stattdessen Map verwenden ($HashMap<Long, Object>$)

```
import com.sun.jna.Memory;

Memory heap = new Memory(HEAP_SIZE);

long value = heap.getLong(address); // 64-bit lesen, address = offset im Heap
// ...
heap.setLong(address, value);
```

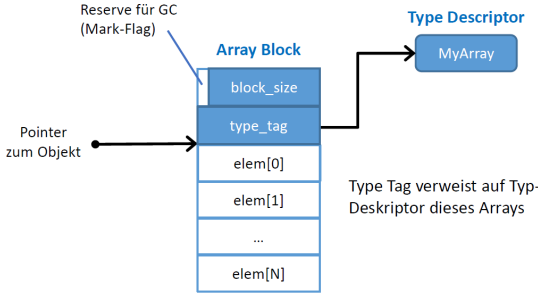
8.2.1 Objektblock im Raw Heap




```
class Rectangle {
  int color;
  Point topLeft;
  Point bottomRight;
}
```


Empfehlung: Deskriptoren in Managed Heap belassen

8.2.2 Array Block



Einfachheit: Array-Länge durch Blockgrösse berechnen (kein Paddding)

8.2.3 Heap-Allokation

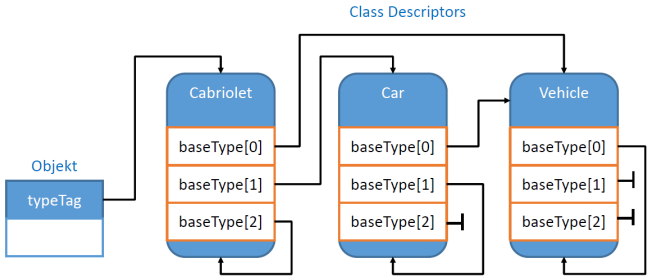
```
Pointer allocate(int size, TypeDescriptor type) {
    int blockSize = size + 16; // Mit Header
    if(freePointer + blockSize > limit) {
        throw new VMException("Out of Memory");
    }
    long address = freePointer;
    freePointer += blockSize;
    heap.setLong(address, blockSize);
    setTypeDescriptor(type, address);
    address += 16;
    return new Pointer(address);
}

Pointer allocateObject(ClassDescriptor type) {
    int size = type.getAllFields().length * 8; // Einfaches Layout
    return allocate(size, type);
}

Pointer allocateArraay(ArrayDescriptor type, int length) {
    int size = length * 8;
    return allocate(size, type);
}
```

9 Typ Polymorphismus

Dynamische Typ-Bestimmung

- Typ-Deskriptor = Dynamischer Typ des Objektes

9.1 Ancestor Tables

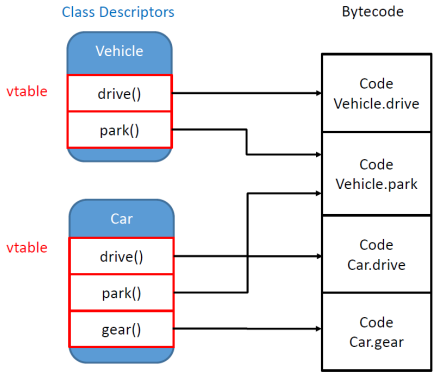


ancestor[i] = Zeiger auf Class Descriptor der Stufe i

9.2 Implementierung

```
private void instanceofTest(Object operand) {
    var instance = checkPointer(pop());
    if (instance == null) {
        push(false);
    } else {
        var targetType = checkClassDescriptor(operand);
        push(typeTest(instance, targetType));
    }
}

private void checkCast(Object operand) {
    var instance = checkPointer(pop());
    push(instance);
    var targetType = checkClassDescriptor(operand);
    if (!typeTest(instance, targetType)) {
        throw new VMException("Invalid cast");
    }
}
```

9.3 Virtual Method Table



9.3.1 Lineare Erweiterung

Jede virtuelle Methode hat einen Eintrag

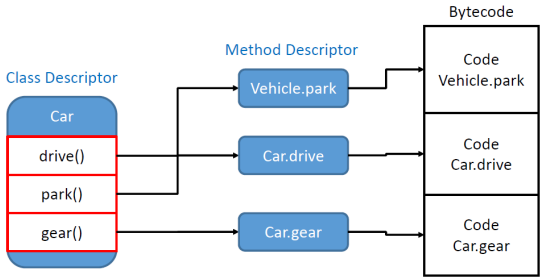
- Methoden der Basisklasse oben
- Neu deklarierte Methoden der Subklasse unten
- Funktioniert nur bei Single Inheritance

9.3.2 Method Position

- Jede virtuelle Methode hat fixe Position in vtable
- Position ist statisch bekannt im deklarierten Typ

9.3.3 Method Descriptor

- Zusätzliche Indirektion über Methoden-Deskriptor
- Interpreter braucht Infos zu Typen von Params/Locals

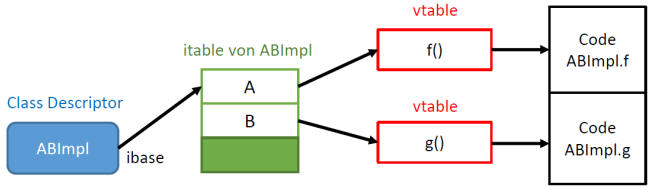


9.3.4 Implementation

```
private void invokeVirtual(Object operand) {
    var staticMethod = checkMethodDescriptor(operand);
    var parameterTypes = staticMethod.getParameterTypes();
    var arguments = new Object[parameterTypes.length];
    for (int index = arguments.length - 1; index >= 0; index--) {
        arguments[index] = pop();
        checkType(arguments[index], parameterTypes[index]);
    }
    var target = checkPointer(pop());
    invokeVirtual(staticMethod, target, arguments);
}

private void invokeVirtual(MethodDescriptor staticMethod, Pointer target, Object[]
    arguments) {
    if (target == null) {
        throw new VMException("Null dereferenced");
    }
    var type = checkClassDescriptor(heap.getDescriptor(target));
    var position = staticMethod.getPosition();
    var vtable = type.getVirtualTable();
    var dynamicMethod = vtable[position];
    var locals = initLocals(dynamicMethod.getLocalTypes());
    if (useJIT && jitEngine.supports(dynamicMethod)) {
        performJITCall(dynamicMethod, arguments);
    } else {
        callStack.push(new ActivationFrame(dynamicMethod, target, arguments, locals));
    }
}
```

9.4 Interface Support

- Interfaces global durchnummerieren
- Pro Class Descriptor eine Interface Tabelle (itable)
  - Interfaces nach Nummer = Position eintragen
- Einträge in *itable* verweisen auf *vtable*
- Class Deskriptor verweist auf *itable* (ibase-Pointer)



9.4.1 Verzahnte Interface Tabelle

Grund

- Einfache Interface Tabellen sind ein Speicherproblem
- Lange itables bei vielen Interfaces
- Viele Lücken wegen globaler Nummerierung

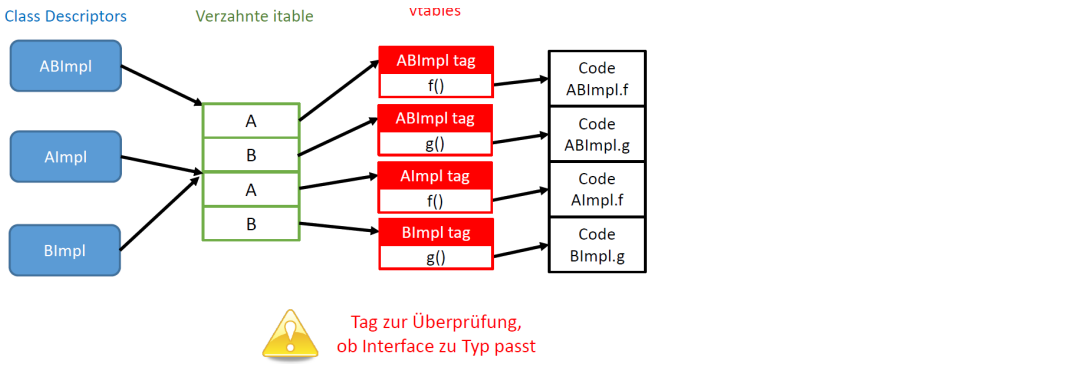
Konzept

- *itables* im Speicher kollisionsfrei übereinanderlegen
- Muss nun prüfen, ob Eintrag für Typ gültig ist (Vermerk des Class Descriptors in *vtable*)

9.4.2 Verschiedene Offsets

- Verzahnung auch mit veschiedenen Offsets möglich

9.4.3 Gesamtbild



Tag zur Überprüfung,
ob Interface zu Typ passt

10 Garbage Collection

Typ-Deskriptor Zweck
- Ancestor Table für Typ-Test & Cast
- Virtual Method Table für Dynamic Dispatch
- Interpreter Metadata bei Field- / Array Typen
- **Neu:** Pointer Offsets für Garbage Collection

10.1 Explizite Freigabe

- Delete Statement zum Deallozieren eines Objektes

10.1.1 Probleme

- Dangling Pointers: Referenz auf gelöschtes Objekt
- Memory Leaks: Verweiste Objekte, die nicht abräumbar sind

10.1.2 Dangling Pointer

- Zeigt auf Lücke oder falsches Objekt im Heap
- Lese nicht berechtigten Speicher (Security Issue)
- Überschreibe fremden Speicher (Safety + Security Issue)

10.1.3 Memory Leak

- Unbenötigtes Objekt, das nicht löschbar ist
- Es gibt keine benutzbaren Referenzen mehr darauf

10.2 Garbage Collection

- Laufzeitsystem kümmert sich um die automatische Freigabe von Garbage
- Garbage = Objekte, die nicht mehr erreichbar sind und daher nicht mehr gebraucht werden

Ziel:
- Memory Safety
- Keine Dangling Pointers
- Keine Memory Leaks

10.2.1 Reference Counting

- RC pro Objekt
- Anzahl eingehender Referenzen
- Zyklische Objektstrukturen werden mit Reference Counting nie zu Garbage

Vorteil
- Sofortige Deallokation

Nachteile
- Falsch bei Zyklen
- Ineffizient

Untauglich

10.2.2 Transitive Erreichbarkeit

- Objekte beibehalten, die das Programm noch zugreifen könnte
- Ausgehend von Ankerpunkten (Root Set)

Root Set
- Referenzen in statischen Variablen
- Referenzen in Activation Frames auf Call Stack
- Referenzen in Register

10.3 Mark & Sweep Algorithmus

Mark Phase
- Markiere alle erreichbaren Objekte

Sweep Phase
- Lösche alle nicht markierten Objekte

10.3.1 Mark Phase

```
private void mark() {
    for(var root: getRootSet(stack)) {
        traverse(root);
    }
}

private void traverse(Pointer current) {
    if(current == null) {
        return;
    }
    long block = heap.getAddress(current) - Heap.BLOCK_HEADER_SIZE;
    if(!isMarked(block)) {
        setMark(block);
        for(var next: getPointers(current)) {
            traverse(next);
        }
    }
}
```

10.3.2 Rekursive Traversierung

- GC braucht zusätzlich Speicher für Stack
- Problematisch, da Speicher bei GC meist sowiso knapp ist
- Es exisitieren Algorithmen zur Traversierung ohne Zusatzspeicher (Pointer Rotation Algo)

10.3.3 Sweep Phase

- Linearer Scan über den gesamten Heap, alle Blöcke

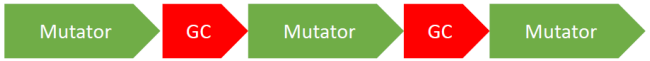
```
private void sweep() {
    var current = Heap.HEAP_START;
    while(current < Heap.HEAP_SIZE) {
        if(!isMarked(current) && !freeList.contains(current)) {
            free(current);
        }
        clearMark(current);
        current += heap.getBlockSize(current);
    }
}

private void free(long address) {
    freeList.add(address);
}
```

10.4 Ausführungszeitpunkt

Delayed Garbage Collection
- Garbage wird nicht sofort erkannt und freigegeben
- GC läuft spätestens, wenn der Heap voll ist (Check beim Allozieren)
- Eventuell prophylaktisch früher

10.5 Stop & Go

- GC läuft sequentiell und exklusiv
- Mutator = Produktives Programm
- Mutator ist während GC unterbrochen



10.6 Root Set Erkennung

Pointer auf dem Call Stack
- Pointers in Parameter
- Pointers in lokalen Variablen
- Pointers auf Evaluation Stack
- this-Referenz

10.7 Pointers im Objekt

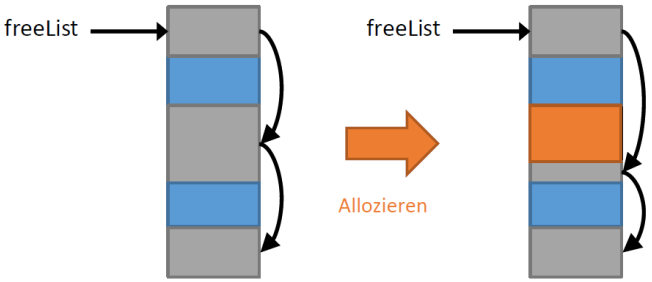
```
private Iterable<Pointer> getPointers(Pointer current) {
    var list = new ArrayList<Pointer>();
    var descriptor = heap.getDescriptor(current);
    if(descriptor instanceof ClassDescriptor classDescriptor) {
        var fields = classDescriptor.getAllFields();
        for(var i = 0; i < fields.length; i++) {
            var type = fields[i].getType();
            if(isPointerType(type)) {
                var value = heap.readField(current, i);
                if(value != null) {
                    list.add((Pointer) value);
                }
            }
        }
    } else if (descriptor instanceof ArrayDescriptor arrayDescriptor) {
        var length = heap.getArrayLength(current);
        for (int i = 0; i < length; i++) {
            var value = heap.readElement(current, i);
            var type = arrayDescriptor.getElementType();
            if (value != null && isPointerType(type)) {
                list.add((Pointer)value);
            }
        }
    }
    return list;
}

private boolean isPointerType(Object pointer) {
    return pointer instanceof ClassDescriptor || pointer instanceof ArrayDescriptor;
}
```

10.8 Free List

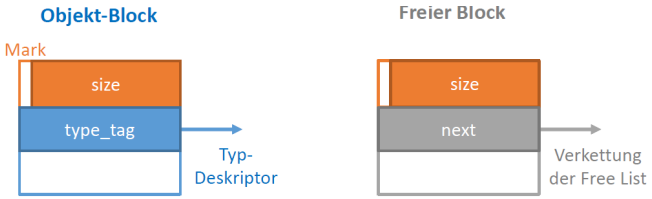
- Freie Blöcke linear verketten

10.8.1 Neue Heap Allozierung

- Traversiere Free List bis zu passendem Block
- Überschuss des Blockes wieder in free List einreihen



10.8.2 Heap Block Layout



10.8.3 Strategien

First Fit
- Keine Sortierung
- Suche erst passenden Block

Best Fit
- Nach aufsteigender Grösse sortiert
- Unbrauchbar kleine Fragmente

Worst Fit
- Nach absteigender Grösse sortiert
- Finde passenden Block sofort

Segregated Free List
- Mehrere Free Lists mit verschiedenen Grössenklassen
- e.g. 64..128, 128..196, 196..259, ...

Benachbarte freie Blöcke verschmelzen
- Am einfachsten in der Sweep Phase

Buddy System
- Diskrete Blockgrössen nach Adresse geordnet
- Exponentielle Blockgrösse
- Sehr schnelles Verschmelzen & Allozieren & Freigabe
- Aber grosse interne Fragmentierung (unbrauchbare Reste)

11 GC Vertiefung

Pointers in C++
- expliziter Zahlenwert
- Jedes Wort kann Pointer sein
- Keine GC Metadaten in C/C++

11.1 Finalizer

- Methode, die vor Löschen des Objektes läuft (Abschlussarbeit wie z.B. Verbindung schliessen)
- Von GC initiiert, wenn Objekt Garbage geworden ist

```
class Block {
    @Override
    protected void finalize() {
        // ...
    }
}
```

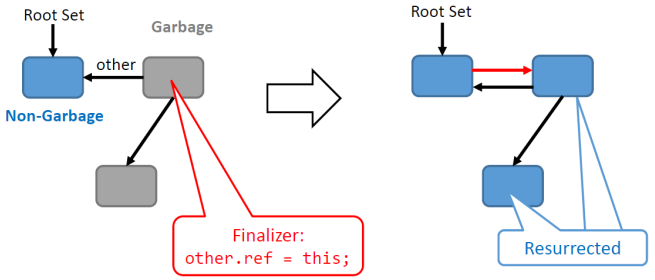
11.1.1 Separate Finalisierung

- Finalizer wird nicht in GC-Phase ausgeführt, sondern erst später

Gründe
- Finalizer dauert evtl. beliebig lange (blockiert sonst GC)
- Finalizer kann neue Objekte allozieren (korrumpiert GC)
- Programmierfehler im Finalizer (evtl. Crash des GC)
- Finalizer kann Objekt wieder weiterleben lassen (Resurrection)

11.1.2 Resurrection

- Finalizer kann bewirken, dass Objekt wieder lebendig wird und kein Garbage mehr ist
- Nicht nur das eigene Objekt, sondern auch indirekt andere Objekte können wiederauferstehen



11.1.3 Internals

Finalizer Set
- Registrierte Finalizer

Pending Queue
- Noch auszuführende Finalizer
- Garbage mit Finalizer werden in Pending Queue eingetragen
- Einfügen bewirkt Resurrection: Neue GC Phase nötig

11.1.4 Konsequenzen

GC braucht 2 Mark Phasen
- Markiere und erkenne Garbage mit Finalizer
- Markiere von Pending Queue erneut, dann Sweep

Objekt mit Finalizer braucht min 2 GC Durchläufe bis zur Freigabe
- Speicher kann evtl. nicht schnell genug frei werden

11.1.5 Programmieraspekte

- Reihenfolge der Finalizer ist unbestimmt
- Laufen beliebig verzögert
- Sind nebenläufig zum Hauptprogramm

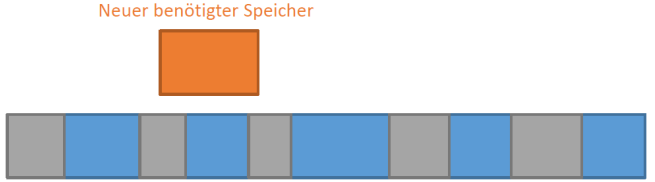
11.2 Weak Reference

- Zählt nicht als Referenz für GC
- Für Objekt Caches

Internals
- Indirektion via Weak Reference Table
- GC Mark ignoriert Table-Einträge
- GC Sweep nullt Einträge, falls Zielobjekte gelöscht

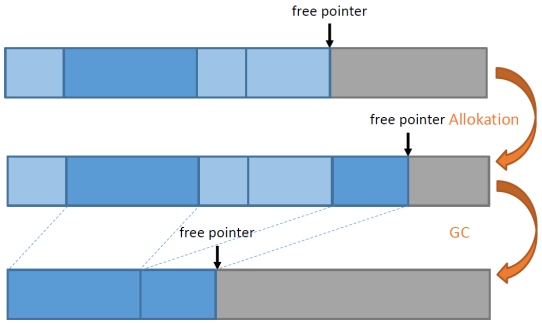
11.3 Externe Fragmentierung

Viele kleine Lücken im Heap durch Allozieren & Free
- Spätere grössere Allokation passt in keine Lücke
- Obschon Summe der freien Blöcken genügend wäre



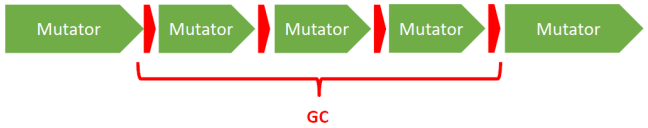
11.4 Compacting GC

- Auch Mark & Copy GC
- Allokation am Heap-Ende (supereffizient)
- GC schiebt Objekte wieder zusammen
- Bei Verschieben müssen Referenzen nachgetragen werden
- Konservative Methode daher unmöglich (C/C++ nicht möglich)



11.5 Inkrementeller GC

- GC soll quasi-parallel zum Mutator laufen
- Nur kleinste Unterbrüche (Inkrements)
- Kein Stop & Go / Stop the World



11.5.1 Generational GC

Zeitspiegelungsheuristik

- Junge Objekte: kurze Lebensdauer
- Alte Objekte: lange Lebensdauer

Alter	Generation	GC-Frequenz	GC Pause
neu	G0	hoch	kurz
mittel	G1	mittel	mittel
alt	G2	tief	lang

Root Sets bei Generationen

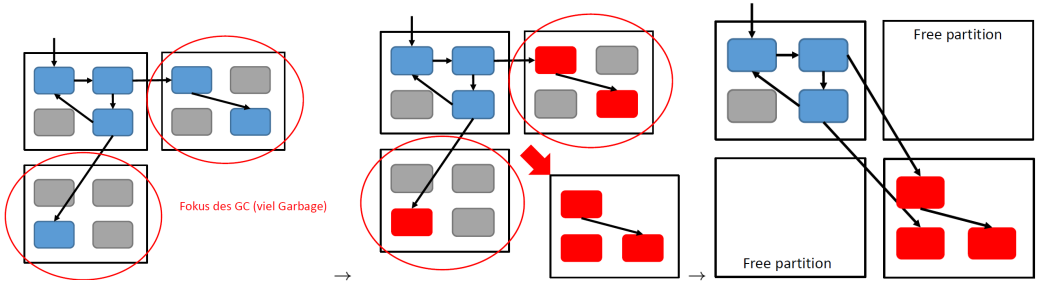
- Referenzen von alten zu neuen Generationen: Zusätzlicher Root Set der neuen Gen
- Write Barriers: Schreiben von Referenzen in alte Generationen erkennen
- Bei GC der alten Generationen müssen auch neuere mit aufgeräumt werden

11.5.2 Partitioned GC

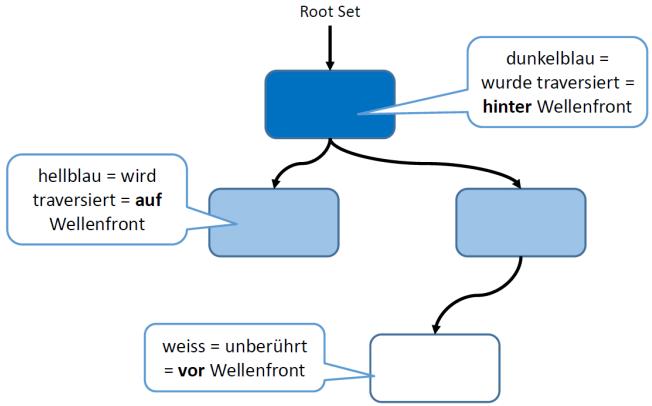
- Heap in Partitionen zerlegen
- Ziel: kurze GC Pausen
- Nebenläufiges Markieren mit Snapshots: Relevante nebenläufige Updates erkennen
- GC fokussiert auf Partitionen mit viel Garbage
- Evakuiere lebende Objekte in neue Partitionen

Problem

- Zyklischer Garbage zwischen Partitionen
- Braucht immer noch Full GC (Stop the World)

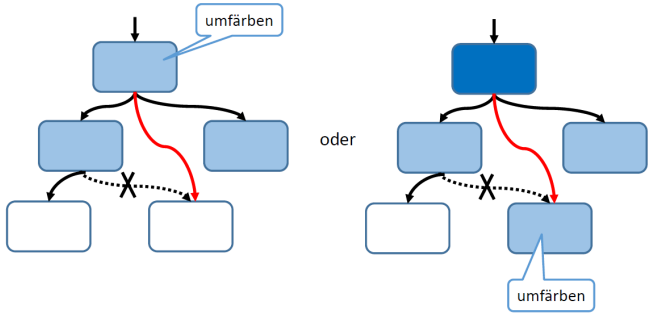


11.6 Wellenausbreitungsmodell



11.7 Isolationsproblem

- Referenzzuweisung durch Mutator während GC
- Kritisch: Neue Referenz von dunkelblau zu weiss
- **Heilung:** Quelle oder Ziel hellblau färben (Write Barrier)



12 Just-In-Time Compiler

- Effizientere Ausführung als Interpretation
- Bytecode direkt in nativen Prozessor-Code übersetzen und ausführen
- Nicht unbedingt alles, sondern nur kritische Teile

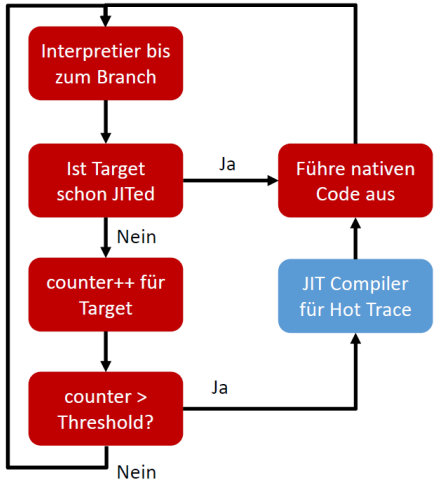
12.1 Hot Spot

- Performance-kritisher Code Abschnitt
- Wird häufig ausgeführt
- JIT-Kompilierung loht sich hierfür am meisten

12.2 Profiling

- Interpreter zählt Ausführung gewisser Code-Teile (Methoden, Traces (Code-Pfade))
- Falls häufig ausgeführt, JIT für den Teil anwerfen

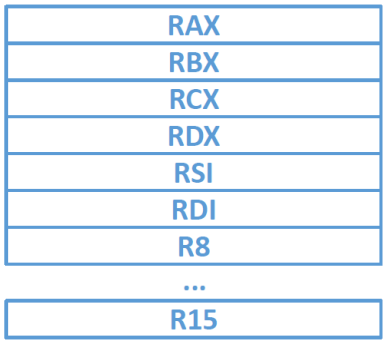
12.3 Vorgehen

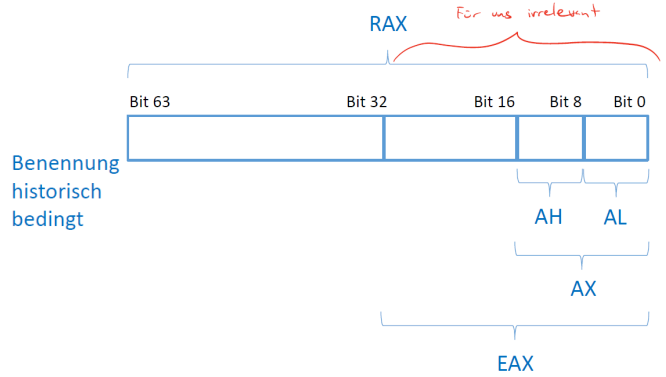


12.4 Intel 64 Architektur

Spezielle Register

- RSP: Stack Pointer
- RBP: Base Pointer
- RIP: Instruction Pointer





12.4.1 Elementare Instruktionen

`ADD RAX, RBX`	RAX += RBX
`SUB RAX, RBX`	RAX -= RBX
`IMUL RAX, RBX`	RAX *= RBX
`IDIV RBX`	RDX muss vorher 0 sein bei nicht-negativen RAX, sonst -1. RAX /= RBX, RDX = RAX % RBX

Signed Mul und Div

IDIV benutzt fixe Register (Register Clobbering)

`MOV RAX, 100`	Setze 100 in Register RAX
`MOV RAX, RBX`	Kopiere RBX in RAX

12.4.2 IDIV Vorbereiten

- Fixe 128-bit Division von RDX:RAX
- Setze RDX je nach RAX
  - 0 falls RAX >= 0
  - -1 falls RAX < 0

`CDQ`	Vorzeichenbehaftete Konversion von RAX nach RDX:RAX (Convert to Quad Word)

12.5 Register Allokation

Lokale Register Allokation

- Für Ausdrucksauswertung (Evaluation Stack)
- Evaluation Stack Einträge auf Register abbilden
- Cross Compiler führt Stack an belegten Register
- Pro übersetzte Bytecode-Instruktion wird Stack nachgeführt

Globale Register-Allokation

- Veriablen in Register speichern
- Deutlich schneller als Speicherzugriffe

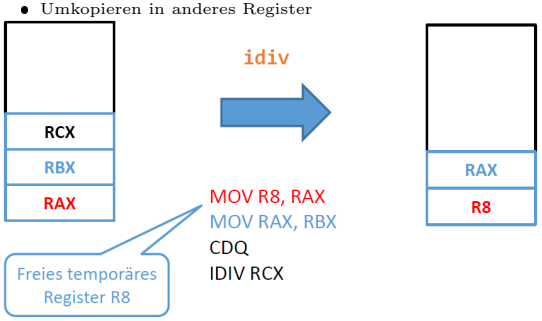
Registeranzahl ist beschränkt!

12.6 Register Clobbering

- Evakuiere in neues Register bei Instruktionen mit fixem Operand (z.B. IDIV)

12.6.1 Register Relocation

- Umkopieren in anderes Register



12.7 Intel Branches

- Bedingte Sprünge basierend auf Condition Code
- Condition Code aus vorherigem Vergleich

`CMP reg1, reg2`	Compare
`JE label`	Jump if equal
`JNE label`	Jump if not equal
`JG label`	Jump if greater (reg1 > reg2)
`JGE label`	Jump if greater equal (reg1 >= reg2)
`JL label`	Jump if less (reg1 < reg2)
`JLE label`	Jump if less (reg1 <= reg2)
`JMP label`	Unconditional jump