1 Einführung

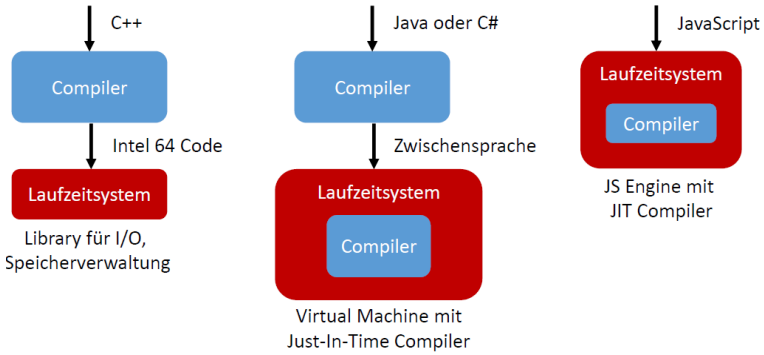
Wieso Compilerbau?

- Sprachkonzepte verstehen
- Einschränkungen und Kosten von Sprachfeatures beurteilen können
- Konzepte in verwandten Bereichen einsetzen: Converter, Analysen, Entwicklertools, Algorithmen

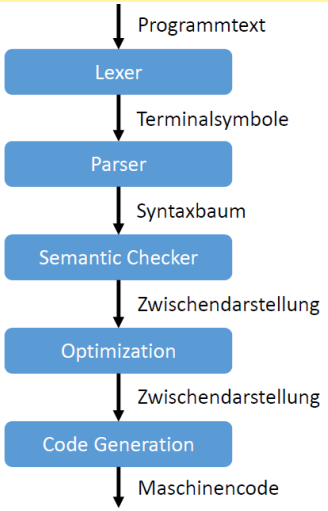
Compiler: Transformiert Quellcode einer Programmiersprache in ausführbaren Maschinencode.

Runtime System: Unterstützt die Ausführung mit software- und Hardware-Mechanismen

1.1 Architekturen

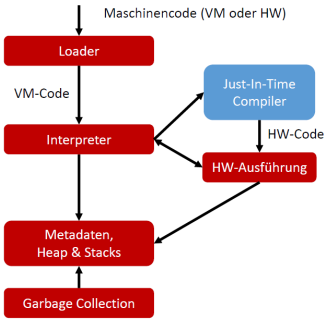


1.2 Aufbau Compiler



- Lexer (Lexikanische Analyse, Scanner)
  - Zerlegt Programmtext in Terminalsymbole (Tokens)
- Parser (Syntaktische Analyse)
  - Erzeugt Syntaxbaum gemäss Programmstruktur
- Semantic Checker (Schemantische Analyse)
  - Löst Symbole auf, prüft Typen und semantische regeln
- Optimization (Optional)
  - Wandelt Zwischendarstellung in effizientere um
- Code Generation
  - Erzeugt ausführbaren maschinencode
- Zwischendarstellung (Intermediate Representation)
  - Beschreibt Programm als Datenstruktur (diverse varianten)

1.3 Aufbau Laufzeitsystem



- Loader
  - Lädt maschinencode in Speicher, veranlasst Ausführung
- Interpreter
  - Liest Instruktionen und emuliert diese in Software
- JIT Compiler
  - Übersetzt Code-Teile in hardware-Instruktionscode
- HW-Ausführung (nativ)
  - Lässt Instruktionscode direkt auf HW-Prozessor laufen
- Metadaten, heap + Stacks
  - Merken programminfos, Objekte und Prozeduralaufrufe
- Garbage Collection
  - Räumt nicht erreichbare Objekte ab

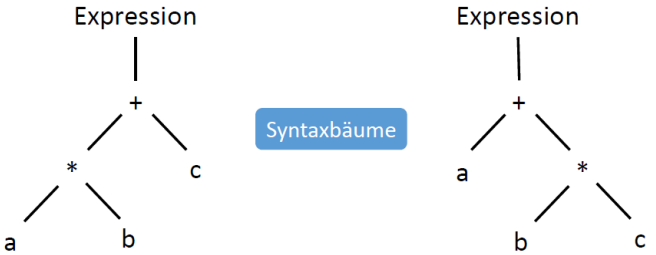
1.4 EBNF (Extended Backus Naur Form)

Definition einer Programmiersprache: Syntax (mittels Regeln/Formeln), Semantik (meist in Prosa)

1.4.1 EBNF Konstrukte

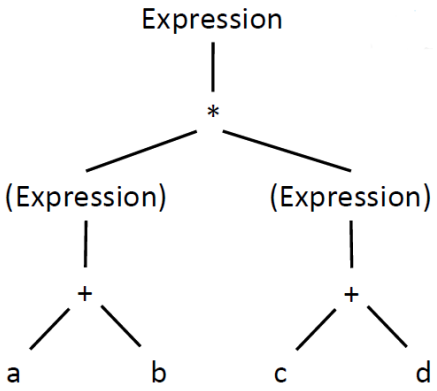
	Beispiel	Sätze
Konkatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" oder "B"
Option	["A"]	leer oder "A"
Wiederholung	{ "A" }	leer, "A", "AA", "AAA", etc.

1.4.2 Arithmetische Adrücke



Expression = Term | Expression "+" Term.
Term = Variable | Term "*" Variable.
Variable = "a" | "b" | "c" | "d".

1.4.3 Explizite Klammerungen



Expression = Term | Expression "+" Term.
Term = Factor | Term "*" Factor.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".

Expression = Term { "+" Term }.
Term = Factor { "*" Factor }.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".

2 Lexikanische Analyse

→ **Kümmert sich um die lexikanische Analyse**

Input: Zeichenfolge (Programmtext)

Output: Folge von Terminalsymbolen (Tokens)

Aufgaben:

- Fasst Textzeichen zu Tokens zusammen
- Eliminiert Whitespaces und Kommentare
- Merkt Position in Programmcode für Fehlermeldung/Debugging

Nutzen:

→ Erleichtert spätere syntaktische Analyse (Parser)

- Abstraktion: Parser muss sich nicht um Textzeichen kümmern
- Einfachheit: Parser braucht Lookahead pro Symbol, nicht Textzeichen
- Effizienz: Lexer benötigt Stack im Gegensatz zu Parser

2.1 Tokens

- **Statisch:** Keywords, Operationen, Interpunktion
  - if, else, while, *, &&, ;
- **Identifiers**
  - MyClass, readFile, name2
- **Zahlen**
  - 123, oxfe12, 1.2e-3
- **Strings**
  - "Hello!", "", "01234", "\n"
- **Evt weitere**
  - Einzelne Characters wie 'a', '0'

2.2 Reguläre Sprachen

Regulär: Als EBNF ohne Rekursion ausdrückbar!!

```
Integer = Digit { Digit }.
Digit = "0" | ... | "9".
Ausdruck = [ "(" Ausdruck ")" ].
```

Regulär

Nicht regulär

```
Integer = Digit [ Integer ].
```

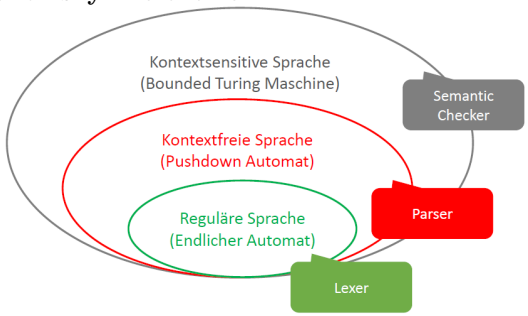
Umformung in äquivalente Syntax

```
Integer = Digit { Digit }.
```

keine Rekursionen

Regulär

Chomsky Hierarchie



2.3 Identifier

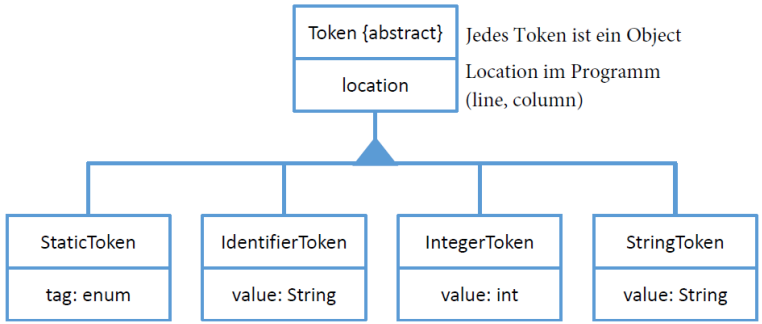
```
Identifier = Letter { Letter | Digit }.
Letter = "A" | ... | "Z" | "a" | ... | "z".
Digit = "0" | ... | "9".
```

- Bezeichner von Klassen Methoden, Variablen etc.
- Beginnt mit Buchstabe, danach Ziffern erlaubt
- (Java unterstützt auch Underscores, wir nicht)

2.4 Sonstiges

- **Maximum Munch:** Lexer absorbiert möglichst viel in einem Token
- **Whitespaces:** Von Lexer übersprungen, trennt Tokens, Tokens evt auch ohne Whitespac getrennt
- **Von Lexer übersprungen**
  - Blockkommentare: Nicht schachtelbar, weil sonst nicht mehr regulär
  - Zeilenkommentar: Bis Newline

2.5 Token-Model



2.6 Implementation

2.6.1 Tags für statische Tokens

Tipp: Reservierte Typnamen (void, boolean, int, string) und Werte (null, true, false) als Identifier im Lexer verarbeiten.

```
public enum Tag {
    CLASS,ELSE ,IF ,RETURN ,WHILE, ...
    AND, OR, PLUS, MINUS, SEMICOLON, ...
}
```

2.6.2 Lexer Gerüst

```
class Lexer {
    private final Reader reader;
    private char current; // One character lookahead
    private boolean end;

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

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

2.6.3 Token Stream lesen

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

2.6.4 Lexer Kernlogik

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

3 Parser Einführung

→ **Kümmert sich um die syntaktische Analyse**
Input: Folge von Terminalsymbolen (Tokens)
Output: Syntaxbaum/Parse Tree

Kontextfrei: Parser beschränkt sich auf kontextfreie Sprachen (in EBNF beschreibbar). Kontextabhängige Aspekte wie Boolean lassen sich nicht addieren etc. übernimmt der Semantic Checker
Aufgaben:

- Finde eindeutige Ableitung der Syntaxregeln, um einen gegebenen Input herzuleiten
- Analysiert die gesamte Syntaxdefinition
- Erkennt, ob Eingabetext Syntax erfüllt oder nicht
- Eindeutige Ableitung erwünscht
- Erzeugt Syntaxbaum

3.1 Concrete vs. Abstract Syntax Tree

Concrete: Ableitung der Syntaxregeln als Baum widerspiegelt
Abstract: Unwichtige Details auslassen, Struktur vereinfachen und für Weiterverarbbeitung massschneidern
→ Beides sind mögliche Intermediate Representations
→ Generierter Parser kann Concrete Syntax Tree liefern
→ Selbst implementierter Parser kann Abstract Syntax Tree liefern

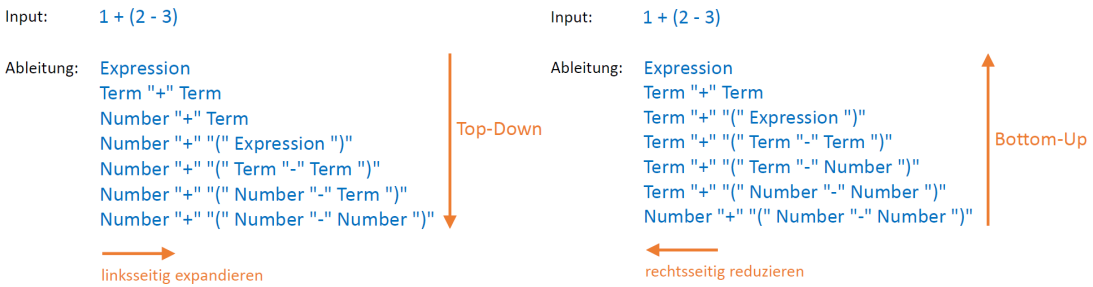
3.2 Parser Strategien

3.2.1 Top-Down

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

3.2.2 Buttom-Up

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



3.2.3 Recoursive Descent

- Pro Nicht-Terminalsymbol eine Methode
- Funktioniert bei rekursiven und nicht-rekursiven Produktionen

Diskusion:

- Recoursive Descent ist **Top-Down Parser**
  - Implizierter Stack durch Methodenaufrufe
  - Entspricht Push-Down Automat
- Zielorientierte Satzzerlegung (Predictive Direct)
  - Immer klar, welche Produktion genommen wird, Bevorzugte Vorgehensweise
- Anderer Ansatz: Backtracking
  - Falls unklar welche Produktion zu nehmen ist: Wähle Produktion aus, bei Syntaxfehler Undo und nächste probieren

3.3 Implementation

3.3.1 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();
    }
}
```

3.3.2 Parser Einstieg

```
private ProgramNode parseProgram() {
    var classes = new ArrayList<ClassNode>();
    try {
        while (!isEnd()) {
            next();
            classes.add(parseClass());
        }
    } catch(IllegalArgumentException e) {
        error(e.getMessage());
    }
    return new ProgramNode(location, classes);
}
```

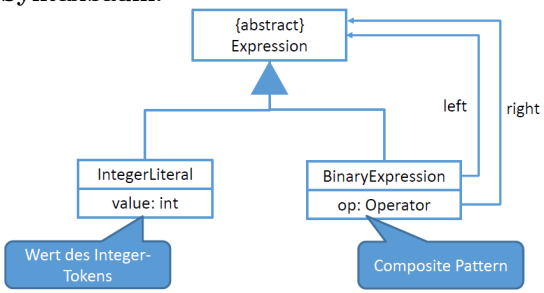
3.4 Parsen mit längerem Lookahead

```
// Statement = Assignment | Invocation.
// Assignment = Identifier -Ëxpression.
// Invocation = Identifier "("")".

// Umwandeln zu:
// Statement = Identifier (AssignmentRest | InvocationRest).
// AssignmentRest = -Ëxpression.
// Invocationrest = "("")".

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

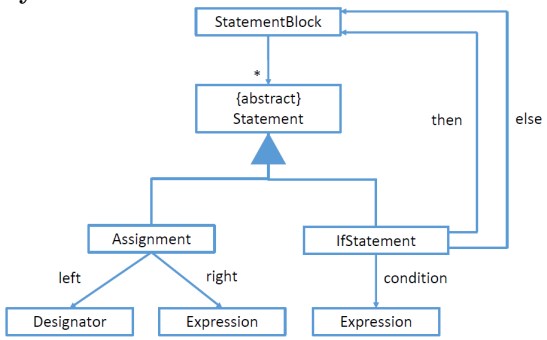
4 Parser Vertiefung

4.1 Syntaxbaum

Syntaxbaum:



Syntaxbaum:



4.2 Designfragen

- Abstrakt vs. konkret
  - Abstract Syntax Tree bei Eigendesign
  - Concrete Syntax Tree bei generiertem Parser
- Weitere Expression-Subklassen
  - UnaryExpression (z.B. für -3 oder +4)
  - Andere Literal-Typen (z.B. boolean, string)
  - Designator (z.B. für x oder y[0].z)
- Source Code Positionen merken
  - Für fehlermeldungen und Debugging
  - Von Lexer-Symbolstrom übernehmen

4.2.1 Term parsen

```
// 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();
        }
        return expression();
    } else {
        error();
    }
}
```

4.3 Syntaxfehler-Behandlung

- Weitermachen bei Fehler → Neuen Einstiegspunkt suchen
- Hypothesen nötig
  - Interpunktionsfehler sind häufig (z.B. fehlendes Semikolon)
  - Vergessener Operator ist selten (z.B. fehlendes Plus)
- Häufige Fehlerarten
  - Fehlendes Symbol wie Semikolon oder Klammer → ignorieren
  - Falsches Symbol wie falscher Klammertyp → ersetzen

Nicht erkannte Fehler: Müssen vom Semantic Checker geprüft werden

- Inkompatible Typen
- Anzahl Argumente ungleich Anzahl Parameter
- Nicht deklarierte Variablen/Methoden
- Ungültige Operanden

5 Semantische Analyse

→ **Kümmert sich um die semantische Analyse**
Input: Syntaxbaum (konkret oder abstrakt)
Output: Abstrakter Syntaxbaum + Symboltabelle

5.1 Semantische Prüfung

Prüfe, dass das Programm gemäss Sprchregeln Sinn macht.

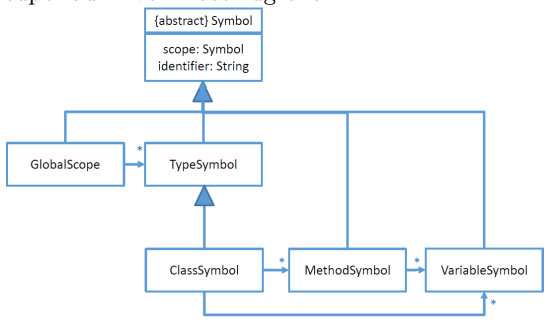
- Deklarationen
  - Jeder Identifier ist eindeutig deklariert
- Typen
  - Typregeln sind erfüllt
- Methodenaufrufe
  - Argumente und Parameter sind kompatibel
- Weitere Regeln
  - z.B. Keine zyklische Vererbung, nur eine main()-Methode

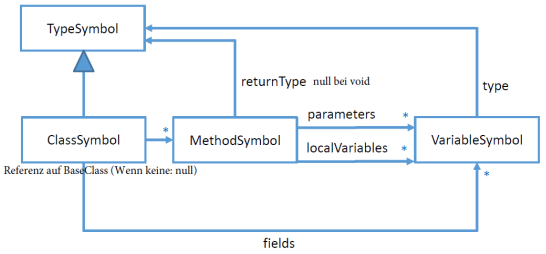
Benötigte Informationen:

- Deklarationen: Variablen, Methoden, Klassen
- Typen:
  - Vordefinierte Typen (int, boolean etc.)
  - Benutzerdefinierte Typen (Klassen)
  - Arrays
  - Typ-Polimorphismus (Vererbung)

5.2 Symboltabelle

Datenstruktur zur Verwaltung der Deklarationen.
Wiederspiegelt hierarchische Bereiche im Programm.
→ Global Scope einführen, um mehrere Klassen zu managen.
Shadowing: Deklarationen in inneren Bereichen verdecken gleichnamige von äusseren Bereichen.
Hiding: Base Klasse und Sub Klasse haben beide die Variable x deklariert. → Sub Klasse muss mit super auf x von Base zugreifen.





5.2.1 Besonderheiten

- Vordefinierte Types: int, boolean, string
  - Als Inbuild Type in Global Scope einfügen
- Vordefinierte Konstanten: true, false, null, this
  - true, false, null als Konstanten in Global Scope
  - null ist Poly-Typ (kompatibel zu allen Referenztypen)
  - this speziell bei Analyse behandeln

- Vordefinierte Methoden: writeString etc.
- Vordefinierte Variablen: length
  - Nur für Array-Typen, ist read-only

5.3 AST verknüpfen

Symboltabelle enthält Mapping Symbol → AST

5.3.1 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 Pattern

Forward-Referenzen: Typ-Namen und Designatoren noch nicht auflösen

5.3.2 2. Typen bei Symbolen auflösen

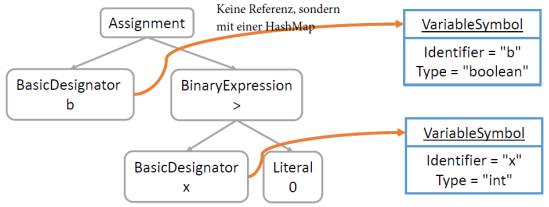
- Für Variablentyp, Parametertyp, Rückgabetyp etc.
- Brauche Suche für Identifier auf Symboltabelle
- Welches Symbol deklariert Identifier "id"?
  - Suche beim innerstem Scope beginnen

Suchfunktion:

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

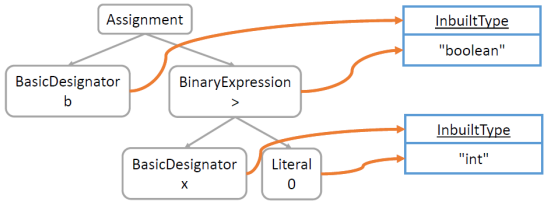
5.3.3 3. Deklarationen in AST auflösen

- Traversiere Ausführungscode in AST (Method Body)
- Jeden Designator auflösen (Deklaration zuordnen)



5.3.4 4. Typen in AST bestimmen

- Typ zu jeder Expression zuordnen
  - Literal: definierter Typ
  - Designator: Typ der Deklaration
  - Unary/Binary Expression: Resultat des Operators



5.4 Semantic Checks

- Alle Designators beziehen sich auf Variablen/Methoden
- Typen stimmen bei Operationen
- Kompatible Typen bei Zuweisung
- Argumentliste passt auf Parameterliste
- Bedingung in if, while sind boolean
- Return-Ausdruck passt
- Keine Mehrfachdeklarationen
- Kein Identifier ist reserviertes Keyword
- Exakt eine main()-Methode
- Array Length ist read-only

```
@Override
public void visit(BinaryExpressionNode node) {
    Visitor.super.visit(node); // post-order traversal
    var leftType = symbolTable.findType(node.getLeft());
    var rightType = symbolTable.findType(node.getRight());
    switch (node.getOperator()) {
        case PLUS -> {
            // error(), falls Type nicht int ist
            checkType(leftType, globalScope.getIntType());
            checkType(rightType, glbalScope.getIntType());
            symbolTable.fixType(node, GlobalScope.INT_TYPE);
        }
    }
}
```

6 Code Generierung

→ **Erzeugung von ausführbarem Maschinencode**
Input: Zwischendarstellung (Symboltabelle + AST)
Output: Maschinencode
Mögliche Zielmaschinen: Reale Maschine, (z.B. Intel 64, ARM Prozessor) oder virtuelle Maschine (z.B. Java VM, .NET CLI)
Kernkonzepte:

- Virtueller Stack-Prozessor (also keine Register)
- Branch Instructions (Goto) für Bedingungen wie if/while
- Metadaten wie z.B. Klassen, Methoden und Variablen die existieren

6.1 Auswertungs-Stack

- Instruktionen benutzen Auswertungs-Stack
- Jede Instruktion hat definierte Anzahl von Pop und Push Aufrufen
- Eigener Stack pro Methodenaufruf (Am Anfang und Ende leer)
- Stack hat unbeschränkte Kapazität

6.1.1 Load/Store Numerierung

- this Referenz: Index 0 (virtuelle Methode)
- Danach n Parameters: 1..n
- Danach m lokale Variablen: Index n+1..n+m

```
// Beispiel Instruktion imul:
pop y
pop x
z = x * y
push z
```

6.2 Metadaten

Werden gebraucht für Fehlermeldungen, Allozieren von Speicher, Vererbung

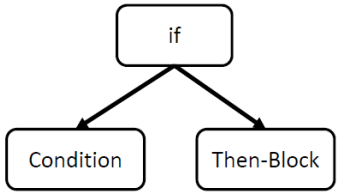
- Zwischensprache kennt alle Informationen zu
  - Klassen (Namen, Typen der Fields, Methoden)
  - Methoden (Namen, Parametertypen, Rückgabetyp)
  - Lokale Variablen (Typen)
- Kein direktes Speicherlayout festgelegt
- Nicht enthalten:
  - Namen von lokalen Variablen und Parameter → Sind nur nummeriert

6.2.1 Code-Generierung

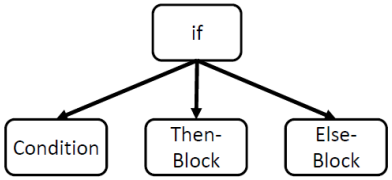
- Traversiere Symboltabelle
  - Erzeuge Bytecode Metadaten
- Traversiere AST pro Methode (Visitor)
  - Erzeuge Instruktionen via Bytecode Assembler
- Serialisiere in Output Format

6.2.2 Traversierungsreihenfolge

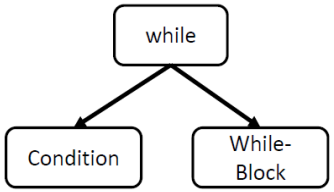
- Bei Expressions: Immer Post-Order
- Bei Statements: Je nach Code-Template
  - Assignment: Rechts zuerst, dann Code Muster
  - If, If-Else, While etc. komplizierter




```
        Condition
        if_false target
        Then-Block
target:
```




```
        Condition
        if_false target0
        Then-Block
        goto target1
target0: Else-Block
target1:
```




```
target0: Condition
    if_false brfalse target1
        While-Block
    goto br target0
target1:
```

```
// While Statement
@Override
public void visit (WhileStatementNode node) {
    var beginLabel = assembler.createLabel();
    var endLabel = assembler.createLabel();
    assembler.setLabel(beginLabel);
    node.getCondition().accept(this);
    assembler.emit(IF_FALSE, endLabel);
    node.getBody().accept(this);
    assembler.emit(GOTO, beginLabel);
    assembler.setLabel(endLabel);
}
```

6.2.3 Short Circuit

```
// a && b
if a then b
else false

// a || b
if !a then b
else true
```

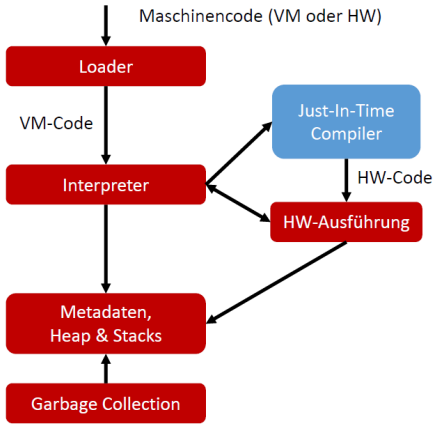
6.3 Methodenaufrufe

- Statisch
  - Vordefinierte Methoden: readInt(), writeInt() etc.
- Sonst immer virtuell (dynamisch)
  - An Objekt gebunden z.B. x.run() oder this.run()

6.3.1 Virtueller Methodenaufruf

1. Argumente von Methode sind auf dem Stack (letztes zuoberst), zuunterst ist Objektreferenz (this)
2. Call-Instruktion
3. Call entfernt Argumente & Objektreferenz und legt Rückgabewert auf Stack (falls nicht void) → Assembler Code ret ist aber auch bei void-Methode nötig

7 Virtual Machine



7.1 Loader

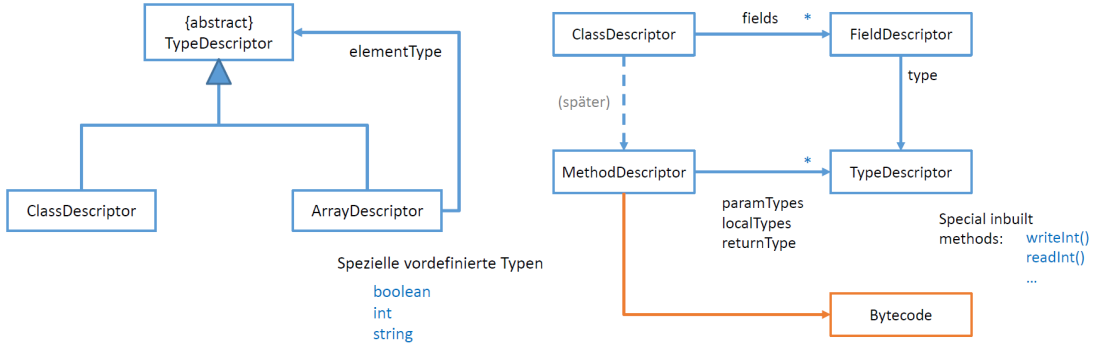
- Lädt Zwischencode (File) in Speicher
- Alloziert Speicher
  - Metadaten für Klassen, Methoden, Variablen, Code
- Definiert Layouts
  - Speicherbereiche für Fields/Variablen/Parameter
- Address Relocation
  - Löst Verweise auf zu Methoden, Typen, anderen Assemblies
- Initiiert Programmausführung
  - Interpreter oder Compilation (JITer)
- Optional: Verifier zum erkennen von falschem IL-Code oder anderen Fehlern (Stack over/underflow, Typefehler, illegaleSprünge etc.)
  - Sonst: Überprüfunfen zur laufzeit (unser Approach)

7.1.1 Deskriptoren

Laufzeitinfo für Typen & Methoden:

- Typen: Klassen, Arrays oder Basistypen
- Klassen: Field-Typen
- Methoden: Typen von Parameter & Locals, Rückgabetyp, Bytecode

Zusätzlich gut zum merken: Parent Klasse, Virtual Method Table



7.1.2 VM: Managed & Unmanaged

Da wir für die VM Java verwenden kriegen wir Managed Runtime Support. Wir wollen aber einen eigenen GC bauen.
→ Kleine Unmanaged Teile neben der Java VM: Heap und HW-Execution (JIT).

7.2 Interpreter

- Interpreter Loop
  - Emuliert Instruktion nach der anderen
- Instruction Pointer (IP)
  - Adresse der nächsten Instruktion
- Evaluation Stack
  - Für virtuellen Stack Prozessor
- Locals & Parameters
  - Für aktive Methode
- Method Descriptor
  - Für aktive Methode

7.2.1 Ausführung

```
// execute() emuliert Instruktion je nach Op-Code
switch(instruction.getOpCode()){
```

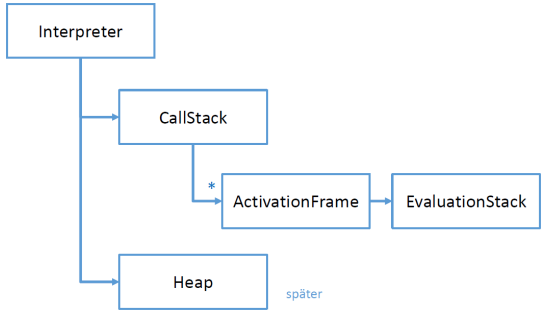
```
    case LDC -> push(instruction.getOperand());
    case IADD -> {
        var right = pop();
        var left = pop();
        var result = left + right;
        push(result);
    }
}
```

7.2.2 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

Call Stack Design:
Managed Call Stack im Interpreter. Unmanaged bei HW-Execution

7.3 Gesamtbild



Verifikation im Interpreter

- Korrekte Benutzung der Instruktionen
  - Typen stimmen (bei Operatoren, Aufrufen etc.)
  - Methodenaufrufe stimmen (Argumente, Rückgabe etc.)
  - Sprünge sind gültig
  - Op-Codes stimmen
- Typen sind bekannt
  - Metadaten (Typen der Fields/Locals/Parameters)
  - Werte auf Evaluation Stack haben Typ

Sicherheitsmassnahmen

- Korrekter Bytecode und Typenkonsistenz prüfen
- Variablen immer initialisieren (auch lokale)
- Checks durchführen (Null, Array-Index etc.)
- Stack Overflow und Underflow Detections
- Kompatibilität von externen Verweisen (hier nicht)
- Garbage Collection

Interpreter vs Kompilation

- Interpreter ist ineffizient
  - Dafür aber flexibler und einfach zu entwickeln
  - Akzeptabel für selten ausgeführten Code
- Kompilierter HW-Prozessor Code ist schneller
  - JIT Compilation für Hot Spots
  - Kompilation kostet, Laufzeit macht es (allenfalls) wett

8 Objekt Orientierung

OO in Lexer & Parser

- new-Operator für Klassen (new Point())
- Indirekte Zugriffsausdrücke (Designators)
- Type Cast und instanceof-Operator

OO in Semantic Checker

- Zuweisungs-Kompatibilität der Typen
  - null passt auf alle Referenztypen
  - Subklasse passt auf Basisklasse (impl. Upcast)
- Zuweisungen/Parameterübergabe/Rückgabe
- Vergleiche zwischen 2 Variablen: Gegenseitig zuweisungskompatibel
- Keine zyklische Vererbung
- Overriding stimmt: Gleiche Signatur/Return value

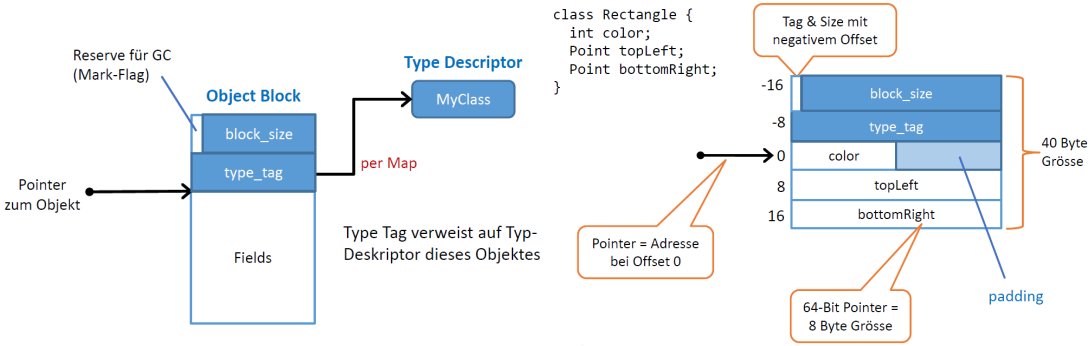
8.1 Heap

Ablage erzeugter Objekte auf dem Heap. → Linearer Adressraum
Warum nicht auf Stack? Würden sonst verloren gehen wenn das Activation Frame abgebaut wird.
→ Nicht mehr an Methodeninkarnation gebunden
Objekt-referenzen:

- Verweis zum Objekt im Heap
  - Repräsentiert durch Speicheradresse des Objektes
- Variablen speichern nur Referenzen
  - Falls vom referenztyp, d.h. Klasse, Array, String

8.1.1 Raw Heap

- 64-Bit Adressen/Pointer
- Raw Heap kann keine Java referenzen speichern
  - Sonst würde der GC im Managed Bereich nicht mehr funktionieren
- Stattdessen Map (z.B. HashMap<Long, Object>) verwenden



8.1.2 Einfache Heap Allokation

```
Pointer allocate(int size, TypeDescriptor type) {
    int blockSize = size + 16 // Mit Header
    if (freePointer + blockSize > limit) {
        throw new VMException("Out of Memory"); // Ohne GC
    }
    long address = freePointer;
    freePointer += blockSize;
    heap.setLong(address, blockSize);
    setTypeDescriptor(type, address); // at offset 8
    address += 16;
    return new Pointer(address);
}
```

9 Typ Polymorphismus

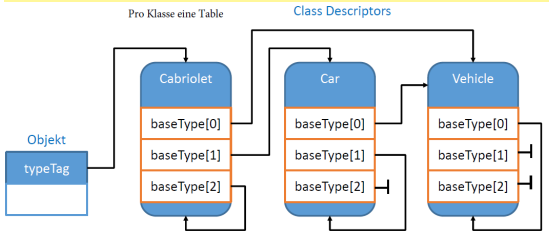
9.1 Vererbung

→ **Zur Einfachheit nur Single Inheritance**
Sonst: Diamond Problem (Wenn in oberster Klasse Instanzvariablen vorhanden wären)
Zwei Aspekte:

- Code Reuse: Subklasse erbt variablen & Methoden der Basisklasse
- Typ-Polimorphismus: Objekt der Subklasse ist auch vom Typ der Basisklasse
  - Dynamischer Typ: Zur laufzeit, Statischer Typ: Deklarierter → Compiler

Type Test & Cast: Entscheiden mit Descriptor → Vererbungshierarchie so lange traversieren, bis es keine Basisklasse mehr gibt, oder man den Typ gefunden hat.

9.2 Ancestor Tables



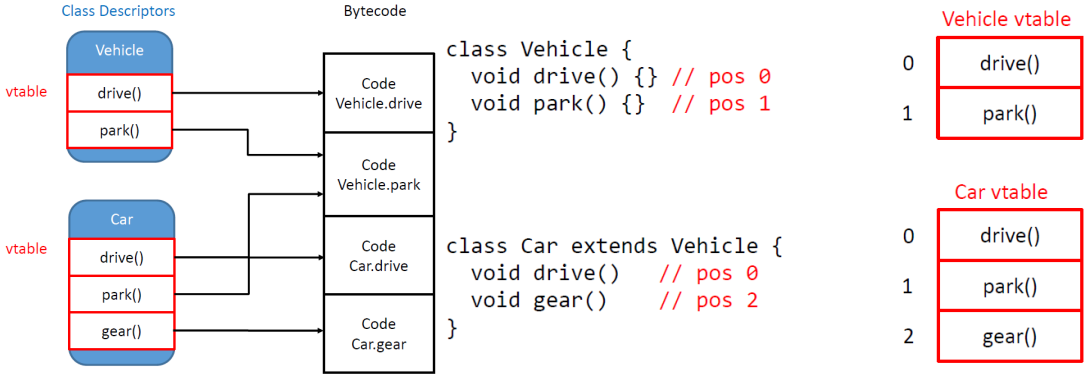
- Konstante Zeit für Type Tests & Casts
  - Alternative: Dynamische Tabelle mit Längenprüfung
- Anzahl Stufen kann begrenzt sein
- Funktioniert nur bei Single-Inheritance

9.3 Casts

```
// instanceof
var instance = checkPointer(pop());
var target = checkClassDescriptor(instruction.Operand);
var desc = heap.getDescriptor(instance);
var level = target.getAncestorLevel();
var table = desc.getAncestorTable();
push(table[level] == target);
// Zusätzlich prüfen:
// - Wenn instance null ist: False
// - Dynamische Länge der table -> Level könnte grösser sein als
    table.length

// checkcast
if (!checkBoolean(pop())) {
    throw new VMException("Invalid cast");
}
push(instance)
// Unterschied zu instanceof:
// - null kann man immer casten !!
// Im null-Fall beachten:
// null in instanceof T -> False
// (T)null -> succeeds
```

9.4 Virtuelle Methoden

Instanzmethoden können überschrieben werden. → Bei virtuellen Methodenaufruf entscheidet der dynamische Typ.



- Methoden von Basisklasse oben, eigene unten
- Position von jeder Methode ist statisch bekannt im deklariertem Typ
- Funktioniert nur bei Single Inheritance
- Loader präpariert die vtable und position

```
// invokevirtual
// pop arguments
var staticMethod = checkMethodDescriptor(instruction.Operand);
var target = checkPointer(pop());
var desc = heap.getDescriptor(target);
int pos = staticMethod.getPosition();
var vtable = desc.getVirtualTable();
var dynamicMethod = vtable[pos];
// call dynamic method
```

9.5 Interfaces

- Interfaces global durchnummerieren
- Pro Class decriptor eine Interface Table (itable)
- Einträge in itable verweisen auf vtable
- Class Descriptor verweist auf itable (ibase-Pointer)

10 Garbage Collection 1

Zweck des Typ-deskriptors bis anhin:

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

Explizite Freigabe:

- Könnte mit delete-Statement gelöst werden (C/C++)
- Probleme:
  - Dangling Pointers: Referenz auf gelöschtes Objekt
  - Memory Leaks: Verwaiste Objekte, die nicht abräumbar sind

10.1 Reference Counting

Idee: Pro eingehende Referenz Counter inkrementieren → Sofortige Deallokation wenn Counter wieder auf 0

Problem: Bei zyklischen Abhängigkeiten wir Objekt nie zu Garbage. Ineffizient

10.2 Transitive Erreichbarkeit

- Objekte beibehalten, die das Programm noch zugreifen könnte
- Alle direkt und indirekt über referenzen vom Programm erreichbaren Objekte
- Ausgehend von Ankerpunkten (Root Set)

Root Set:

- Referenzen in statischen Variablen (soweit benutzt)
- Referenzen in Activation Frames auf Call Stack
- Referenzen in Register (soweit benutzt)

10.3 Mark & Sweep Algorithmus

Mark Phase: Erreichbare Objekte markieren
Sweep Phase: Alle nicht markierten Objekte löschen und Markierungen zurücksetzen
Rekursive Traversierung

- GC braucht zusätzlichen Speicher für Stack
- Problematisch, da Speicher sowieso schon knapp
- Es existieren Algorithmen, für Traversierung ohne zusätzlichem Speicher

10.4 Ausführungszeitpunkt

- Delayed Garbage Collection
  - Garbage wird nicht sofort erkannt und freigegeben
- GC läuft spätestens wenn Heap voll ist
  - Check beim Allozieren
- Eventuel prophylaktisch früher
  - Insbesondere bei Finalizer

Stop & Go:

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

10.5 Root Set Erkennung

Alle Pointers auf Call Stack:

- Pointer in Parameter
- Pointer in lokalen Variablen
- Pointer auf Evaluation Stack
- this-Referenz

10.6 Neue Heap Allozierung

- Traversiere freeList bis zu passendem Block
- Überschuss des Blockes wieder in freeList einreihen

Free List 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 FreeLists mit verschiedenen Grössenklassen

Weitere Möglichkeiten:

- Benachbarte freie Blöcke verschmelzen (Am einfachsten in Sweep Phase)
- Buddy System
  - Diskrete Blockgrössen nach Adresse geordnet
  - Exponentielle Blockgrössen (z.B. 2er Potenz, Fibonacci)
  - Sehr schnelles Verschmelzen & Allozieren & Freigabe
  - Grosse interne Fragmentierung (unbrauchbare Reste)
- Compacting Garbage Collection

10.7 Implementation

```
Iterabley<Pointer> getRootSet(CallStack callStack) {
    var list = new ArrayList<Pointer>();
    for (var frame : callStack) {
        // collectPointers: Fügt vorhandene Pointers der list hinzu
        collectPointers(frame.getParameters(), list);
        collectPointers(frame.getLocals(), list);
        collectPointers(frame.getEvaluationStack().toArray(), list)
            ;
        list.add(frame.getThisReference());
    }
    return list
}

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

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

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

11 Garbage Collection 2

GC braucht Metadaten:

- Pointer Offsets in Objekt
  - Via Typ-Deskriptor
- Pointer Offsets in Stack
  - Via Method Descriptor und Interpreter-Typen
- Pointers in register
  - Bei Interpreter nicht nötig

11.1 Finalizer

- Eine Methode, die vor Löschen des Objekts läuft
  - Abschlussarbeit: Verbindungen schliessen etc.
- Von GC initiiert, wenn Objekt zu Garbage geworden ist

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

- Finalizer dauert evtl. beliebig lange
  - Würde GC blockieren
- Finalizer kann neue Objekte allozieren
  - Korrumpiert GC
- Programmierfehler im Finalizer
  - Evtl. Crash des GC
- Finalizer kann Objekt wieder weiterleben lassen
  - Resurrection

11.1.1 Resourrection

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

11.1.2 Finalizer Internals

Finalizer Set: Registrierte Finalizer

Pending Queue: Noch auszuführende Finalizer

Wenn Objekt Garbage wird: Aus Finalizer Set löschen und in Pending Queue hinzufügen. Erst nach Pending Queue definitiv löschen.

Konsequenzen:

- GC braucht 2 Mark Phasen
  - Markiere und erkenne Garbage mit Finalizer
  - Markiere von Pending Queue erneut, dann Sweep
- Speicher kann evtl. nicht schnell genug frei werden

Programmieraspekte:

- Reihenfolge der Finalizer ist unbestimmt
- Finalizer laufen beliebig verzögert
- Finalizer sind nebenläufig zum Hauptprogramm
  - Seperater Thread oder unbestimmt verzahnt
- Läuft der Finalizer nach resurrection wieder?
  - Nicht in java

11.2 Weak References

- Weak References zählt nicht als referenz für GC
- Garbage kann mit Weak Reference noch so lange erreicht werden, bis es abgeräumt worden ist
- Weak Reference wird nach Freigabe von Garbage Objekt auf null gesetzt
- Weak Reference vor oder nach Finalizer freigeben? → Weak, Soft, Phantom Reference

11.3 Compacting GC

Problem: Grosses Objekt müsste auf Heap alloziert werden, aber passt in keine Lücke (obschon genügend freier Speicher vorhanden wäre)
Lösung:

- Mark & Copy GC
- Allokation am Heap-Ende
- GC schiebt Objekte wieder zusammen
- Beim Verschieben müssen alle Referenzen nachgetragen werden
- Konservative Methode daher unmöglich

11.4 Inkrementeller GC

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

Inkrementelle GCs

- Generational GC
  - Junge Objekte schneller freigeben
  - Junge Objekte: Kurze Lebenserwartung, Alte Objekte: Lange Lebenserwartung
- Partioned GC
  - Heap partitionsweise verwalten
- Weitere Verfahren
  - Concurrent GC, Real-Time GC

12 JIT Compiler

- Effizientere Ausführung als Interpretation
- Bytecode direkt in nativen Prozessor-Code übersetzen und ausführen
- Nicht unbedingt alles, sondern nur Performance-kritische Teile
- z.B. Häufig ausgeführter Code (>= 50 mal)

Profiling:

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

12.1 Intel 64 Architektur

- Instruktionen benutzen Register
- 14 allgemeine register für Ganzzahlen
  - RAX, RBX, RCX, RDX, RSI, RDI, R8..R15

12.2 Arithmetische 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)

12.2.1 IDIV vorbereiten

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

```
// Rechnung: (x - 1) / 2
// x sein in RAX
MOV RBX, 1
SUB RAX, RBX // RAX = x - 1
MOV RBX, 2
CDQ // RDX für IDIV vorbereiten
IDIV RBX // RAX = (x - 1) / 2
// Resultat in RAX
```

12.3 Register Allokation

Passende register für Code-Fragmente bestimmen:

- Registerauswahl Hängt von vorherigen Instruktionen ab
- Durch Instruktion werden register frei oder belegt

12.3.1 Lokale register-Allokation

- Für Ausdrucksauswertung (Evaluation Stack)
- Evaluation Stack Einträge auf Register abbilden → Register Stack

Vorgehen:

- Cross Compiler führt Stack an belegten Register
  - Register Stack entspricht also dem Evaluation Stack
- Pro übersetzte Byte-Code Instruktion wird der Stack nachgeführt

Register Clobbering

- IDIV überschreibt RAX und RDX als Seiteneffekt
- Sichere benötigte Werte von RAX und RDX vor IDIV

12.3.2 Globale Register-Allokation

- Variablen in Register speichern (in bestimmten Abschnitten)
  - Insbesondere Locals und Params
- Deutlich schneller als Speicherzugriffe
- Idee: Aktuelle Allokation merken

12.3.3 Intel Branches

- Bedingte Sprünge basierend auf Condition Code
- Condition Code aus vorherigem Vergleich
  - Zuerst: CMP RAX, RBX und dann JE target

12.4 Implementation

```
switch (opCode) {
    // Load Constant
    case LDC -> {
        var target = acquire();
        var value = (int)instruction.getOperand();
        assembler.MOV_RegImm(target, value);
        push(target);
    }

    // Load
    case LOAD -> {
        X64Register reg;
        var index = (int)instruction.getOperand();
        var nofParams = allocation.parameters.size();
        var nofLocals = allocation.locals.size();
        if (index <= nofParams) {
            reg = allocation.parameters().get(index - 1);
        } else if (index <= nofParams + nofLocals) {
            reg = allocation.locals().get(index - 1 - nofParams);
        }
        push(reg);
    }

    // Store
    case STORE -> {
        var source = pop();
        // Check, ob locals oder params (wie bei load)
        var target = allocation.locals().get(index - 1 - nofParams)
            ;
        assembler.MOV_Regreg(target, source);
        release(source);
    }

    // ISub
    case ISUB -> {
        var subtrahend = pop();
        var minuend = pop();
        var difference = acquire();
        assembler.MOV_RegReg(difference, minuend);
        assembler.MOV_RegReg(difference, subtrahend);
        release(subtrahend);
        release(minuend);
        push(difference);
    }

    // IDiv
    case IDIV -> {
        reserve(RAX);
    reserve(RDX);
    forceStack(1, RAX);
    var divisor = pop();
    pop();
    assembler.CDQ();
    assembler.IDIV(divisor);
    push(RAX);
    release(divisor);
    release(RDX);
    }

    // Any Compare Operation
    case CMPEQ -> {
        var right = pop();
    var left = pop();
    assembler.CMP_RegReg(left, right);
    previous = opCode;
    release(left);
    release(right);
    }

    // Verzweigung (Jump)
    case if_true -> {
        var offset = (int)instruction.getOperand();
        var target = code[position + 1 + offset];
        var label = labels.get(target);
        matchAllocation(label);
        switch(previous) {
            case CMPEQ -> assembler.JE_Rel(label);
            case CMPNE -> assembler.JNE_Rel(label);
            case ICMPLT -> assembler.JL_Rel(label);
            case ICMPLE -> assembler.JLE_Rel(label);
            case ICMPGT -> assembler.JG_Rel(label);
            case ICMPGE -> assembler.JGE_Rel(label);
            default -> throw new AssertionError("Unsupported
                operand before if_true in JIT compiler");
        }
    }
}
```

13 Code Optimierung

Aufgabe der Optimierung:

- Transformation von Intermediate Representation/Maschinencode zu effizienteren Version
- Mögliche Intermediate Representations:
  - AST + Symbol Table
  - Bytecode
  - Anderer Intermediater Code (z.B. Three Address Code)
- Meist eine Serie von Optimierungsschritten

13.1 Optimierte Arithmetik

Multiplikation/Division: Bit Shiften mit der Potenz (mal = links, geteilt = rechts)
Modulo: x % 32 → x & 31

13.2 Algebraische Vereinfachung

Expr / 1 → Expr
Expr * 0 → 0
- - Expr → Expr
1 + 3 → 4

13.3 Loop Invariant Code

```
// Wiederholt ausgewertete Teilausdrücke in temp Variable speichern
while (x < N * M) {
    k = y * M;
    x = x + k;
}
```

```
// Optimiert:
k = y * M;
temp = N * M;
while (x < temp) {
    x = x + k;
}
```

13.4 Common Subexpressions

```
// Wiederholt ausgewertete Teilausdrücke in temp Variable speichern
x = a * b + c
...
y = a * b + d

temp = a * b;
x = temp + c;
...
y = temp + d;
```

13.5 Dead Code

```
a = readInt();
b = a + 1;
writeInt(a);
c = b / 2; // Nicht gebraucht

// Schritt 1
a = readInt();
b = a + 1; // Nicht mehr gebraucht
writeInt(a);

// Schritt 2
a = readInt();
writeInt(a);
```

13.6 Copy Propagation

```
// Idee: Ersetze Variable durch zuletzt zugewiesenen Ausdruck
t = x + y;
u = t
writeInt(u);

// Schritt 1
t = x + y;
u = x + y
writeInt(u);

// Schritt 2
t = x + y;
u = x + y
writeInt(x + y);

// Schritt 3
writeInt(x + y);
```

13.7 Constant Propagation

Auch Constant Folding genannt. → Konstante Ausdrücke ersetzen.

13.8 Partial Redundancy

Optimieren, dass eine Redundante Berechnung nur ein mal pro Pfad ausgeführt wird.

13.9 Erkennung von Optimierungspotential

- Static Single Assignment
- Peephole Optimization
- Dataflow Analysis

13.9.1 Static Single Assignment

- Code-Transformation für einfachere Analyse & Optimierung
- Jedes mal wenn x ein neuer Wert zugewiesen wird, eine neue Variable x2 anlegen
  - Phi Function bei Verzweigungen
- Relativ kompliziert und teuer
- Günstigere Techniken gewünscht wie z.B. Peephole

13.9.2 Peephole Optimization

- Optimierung für sehr kleine Anzahl Instruktionen
- In JIT Compiler für Intermediate Code oder Maschinencode benutzt
- Sliding Window mit z.B. 3 Instruktionen

```
ldc 2
ldc 1
imul      -->  ldc 2
ldc 4          ldc 4    -->  ldc 6
iadd           iadd
```