1 Einführung

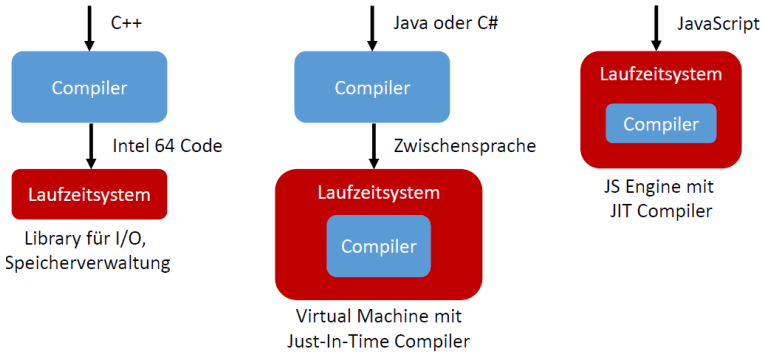
Wieso Compilerbau?

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

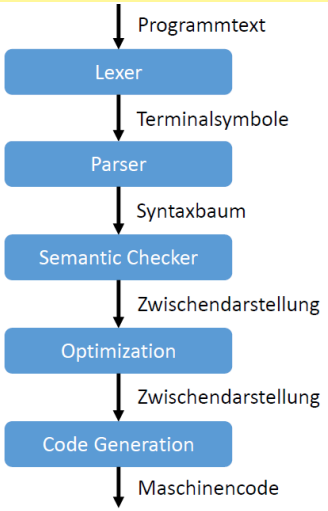
Compiler: Transformiert Quellcode einer Programmiersprache in ausführbaren Maschinencode.

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

1.1 Architekturen

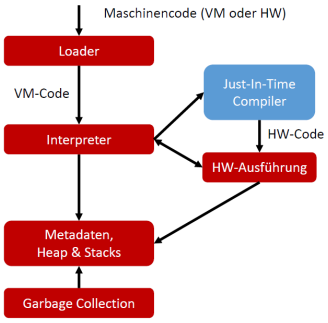


1.2 Aufbau Compiler



- 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

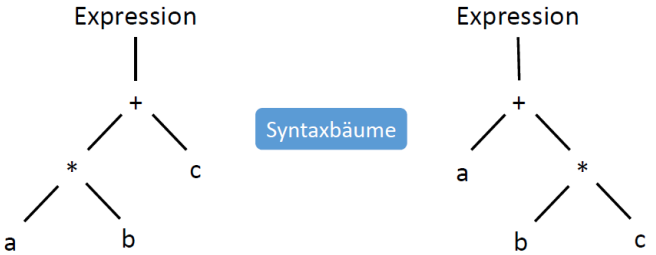
1.4 EBNF (Extended Backus Naur Form)

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

1.4.1 EBNF Konstrukte

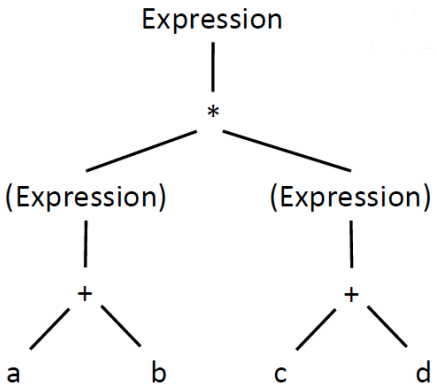
	Beispiel	Sätze
Konkatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" oder "B"
Option	["A"]	leer oder "A"
Wiederholung	{ "A" }	leer, "A", "AA", "AAA", etc.

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 ].
```

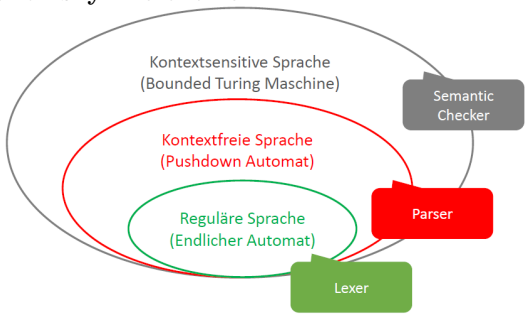
Umformung in äquivalente Syntax

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

keine Rekursionen

Regulär

Chomsky Hierarchie



2.3 Identifier

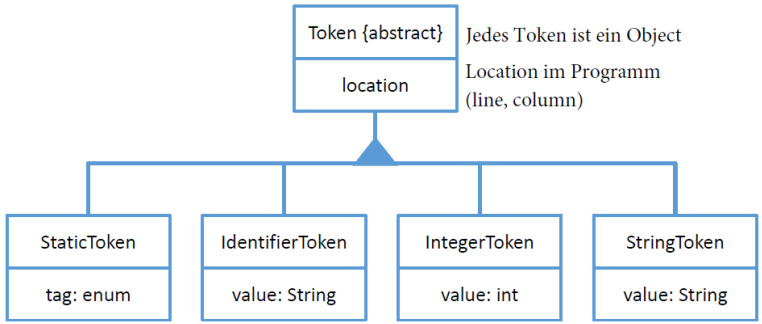
```
Identifier = Letter { Letter | Digit }.
Letter = "A" | ... | "Z" | "a" | ... | "z".
Digit = "0" | ... | "9".
```

- 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

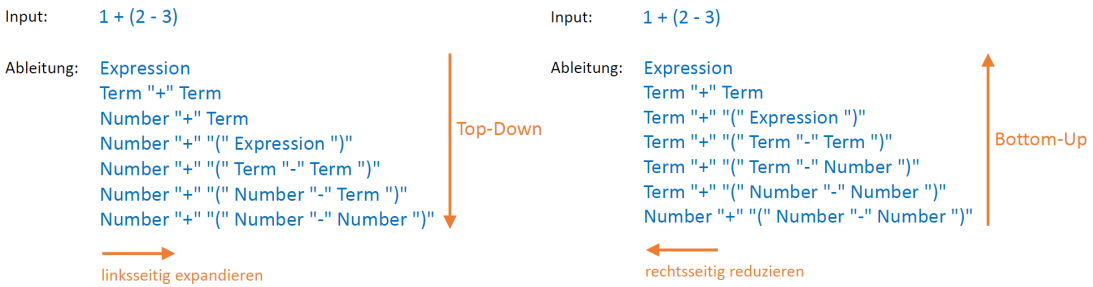
3.2 Parser Strategien

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);
}
```

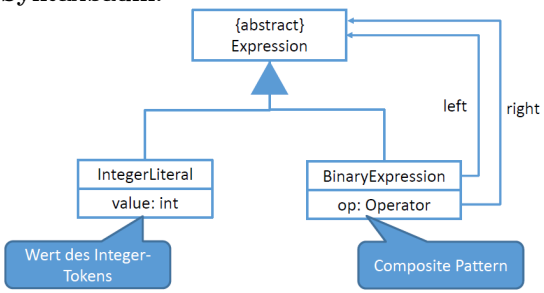
3.4 Parsen mit längerem Lookahead

```
// 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();
    }
}
```

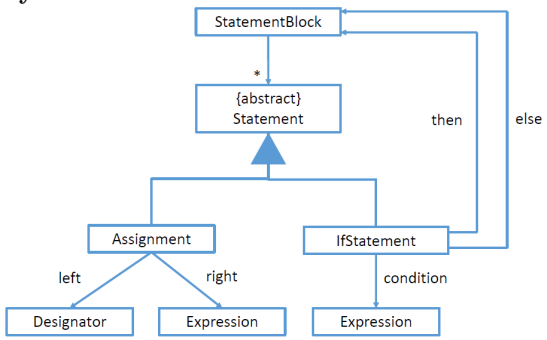
4 Parser Vertiefung

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.

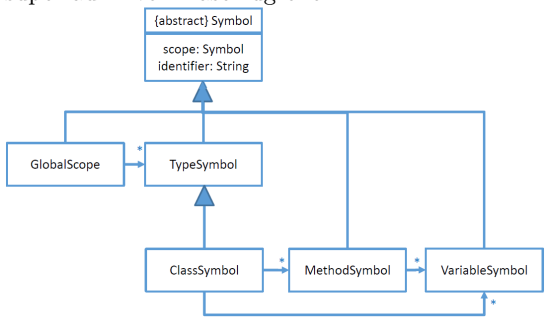
- Deklarationen
  - Jeder Identifier ist eindeutig deklariert
- Typen
  - Typregeln sind erfüllt
- Methodenaufrufe
  - Argumente und Parameter sind kompatibel
- Weitere Regeln
  - z.B. Keine zyklische Vererbung, nur eine main()-Methode

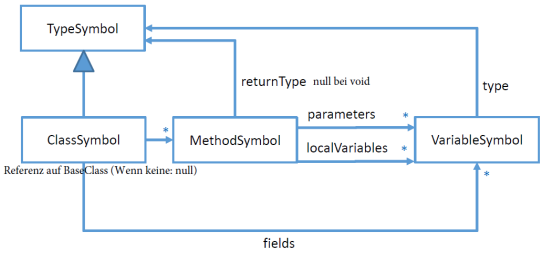
Benötigte Informationen:

- 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

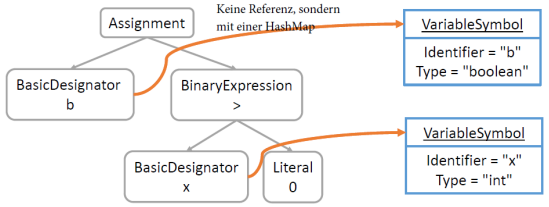
- Für Variablentyp, Parametertyp, Rückgabetyp etc.
- Brauche Suche für Identifier auf Symboltabelle
- Welches Symbol deklariert Identifier "id"?
  - Suche beim innerstem Scope beginnen

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
}
```

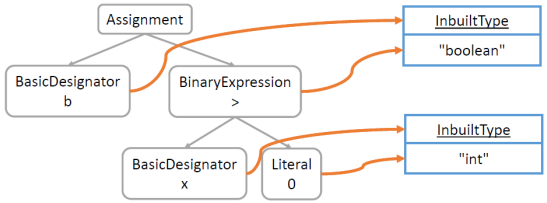
5.3.3 3. Deklarationen in AST auflösen

- 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

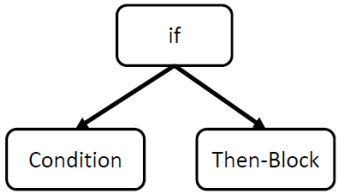
- Zwischensprache kennt alle Informationen zu
  - Klassen (Namen, Typen der Fields, Methoden)
  - Methoden (Namen, Parametertypen, Rückgabetyp)
  - Lokale Variablen (Typen)
- Kein direktes Speicherlayout festgelegt
- Nicht enthalten:
  - Namen von lokalen Variablen und Parameter → Sind nur nummeriert

6.2.1 Code-Generierung

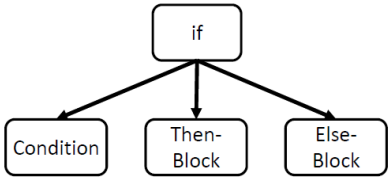
- Traversiere Symboltabelle
  - Erzeuge Bytecode Metadaten
- Traversiere AST pro Methode (Visitor)
  - Erzeuge Instruktionen via Bytecode Assembler
- Serialisiere in Output Format

6.2.2 Traversierungsreihenfolge

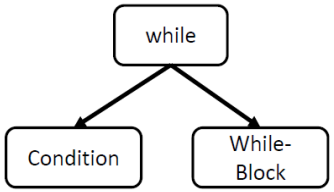
- Bei Expressions: Immer Post-Order
- Bei Statements: Je nach Code-Template
  - Assignment: Rechts zuerst, dann Code Muster
  - If, If-Else, While etc. komplizierter




```
        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
```

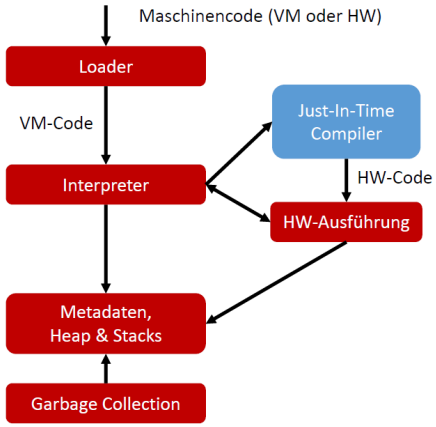
6.3 Methodenaufrufe

- 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

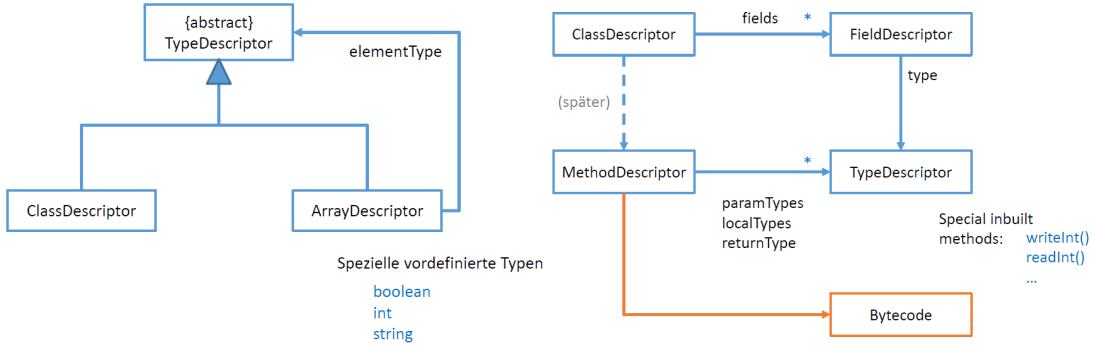
- Lädt Zwischencode (File) in Speicher
- Alloziert Speicher
  - Metadaten für Klassen, Methoden, Variablen, Code
- Definiert Layouts
  - Speicherbereiche für Fields/Variablen/Parameter
- Address Relocation
  - Löst Verweise auf zu Methoden, Typen, anderen Assemblies
- Initiiert Programmausführung
  -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

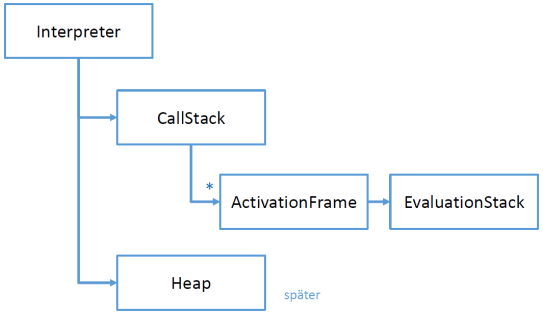
```
    case LDC -> push(instruction.getOperand());
    case IADD -> {
        var right = pop();
        var left = pop();
        var result = left + right;
        push(result);
    }
}
```

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

9 Typ Polymorphismus

10 Garbage Collection 1

11 Garbage Collection 2

12 JIT Compiler

13 Code Optimierung