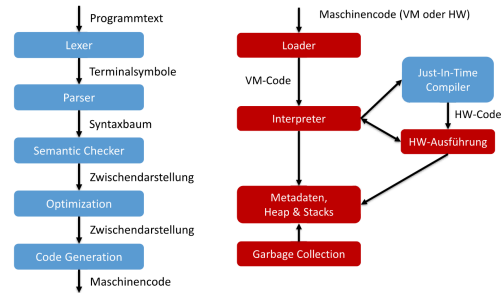


## 1. LAUFZEITSYSTEME

Source Code → Compiler → Maschinencode → Laufzeitsystem



**Syntax:** Struktur des Programms

**Semantik:** Bedeutung des Programms

## 2. EBNF-SYNTAX

Kann **kontextfreie Grammatiken** (Extended Backus-Naur Form) darstellen.

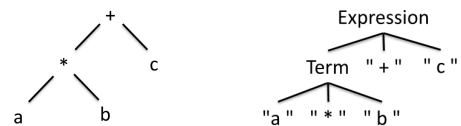
Begriff	Beispiel	Sätze
Konkatenation	"A" "B"	«AB»
Alternative	"A"   "B"	«A» oder «B»
Option	["A"]	∅ oder «A»
Wiederholung	{"A"}	∅, "A", "AA", ...

**Beispiel:**  $a * b + c$

**Expression** = Term | Expression "+" Term.

**Term** = Variable | Term "\*" Variable.

**Variable** = "a" | "b" | "c" | "d".

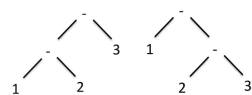


Darf nicht **mehrdeutig** sein:

**Expression** = Number | Expression "-"

**Expression.**

**Number** = "1" | "2" | "3".



**Besser:** **Expression** = Number { "-" Number }.

## 3. LEXIKALISCHE ANALYSE

**Input:** Zeichenfolge, **Output:** Folge von Tokensymbolen (*Tokens*).

Kann **Reguläre Sprachen** analysieren. (Regulär = hat rekursionsfreien EBNF oder kann in rekursionsfreien EBNF umgewandelt werden)

Eliminiert **Whitespaces** und **Kommentare**, merkt **Positionen** im Code.

**Maximum Munch:** Lexer ist greedy.

**Vorteile:** Abstraktion (Parser muss sich nicht um Textzeichen kümmern), Einfachheit (Parser hat nur noch Lookahead pro Symbol), Effizienz (benötigt keinen Stack).

Hat **one-character-lookahead**, um Typ immer bestimmen zu können.

**Tokens:** **Fixe / Static Token (1)** (Keywords, Operatoren, Interpunkt.), **Identifiers (2)** MyClass, **Integer (3)** 123, **Strings (4)** "Hello", **Characters (5)** 'a'

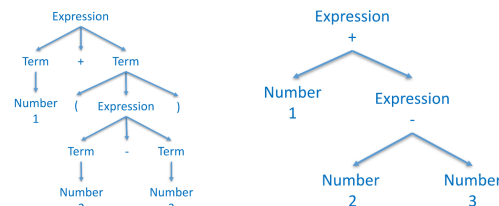
**while** (i < 100) {x = x + 1; }

"while" (1), "(" (1), "x" (2), "<" (1), 100 (3), ")" (1), "{" (1), "=" (1), "+" (1), "1" (3), ";" (1), "}" (1)

**Fehler:** Invalid Symbol, Unclosed stuff, overflow

## 4. REC. DESCENT PARSER

**Input:** Tokens, **Output:** Syntaxbaum  
Parser erkennt, ob Eingabetext den **Syntax** erfüllt. Funktioniert mit **kontextfreien Sprachen** (als EBNF ausdrückbar + Stack). **Concrete Syntax Tree** (Parse Tree): Vollständige Ableitung, erleichtert Parsen. **Abstract Syntax Tree:** Minimale Ableitung.



## Parser-Klassen

- **L** für von links, **R** für von rechts
- **L** für Top down, **R** für Bottom up
- **Zahl** für Anzahl Token Lookahead

Input: 1 + (2 - 3)

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

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

Input: 1 + (2 - 3)

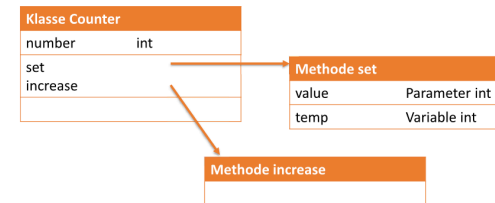
LR-Parser ist mächtiger als LL-Parser, kann Linksrekursion behandeln.  $E = [E] "x"$

## 5. SEMANTISCHE ANALYSE

Syntaktisch ✓ ≠ Semantisch ✓

**Input:** Syntaxbaum, **Output:** Symboltabelle. Prüft, ob das Programm korrekt ist, **kontextsensitive Grammatik** (Designators aufgelöst, alles deklariert, Typregeln erfüllt, Argumente und Parameter kompatibel, keine zyklische Vererbung, nur eine main Methode, ...).

**Symboltabelle:** Datenstruktur zur Verwaltung von Deklarationen.



```
class Counter { int number;  
void set(int value) { int temp;  
temp = number; number = value; }  
void increase()  
{ number = number + 1; } }
```

**Shadowing:** Deklaration innen verdecken gleichnamige äussere Scopes.

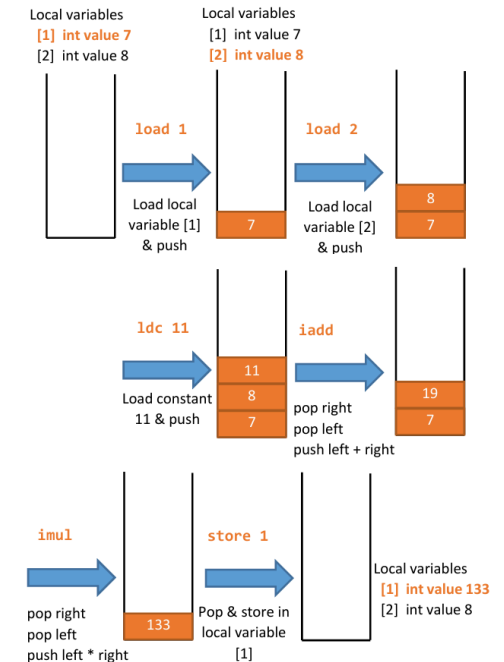
**Global Scope:** Enthält vordefinierte Typen, Konstanten, this, Built-in Methoden und array.length

## 6. CODE-GENERIERUNG

**Input:** Zwischendarstellung, **Output:** Ausführbarer Maschinencode

**Visitor Pattern:** Traversieren des AST pro Methode

```
while (x < 10) { x = x + 1; }  
begin: load 1  
ldc 10  
icmplt  
if_false end  
load 1  
ldc 1  
iadd  
store 1  
goto begin  
end: ...
```



1. **this-Referenz:** Index 0
2. **n Params:** Index 1...n
3. **m lokale Variablen:** Index n + 1...n + m

## 7. CODE-OPTIMIERUNG

**Arithmetik** (Zweierpotenzen in Bit-Operation umwandeln), **Algebraisch** (Redundante Operatoren entfernen, konstante Literale zusammenfassen), **Loop-Invariant Code Motion** (Unveränderter Code aus Schleife nehmen), **Common Subexpressions** (Wiederholt ausgewertete Teilausdrücke zusammenfassen), **Dead Code Elimination** (Nicht Verwendetes entfernen), **Copy Propagation** (redundante Load und Stores entfernen), **Constant Propagation** (konstante Variablen durch Konstante ersetzen), **Partial Redundancy Elimination** (Expressions in Pfaden so wenig wie möglich evaluieren)

**Static Single Assignment (SSA):** Variablen werden umbenannt, damit jede nur ein einziges Mal zugewiesen wird (Veränderungen schnell erkennbar).

Bei Verzweigungen:  $\varphi(x_1, x_2)$

**Peephole Optimization:** Sliding Window, Optimierungen werden auf diesen kleinen Bereich vorgenommen.

## 8. VIRTUAL MACHINE

Nützlich für **Mehrplattformensupport**.

**Loader:** **Liest Assembly** und **alloziert** notwendige **Laufzeitstrukturen**. Kreiert Metadaten, initiiert Programmausführung.

**Interpreter:** Arbeitet mit einem **Call Stack**, der aus **Activation Frames** besteht. Jeder **Activation Frame** verwaltet einen **Evaluation Stack**.

**Instruction Pointer:** Adresse der nächsten Instruktion  
(springt bei Branches  $\langle \text{Zahl} \rangle + 1$ )

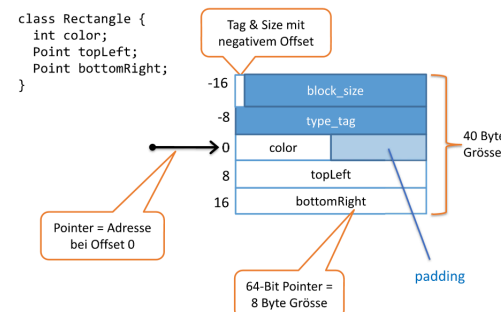
**Evaluation Stack:** Stack der Methode. (item = register)

**Call Stack:** Stack der **Methodenaufrufe**. Verwaltet lokale Variablen und Rücksprungadresse (item = activation frame) **Managed** (mit Klassen modelliert) **Unmanaged** (Funktioniert mit Stack Pointer und Base Pointer)

## 9. OBJEKT-ORIENTIERUNG

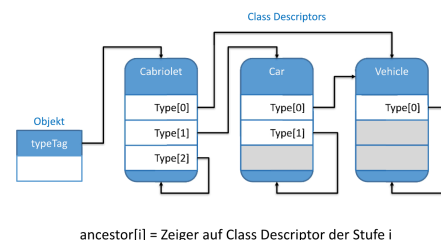
**Heap:** Objekte im Laufzeitsystem werden im Heap gespeichert (Können nicht auf Stack gespeichert werden wegen nicht-hierarchischer Lifetime Dependency). Ist ein linearer Adressraum. **Unabhängig** vom Call Stack. Objekte werden immer durch eine **Referenz** verwiesen.

**Objektblock:** **Mark Flag** (Für GC), **Block Size** (Gesamtgrösse des Blocks), **Type Tag** (Referenz zum Class Descriptor), **Fields** (Inhalt des Blocks)



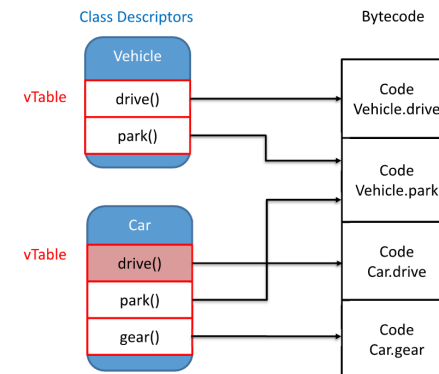
**Typ-Polymorphismus:** **Subklasse** erbt von und erweitert **Basisklasse**. **Downcasts** müssen dynamisch überprüft werden.

**Ancestor Tables:** Jeder Class Descriptor hat eine (Nur bei Single Inheritance).



**Virtuelle Methoden:** Methoden können **überschrieben** werden, Inhalt der bestehenden Methode wird ersetzt. **Virtual Method Table:** Jeder **Klassendeskriptor** hat eine **vTable** mit den Methoden (zu oberst von Basisklasse, bei Overriding wird nicht ersetzt, sondern ergänzt).

**Typdeskriptoren:** Werden vom Loader generiert. Nützlich für Type Checking, Ancestor Table, vTables (im Bild).



**iTable:** Global durchnummeriert, jede Methode hat in ihrer iTable an der Stelle Interfaces, wo sie sich auch in der globalen Tabelle befinden. Die Einträge verweisen auf vTables.

## 10. GARBAGE COLLECTION

**Dangling Pointer:** Referenz auf bereits gelöscht Element

**Memory Leak:** Verwaiste Objekte

**Garbage:** Nicht mehr verwendete und erreichbare Objekte

**Reference Counting:** Counter pro Objekt mit eingehenden Referenzen (Problematisch bei Zyklen).

**Ablauf: Mark Phase** (Ausgehend vom Root Set (Call Stack) werden alle erreichbaren Objekte markiert)

**Sweep Phase** (Alle nicht markierten Objekte werden gelöscht)

**Free List:** Liste der freien Blöcke, wird bei Allokierung traversiert. Nebeneinanderliegende werden wieder verschmolzen.

**Stop & Go:** GC läuft **sequenziell** und **exklusiv**. Mutator muss warten. Mark-Phase hätte Probleme bei Parallelität wegen Heap-Veränderungen, Sweep-Phase würde aber funktionieren.

**Compacting GC / Moving GC:** Schiebt Objekte im Heap wieder zusammen. Der freie Speicher befindet sich zu hinterst im Heap. Referenzen müssen **nachgetragen** werden. **Mark-Phase** gleich wie bei Stop & Go, **Sweep-Phase** hätte dann der Mutator Zugriff auf verschobene Adressen. (Vorteile: Eliminiert External Fragmentation, schnelle Speicherallozierung)

## 11. JIT COMPILER

**Profiling:** Ausführung von Code-Teilen zählen, um **Hot Spots** (oft ausgeführter Code) zu erkennen.

**Intel 64 Architektur:** Instruktionen benutzen **Register** (RSP: Stack Pointer, RBP: Base Pointer, RIP: Instruction Pointer)

```
load 1 // x laden
ldc 1 // 1 laden
isub // x - 1
ldc 3 // 3 laden
idiv // RAX = (x - 1) / 3
# x64 Code: x sei in RAX
MOV RBX, 1 # 1 laden
SUB RAX, RBX # RAX = x - 1
MOV RBX, 3 # 3 laden
CDQ // RDX vorbereiten
IDIV RBX # RAX = (x-1)/3
# Resultat ist in RAX
```

**Lokale Register-Allokation:** Jeder Eintrag des Evaluation Stack wird auf ein Register abgebildet.

**Globale Register-Allokation:** Zusätzlich auch lokale Variablen und Parameter.