Compilerbau | ComBau

Zusammenfassung

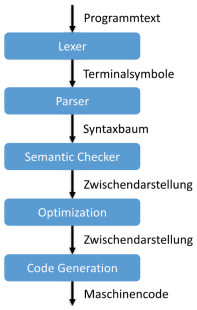
INHALTSVERZEICHNIS

1. LAUFZEITSYSTEME

- ***Compiler:*** Transformiert Quellcode in Maschinencode.
- ***Runtime System:*** Unterstützt die Programmausführung mit Software- und Hardware-Mechanismen. Kann prozessor-nativ *(C++)*, VM-kompiliert *(Java, C++)* oder interpretiert sein *(Python, JavaScript)*.

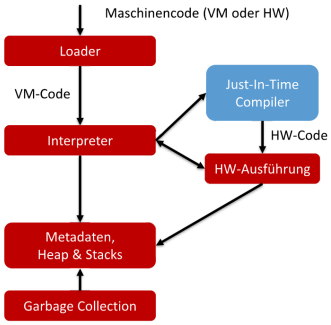
Source Code → Compiler → Maschinencode → Laufzeitsystem

1.1. AUFBAU COMPILER



- ***Lexer:*** Zerlegt Programmtext in Terminalsymbole *(Tokens)*
- ***Parser:*** Erzeugt Syntaxbaum gemäss Programmstruktur
- ***Semantic Checker:*** Löst Symbole auf, prüft Typen und semantische Regeln
- ***Optimization:*** Wandelt Zwischendarstellung in effizientere Darstellung um *(Optional)*
- ***Code Generation:*** Erzeugt ausführbaren Maschinencode
- ***Zwischendarstellung:*** Beschreibt Programm als Datenstruktur

1.2. 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:*** Lässt Instruktionscode direkt auf Prozessor laufen
- ***Metadaten, Heap & Stacks:*** Verwaltung von Programminfos, Objekten und Prozeduraufrufen
- ***Garbage Collection:*** Automatische Freigabe von nicht erreichbaren Objekten

1.3. DEFINITION EINER PROGRAMMIERSPRACHE

- ***Syntax:*** definiert Struktur des Programms *(Bewährte Formalismen für Syntax)*
- ***Semantik:*** definiert Bedeutung des Programms *(Meist in Prosa beschrieben)*

Eine Sprache ist die ***Menge von Folgen von Terminalsymbolen***, die mit der Syntax herleitbar sind.
Die ***Syntax*** einer Sprache ist formal definiert durch:

- ***Menge von Terminalsymbolen*** *(Können nicht weiter ersetzt werden: `"1"`, `"Hallo"`)*
- ***Menge von Nicht-Terminalsymbolen*** *(Können in der Syntax weiter ersetzt werden: `Term`, `Expression`)*
- ***Menge von Produktionen*** *(Definition einer Regel: `Expression = Term`)*
- ***Startsymbol*** *(In ComBau nicht weiter relevant)*

1.3.1. Formalismus zur Syntax-Definition

Beschreibung der Syntax mittels Regeln/Formeln:

```
Language = Subject Verb
Subject = "Anna" | "Paul"
Verb = "talks" | "listens"
```

Mit dieser Sprache können folgende Sätze gebildet werden: «Anna talks», «Anna listens», «Paul talks» und «Paul listens».

Rekursion mit vereinfachter EBNF

`Expression = "(" ")" | "(" Expression ")"` → «()», «(())», «((()))», usw.

2. EBNF-SYNTAX

Die EBNF-Syntax *(Extended Backus-Naur Form)* kann ***kontextfreie Grammatiken*** darstellen.

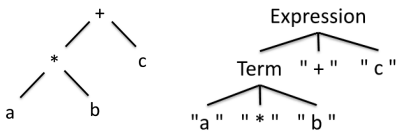
Begriff	***Beispiel***	***Sätze***
Konkatenation	`"A" "B"`	«`AB`»
Alternative	`"A" | "B"`	«`A`» oder «`B`»
Option	`["A"]`	leer oder «`A`» *(gleich wie `"A" | ""`)*
Wiederholung	`{"A"}`	leer, «`A`», «`AA`», «`AAA`», etc.

Für stärkere Bindung können runde Klammern verwendet werden. Ein « | » bindet schwächer als andere Konstrukte.

Mit der EBNF-Syntax kann das Rekursionsbeispiel von oben anders dargestellt werden.
Aus `Expression = "(" ")" | "(" Expression ")"` wird so `Expression = "(" [Expression] ")"`.

2.1. BEISPIEL $a * b + c$

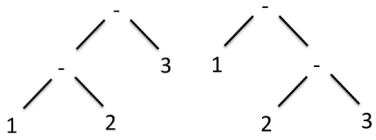
```
Expression = Term | Expression "+" Term. (* bindet stärker als +)
Term = Variable | Term "*" Variable.
Variable = "a" | "b" | "c" | "d".
```



2.2. MEHRDEUTIGKEIT

Der Syntax des EBNF darf nicht ***mehrdeutig*** sein.
Problematisches Beispiel:

```
Expression = Number | Expression "-" Expression.
Number = "1" | "2" | "3".
```



Mit der Formel $1 - 2 - 3$ können aufgrund der Rekursion von `Expression` verschiedene Syntaxbäume gebildet werden, was die Syntax ***unbrauchbar*** macht. ***Behoben*** werden kann dies durch die ***Erzwingung der Linksassozivität***:

```
Expression = Term { "-" Term }.
Term = Number.
```

2.3. SPEZIALFÄLLE

- `"'"` ist ein Anführungszeichen (") als Terminalsymbol
- ***`"A" | .. | "Z"`:*** ist ein Zeichen zwischen A und Z
- ***Whitespaces:*** Werden nicht in der Syntax erfasst, ausser wenn relevant *(z.B. in Python)*
- ***Kommentare:*** Werden nicht in der Syntax erfasst

3. LEXIKALISCHE ANALYSE

Der Lexer kümmert sich um die ***lexikalische Analyse***. Der Input ist eine ***Zeichenfolge*** *(Programmtext)*, der Output eine ***Folge von Terminalsymbolen*** *(Tokens)*.

Der Lexer ***fasst Textzeichen zu Tokens zusammen*** *(Bsp: «1234» ergibt Integer 1234)*, ***eliminiert Whitespaces*** und ***Kommentare*** und ***merkt Positionen*** im Programmcode *(für Fehlermeldungen)*.
Er erleichtert die spätere syntaktische Analyse:

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

3.1. TOKENS

- ***Fixe Tokens:*** Keywords, Operatoren, Interpunktion *(`if`, `else`, `while`, `*`, `&&`, …)*
- ***Identifiers:*** `MyClass`, `readFile`, `name2`, etc.
- ***Zahlen:*** `123`, `0xfe12`, `1.2e-3`, etc.
- ***Strings:*** `"Hello!"`, `01234`, `\n`, etc.
- ***Characters:*** `'a'`, `'0'`, etc. *(nicht in SmallJ)*

3.2. REGULÄRE SPRACHEN

Lexer unterstützen nur ***reguläre*** Sprachen. Reguläre Sprachen sind Sprachen, deren EBNF ***ohne Rekursion*** ausdrückbar sind. *(Eine rekursive EBNF kann auch eine reguläre Sprache sein, solange sie auch ohne Rekursion dargestellt werden könnte)*.

Beispiele Regulär:

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

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

Kann zu obigem Beispiel umformuliert werden, ist also auch ohne Rekursion lösbar.

Beispiele Nicht-Regulär:

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

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

Muss sich merken, wie viele offene Klammern vorhanden sind, um gleich viele schliessende zu haben.

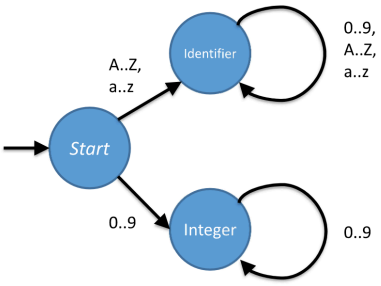
3.2.1. Theoretische Einordnung

Reguläre Sprache *(Endlicher Automat, für Lexer)* << ***Kontextfreie Sprache*** *(Pushdown Automat mit Gedächtnis, für Parser)* << ***Kontextsensitive Sprache*** *(Bounded Turing Maschine, für Semantic Checker)*

3.3. IDENTIFIER

Bezeichner von Klassen, Methoden, Variablen etc. Beginnt mit einem Buchstaben, danach sind Buchstaben und Ziffern erlaubt *(Java erlaubt auch Underscores)*.

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



3.4. MAXIMUM MUNCH

Der Lexer absorbiert ***so viele Zeichen wie möglich*** in einem Token, er ist also ***greedy***. *(`my1234Name` wird als einziges Token gelesen und nicht als drei einzelne Token)*.

3.5. WHITESPACES UND KOMMENTARE

Werden vom Lexer ***übersprungen***. Ein Whitespace ***trennt*** jedoch zwei Tokens. Die Trennung erfolgt aber manchmal auch ***ohne Whitespace***. *(z.B `1234name` wird in Integer und Identifier getrennt)*

3.6. IMPLEMENTATION

3.6.1. Token-Repräsentation

Eine abstrakte `Token` Basisklasse beschreibt die ***Position*** des Tokens im Sourcecode. Die Subklassen implementieren die einzelnen Tokentypen: ***`IntegerToken`***, ***`IdentifierToken`*** und ***`ValueToken`*** enthalten den entsprechenden Wert, während ***`FixToken`*** einen ***`TokenTag`*** enthält. Dies ist ein Enum, welches alle Keywords, Operatoren und Interpunktion enthält, aber keine Typen und Konstanten *(`true`, `false`, `null`)*.

Jeder Token-Typ hat einen ***eigenen Lexer***, der Input wird charakterweise gelesen. Um den Typ zuzuweisen, reicht der aktuelle Charakter jedoch nicht immer aus. Der Lexer hat darum einen ***one character lookahead***, mit welchem der momentane und der nächste Charakter gelesen wird, um den Typ zu bestimmen.

Zuerst werden alle Whitespaces entfernt. Bei den ***Operatoren und Interpunktionen*** kann mit einem Switch-Case das entsprechende `FixToken` ausgegeben werden.
Ist der Charakter eine ***Ziffer***, wird ein Integer-Token erstellt. Dazu wird die Ziffer in einen `char` gecastet, was im ASCII-Wert resultiert; minus den ASCII-Wert von `0` gerechnet ergibt den Ziffernwert. Die momentane Zahl wird mal 10 gerechnet und die Ziffer hinten angehängt. Ebenfalls muss noch geprüft werden, ob der Integer gültig ist ($\texttt{MIN\_VALUE} \leq i \leq \texttt{MAX\_VALUE}$).

```
// IntegerTokenLexer
int value = 0;
while (!IsEnd() && IsDigit()) {
  // Char cast to int → ASCII value
  int digit = Current - '0';
  value = value * 10 + digit;
  readNext();
}
token = new IntegerToken(...,
value);

// NameTokenLexer
string name = Current.ToString();
Next();
while (!IsEnd()
  && (IsLetter() || IsDigit())) {
    name += Current;
    Next();
}
token = Keywords.TryGetValue(name,
out var tag)
  ? new FixToken(..., tag)
  : new IdentifierToken(..., name)
```

Bei einem ***Buchstaben*** kommt entweder ein Identifier oder ein Keyword in Frage. Dazu wird nach dem Erreichen des ersten nicht-alphanumerischen Zeichen geprüft, ob sich der Name im `TokenTag` Enum befindet.

Trifft der Lexer auf einen `/`, wird geprüft, ob darauf ein `/` oder `*` folgt: Dann sind die nachfolgenden Zeichen ***Kommentare*** und werden bis zur nächsten Newline *(Zeilenkommentar)* oder `*/` *(Blockkommentar)* ignoriert. Folgt keines dieser Zeichen, wird ein ***Divisions-`FixToken`*** ausgegeben.

In anderen Sprachen gibt es ***Spezialfälle***, welche separat behandelt werden müssen: `char`-Literale, String/Char-Escaping, Hexadezimale Integers und Kommazahlen. Ebenfalls wird Error Handling für nicht geschlossene Blöcke *(Kommentare, Loops, Strings)*, zu grosse/kleine Zahlen und ungültige Zeichen implementiert. Diese Fehlermeldungen sollen dem Nutzer mit der entsprechenden Zeilenposition angegeben werden. Speziell soll im Lexer bei fehlerhaftem Input ***kein Infinite Loop*** ausgelöst werden.

```
// SlashTokenLexer
case '/': SkipLineComment();
case '*': SkipBlockComment();
default:
  token = new FixToken(DIVISION);
void SkipLineComment() {
  CheckNext('/');
  while (!IsEnd && Current != '\n')
  { next(); }
}

void SkipBlockComment() {
  do {
    Next();
    while (!IsEnd && Current != '*')
      { Next(); }
    Next(); // look for / after *
  } while (!IsEnd && Current != '/')
}
```

3.7. LEXER-GENERATOREN

Verschiedene Tools können Lexer aus EBNF oder Regex ***automatisch erstellen***. Diese ***ersparen Programmierarbeit*** und Flüchtigkeitsfehler, generieren aber ***verbosen Code***, haben ***feste Token-Representation*** und ***erstellen Abhängigkeiten***.

4. RECURSIVE DESCENT PARSER

Der ***Parser*** kümmert sich um die syntaktische Analyse. Der Input ist eine ***Reihe von Terminalsymbolen*** des Lexers und der Output ein ***Syntaxbaum***.

4.1. KONTEXTFREIE SPRACHE

Der Parser kann mit ***kontextfreien*** Sprachen umgehen. Eine ***kontextfreie*** Sprache ist als beliebige EBNF ausdrückbar und hat einen Push-Down Automat (Stack). Viele Aspekte sind aber ***kontextabhängig***, z.b. ***Variablen*** sind vor Gebrauch deklariert, ***Booleans*** sind nicht addierbar, ***Argumente*** müssen auf Parameter passen, etc. Die ***Kontextabhängigkeit*** wird später im ***semantic Checker*** geprüft.

4.1.1. Beispiel

Syntax

```
Expression = Term { ( "+" | "-" ) Term }.
Term = Number | "(" Expression ")".
Number = "0" | ... | "9".
```

Input

```
1 + ( 2 - 3 )
```

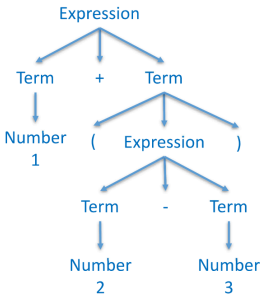
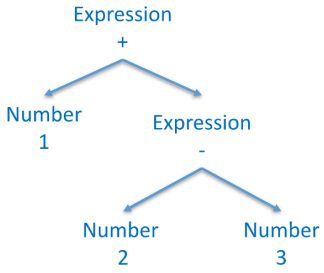
Ableitung

```
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Expression "-" Term ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"
```

4.2. AUFGABE DES PARSERS

Der Parser ***analysiert*** die gesamte Syntaxdefinition *(mit oder ohne rekursive Regeln)*. Er erkennt, ob der Eingabetext ***die Syntax erfüllt*** oder nicht. Hier ist eine ***eindeutige Ableitung*** gewünscht, weil es sonst in der Syntaxdefinition ein Problem gibt. Er ***erzeugt*** den Syntaxbaum für weitere Compiler-Schritte.

Die ***Ableitung der Syntaxregeln*** wird als Baum widerspiegelt.

Concrete Syntax Tree (Parse Tree)	***Abstract Syntax Tree***
Vollständige Ableitung der Syntaxregeln, welche alle Grammatik-Regeln der Sprache widerspiegelt. Parser-Generatoren können nur Parse-Trees liefern.	***Unwichtige Details*** werden ***weggelassen***, Struktur wird ***vereinfacht*** und für Weiterverarbeitung optimiert. Selbst implementierter Parser kann direkt Abstract Syntax Tree liefern.
	

4.3. PARSING-STRATEGIEN

4.3.1. Parser-Klassen nach D.E. Knuth

Parser können je nach ihrer Arbeitsweise in ***Klassen*** eingeteilt werden. Die Notation besteht aus zwei Buchstaben und einer Zahl.

- ***Erster Buchstabe:*** `L` für links nach rechts und `R` für von rechts nach links
- ***Zweiter Buchstabe:*** `L` für Top-Down Parser, `R` für Bottom-Up Parser
- ***Zahl:*** für Anzahl Symbol Lookahead

Beispiel: `LL(1)`: links nach rechts, Top-Down, ein Token lookahead

Top-Down (LL)	***Bottom-Up (LR)***
– Beginne mit ***Start-Symbol*** – Wende ***Produktionen*** an – ***Expandiere*** Start-Symbol auf Eingabetext `Expr → Term + Term → ... → 1 + ( 2 - 3 )`	– 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 Number "+" Term Number "+" "(" Expression ")" Number "+" "(" Term "-" Term ")" Number "+" "(" Number "-" Term ")" Number "+" "(" Number "-" Number ")" Top-Down linksseitig expandieren	Ableitung : Expression Term "+" Term Term "+" "(" Expression ")" Term "+" "(" Term "-" Term ")" Term "+" "(" Term "-" Number ")" Term "+" "(" Number "-" Number ")" Number "+" "(" Number "-" Number ")" Bottom-Up Input: 1 + (2 - 3) rechtsseitig reduzieren

4.4. TOP-DOWN PARSING

In einem Recursive Descent Parser hat eine Expression weitere Terme, ein Term hat allenfalls eine weitere Expression → ***Rekursive Definition***. Um dies abzubilden, muss beim Parsen ebenfalls ***Rekursion*** verwendet werden.

Pro Nicht-Terminalsymbol wird ***eine Methode*** implementiert. Wenn ein Nicht-Terminalsymbol vorkommt, wird die entsprechende Methode ***aufgerufen***. Diese Methode funktioniert bei rekursiven und nicht-rekursiven Syntaxen. Der ***Stack*** ist ***implizit*** durch Methodenaufrufe gegeben. Die bevorzugte Vorgehensweise ist die ***zielorientierte Satzzerlegung*** *(direct prediction)*: Die Syntax wird ***analysiert*** und ***systematische Kriterien*** für das Parsen ***bestimmt***, damit immer klar ist, welche Produktion als nächstes angewendet wird. Ist dies nicht möglich, muss mit ***Backtracking*** gearbeitet werden: Bei einem Syntaxfehler wird der letzte Schritt rückgängig gemacht und ein anderer Pfad gewählt. Wird ein Pfad gefunden, ist die Syntax ***valide***.

4.4.1. One Token Lookahead

```
Statement = Assignment | IfStatement.
Assignment = Identifier "=" Expression.
IfStatement = "if" "(" Expression ")" Statement.
```

```
void ParseStatement() {
if (???) ParseAssignment();
else if (???) ParseIfStatement(); }
```

Nach welcher ***Bedingung*** soll hier verzweigt werden? Um dies zu entscheiden, kann ***one Token Lookahead*** verwendet werden. Mit dieser Methode werden ***mögliche erste Terminalsymbole*** bestimmt, die mit einer Produktion ableitbar sind – die ***FIRST-Menge***.

```
FIRST(Assignment) = { Identifier }
FIRST(IfStatement) = { "if" }
```

```
void ParseStatement() {
  if (IsIdentifier() /* FIRST(Assignment) */) ParseAssignment();
  else if (Is(TokenTag.IF) /* FIRST(IfStatement) */) ParseIfStatement();
}
```

`FIRST()` kann auch ***mehrere Elemente*** enthalten:

```
FIRST(LoopStatement) = { "while", "do" }
if (Is(TokenTag.WHILE) || Is(TokenTag.DO)) { ParseLoopStatement(); }
```

4.4.2. K Tokens Lookahead

Es kann auch sein, dass Lookahead von einem Token ***nicht ausreicht***. In einem solchen Fall brauchen wir einen ***Lookahead mit k Tokens***.

```
Statement = Assignment | MethodCall.
Assignment = Identifier "=" Expression.
MethodCall = Identifier "(" ")".
```

```
FIRST(Assignment) = { Identifier }
FIRST(MethodCall) = { Identifier }
```

k = 2 würde in diesem Beispiel ausreichen. Um dies zu implementieren, ist eine ***technische Syntax-Umformung*** notwendig, damit ein ***one Token Lookahead*** wieder ausreicht – der gemeinsame Teil wird ***zusammengefasst***.

```
Statement = Identifier ( AssignmentRest | MethodCallRest ).
AssignmentRest = "=" Expression.
MethodCallRest = "(" ")".
```

```
void ParseStatement() {
  var identifier = ReadIdentifier();
  if (Is(TokenTag.ASSIGN)) ParseAssignmentRest(identifier);
  else if (Is(TokenTag.OPEN_PARENTHESIS)) ParseMethodCallRest(identifier);
}
```

4.4.3. Linksrekursion

Eine Linksrekursion wie hier: `Sequence = Sequence ["s"]` sollte vermieden werden. Jedes ***Nicht-Terminalsymbol*** löst einen Aufruf der ***jeweiligen `Parse()`-Methode*** auf. Da `Sequence` ganz links steht, muss die `ParseSequence()`-Methode ***vor*** der Überprüfung auf `s` aufgerufen werden – es gibt somit ***keine Abbruchbedingung*** der Rekursion und sie ist endlos. Deshalb sollte Linksrekursion ***vermieden*** werden, stattdessen sollte die ***EBNF-Repetition*** verwendet werden: `Sequence = {"s"}`

```
// Sequence = Sequence ["s"] Parse
void ParseSequence() {
  ParseSequence(); // Recursion Error
  if (!AtEnd) {
    if (Current == 's') {
      Next();
    }
    else {
      Error();
    } } }
```

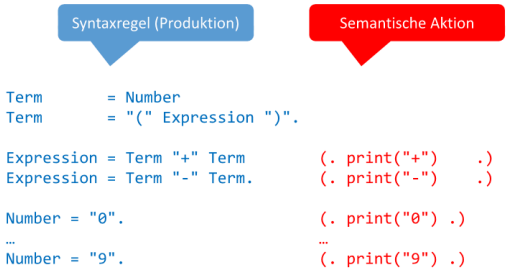
```
// Sequence = {"s"} Parse
void ParseSequence() {
  while (!AtEnd) {
    if (Current == 's') {
      Next();
    }
    else {
      Error();
      break;
    } } }
```

5. ALTERNATIVE PARSING-METHODEN

5.1. SYNTAXGESTEUERTE ÜBERSETZUNG

- ***Annotationen*** zu den Syntaxregeln: Attribute zu den Symbolen, Semantische Regeln pro Syntax-Regel, Semantische Aktionen innerhalb von Syntax-Regeln.
- ***Zweck:*** Zusätzliche Aktionen beim Parsen *(Type Checks, Syntaxbaum erzeugen, Code-Generierung, Direkte Auswertung von konstanten Expressions)*
- Wird oft in ***Parser-Generatoren*** verwendet *(z.B. yacc, bison)*

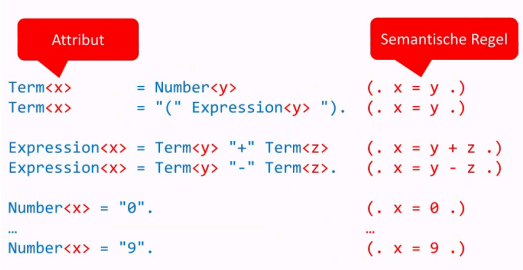
Beispiel: *Bei Erkennung des Syntaxteils wird das entsprechende Zeichen ausgegeben*



5.1.1. Vorteile / Nachteile

- ***Vision:*** Alles in der Grammatik beschrieben. Ganzer Compiler von A bis Z generieren, quasi ein «Compiler-Compiler».
- ***Praxis:*** Unübersichtlich und kompliziert *(Seiteneffekte im Parser, Verteilte Code-Snippets in der Regeln, Syntax & Semantik gemischt, meist nur für Syntaxbaum-Erzeugung genutzt)*

Beispiel: *Weise Symbole Attribute zu und verwende sie in semantischen Regeln (hier als Type Checks)*



5.2. BOTTOM-UP PARSER

Lese Token im Text ***ohne fixes Ziel***. Prüfe nach ***jedem Schritt***, ob gelesene Folge einer ***EBNF-Regel*** entspricht.

- ***Wenn ja:*** Reduziere auf Nichtterminal-Symbol (`REDUCE`)
- ***Wenn nein:*** Lese weiteres Token im Input (`SHIFT`)

Am Schluss bleibt ***Startsymbol*** übrig, ansonsten war ein ***Syntaxfehler*** vorhanden.

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	

5.2.1. Vereinfachte Parsing-Strategie

Lässt sich mit einer Tabelle abbilden. ***Probleme:*** Performance, keine Zustandsabhängigkeit.

5.3. LR-PARSER

Ein LR-Parser ist ***mächtiger*** als ein LL-Parser, weil er z.B. ***Linksrekursion*** behandeln kann *(z.B. `E = [E] "x"`)*.

- ***`LR(0)`:*** Parse-Tabelle ohne Lookahead erstellen. Aktueller Parse-Table-Zustand reicht, um zu entscheiden.
- ***`SLR(k)`:*** Simple `LR`. Lookahead bei `REDUCE`-Schritten, um Konflikt zu lösen. Keine neuen Zustände.
- ***`LALR(k)`:*** Look-Ahead `LR`. Analysiert Sprache auf `LR(0)`-Konflikte. Benutzt Lookahead bei Konfliktstellen und fügt dann neue Zustände in der Parse Table ein.
- ***`LR(k)`:*** Pro Grammatikschritt + Lookahead ein Zustand. Nicht praxistauglich, da zu viele Zustände.

In der Praxis genügt meist ein ***`LL(k)`-Parser***, die Grammatik kann oft dahingehend angepasst werden. C++, Java und C# haben handgeschriebene LL-Parser, obwohl ihre Grammatik nicht für LL entworfen wurde. Die Grammatik wurde stellenweise angepasst oder es werden längere Lookaheads verwendet. Bei ***Parser-Generatoren*** sind `LARL(k)` und `LL(k)` verbreitet.

5.3.1. Details

- ***Parser-Bauteile:*** Input-Symbol-Stream mit Lookahead, Zustandsmaschine *(um nach einer Reduktion in einen neuen Zustand zu gelangen)*, Ableitungs-Stack *(speichert die erkannten Symbole und aktuellen Zustände)* und Parse-Tabelle *(Aktionen pro Zustand & nächster Input)*.
- ***4 mögliche Aktionen*** beim Analysieren jedes Symbols:
  - `SHIFT` *(lesen und pushen des nächsten Symbols auf den Stack)*
  - `REDUCE` *(den Stack auf die linke Seite einer Produktion reduzieren)*
  - `ACCEPT` *(Syntaktische Analyse erfolgreich)*
  - `ERROR` *(Syntaktischer Fehler aufgetreten)*

5.3.2. Konstruktion

1. ***Grammatik anpassen:*** Umformulierung in BNF *(Dediziertes Startsymbol einführen, EBNF-Wiederholung durch Rekursion ersetzen, EBNF-Optionen eliminieren, indem sie in mehrere Produktionen aufgeteilt werden)*
2. ***Zustandsmaschine berechnen:***
   - ***Item:*** *(Produktion mit Kennzeichnung, wie weit der Parser bereits analysiert hat – in den Folien-Beispielen ein «●»)*
   - ***Handle:*** *(Item, das Kennzeichnung am Schluss hat. Bedeutet, dass Produktion reduziert werden kann)*
   - ***Closure:*** *(Enthält alle equivalenten Produktionen eines Item, d.h. befindet sich die Kennzeichnung bei einem nicht-Terminalsymbol wird dieses aufgelöst, bis alle mögliche Varianten dieser Produktion gefunden wurden)*
   - ***Goto:*** *(Set eines Terminalsymbols, dass alle BNF-Regeln mit diesem Symbol darin enthält. Goto-Sets werden so lange aufgelöst, bis daraus ein vollständiges Zustandsdiagramm erstellt werden kann)*
3. ***Parse-Tabelle bauen:*** `FOLLOW(X)`-Set *(Menge aller Terminalsymbole, die auf das Nicht-Terminalsymbol X folgen können)*

5.3.3. Parser-Tabelle

Der Parser muss wissen, was er als ***nächsten Schritt*** tun muss. Diese möglichen Schritte werden in einer ***Parser-Tabelle*** abgespeichert. Die Erzeugung ist ***kompliziert***, ***Entscheidungskonflikte*** sind möglich.

- ***Shift-Reduce-Konflikte*** *(Es sind sowohl `Shift` als auch `Reduce` möglich)*
- ***Reduce-Reduce-Konflikte*** *(Es gibt mehrere mögliche Produktionen, auf welche reduziert werden kann)*

Um diese aufzulösen, gibt es verschiedene Vorgehensweisen: ***Zusätzliche Regeln*** im Parser zur Konfliktbehebung, ***Anpassung der Grammatik*** oder ***grösserer Lookahead***.

---

6. SEMANTISCHE ANALYSE

Ein Programm kann ***syntaktisch richtig***, aber ***semantisch trotzdem falsch*** sein.

```
boolean x;
if ( x + x ) { int x; x = 0; } // bool nicht addierbar, x ist mehrfach deklariert
```

Der ***Semantic Checker*** kümmert sich um die semantische Analyse. Er wendet ***kontextsensitive Regeln*** an, welche z.B. prüfen, ob zwei Werte addierbar sind oder ob eine Variable mehrfach definiert wird. Der Input ist ein konkreter oder abstrakter ***Syntaxbaum*** und der Output eine ***Zwischendarstellung*** *(Abstrakter Syntaxbaum + Symboltabelle)*.

6.1. SEMANTISCHE PRÜFUNG

Prüfe, dass Programm ***alle*** über die Syntax hinausgehenden ***Regeln*** enthält.

- ***Deklaration:*** Jeder Identifier ist eindeutig deklariert *(keine Mehrfachdeklarationen, kein Identifier ist ein reserviertes Keyword)*
- ***Typen:*** Typregeln sind erfüllt *(Typen stimmen bei Operatoren, Kompatible Typen bei Zuweisungen, Argumentliste passt auf Parameterliste, Bedingungen in `if`, `while` sind `boolean`, `return`-Ausdruck hat Wert des Return Types)*
- ***Methodenaufrufe:*** Argumente und Parameter sind kompatibel
- ***Weitere Regeln:*** z.B. keine zyklische Vererbung, nur eine `main()`-Methode, `array.length()` ist read-only

6.1.1. Benötigte Informationen

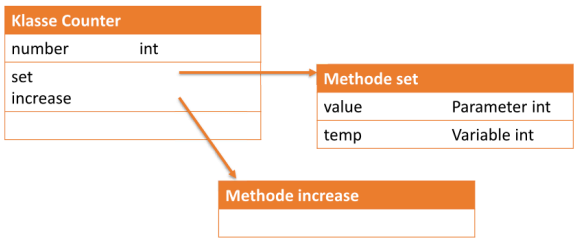
- ***Deklarationen:*** Variablen, Methoden, Parameter, Felder, Klassen, Interfaces
- ***Typen:*** Vordefinierte Typen *(`int`, `boolean` etc.)*, Benutzerdefinierte Typen *(Klassen, Interfaces)*, Arrays, Typ-Polymorphismus- und Vererbungsinformationen

6.2. SYMBOLTABELLE

Beispiel Deklarationen

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

Die Symboltabelle ist eine Datenstruktur zur Verwaltung der Deklarationen. Widerspiegelt hierarchische Bereiche im Programm, die ***Scopes***.



6.2.1. Shadowing

Deklarationen in inneren Scopes ***verdecken gleichnamige*** des äusseren Scopes. ***Ausnahme:*** Innerhalb Scopes der gleichen Methode ist Shadowing meist verboten *(z.B selbe Variable in- und ausserhalb eines `while`-Loop definiert)*

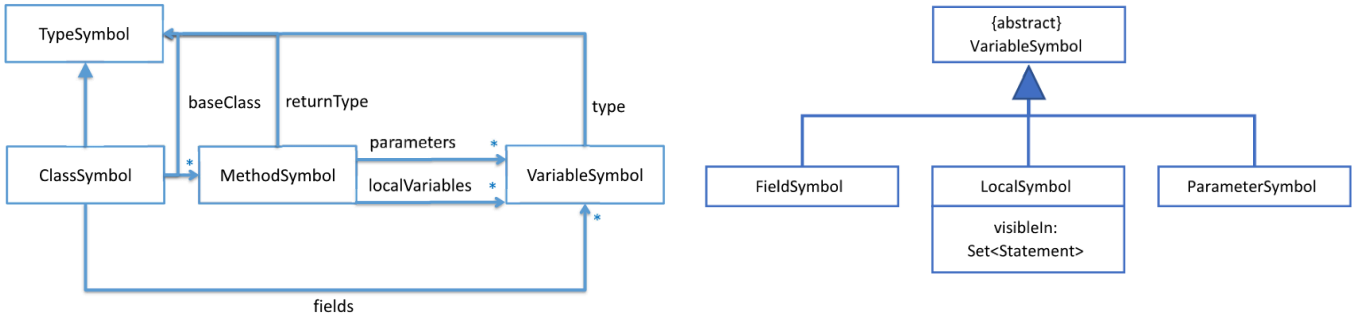
6.2.2. Global Scope

Wenn es mehrere Klassen und Interfaces gibt, gibt es für jede Klasse eine Tabelle mit den klassenspezifischen Deklarationen. Es gibt jedoch auch ***Deklarationen im Global Scope***. Hier werden alle Klassen des Programms sowie globale Variablen referenziert.

Auch vordefinierte ***Typen*** *(Value Types wie int und boolean, Reference Types wie string)*, ***Konstanten*** *(true, false, null)*, ***this*** *(in Methoden als read-only Parameter)*, ***Built-in Methoden*** *(`writeString()` etc.)* und ***Felder*** *(`length`: read-only und nur für Array-Typen)* (aber keine Keywords!) werden hier eingetragen. Sie werden hier definiert, um die Implementation des Lexers und Parsers zu vereinfachen.

Array Typen: Sind im Source ***nicht*** mit eigenem Namen definiert. Haben eine ***String-Repräsentation*** der Form `<Element-Typ>"[]"`, damit sie via Namen nachgeschlagen werden können.

6.2.3. Aufbau der Symboltabelle



6.3. VORGEHEN

1. ***Erstellen*** aller deklarierten Symbole *(und Eintragen in die Symboltabelle)*
2. Typen aller Deklarationen ***auflösen***
3. Verwendung der Deklarationen in AST *(Abstract Syntax Tree)* ***auflösen***
4. Typen in AST ***prüfen***

6.3.1. Erstellen aller deklarierten Symbole

Die Symbole werden zuerst durch eine ***Registrierung aller Deklarationen im Programm*** konstruiert. Anschliessend wird der AST von ***oben nach unten traversiert***, ausgehend vom Global Scope. Pro Klasse, Interface, Methode, Parameter und Variable wird das ***entsprechende Symbol*** in den ***richtigen Bereich eingefügt***.

In dieser Phase werden die Deklarationen ***nur gesammelt und registriert***, die Typen werden ***noch nicht aufgelöst***.

6.3.2. Typen aller Deklarationen auflösen

In dieser Phase ***löst*** der Semantic Checker ***alle Typen*** der in der ersten Phase hinzugefügten Deklarationen ***auf*** *(Felder, Parameter, lokale Variablen, Return type, Basisklasse)*. Für jeden Typ wird der ***entsprechende Zieltyp*** in der Symboltabelle gesucht. «Welches Symbol deklariert den Identifier?» Dies geschieht von ***innen nach aussen***.

6.3.3. Verwendung der Deklarationen in AST auflösen

In diesem Schritt werden die Anweisungen aller Methoden im AST traversiert und ***jeder Designator aufgelöst***. Ein ***Designator*** ist entweder eine ***Variable***, ein ***Arrayelement*** oder eine ***Methode***. Bei einfachen Designatoren kann in der ***Symboltabelle*** nach dem Namen gesucht werden. Bei einem ***Member Access*** *(z.B `car.seats`)*, muss zuerst das Feld *(`seats`)* im Scope der Klasse *(`car`)* nachgeschlagen werden. Bei einem ***Arrayzugriff*** *(z.B. `a[3]`)* wird nach der Array-Typdefinition gesucht und daraus der Elementtyp erfasst.

6.3.4. Typen in AST prüfen

Mit einer ***Post-Order-Traversierung*** werden die Typen zuerst in den unteren Knoten bestimmt. Die Typen der Blattknoten sind vordefinierte Typen wie `int` oder `string`. ***Designatoren*** haben die ***Typen***, welche in der vorherigen Phase ***bestimmt*** wurden. Unäre oder Binäre Ausdrücke erhalten ihren Typ je nach Operanden.
Da in SmallJ der + Operator nur für `int` definiert ist, kann als Typ also implizit `int` angenommen werden.

6.4. IMPLEMENTATION

Wie beim Lexer und Parser arbeitet die ***Semantische Analyse*** mit dem ***Visitor Pattern***. Dabei fungiert das Feld `ExpressionType` als globaler Zustand, welche den Typ des zuletzt geprüften Nodes beinhaltet. Im nebenstehenden `BinaryExpressionNode` wird zuerst rekursiv der linke, dann der rechte Kindknoten geparst, um deren Typen zu erhalten. Gehört der Operator des Nodes zu einer arithmetischen Operation, wird geprüft, dass sowohl der linke, als auch rechte Teil des Ausdrucks einer `int`-Variable zuweisbar sind. Ist das der Fall, wird für diesen Node der Typ `int` in `ExpressionType` geschrieben.

```
TypeSymbol? ExpressionType { get; private set; }
override void Visit(BinaryExpressionNode node) {
  var left = node.Left; var right = node.Right;
  left.Accept(this);
  var leftType = ExpressionType;
  right.Accept(this);
  var rightType = ExpressionType;
  switch (node.Operator) {
    case Operator.DIV or Operator.MINUS or ...:
      CheckAssignableTo(left, leftType, @int);
      CheckAssignableTo(right, rightType, @int);
      ExpressionType = @int; break;
  } }
```

6.5. INTERMEDIATE REPRESENTATION

Die Symboltabelle mit aufbereitetem AST ergibt eine ***Zwischendarstellung*** für das Compiler-Backend. Damit kann mit der ***Code Generierung*** weitergefahren werden.

---

7. CODE GENERIERUNG

Der ***Code Generator*** kümmert sich um die Erzeugung von ausführbarem Maschinencode. Der Input ist eine ***Zwischendarstellung*** und der Output ***(Virtual) Maschine Code*** in einem ***Assembly- bzw. Object File***.

7.1. COMPILER FRONTEND/BACKEND

- ***Frontend:*** Java, C#, Scala, …
- ***Backend:*** x64 Assembly, Java Bytecode, .NET CLI, …

AST & Symboltabelle trennt das ***Frontend*** vom ***Backend***. Diese Trennung erlaubt ***Mehrsprachen-*** und ***Mehrplattformenkompatibilität*** im Compiler, da alle Sprachen auf ***dieselbe Intermediate Language*** *(IL)* kompiliert werden und aus diesem der Maschinencode für die entsprechende Platform generiert werden kann.

7.2. STACK-PROZESSOR

Instruktionen benutzen einen ***Auswertungs-Stack*** anstelle von CPU-Registern. Jede Instruktion hat eine ***definierte Anzahl*** von ***Pop***- und ***Push-Aufrufen***. Am Anfang und am Schluss der Methode ist der ***Evaluations-Stack*** leer. Um eine Instruktion ***ausführen*** zu können, müssen vor dem Aufruf die entsprechenden Werte auf den ***Evaluation Stack*** gepusht werden.
Jede Methode hat ihren ***eigenen Evaluation Stack***, dieser muss zu Beginn und am Ende der Methode ***leer*** sein.

7.3. INSTRUKTIONEN

7.3.1. Beispiel: `while`-Bytecode

```
while (x < 10) {
  x = x + 1;
}
```

```
begin: load 1 ; "x" has index 1
       ldc 10
       icmplt
       if_false end ; jump to code after loop if statement is false
       load 1
       ldc 1
       iadd ; x + 1
       store 1
       goto begin
end:
```

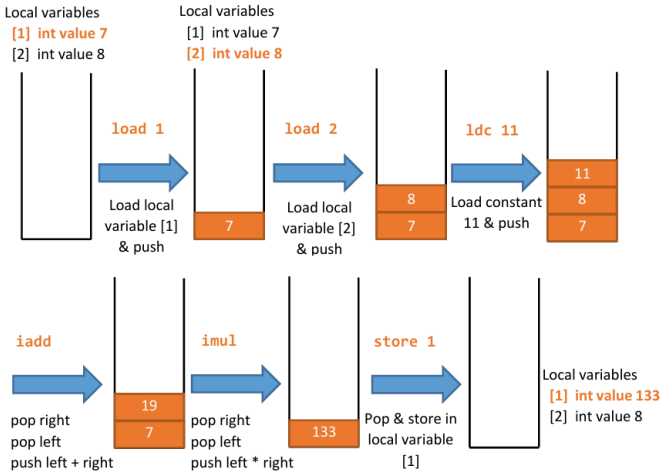
SmallJ-Instruktion	*Bedeutung*	*Auswertungs-Stack*
`ldc <const>`	Lade Konstante *(`int`, `boolean`, `string`)*	1 Push
`iadd`	Integer Addition	2 Pop, 1 Push
`isub`	Integer Subtraktion	2 Pop, 1 Push
`imul`	Integer Multiplikation	2 Pop, 1 Push
`idiv`	Integer Division	2 Pop, 1 Push
`irem`	Integer Modulo	2 Pop, 1 Push
`ineg`	Integer Negation	1 Pop, 1 Push
`bneg`	Boolesche Negierung	1 Pop, 1 Push
`load <num>`	Lade Parameter oder lokale Variable	1 Push
`store <num>`	Speichere Parameter oder lokale Variable	1 Pop
`cmpeq`	Compare Equal *(Verschiedene Typen)*	Pop right, Pop left, Push boolean
`cmpne`	Compare Not Equal *(Verschiedene Typen)*	Pop right, Pop left, Push boolean
`icmpgt`	Integer Compare Greater Than	Pop right, Pop left, Push boolean
`icmpge`	Integer Compare Greater Equal	Pop right, Pop left, Push boolean
`icmplt`	Integer Compare Less Than	Pop right, Pop left, Push boolean
`icmple`	Integer Compare Less Equal	Pop right, Pop left, Push boolean
`goto <label>`	Branch/Jump *(Bedingungslos)*	—
`if_true < label>`	Branch falls true	1 Pop
`if_false <label>`	Branch falls false	1 Pop

7.3.2. Load / Store Indexierung für Locals

Mit `load` / `store` können Parameter und lokale Variablen verwendet werden. Die Parameter werden zuerst deklariert, damit sie innerhalb der Methode nicht redefiniert werden können.

1. ***«this»*** Referenz: Index 0 *(Virtuelle Methode, read-only)*
2. n ***Parameter***: Index $1 \ldots n$
3. m ***lokale Variablen***: Index $n+1 \ldots n+m$

```
void swap(int a, int b) {
  int temp;
  temp = a; // load 1 (a) / store 3 (temp)
  a = b; // load 2 (b) / store 1 (a)
  b = temp; // load 3 (temp) / store 2 (b)
  ... }
```



7.4. METADATEN

Durch die Metadaten kann die ***Speicher-*** und ***Typensicherheit*** zur Runtime ***verifiziert*** werden. Dies ist insbesondere beim ***Laden von Libraries*** wichtig, um Kompatibilität zu garantieren.
Die Zwischensprache kennt alle Informationen zu

- ***Klassen:*** Namen, Typen der Fields und Methoden
- ***Methoden:*** Namen, Parametertypen und Rückgabetyp
- ***Lokale Variablen:*** Typen und Namen

Es ist ***kein direktes Speicherlayout*** festgelegt, dies liegt in der Hand der Runtime-VM. Namen von ***lokalen Variablen und Parameter*** sind ***nicht enthalten***, sie werden nur ***durchnummeriert***, weil dies Platz & Performance spart.

7.5. CODE-GENERIERUNG

1. ***Traversiere Symboltabelle***: Erzeuge Bytecode Metadaten
2. ***Traversiere AST pro Methode (Visitor):*** Erzeuge Instruktionen via Bytecode Assembler
3. ***Serialisiere in Output Format:*** Mittels .NET Standard Library oder Proprietäres Binärformat

7.5.1. Bytecode Assembler

While-Loop von «Beispiel: `while`-Bytecode» (Seite 11) in Bytecode Assembler:

```
var assembler = new Assembler(...);
var begin = assembler.CreateLabel();
var end = assembler.CreateLabel();
assembler.SetLabel(begin);
assembler.Emit(LOAD, 1);
...
```

7.5.2. Backpatching

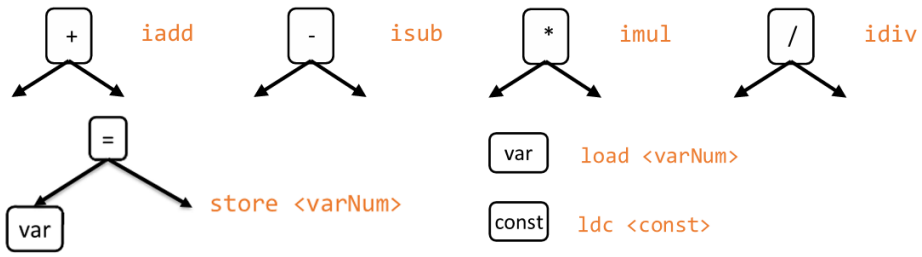
Nach der Generierung des Bytecodes sollen die ***Labels*** in den Jumps in ***relative Offsets***, die ***Branch Offsets***, umgewandelt werden. Diese geben an, ***wie viele*** Instruktionen nach oben/unten gesprungen werden soll, bis das Label erreicht wird. Die Labels werden danach entfernt.

```
if_false end → if_false<5>
goto begin → goto<-9>
```

7.5.3. Template-Basierte Generierung

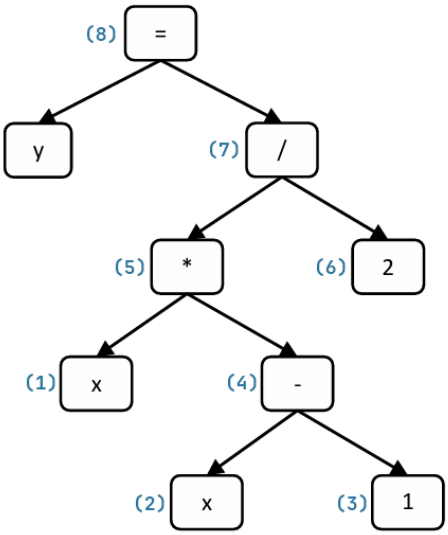
Postorder-Traversierung: Kinder zuerst besuchen und jeweils Template für erkanntes Teilbaum-Muster anwenden.

Verschiedene Teilbaum-Muster



Beispiel

```
// Assignment → rechter Node, dann Post-Order
(1) load 1
// load <varNum> wird angewendet, x hat Wert 1
(2) load 1
// load <varNum> wird erneut angewendet
(3) ldc 1
// ldc <const> wird angewendet
(4) isub
// isub Teilbaum erkannt
(5) imul
// imul Teilbaum erkannt
(6) ldc 2
// ldc <const> wird angewendet
(7) idiv
// idiv Teilbaum erkannt
(8) store 2
// store <varNum> wird ohne Rekursion
angewendet, y hat Wert 2
```



Traversierungsreihenfolge

- ***Bei Expressions:*** Immer Post-Order
- ***Bei Statements:*** Je nach Code-Template
  - ***Assignment:*** Beim obersten Knoten zum Kindknoten rechts traversieren *(Statement rechts von =)* → Post-Order zum Knoten ganz links *(Target)* traversieren und auswerten → Den Knoten rechts *(Wert)* auswerten → Beim obersten Knoten links *(Variable links von =)* `store`-Instruktion emitten
    (`a = 2 * b` → `ldc 2, load b, imul, store a`)
  - ***Member Access:*** Klassenvariabel mit `load` laden, bevor auf Feld zugegriffen werden kann
    (`a.f = 7` → `load a, ldc 7, store f`)
  - ***If-Statement:***

```
        <Condition>
        if_false else
        <Then-Block>
        goto endif
else: <Else-Block>
endif:
```

  - ***While-Statement:***

```
whileStart: <Condition>
            if_false whileEnd
            <While-Block>
            goto whileStart
whileEnd:
```

7.5.4. Bedingte Auswertung

Die ***&&*** und ***||*** Operatoren haben eine ***Short-Circuit Logik***: Das rechte Statement wird nur evaluiert, wenn es für das Resultat ***entscheidend*** ist. Ist die linke Seite bei && bereits `false` bzw. bei || schon `true`, kann direkt im Code weitergefahren werden.

```
// Pseudo code for "&&"
// a && b ⇒ a ? b : false
       left
       if_false short
       right
       goto end
short: ldc false
end:
```

```
// Pseudo code for "||"
// a || b ⇒ a ? true : b
       left
       if_true short
       right
       goto end
short: ldc true
end:
```

7.5.5. Methodenaufruf

- ***Statisch:*** Vordefinierte Methoden *(`readInt()`, `writeString()`)* mit `invokestatic`
- ***Dynamisch:*** An Objekt gebunden *(`x.run()`, `this.delete()`)* mit `invokevirtual`. Bei `this` wird immer `load 0` generiert, auch wenn es im Code selber implizit ist.

Bei ***virtuellen Methodenaufrufen*** muss die Objektreferenz zuunterst auf dem Evaluation Stack liegen, darüber befinden sich ***die Parameter*** von vorne nach hinten. Am Ende der Methode befindet sich eine ***`ret`-Instruktion***, um die Methode zu verlassen. Hat die Methode einen ***Return Type*** *(nicht `void`)*, muss dieser Wert vor `ret` auf den Stack gepusht werden. Dieser wird dann auf den Stack des Aufrufers gepusht. Der Evaluation Stack der aufgerufenen Methode muss am Schluss leer sein.

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

int five;
five = 5;
sum(10, five);
```

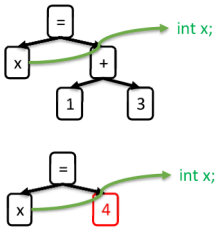
```
ldc 5             ; load number '5'
store 1           ; store '5' into 'five'
load 0            ; load object reference of function (here
'this')
ldc 10            ; load first parameter (constant)
load 1            ; load second parameter (variable)
invokevirtual sum ; call the 'sum' function
load 1            ; load parameter 'x'
load 2            ; load parameter 'y'
iadd              ; add numbers and put result on stack
ret               ; return value on stack and exit method
```

8. CODE-OPTIMIERUNG

Code-Optimierungen können an ***verschiedenen Orten im Compiler*** angewendet werden.
- ***Ahead-of-Time Compiler:*** Vor *(Plattform-unabhängig)* oder nach *(VM-Abhängig)* der Code-Generierung
- ***JIT-Compiler:*** Vor *(Maschinen-unabhängig)* oder nach *(Maschinen-Abhängig)* der Code-Generierung

8.1. AUFGABEN DER OPTIMIERUNG

Die Aufgabe der Optimierung ist es, eine ***Zwischendarstellung*** in eine ***effizientere***, aber ***gleichwertige*** Darstellung umzuwandeln.



Mögliche Zwischendarstellungen
- ***AST + Symbol-Tabelle***
- ***Bytecode***
- ***Three-Address-Code mit Registern*** *(Zwei Operanden und ein Resultat)*

8.2. OPTIMIERTE ARITHMETIK

Je nach Maschine können gewisse Operationen ***«teurer»*** sein als andere. Dies gilt insbesondere für die ***Multiplikation***, ***Division*** und ***Modulo***. Diese Operationen umzuschreiben kann zu einer ***Leistungssteigerung*** führen.

8.2.1. Bit-Operationen

Multiplikation, Division oder Modulo mit einer ***Zweierpotenz*** können in eine ***Bit-Operation*** umgewandelt werden.

Original	*Optimiert*
`x * 32`	`x << 5` *($2^5 = 32$, Bitshift um 5 Stellen nach links)*
`x / 32`	`x >> 5` *(Bitshift um 5 Stellen nach rechts)*
`x % 32`	`x & 31` *(Bitwise and, «isoliert» die 5 hintersten Bits, welche kleiner als 32 sind und den Rest von $x \div 32$ darstellen)*

8.2.2. Algebraische Vereinfachung

- Entfernung von ***redundanten Operatoren*** *(`Expr / 1` ⇒ `Expr`, `Expr * 0` ⇒ `0`)*
- AST-Bäume, die nur aus ***konstanten Literalen*** bestehen, können ***zusammengefasst*** werden *(`1 + 3` ⇒ `ldc 4`)*
- Sonstige ***redundanten*** AST-Bäume *(Doppel-Minus: `--Expr` ⇒ `Expr`)*

Diese Expressions dürfen aber nur vereinfacht werden, wenn sie ***keine Seiteneffekte*** besitzen
(Keine Exceptions, kein Schreiben von Variablen, keine I/O-Operationen)

8.2.3. Loop-Invariant Code

Code in einer ***Schlaufe***, welcher sich jedoch nicht verändert, kann aus der Schlaufe ***herausgenommen*** werden. So muss er ***nicht*** bei jedem Durchlauf ***neu evaluiert*** werden. Dies nennt man ***Code Motion***.

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

8.3. COMMON SUBEXPRESSIONS

Wiederholt ausgewertete ***Teilausdrücke*** können ***zusammengefasst*** werden, vorausgesetzt, sie ***verändern*** sich zwischen den Auswertungen ***nicht***. Dieser Schritt heisst ***Common Subexpression Elimination (CSE)***.

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

8.4. DEAD CODE ELIMINATION

Wird eine Variable geschrieben, aber nachher ***nicht weiterverwendet***, ist das ***Dead Code*** und kann ***entfernt*** werden. Dabei ist darauf zu achten, dass nach der Entfernung von Dead Code allfällig ***neu entstandener Dead Code*** ebenfalls entfernt werden muss. Dies ***verkleinert*** die Code-Grösse, was die ***Laufzeit*** durch weniger Cache-Benutzung und Ladezeiten ***verbessert***.

```
a = readInt();
b = a + 1;
writeInt(a);
c = b / 2; // wird nicht verwendet
```
⇒
```
a = readInt();
b = a + 1; // wird nicht verwendet
writeInt(a);
```
⇒
```
a = readInt();
writeInt(a);
```

8.5. COPY PROPAGATION

Code, der redundante `loads` und `stores` beinhaltet, kann ebenfalls ***vereinfacht*** werden. Dabei wird bei jedem Lesen einer Variable diese durch ihr ***letztes Assignment ersetzt***. Hier im Beispiel zusammen mit Dead Code Elimination:

```
t = x + y;
u = t;
writeInt(u);
```
⇒
```
t = x + y;
u = x + y; // ←
writeInt(u);
```
⇒
```
t = x + y;
u = x + y;
writeInt(x + y); // ←
```
⇒
```
writeInt(x + y);
```

8.6. CONSTANT PROPAGATION

Wird auch ***Constant Folding*** genannt. Wenn durch eine statische Analyse festgestellt wird, dass mehrere Variablen zur Laufzeit ***garantiert immer den gleichen Wert*** haben, sind diese ***konstant*** und können entsprechend durch diese Konstante ***ersetzt*** werden. Anschliessend kann in einem weiteren Schritt Dead Code und Duplikate entfernt werden.

```
a = 1;
if ( ... ) {
  a = a + 1; // a immer 2
  b = a;     // b immer 2
} else {
  b = 2;     // b immer 2
}
c = b + 1;   // c immer 3
```
⇒
```
a = 1;
if ( ... ) {
  a = 2;
  b = 2;
} else {
  b = 2;
}
c = 3;
```

8.7. PARTIAL REDUNDANCY ELIMINATION

Eine Expression ist partial redundant, wenn sie in ***einigen***, aber nicht allen Pfaden ***erneut berechnet wird***.

Im Beispiel wird `x + 4` beim Durchlauf des `if`-Pfades ***zweimal*** evaluiert, im `else`-Pfad jedoch nur ***einmal***.
Dies kann behoben werden, indem die Expression ***in beiden Pfaden*** evaluiert wird, dafür am ***Schluss nicht mehr***.
So wird sie in beiden Pfaden ***nur jeweils einmal*** evaluiert. Dafür gibt es in diesem Beispiel ***mehr Lese- und Schreiboperationen***.

```
if ( ... ) {
  y = x + 4;
} else {
  ...
}
z = x + 4;
```
⇒
```
if ( ... ) {
  t = x + 4;
  y = t;
} else {
  ...
  t = x + 4;
}
z = t;
```

8.8. ERKENNUNG VON OPTIMIERUNGSPOTENTIAL

Es ist immer ein ***Abwägen***, welche Optimierung am sinnvollsten ist. Eine Optimierung kann wiederum ein anderes Optimierungspotential auslösen.

Es gibt verschiedene ***Techniken***, um Optimierungspotential zu erkennen:
- Static Single Assignment
- Peephole Optimization
- Dataflow Analysis *(Wird in ComBau nicht behandelt)*

8.8.1. Static Single Assignment (SSA)

Static Single Assignment ist eine Codetransformation, die verschiedene Analysen und Optimierungen ***erleichtert***, die auf dem ***Lesen*** und ***Schreiben*** von ***Variablen*** beruhen.

Es kann auf einen ***Abstract Syntax Tree*** *(AST)*, auf ***Intermediate Code*** oder auf ***Maschinencode*** angewendet werden. ***Das Ziel ist einfach:*** Jede Variable wird ***nur einmal*** im Code zugewiesen. Der Code wird also wie folgt ***umgeschrieben***, sodass bei jeder ***neuen Zuweisung*** ein ***anderer Variablenname*** verwendet wird:

```
x = 1; x = 2; y = x;     ⇒     x1 = 1; x2 = 2; y1 = x2;
```

Damit ist schnell ersichtlich, ob sich eine Variable ***geändert*** hat oder nicht.

Bei ***Verzweigungen*** wird es etwas komplexer, weil nicht klar ist, welche Version der Variable weiterverwendet wird. Dafür gibt es die ***φ(x1, x2)-Funktion***: Falls der erste Pfad ausgeführt wird, wähle den ersten Wert, ansonsten den zweiten. In echtem Maschinencode wird die φ-Funktion durch eine ***temporäre Variable*** realisiert.

```
if ( ... ) { x1 = 1; } else { x2 = 2; }
y1 = φ(x1, x2)
```

Common Subexpressions in SSA

Common Subexpressions sind in SSA ***direkt entscheidbar***, weil sofort erkannt wird, ob sich die Variablen in der Zwischenzeit ***geändert*** haben.

```
x1 = a1 * b1 + c1;
...
y1 = a1 * b1 + d1; //a und b enthalten denselben Wert wie vorhin, können also zusammengefasst
werden
...
z = a1 * b2 + d3; // b und d wurden in der Zwischenzeit geändert
```

Dead Code Elimination in SSA

```
x1 = 1; // x1 wird nie gelesen ⇒ Dead Code, zu entfernen
x2 = 2; // wird gelesen, darf nicht entfernt werden
y1 = x2 + 1;
writeInt(y1);
```

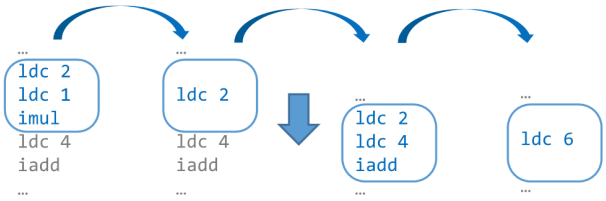
SSA-Berechnung

Die ***Umwandlung der Variablen*** und vor allem die ***Platzierung der Phis*** ist sehr ***aufwendig***. In simplen oder performance-optimierten Compilern wie bei JIT wird darum häufig darauf verzichtet. Hier sind günstigere Techniken gewünscht, wie zum Beispiel die ***Peephole Optimization***.

8.8.2. Peephole Optimization

Die Peephole Optimization ist eine sehr ***einfache und günstige*** Technik, um Optimierungen vorzunehmen. Im JIT-Compiler wird sie für Intermediate Code oder Maschinencode benutzt.

Die Technik verwendet ein ***«Sliding Window»*** mit z.B. jeweils 3 Instruktionen. Es werden ***nur*** diese 3 Instruktionen angesehen und ***mögliche Optimierungen angewendet***. Anschliessend wird das Fenster um 1 weitergeschoben.



8.9. ZUSAMMENFASSUNG

Techniken	***Optimierung, die die Technik ermöglicht***
Template-Based Code Gen, Peephole Optimization	Optimierte Arithmetik, Algebraische Vereinfachung
Static Single Assignment (SSA)	Common Subexpression Elimination, Dead Code Elimination, Copy Propagation, Constant Propagation, Partial Redundancy Elimination

9. VIRTUAL MACHINE

Um die mit dem eigenen Compiler in die Intermediate Language kompilierten Programme laufen zu lassen, brauchen wir eine ***Laufzeitumgebung***, also eine ***Virtuelle Maschine***.

Nutzen: ***Mehrplattformensupport*** *(Der gleiche Compiler kann für verschiedene Plattformen verwendet werden, es muss nur die VM portiert werden, damit alle Programme laufen)*, ***Mehrsprachigkeit*** *(kann mehrere Sprachen unterstützen, welche auf dieselbe IL kompilieren)*, ***Sicherheit*** *(VM verhindert elementare Sicherheitsprobleme wie Speicherfehler und erlaubt Sandboxing)*.

9.1. AUFBAU EINER VIRTUELLEN MASCHINE

Loader → Interpreter *(mit Just-in-Time Compiler)* → Metadaten, Heap & Stacks → Garbage Collection

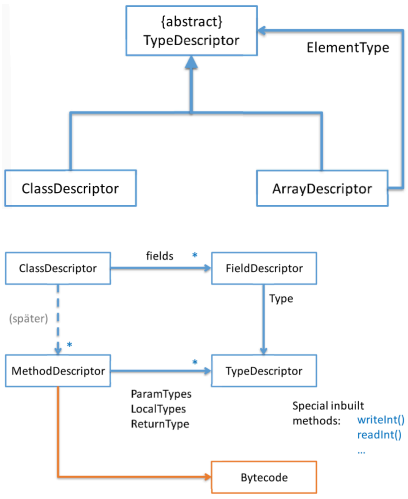
9.2. LOADER

Der Loader ***liest*** das ***Assembly*** bzw. Object File und ***alloziert*** die notwendigen ***Laufzeitstrukturen***. Er kreiert ***Metadaten*** für Klassen, Methoden, Variablen, Code, z.B die Definition der Speicher-Layouts für Fields/Variablen/Parameter *(Typengrösse, Offsets)*. Ebenfalls löst er die Verweise zu Methoden/Typen/anderen Assemblies auf absolute Speicheradressen auf; die ***Relocation***. Danach ***initiiert*** er die Programmausführung und macht optional eine ***Code Verifikation***. Schlussendlich instanziert die VM die Klasse, welche die Main-Methode enthält und ruft sie dynamisch auf.

Der ***Verifier*** überprüft, ob der Bytecode ***valide*** ist. Die meisten dieser Checks können aber auch zur Laufzeit durchgeführt werden. Geprüft werden unter anderem Type Errors, Evaluation stack Under-/Overflow, Undefinierte Variablen/Methoden/Klassen, Illegale Instruktionen & Branches und weitere sprachspezifische Fälle.

9.2.1. Metadaten

Metadaten sind ***Laufzeitinformationen*** über Typen & Methoden *(Klassen: Feldtypen und Methoden, Arrays: Elementtyp)*. Diese Informationen sind notwendig, um das Programm korrekt auszuführen und Typsicherheit garantieren zu können. Zu diesem Zweck wandelt der Loader die Metadaten in Laufzeitstrukturen um, die ***Descriptors***, welche später weiterverwendet werden.



Der Loader erstellt für jede Klasse einen ***Class Descriptor***, der den Base Type, die beinhalteten Felder und ihre Typen und die beinhalteten Methoden spezifiziert.
Zusätzlich gibt es auch ***Method Descriptors***, die Parametertypen, Returntypen und die lokalen Variablen spezifizieren.

9.2.2. Code Patching

Der Bytecode wird ***direkt*** in den Speicher der VM ***geladen***. Bestimmte Operanden müssen jedoch ***angepasst*** werden, damit sie ***korrekt weiterverwendet*** werden können. Zum Beispiel müssen ***spezifische Referenzen*** *(auf Klassen, Methoden, Variabeln, Typen…)* ***oder Offsets*** angepasst werden, damit sie eine ***Referenz*** auf den entsprechenden Descriptor bilden. Diesen Vorgang nennt man ***Code Patching***.

Original	*Patched*
`invokevirtual MyMethod`	`callvirt <method_desc>`
`new MyClass`	`new <class_desc>`
`newarr MyType`	`newarr <type_desc>`
`getfield MyField`	`getfield <field_desc>`
`putfield MyField`	`putfield <field_desc>`
`checkcast MyClass`	`checkcast <class_desc>`
`instanceof MyClass`	`instanceof <class_desc>`

9.3. INTERPRETER

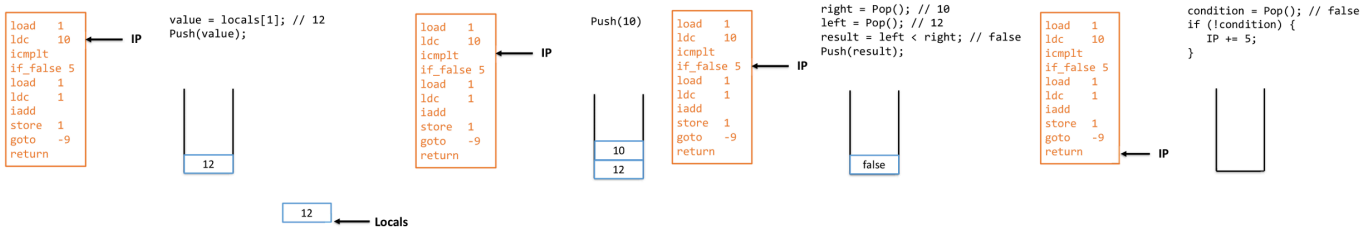
Der Interpreter ***führt den Code der Zwischensprache aus***. Er verfügt über einen ***Interpreter Loop***, die eine Anweisung nach der anderen emuliert. Er besteht aus drei Schritten: ***Fetch*** *(Instruktion laden)*, ***Decode*** *(Instruktion und Operanden auslesen)* und ***Execute*** *(Instruktion ausführen)*.

9.3.1. Bestandteile

- ***Interpreter Loop:*** Emuliert eine Instruktion nach der anderen `while (true) { Execute(code[IP++]) }`
- ***Instruction Pointer (IP):*** Adresse der nächsten Instruktion
- ***Evaluation Stack:*** Für den virtuellen Stack Prozessor
- ***Locals & Parameters:*** Von der aktiven Methode
- ***Method Descriptor:*** Für aktive Methode

9.3.2. Ablauf

1. Der Interpreter Loop holt die ***erste Instruktion*** `load 1` beim IP, wir laden also die lokale Variable mit Value `12` und pushen diese auf den ***Evaluation Stack***.
2. Anschliessend ***bewegt sich der IP eines nach vorne*** zur nächsten Instruktion `ldc 10`, womit der Interpreter Loop `10` auf den Evaluation Stack pusht.
3. Die nächste Instruktion `icmplt` poppt ***zuerst den rechten*** und ***dann den linken Operand*** für den ***Vergleich*** vom Evaluation Stack. Hier wird nun `10 < 12` verglichen, was `false` ergibt. Dieses ***Resultat*** wird wieder ***auf den Stack gepusht***.
4. Nun kommt die Instruktion `if_false`, welche das Resultat der letzten Instruktion vom Stack poppt, und weil dieses `false` ist, wird nun der ***Branch verwendet*** und der IP springt um 5 nach vorne. Da jedoch der IP ***zuerst zur nächsten Instruktion*** geht und dann erst springt, gehen wir ***insgesamt um 6*** nach vorne.



9.3.3. Ausführung des Interpreter-Loops

`Execute()` ***emuliert Instruktion*** je nach Op-Code.

```
while (true) {
  var instruction = code[instructionPointer];
  instructionPointer++;
  execute(instruction);
}
```

Der Loop läuft, bis das Programm ***beendet*** ist oder ein Fehler auftritt. Der IP wird vor der zu aufrufenden Funktion inkrementiert, deswegen gelten die Offsets von ***relativen Jumps*** auch erst ab der Instruktion ***danach***.

```
switch (instruction.OpCode) {
  case LDC:
    Push(instruction.Operand);
    break;
  case IADD:
    var right = Pop();
    var left = Pop();
    var result = left + right
    Push(result);
    break;
  ...
}
```

9.4. CALL STACK

Ein Programm beinhaltet ***mehrere Methoden*** und ***Sprünge*** zwischen den Methoden. Damit pro Methode der Zustand des Evaluation Stacks zwischengespeichert werden kann, braucht es neben dem Evaluation Stack auch einen ***Call Stack***. Dieser speichert die ***Activation Frames***, ein Frame pro aufgerufene Methode. Die aktive Methode liegt ***zuoberst*** auf dem Stack. Jeder ***Activation Frame*** speichert ***Parameter***, ***lokale Variablen*** und den ***Evaluation Stack*** pro aktive Methode.

Jedes Mal, wenn eine Methode mithilfe des Opcodes `invokevirtual` aufgerufen wird, wird ein ***neues Activation Frame*** auf den Call Stack gepusht.
Wenn zum Beispiel eine Methode `g()`, die ***bereits aufgerufen wurde***, also bereits auf dem Call Stack ist, ***rekursiv*** erneut aufgerufen wird, wird diese ein ***zweites Mal auf den Call Stack gepusht***, mit einem ***neuen Set*** an lokalen Variablen und einem ***neuen, leeren Evaluation Stack***.
Bei einem ***Rücksprung*** durch den Opcode `return` aus der Methode `g()` wird das oberste Frame wieder vom Call Stack ***entfernt***.

9.4.1. Methodenaufruf

Bei einem ***Methodenaufruf*** ruft der ***Caller*** *(Methode, welche den Function Call aufruft)* den ***Callee*** *(Zu aufrufende Methode)* auf. Zuerst wird der Method Descriptor des Callees bestimmt, indem der Operand des `invokeVirtual`-Opcodes evaluiert wird. Dieser spezifiziert die ***Anzahl Parameter***; basierend darauf werden die Argumente und anschliessend die `this`-Referenz auf das Objekt der auszuführenden Methode von hinten vom Stack gepoppt.

```
var met =
(MethodDescriptor)instruction.Operand;
var nofParams = met.ParameterTypes.Length;
var args = new object[nofParams];
for(int i = args.Length - 1; i ⩾ 0; i--) {
  args[i] = Pop();
}
var target = Pop(); // 'this' reference
var af = new ActivationFrame(met, target,
args);
callStack.Push(af);
```

9.4.2. Methodenrücksprung

Wird eine Methode ***beendet***, wird im aktuellen Method Descriptor geprüft, ob sie einen ***Return Type*** hat oder `void` ist. Hat sie einen, wird dieser Wert vom Stack gepoppt und nach der Entfernung des Activation Frames auf den Evaluation Stack des Callers gepusht.

```
var met = activeFrame.Method;
var hasReturn = met.ReturnType ≠ null; //
void?
object result = hasReturn ? Pop() : null;
callStack.Pop(); // go back to caller
if (hasReturn) { Push(result); }
```

9.4.3. Design

Ein Call Stack kann auf ***zwei Arten*** aufgebaut werden:
– ***Managed Call Stack:*** Im Interpreter, Objekt-orientierte Darstellung *(Fokus auf Komfort)*
– ***Unmanaged Call Stack:*** Bei HW-Execution *(JIT)*, Kontinuierlicher Speicherblock *(Fokus auf Effizienz)*

Managed Call Stack

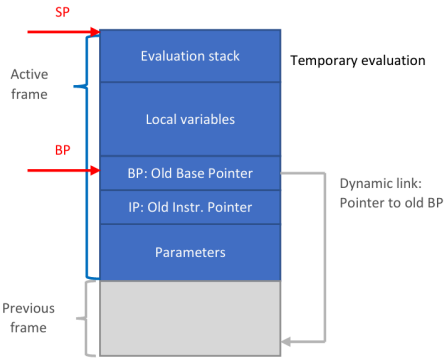
Wird mit ***Klassen*** modelliert. Ein Aktivierungsframe ist eine ***Instanz einer Klasse***, die einen Verweis auf den Method Descriptor sowie die lokalen Variablen, Parameter, this-Referenz, Evaluation Stack und den aktuellen IP des Callers enthält.

```
class ActivationFrame {
  public MethodDescriptor Method { get; }
  public Pointer ThisReference { get; }
  public object[] Arguments { get; }
  public object[] Locals { get; }

  public EvaluationStack EvaluationStack { get; }
  public int InstructionPointer { get; }
}

class CallStack {
  private readonly Stack<ActivationFrame> _stack;
}
```

Unmanaged Call Stack

Folgt dem klassischen ***Design eines Betriebssystems***. Bevorzugter Weg, wenn die VM in einer ***low-level Sprache*** implementiert oder JIT-Code ausführt wird. Der IP *(auch Program Counter genannt)* ist ***eine Adresse*** im generierten Code Block. Ausserdem wird ein ***zusammenhängender virtueller Speicherblock*** für den Stack alloziert.

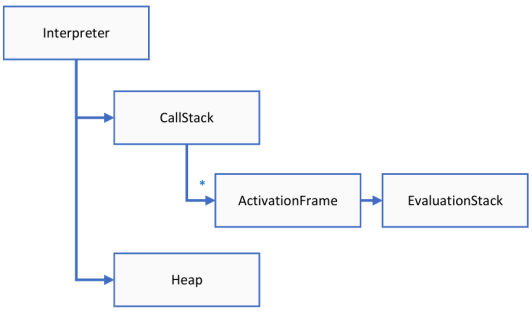
Der Frame der aktiven Methode ist durch den ***Stack Pointer*** *(SP)* und den ***Base Pointer*** *(BP, auch Frame Pointer genannt)*, abgegrenzt. Oberhalb des BPs werden die ***lokalen Variablen*** und der ***Evaluation Stack*** der ***aktiven Methode*** gespeichert. Unterhalb des BPs befinden sich die ***Parameter*** der aktiven Methode und der SP und BP des Callers, zu letzterem wird bei ***Return zurückgesprungen***.



9.5. LAUFZEITSTRUKTUREN

Die Abbildung rechts dient als Zusammenfassung der Laufzeitstrukturen einer virtuellen Maschine. Der ***Interpreter*** arbeitet mit einem ***Call Stack*** der aus ***Activation Frames*** besteht. Jeder ***Activation Frame*** verwaltet einen ***Evaluation Stack***.

Wenn die VM ***Multi-Threading*** unterstützen würde, müsste das Laufzeitsystem über mehrere Call Stacks verfügen, für jeden ***aktiven Thread*** einen.



9.6. VERIFIKATION

Erkenne und ***verhindere*** falschen Byte-Code, die durch ***Fehler im Compiler*** oder ***böswillige Manipulation*** entstehen.

9.6.1. Verhinderungsstrategien

- ***Statische Analyse*** zur Laufzeit
- ***Überprüfung zur Laufzeit*** *(Unser Approach)*

Folgendes ist zu Überprüfen:

- ***Korrekte Benutzung der Instruktionen*** *(Typen stimmen, Methodenaufrufe stimmen, Sprünge sind gültig, Op-Codes stimmen, Stack-Überlauf oder Unterlauf erkennen)*
- ***Typen sind bekannt*** *(Metadaten, Werte auf dem Evaluation Stack haben einen Typ)*

```
// type safe iadd
int right = CheckInt(Pop());
int left = CheckInt(Pop());
Push(left + right);
```

Weitere Sicherheitsmassnahmen

- Variablen immer ***initialisieren***
- ***Null-dereference*** und index-out-of-bounds Checks
- ***Kompatibilität*** von externen Verweisen
- ***Garbage Collection***

9.7. INTERPRETATION VS. KOMPILATION

Interpretation	***Kompilation***
Ineffizient, dafür ***flexibel*** und ***einfach*** zu entwickeln. Akzeptabel für selten ausgeführten Code.	Kompilierter HW-Prozessor Code ist ***schneller***, JIT-Compilation für Hot Spots für noch mehr Performance. Kompilation ***kostet***, Laufzeit macht es (allenfalls) wett.

10. OBJEKT-ORIENTIERUNG

Die wichtigsten Merkmale der Objekt-Orientierung kommen erst zur Laufzeit zum Tragen. Dazu gehören Themen wie ***Speicherverwaltung*** und ***Typpolymorphie***.

10.1. OO IM COMPILER

In der ***Entwicklung des Compilers*** wurden bereits einige OO-Features eingebaut:

Lexer

Der ***`new()`-Operator*** musste erkannt werden für die Instanzierung von Klassen, die Erstellung von Arrays sowie der `instanceof`-Operator. Das ***`this`-Keyword*** ist zwar reserviert, wurde im Lexer aber als Identifier behandelt und erst später im Semantic Checker weiterverarbeitet.

Parser

Hier wurden ***mehrere Konstrukte*** für die Objekt-Orientierung verwendet: ***Indirect Access Designators***, wie Member Accesses für Felder *(`car.seats`)* oder ***Methoden*** *(`car.drive()`)*, ***Type Casts*** *(`(Vehicle)car`)* und ***Type Tests*** *(`car instanceof Vehicle`)*, als auch die ***Definition von Basisklassen*** *(`class Car extends Vehicle`)* und Interfaces *(`class Lambo implements SportsCar`)*.

Semantic Checker

Die Designator- und Typenauflösung musste OO-Features beinhalten, wie die ***Auflösung von Member- oder Element Access Designators***, die ***`this`-Referenz*** und auch Typen für die ***Objekterzeugung***, ***Arrayerzeugung***, ***Feldzugriffe***, ***Methodenaufrufe***, ***Type Casts*** und ***Type Tests***. Eine ***spezielle Prüfung*** stellt fest, dass ***`this`*** ein ***reserviertes Schlüsselwort*** ist, dass nur ***gelesen*** werden kann.

Auch im ***Semantic Checker*** sind bereits ***OO-Features implementiert***: Zuweisungs-Kompatibilität der Typen, `null` kann jedem Referenztyp zugewiesen werden, Subklasse kann zu Basisklasse zugewiesen werden *(impliziter Upcast)*, Interface-Typ passt auf Klassen, welche Interface implementieren, Typüberprüfungen bei `a == b`.

Ebenfalls wird auf zyklische Vererbungen geprüft: `class A extends B {} class B extends A {}` ist nicht erlaubt. `extends` darf nur bei Klassen, `implements` nur bei Interfaces verwendet werden. Beim Overriding muss dieselbe Signatur und Rückgabetyp verwendet werden. Overloading ist in SmallJ nicht erlaubt.

```
class Base {
  void f(int x) {}
}
```

```
class Sub extends Base {
  int f(string s) {} // Signature doesn't match, not allowed
}
```

Code Generator

Im Code Generator sind ***mehrere Opcodes*** für die OO-Unterstützung nötig: ***`getfield`*** und ***`readfield`*** lesen bzw. schreiben Felder einer Klasse, ***`new`*** initialisiert neue Instanzen und ***`ldc null`*** lädt die `null`-Referenz.

Code	***Stack davor*** (links = oben)	***Instruktion***	***Stack danach***
`p.left`	`p`	`getfield left`	`value`
`p.left = value`	`value, p`	`putfield left`	—
`new Point()`	—	`new Point`	`reference` zu Point Obj.
`null`	—	`ldc null`	`null`

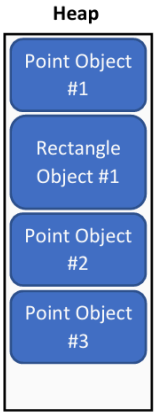
10.2. HEAP

Objekte werden in einem ***speziellen Speicher im Laufzeitsystem***, im sogenannten ***Heap***, gespeichert. Der ***Heap*** ist ein linearer Adressraum, in welchem die Objekte alloziert werden. Jedes Objekt benötigt einen ***individuellen Speicher*** für seine Felder und Typinformationen.

Der Heap ist ***unabhängig*** vom Call Stack. Anders als Parameter oder lokale Variablen, können Objekte normalerweise nicht auf dem Call Stack gespeichert werden, weil Objekte nicht zwingend an die ***Lebensdauer einer Methode gebunden*** sind. Dies nennt man ***Method Escape***.

```
Point getPoint() { return new Point(); }
void setup(Rectangle r) { r.topLeft = new Point(); }
// Points/Rectangles live longer than their creating method
```

```
a = new Point();
b = new Rectangle();
c = new Point();
d = new Point();
```



10.2.1. Allokation

Ohne Garbage Collector gibt es noch ***keine Freigabe / Deallokation***, deshalb wird Speicher einfach ***fortlaufend*** alloziert, bis der ***Heap voll*** ist. SmallJ unterstützt keine expliziten `free`/`delete` Statements und ist deswegen vollständig vom ***Garbage Collector abhängig***.

10.2.2. Deallokation

Es gibt keine ***Hierarchie*** unter Objekten und auch keine hierarchische ***Lebensdauer***. Eine ***Deallokation*** verursacht deshalb ***Lücken*** im Heap, welche mit dem Garbage Collector wieder aufgeräumt werden müssen.

10.3. OBJEKT-REFERENZEN

Auf Objekte wird in SmallJ immer durch eine ***Referenz*** verwiesen *(Call by Reference)*. Die Variablen speichern nur die Referenzen, d.h. die ***Speicheradresse*** des Objektes im Heap. Diese Referenz bzw. dieser Zeiger ermöglicht es, ***zum Heap-Block zu navigieren*** und auf dessen Inhalt zuzugreifen oder Methoden für dieses Objekt aufzurufen – sogenanntes ***Dereferencing***. Das Dereferencing von `null` führt zu einem Laufzeitfehler.

Referenzen können statisch sein oder vom Stack oder Heap kommen.
- ***Stack-Referenzen:*** Referenzen in lokalen Variablen, Parametern oder Evaluation Stacks
- ***Heap-Referenzen:*** Von Instanzfeldern oder Feldern in Arrays
- ***Statische Referenzen:*** In SmallJ nicht vorhanden
- ***Register-Referenzen:*** JIT-kompilierter Code

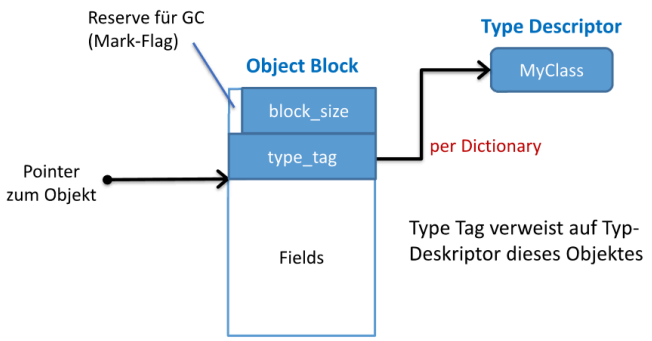
10.4. NATIVE MEMORY ACCESS

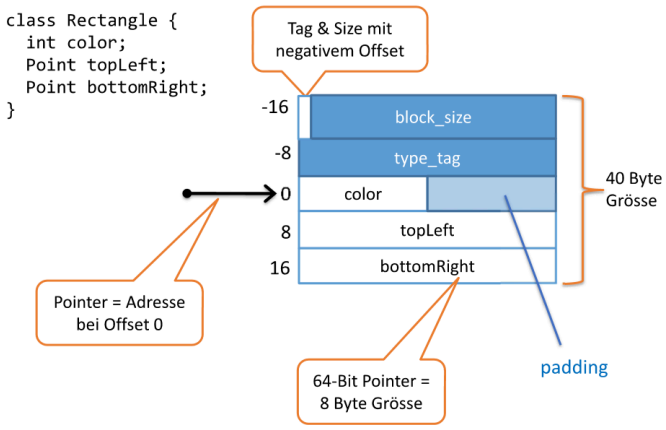
Die VM verwendet einen ***unmanaged Heap***, damit die .NET Runtime bei diesem nicht selbst die Garbage Collection ausführt. Der Heap wird mit 64-Bit Pointern adressiert und darf keine .NET-Referenzen speichern. Stattdessen findet das Mapping über ein `Dictionary<long, object>` statt.

```
ulong[] _heap = new ulong[HeapSize];
long value = Read(address); ... Write(address, value);
```

10.5. OBJEKTBLOCK-LAYOUT

Das interne Layout der Blöcke im .NET Array bzw. Heap wird wie folgt definiert. Jedes dieser Elemente wird durch einen eigenen Eintrag im Array repräsentiert, jeder Eintrag ist 8 Byte gross.
- ***Mark Flag:*** Aktuell noch leer, reserviert für GC
- ***Block Size:*** Gesamtgrösse des Heap Blocks
- ***Type Tag:*** Eine Referenz zum entsprechenden Class Descriptor, Mapping via .NET Dictionary
- ***Fields:*** Hier ist der Inhalt des Blocks gespeichert. Der Pointer zeigt direkt hierhin. Damit alle Felder gleich gross sind, werden sie mit ***Padding*** bis 64 Bit bzw. 8 Byte aufgefüllt.





11. TYP-POLYMORPHISMUS

11.1. VERERBUNG

Zur Vereinfachung implementieren wir nur ***single Inheritance*** *(Jede Klasse kann nur von einer anderen Klasse direkt erben)*

```
class Vehicle { /* ... */ }
class Car extends Vehicle { /* ... */ }
class Cabriolet extends Car { /* ... */ }
```

11.1.1. Code Reuse

Subklasse ***erbt*** Variablen & Methoden und ***erweitert*** Layout der Basisklasse.

```
class Vehicle { int model; int color; }
// type desc: Vehicle
```

```
class Car extends Vehicle { int wheels; }
// type desc: Car, beinhaltet auch model und
wheels
```

11.2. TYP-POLYMORPHISMUS

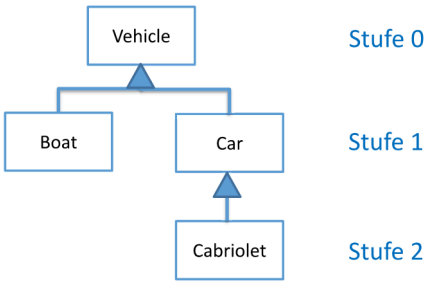
Objekt der Subklasse ist auch vom Typ der Basisklasse(n). Subtyp ist auf Basistyp ***zuweisungskompatibel***.

```
Vehicle v;      // Statischer = deklarierter Typ, definiert zu Compile-Time
v = new Car();  // Dynamischer = effektiver Typ, existiert zu Runtime
                // statischer Typ ≠ dynamischer Typ
```

11.2.1. Type Test & Cast

Für ***Downcasts*** sind ***dynamische Typchecks*** nötig, welche durch `instanceof` realisiert werden. ***Upcasts*** können immer statisch vom Compiler determiniert werden. `null instanceof Car` ergibt immer `false`, während `(Car)null` ohne Fehler erfolgt. Dies funktioniert, da `null` auf jeden Reference-Type zuweisbar ist.

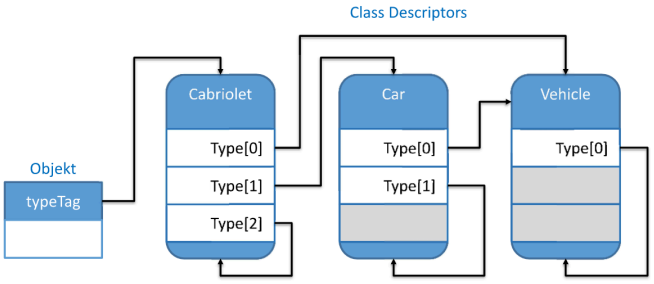
```
Vehicle v; Car c; v = ...;
// Type Test
// Dynamischer Typ von 'v' ist 'Vehicle' oder Subklasse
if (v instanceof Car) {
  // Type Cast
  // Dynamischer Typ von 'v' ist 'Car' oder Subklasse
  c = (Car)v;
}
```



Type Casts ***verändern*** den ***Typ*** des Objektes ***nicht***. Es reicht nicht aus, nur zu überprüfen, ob der dynamische Typ `Car` ist, sondern man muss ***die ganze Kette der Basistypen durchnavigieren***. Das ist jedoch ***ineffizient***. Deshalb gibt es ***fixe Stufen*** für die Vererbung.

11.3. ANCESTOR TABLES

Die ***fixen Stufen der Vererbung*** können für eine ***effizientere Basistypenerkennung*** verwendet werden. Jeder Class Descriptor beinhaltet eine ***Ancestor Table***, die eine ***Tabelle aller Basisklassen*** in der Vererbungsliste der Klasse ist. Der erste Eintrag stellt die oberste Vererbungsstufe dar. Die ***Ancestor Table*** beinhaltet immer auch die ***aktuelle Klasse***. So lassen sich die Basisklassen ***viel effizienter*** überprüfen. Die ***Anzahl*** der Vererbungsstufen ist jedoch typischerweise ***begrenzt***. Dieses System funktioniert nur bei ***single Inheritance***.



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

```
// implementation of 'instanceof'
var instance = CheckPointer(Pop());                    // pointer to the variable/object
var desc = heap.GetDescriptor(instance);               // dereference type tag of object
var target = CheckClassDescriptor(instruction.Operand); // type descriptor of target class
var level = target.AncestorLevel;                      // get ancestor level of target class
var table = desc.AncestorTable;
if (level ⩾ ancestorTable.Length || level > desc.AncestorLevel) {
  throw new VirtualMachineException("Invalid cast");
}
Push(table[level] == target); // check if type at the ancestor level is the same as the target

// additional code for 'checkcast' after the 'instanceof' code
if (!CheckBoolean(Pop())) { throw new VMException("Invalid cast") } Push(instance);
```

Ein ***Compiler*** kann gewisse ***unnötige Type Tests*** oder Casts ***eliminieren***, beispielsweise wenn der dynamische dem statischen Typ entspricht oder wenn ein ***expliziter Upcast*** vorhanden ist.

```
Vehicle v; Car c;
v instanceof Vehicle // dynamic == static type
c instanceof Vehicle // upcast to base class
```

Gewisse Programmiersprachen wie C# haben eine ***Root-Klasse*** *(`object`)*, von welcher alle anderen erben. Diese lebt implizit in der Ancestor Table auf Stufe −1 und der Compiler kann ***jeden Typ-Test*** zu dieser Klasse immer mit ***`true`*** beantworten, solange die Referenz nicht `null` ist. Auch ein Type Cast zu dieser Klasse ist immer erfolgreich.

11.4. VIRTUELLE METHODEN

In SmallJ sind alle Methoden ***Instanzmethoden*** *(nicht statisch)* und ***virtuell*** *(können überschrieben werden, overriding)*. Das bedeutet, dass eine Unterklasse eine ***gleichnamige Methodendeklaration*** *(mit derselben Signatur)* beinhalten kann wie die Basisklasse. In einem solchen Fall wird die Methode ***nicht neu deklariert***, sondern ***der Inhalt*** der geerbten Methode ***wird ersetzt***.

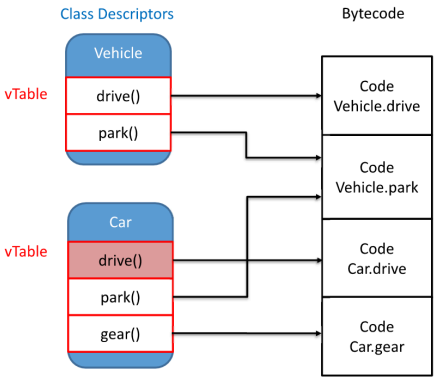
```
class Vehicle { void drive() { ... } void park() { ... } }
class Car extends Vehicle { void drive() { ... } /* überschreibt vehicle.drive() */ ... }
```

11.4.1. Dynamic Dispatch

Der ***dynamische Typ*** *(also Laufzeitsystem)* entscheidet, ***welche Implementation*** aufgerufen wird. Der einzige Zweck des statischen Typ ist, dem Compiler zu zeigen, dass eine deklarierte Methode existiert.
Dynamic Dispatch ist nützlich für die ***Erweiterbarkeit*** von Software, da es erlaubt, künftige Logik auf ***typsichere Art*** und Weise einzubauen *(Es ist garantiert, dass die aufgerufene Methode existiert, obwohl die Implementierung dynamisch ist)*.

```
Car c; c = new Car();
Vehicle v; v = c; // Dynamic Type = Car
v.drive();        // ruft Car.drive() auf
```

11.4.2. Virtual Method Table



Da virtuelle Methodenaufrufe sehr häufig sind, benötigen sie eine schnelle Implementation in der Runtime. Sie werden durch eine ***virtuelle Tabelle*** *(vTable)* realisiert. Jeder ***Klassendeskriptor*** hat eine Tabelle mit ***seinen enthaltenen virtuellen Methoden***. Jeder Tabelleneintrag ***assoziiert*** die Methodenimplementierung für die deklarierte Methode.

Im Beispiel existieren zwei Klassendeskriptoren, einer für `Vehicle` und einer für `Car`. ***`Vehicle`*** hat zwei Tabelleneinträge, da in dieser Klasse nur zwei Methoden deklariert sind; diese verweisen auf die ***ursprüngliche Implementierung***. Da `drive()` in `Car` überschrieben wurde, zeigt dieser Eintrag der vTable von `Car` nun ***nicht mehr*** auf `vehicle.drive()`, sondern auf die ***neue*** `drive()`-Logik in `Car`.

Lineare Erweiterung

Jede virtuelle Methode des Typs hat ***einen Eintrag*** in der vTable. Methoden der Basisklasse sind oben, neu deklarierte Methoden der Subklasse unten. Wird eine Methode ***überschrieben***, wird sie nicht unten angefügt, sondern ***ersetzt***. Somit hat jede virtuelle Methode eine ***fixe Position*** in der vTable. Diese ist im deklarierten Typ ***statisch bekannt***.

Ausführen von Virtual Method Calls

Durch die Virtual Table funktioniert das Aufrufen einer Virtuellen Methode in ***konstanter Zeit*** $(O(1))$. Die ***vTables*** und die dazugehörigen Positionen werden von der VM im Loader generiert.
Diese Schritte werden beim Aufrufen ausgeführt:

1. Der ***Type Tag wird dereferenziert***, um den Klassendeskriptor für den dynamischen Typ des Objekts zu erhalten
2. Der ***Klassendeskriptor enthält den Virtual Table***, in welchem der Eintrag mithilfe der bereits bekannten Position der deklarierten Methode nachgeschlagen wird *(`drive()` hat Position 0 im Typ `Vehicle`)*
3. Der ***entsprechende Method Descriptor wird aufgerufen***, um Informationen über Typen von Parameter, Return Type und Lokalen Variablen zu erhalten
4. Der ***Code*** hinter dem vTable-Eintrag wird ***ausgeführt*** *(overridden `Car.drive()` Implementation)*

11.5. INTERFACES

Mittels Interfaces kann in SmallJ auch eine ***eingeschränkte Mehrfachimplementierung*** angeboten werden. Interfaces stellen ***nur Methoden*** und ***keine Instanzvariablen*** zur Verfügung.

```
interface IA { void f(); }
interface IB { void g(); }
class Ab implements IA, IB { void f() { ... } void g(){ ... } }
```

Bei Interfaces würde wegen der ***Mehrfachimplementierung*** der gleiche Ansatz wie mit den vTables ***nicht*** funktionieren. Bei den ***vTables*** kann davon ausgegangen werden, dass jede Methode sowohl bei der Basisklasse als auch bei der Klasse selbst die ***gleiche Position*** in der Tabelle hat. Da eine Klasse jedoch von ***mehreren*** Interfaces erben kann, stimmt diese ***Reihenfolge nicht überein*** *(Eine Klasse `Ab` erbt von `IA` und von `IB` jeweils eine Methode, die Implementation der Methode von `IB` steht bei der Table der Klasse `Ab` an zweiter Stelle, im Interface `IB` jedoch an erster Stelle)*.

Interfaces müssen deshalb vom Loader ***global durchnummeriert*** werden. Pro Klassendeskriptor wird eine ***Interface-Tabelle*** *(iTable)* generiert. Sie enthält die Interfaces ***an der Stelle***, an welcher sich die Interfaces auch in der ***globalen Tabelle*** befinden. Nicht implementierte Interfaces sind an ihrer Stelle im iTable `null`.
Die Einträge in der ***iTable verweisen*** dann auf die ***vTable*** des ***jeweiligen Interfaces***.

```
Global Interface Table  class Ab implements IA, IB  class A implements IA  class B implements IB
IA → 0                  0 → IA                      0 → IA                  0 → null
IB → 1                  1 → IB                      1 → null                1 → IB
```

Wird eine Methode eines Interfaces ***gecallt***, wird vom Class Descriptor der Klasse auf die iTable verwiesen, wo das Interface von der ***iTable*** geholt und dort dem Verweis in der ***vTable*** auf die korrekte Methode gefolgt wird.

11.5.1. Type Test & Cast bei Interfaces

– ***Type Test:*** `y instanceof IA`
– ***Type Cast:*** `(IA)y`

Verwendet den gleichen Mechanismus wie oben: ***Prüfe***, ob der Interface Eintrag in der ***iTable vorhanden*** ist.

11.5.2. Speicherproblem

Da es eine ***globale Durchnummerierung*** der Interfaces gibt, entstehen bei den ***einzelnen Klassen*** lange ***iTables*** mit ***vielen Lücken*** durch nicht implementierte Interfaces. Somit kann es zu ***Speicherproblemen*** kommen.

Als Abhilfe kann man die iTables mehrerer Interfaces im Speicher ***kollisionsfrei*** übereinanderlegen. *(z.B. iTable S mit Eintrag bei Index 3 mit iTable T mit Einträgen 0 und 4 zu iTable S&T mit Einträgen 0, 3 und 4 zusammenlegen)*
Dafür muss man ***prüfen***, ob der Eintrag für den aktuellen Typ ***gültig*** ist. Dies geschieht mit einem ***Vermerk / Tag*** des Class Descriptors in der vTable.

Weiter ***optimieren*** lässt sich dieses Konzept, indem man die einzelnen iTables zu ***einer einzigen iTable zusammenfasst*** und alle Klassendesktriptoren dann auf einen ***unterschiedlichen Offset*** in dieser «globalen» iTable zeigen, ab welchem ihre implementierten Interfaces eingetragen sind.

12. GARBAGE COLLECTION & SPEICHERFREIGABE

Die ***Speicherfreigabe*** ist ein wichtiger Aspekt der Virtuellen Maschine. Es gibt ***drei verschiedene Arten*** von Speicher mit verschiedenen Verwaltungsoptionen:

– ***Metadaten:*** Keine Freigabe für Typdeskriptoren, Ancestor Tables, vTables etc. nötig
– ***Call Stack:*** Der Activation Frame wird bei Return aus der Methode automatisch freigegeben
– ***Heap:*** Muss mit Garbage Collection aktiv freigegeben werden, da Objekte keine hierarchische Lifetime besitzen

12.1. EXPLIZITE FREIGABE

`delete`-Statement zum manuellen Deallozieren eines Objektes *(Pendant zu new)*. Ist zum Beispiel in C oder C++ so gelöst.

```
x = new T(); ... delete x;
```

Probleme

Mit ***expliziter Speicherfreigabe*** kann es zu ***schwerwiegenden*** Speicherfehler kommen.

Dangling Pointers	***Memory Leaks***
Referenz auf bereits gelöschtes Element *(Kann nicht berechtigten Speicher lesen oder fremden Speicher überschreiben: Security + Safety Issues)* `x = new T(); x = y; delete x;` `// y now points to free space` `// or a different subsequent allocation`	Verwaiste Objekte, die nicht abräumbar sind *(Nicht löschbarer Garbage füllt den Heap)* `x = new T(); x = null;` `// The pointer has been cleared, but the object` `// is still present on the heap` `// unreachable → undeletable without GC`

12.2. GARBAGE COLLECTION

Das Laufzeitsystem kümmert sich um die ***automatische Freigabe*** von Garbage *(nicht mehr benötigte Objekte)*.
Nutzen: ***Memory Safety*** und eine ***Vereinfachung*** der Programmierung.

12.2.1. Garbage

Als Garbage werden Objekte bezeichnet, die ***nicht mehr erreichbar*** sind und daher ***nicht mehr gebraucht*** werden können. Der umgekehrte Fall ist jedoch etwas anders: Es kann Objekte geben, die ***nicht mehr benutzt*** werden, aber trotzdem noch ***erreichbar*** sind. Das System kann jedoch nicht «in die Zukunft schauen», um herauszufinden, ob das Objekt noch verwendet wird, und klassifiziert es darum nicht als Garbage.

12.2.2. Reference Counting

Mit ***Reference Counting*** wird durch einen ***Counter*** *(rc)* pro Objekt festgehalten, ***wie viele eingehende Referenzen*** auf dieses Objekt vorhanden sind. Ist `rc == 0`, ist das Objekt Garbage, die ***Umkehrung*** davon gilt aber ***nicht*** *(Zyklische Referenzen)*.

Bei ***jeder Zuweisung*** von Referenzen muss zusätzlich eine Referenz erstellt/verschoben werden, was ***sehr teuer*** ist.

```
x = y; // wird zu:
y.rc++; x.rc--; if (x.rc == 0) { delete x; } x = y; // rc++ first, in case the object is the same
```

Zyklische Objekte

Zyklische Objekte werden mit Reference Counting ***nie*** zu Garbage, weil alle Elemente im Zyklus ***eine eingehende Referenz*** haben, auch wenn ***keine Referenz von aussen*** mehr vorhanden ist. Ergibt ***Memory Leaks***. Die könnte mit ***weak pointers*** umgangen werden, welche vom Counter nicht mitgezählt werden. Diese können aber wiederum zu Memory Leaks und verfrühter Objekt-Löschung führen. Deshalb ist Reference Counting ***ungeeignet*** für den Garbage Collector.

12.3. GARBAGE COLLECTOR (GC)

Ein Garbage Collector bietet eine ***nachhaltigere Lösung*** für die Rückgewinnung von freiem Heap-Speicher ohne Dangling Pointer oder Memory Leaks. Ein GC ist ein ***automatischer Mechanismus***, welcher den ***Heap analysiert*** und Garbage nach einer nicht-deterministischen Verzögerung ***freigibt***.

12.3.1. Transitive Erreichbarkeit

Objekte, die das Programm ***noch verwenden*** könnte, dürfen ***nicht gelöscht*** werden. Das heisst, ausgehend von ***Ankerpunkten*** *(Root Set)* werden alle direkt und indirekt über Referenzen vom Programm erreichbaren Objekte als ***«Nicht-Garbage» markiert***. Die nicht markierten Objekte sind Garbage und können entfernt werden.

Root Set

Die Analyse der transitiven Erreichbarkeit muss bei bestimmten ***Referenzen der ersten Ebene beginnen***. Diese erste Ebene wird als ***«Root Set»*** bezeichnet und umfasst die folgenden Quellen des laufenden Programms:

- ***Statische Variablen:*** In SmallJ nicht vorhanden
- ***Call Stack:*** Referenzen in Parametern und lokalen Variablen, `this`-Referenz, Evaluation Stack
- ***Register:*** Bei JIT-Compiler relevant

Das Root Set wird in unserem Fall durch die ***Pointer auf dem Call Stack*** erkannt.

```
IEnumerable<Pointer>
GetRootSet(CallStack callStack) {
  var list = new List<Pointer>();
  foreach (var frame in callStack) {
    CollectPointers(frame.Parameters);
    CollectPointers(frame.Locals);

CollectPointers(frame.EvaluationStack);
    list.add(frame.ThisReference);
  }
  return list;
}
```

Eine Referenz kann sich nur auf dem ***Evaluation Stack*** befinden, wenn eine Heapallokation mit ***komplexen Ausdrücken*** ausgeführt wird *(z.B. `setUpVehicles(new Car(), new Cabriolet(), new Truck())`)*. Angenommen, der Heap hat keinen Platz mehr für den neuen `Truck`, befinden sich zu diesem Zeitpunkt bereits `Car` und `Cabriolet` auf dem Evaluation Stack. Diese Objekte ***leben bereits*** und werden nur vom Evaluation Stack ***referenziert***. Würde der Evaluation Stack sich also ***nicht*** im Root Set befinden, würde der GC diese Objekte ***fehlerhafterweise löschen***.

Auch die ***`this`-Referenz*** im Root Set ist essentiell: Das impliziert erstellte `main`-Objekt wird häufig nur vom `this` der `main()`-Methode oder anderen Methodenaufrufen innerhalb des `main`-Objekts referenziert.

12.3.2. Mark & Sweep Algorithmus

- ***Mark Phase:*** Markiere alle erreichbaren Objekte
- ***Sweep Phase:*** Lösche alle nicht markierten Objekte

```
void Collect() { Mark(); Sweep(); }
```

12.3.3. Mark Phase

Traversiere ausgehend vom Root Set alle ***durch Pointer erreichbaren Elemente*** und ***markiere*** diese mit dem ***Mark Flag*** als «Non-Garbage». Dieses Flag wird im Objektblock im ***reservierten Bereich*** vor der `blockSize` gesetzt.

```
void Mark() { foreach (var root in RootSet) { Traverse(root); } }
```

Die Traversierung wird mit ***Depth-First Traversal*** durchgeführt. Jedes ***erreichte Objekt*** wird ***markiert*** und anschliessend durch die von diesem Objekt ausgehenden ***Verweise durchiteriert***, um diese Objekte ***ebenfalls zu markieren***. Da Heap-Strukturen ***zyklisch*** sein können, werden ***bereits markierte Objekte übersprungen***.

```
void traverse(Pointer current) {
  long block = heap.GetAddress(current) - BLOCK_HEADER_SIZE;
  if (!IsMarked(block)) {
    SetMark(block); // Mark Flag im Objekt-Header
    foreach (var next in GetPointers(current)) { // Referenzen im Objekt enumerieren
      Traverse(next);
    }
  }
}
```

Pointer im Objekt

Der GC muss wissen, wo sich im Objekt ***Pointer*** befinden *(Felder eines Referenztypen, Elemente in einem Array eines Referenztypen)*. Um diese zu erfassen, kann der GC den ***Typdeskriptor*** über den Type-Tag des Blocks erhalten. Ist es ein ***Klassendeskriptor***, kann er die Felder mit Referenztypen so erkennen. Bei ***Arrays*** wird im Type Tag geprüft, ob es einen Referenztyp speichert, ansonsten ist kein GC nötig.

```
IEnumerable<Pointer> GetPointers(Pointer current) {
  var descr = heap.GetDescriptor(current);
  var fields = ((ClassDescriptor)descr).AllFields;
  for(var i = 0; i < fields.Length; i++) {
    if (!IsPointerType(fields[i].getType()))
continue;
    var value = heap.ReadField(current, i);
    if (value != null) yield return (Pointer)value;
  }
}
```

Rekursive Traversierung

Der GC braucht ***zusätzlichen Speicher*** für den Call-Stack, was problematisch ist, weil der Speicher beim Aufruf vom GC oft bereits ***knapp*** ist. Es ***existieren*** Algorithmen zur Traversierung ***ohne Zusatzspeicher***, wie z.B. ***Pointer Rotation Algorithmus*** von Deutsch-Schorr-Waite. Für uns reicht jedoch die rekursive Traversierung aus.

12.3.4. Sweep Phase

Lösche linear alle Elemente im Heap, die in der Mark Phase ***nicht*** markiert wurden. Weil diese Phase ***linear*** durch den Heap traversiert, werden ***alle Elemente*** besucht. Bei ***markierten Objekten*** wird das ***Mark Flag entfernt***, ***unmarkierte Blöcke*** werden wieder als ***freie Blöcke registriert***. Blöcke, welche bereits frei sind, werden nicht weiterverarbeitet und einfach so gelassen, wie sie sind.

```
void Sweep() {
  long current = HEAP_START;
  while (current < HEAP_SIZE) {
    if (IsGarbage(current)) { // !Marked && !
Free
      Free(current); // Register free bl. in
heap
    }
    ClearMark(current);
    current += heap.GetBlockSize(current);
  }
}
```

12.3.5. Free List

Freie Blöcke können ***nach der Garbage Collection*** überall im Heap ***verstreut*** sein. Dieses Phänomen wird ***externe Fragmentierung*** genannt. Durch das ***Freigeben von Speicher*** kann nicht mehr der simple Free Pointer verwendet werden, der Heap muss jetzt ***mehrere freie Blöcke verwalten***. Dafür wird die ***free list*** eingeführt, eine ***lineare Linked-List*** mit allen freien Blöcken. Vor der Sweep-Phase werden die freien Blöcke nach und nach in die am Anfang leere ***free list*** eingetragen. Jedes Element in der Free List beinhaltet im Objekt-Block ***anstelle des Type-Tags*** einen ***Pointer auf den nächsten Eintrag*** in der free list. So sind die einzelnen freien Blöcke ***linear miteinander verkettet***.

Neue Heap-Allozierung

Die ***Free-List*** wird ***traversiert***, bis ein ***passender Block*** gefunden ist. Ein allfälliger ***Überschuss*** des Blockes wird ***wieder*** in die Free List ***eingetragen***.

Strategien

- ***First Fit:*** Keine Sortierung, erster passender Block wird für die Allozierung verwendet $O(n)$
- ***Best Fit:*** Die freien Blöcke sind nach aufsteigender Grösse sortiert, um einen möglichst passenden Block zu finden. Dies führt zu unbrauchbar kleinen Fragmenten $O(n)$
- ***Worst Fit:*** Die Blöcke sind nach absteigender Grösse sortiert. Es wird sofort ein passender Block gefunden. Hohe externe Fragmentierung. $O(1)$ für das Allozieren und $O(n)$ für das erneute Einsetzen vom Rest des Free Blocks.
- ***Segregated Free List:*** Mehrere Free Lists mit verschiedenen Grössenklassen *(z.B. Blöcke mit 64-128 bytes, 128-196, etc.)*. Da eine konstante Anzahl von Listen verwendet werden soll, gibt es eine ***Overflow List***, in welche alle «zu grossen» Blöcke eingefügt werden *(z.B. alle Blöcke >= 32kB)*. Sehr kleine Blöcke *(< 64 bytes)* werden nicht wieder eingefügt, da sie zu klein für weitere Verwendung sind und dies ***interne Fragmentierung*** auslösen würde. Allokation und erneutes Einfügen eines Restblocks benötigen nur $O(1)$.
- ***Buddy-System:*** Blockgrössen sind 2er Potenz, falls kein Platz vorhanden wird ein grösserer Block in zwei kleinere aufgeteilt, falls Buddy bei Deallokation auch frei ist, werden beide Blöcke verschmolzen. Durch die exponentiellen Grösseklassen haben die meisten Blöcke aber unbrauchbare Reste, welche nicht weiter verwendet werden können, was ebenfalls wieder zu Interner Fragmentierung führt.
- ***Compacting Garbage Collection:*** Siehe Kapitel «Compacting GC» (Seite 32)

Viele der oben beschriebenen ***Nachteile*** können aber durch einen ***einfachen Trick mitigiert*** werden. Wenn die Heap-Blöcke ***ihre Grösse*** am Anfang und Ende ***speichern***, kann der benachbarte Block trivial bestimmt werden, der Heap ist dann quasi eine ***Double-Linked-List*** an Blöcken. So können Blockreste schnell mit potenziell freien benachbarten Blöcken ***verschmolzen*** werden *(der Vorgänger- und/oder Nachfolgeblock)* und von ihrer momentanen Free List in eine neue Grössenklasse verschoben werden.

12.3.6. Heap Fragmentierung

Um zu verhindern, dass der Heap immer weiter ***zerstückelt*** wird und schlussendlich alle freien Blöcke in der Free List zu klein sind, werden ***benachbarte freie Blöcke*** in der Sweep Phase ***miteinander verschmolzen***. Anschliessend müssen die Block Tags des vorherigen Blocks und des neu verschmolzenen Blocks entsprechend angepasst werden, damit Elemente wieder korrekt in der Liste eingetragen sind.

12.3.7. Ausführungszeitpunkt

Garbage wird ***nicht sofort*** erkannt und freigegeben *(Delayed Garbage Collection)*. Unmittelbare Deallokation war nur mit der Reference Counting-Methodik möglich. Der GC läuft spätestens, wenn der ***Heap voll*** ist. Jedes Mal beim ***Allozieren*** von Speicher *(new-Operator)* wird ***überprüft***, ob noch ***genug Platz*** verfügbar ist, falls nicht, wird der ***GC gestartet***. Er kann aber auch ***prophylaktisch früher*** gestartet werden, insbesondere bei ***Finalizer***.

12.3.8. Stop & Go

Der GC läuft ***sequentiell*** und ***exklusiv***. Das heisst, der ***Mutator*** *(Das produktive Programm)*, wird während dem Ausführen des GC's unterbrochen. So entsteht ein «Stop & Go»-Ablauf *(auch «Stop-the-World» genannt)*, in dem sich der Mutator und der GC abwechseln. Das ***Anhalten des Mutators*** ist essentiell für die korrekte Funktionsweise des Mark-and-Sweep-Algorithmus, da ansonsten ***während des GC-Durchlaufs*** der Heap ***verändert*** werden könnte, womit die Mark-Phase nicht mehr richtig funktionieren würde. Die Sweep-Phase könnte jedoch bei korrekter Synchronisierung *(Locks, atomic instructions)* gleichzeitig laufen *(Lazy Sweep)*.

Stop & Go ist allerdings ungeeignet für zeit- oder performancekritische Anwendungen, da die Ausführung des GC häufig nicht vorhergesagt werden kann und dadurch in für das Programm ungeeigneten Momenten laufen kann.

12.4. FINALIZER

Ein `finalizer` ist eine Methode, die ***vor dem Löschen*** eines Objektes ausgeführt wird. Kann für ***Abschlussarbeiten*** verwendet werden, wie das Schliessen von Verbindungen, Lösen von Concurrent Locks, etc. Wird vom ***GC initiiert***, wenn das Objekt Garbage geworden ist.

12.4.1. Separate Finalisierung

Der Finalizer wird nicht in der GC-Phase ausgeführt, sondern später. Gründe dafür können sein:

- ***Kann lange oder unendliche Laufzeit haben:*** Blockiert sonst GC, bei Stop-and-Go-GC sogar ganzes Programm
- ***Kann neue Objekte allozieren:*** Korrumpiert GC und/oder Heap
- ***Programmfehler im Finalizer:*** Kann zu Crash des GC's führen
- ***Resurrection:*** Ein Objekt, welches gerade finalisiert wird, kann eine Referenz auf ein noch lebendes Objekt besitzen. Durch diese Referenz kann der Finalizer einen auf das Garbage-Objekt zeigenden Pointer im lebenden Objekt kreieren, und damit der Garbage Collection entgehen. Ein wiederbelebtes Objekt lässt nicht nur sich selbst, sondern auch alle anderen direkt oder indirekt referenzierten Garbage im Objekt wiederbeleben. Dies ist allerdings eine komplett valide Operation in einer Runtime, also müssen GC's damit umgehen können.

12.4.2. Internals

Es gibt ein ***finalizer Set*** *(Registrierte Finalizer von allen lebenden Objekten)* und eine ***pending queue*** *(Alle Finalizer von Objekten, welche als Garbage identifiziert wurden und so schnell wie möglich laufen sollten)*.

Garbage mit Finalizer wird im die ***Pending Queue*** eingetragen. Die Pending Queue zählt als ***zusätzliches Root Set*** und somit wird das Element und alle seine Referenzen ***wiederbelebt***. Dies ist nötig, weil das Objekt für die Finalization ***erhalten bleiben muss***. Es ist also aus der Sicht des GCs eine ***zweite Mark-Phase*** innerhalb des selben Durchlaufs nötig. *(**Erste Phase:** Markiere und erkenne Garbage mit Finalizer, welche in die Pending Queue eingefügt werden. **Zweite Phase:** Eine weitere Mark Phase für die Pending Queue, welche jetzt im Root Set ist, wird durchgeführt. Dadurch werden alle Referenzen des Objekts mit dem Finalizer wiederbelebt.)*

Aus der Sicht des ***Objekts mit Finalizer*** sind ***zwei volle GC-Durchläufe*** nötig, um den Garbage vollständig zu entfernen. *(**Erster Durchlauf:** Finalizer wird der Pending Queue hinzugefügt. **Zwischen GC-Läufen:** Finalizer wird ausgeführt. **Zweiter Durchlauf:** Ausgeführter Finalizer ist nicht mehr in der Pending Queue, damit nicht mehr im Root Set und durch Sweep entfernt, falls immer noch Garbage)*

Nach dem Finalizer wird der `pending`-Eintrag ***gelöscht***. Da der Speicher bei sehr wenig freiem Platz eventuell ***nicht schnell genug freigegeben*** werden kann, lassen die meisten Systeme den GC ***prophylaktisch*** laufen, damit die Finalizer noch fertig laufen können, bevor der Heap-Speicher erschöpft ist.

Grundsätzlich ist es ***nicht empfohlen***, die Garbage Collection ***manuell*** auszuführen, da dies mit den Heuristiken des GCs konfliktieren kann und damit die Performance reduziert. Es kann aber ***vor zeit- und speicherintensiven*** Operationen nützlich sein, so viel freien Platz wie möglich zu schaffen.

```
// C# manual GC Run
GC.Collect();
GC.WaitForPendingFinalizers();
```

12.4.3. Probleme

- Die ***Reihenfolge*** der Finalizer ist ***unbestimmt***, da sie keine hierarchische Lifetime besitzen. Die Finalizer können also auch bei Vererbungen in Klassen in einer unerwarteten Reihenfolge laufen.
- Finalizer laufen ***beliebig verzögert***, oder möglicherweise gar nicht, wenn `main()` fertig
- Sie laufen ***nebenläufig*** zum Hauptprogramm, z.B. in einem eigenen Finalizer-Thread oder zwischen der normalen Programmausführung. Dies kann zu Concurrency-Problemen führen.
- Frage, ob der Finalizer nach einer Wiederbelebung erneut laufen soll. Dies ist in Java und C# nicht der Fall, in C# kann ein Finalizer aber mit `GC.ReRegisterForFinalize(this);` erneut ausgeführt werden.

12.5. WEAK REFERENCE

Weak References zählen ***nicht*** als Referenz für den GC, im Gegensatz zu den normalen ***Strong References***. Garbage mit Weak References kann aber noch solange erreicht werden, bis es abgeräumt worden ist. ***Nach Freigabe*** des verwiesenen Objektes wird die ***Referenz auf*** `null` gesetzt. Wird für Implementation von Caches verwendet.

Meist werden ***Weak References*** intern via ***Hash Table*** implementiert. Jede Weak Reference zeigt auf einen ***Tabelleneintrag***, welcher dann auf das Target Object zeigt. Diese werden dann in der Mark Phase des GCs ignoriert und in der Sweep Phase werden alle Einträge geleert, bei welchen das Target Object gelöscht wurde.

Short Weak Reference	*Long Weak Reference*
Wird zurückgesetzt, sobald der GC das Target Objekt als Garbage identifiziert hat, bevor der Finalizer läuft. Dadurch kann es im Finalizer zu Wiederbelebung des Targets kommen.	Wird zurückgesetzt, bevor der GC das Target Object definitiv löscht, also nach dem Finalizer. Hat das Objekt also noch verbleibende Finalizer oder wird wiederbelebt, zeigt die Weak Reference immer noch darauf.

12.6. COMPACTING GC

Durch Allozieren und Löschen entstehen ***viele kleine Lücken*** im Heap. Obwohl in Summe genug freier Speicher verfügbar wäre, kann es so sein, dass ein neues Objekt ***keinen Platz*** mehr hat, weil alle Lücken ***zu klein*** sind.

Der ***Compacting GC*** *(auch* **Mark & Copy GC** *oder* **Moving GC** *genannt)* ***schiebt die Objekte wieder zusammen***, sodass sich der gesamte freie Platz am Ende des Heaps befindet. Hier genügt wieder ein simpler Free Pointer, eine Free List wird nicht benötigt. Allokationen werden am ***Heap-Ende*** gemacht, bei Verschiebungen müssen alle Referenzen ***nachgetragen*** werden. Da dies im gesamten System geschehen muss, wird separat der gesamte Speicherplatz nach zu verschiebenden Referenzen durchsucht. Dies kann auf verschiedene Arten geschehen:

- ***Remembered Set:*** Speichert die Adressen aller eingehenden Pointer pro Objekt
- ***Forwarding Pointer:*** Ein zweiter Pointer zeigt auf die momentane Objektadresse. Zeigt dieser nach dem Ausführen des GCs nicht mehr auf das eigene Objekt, wurde das Objekt verschoben und der Originalpointer muss angepasst werden. Benötigen viel Speicher und jeder Pointer muss zwei Mal dereferenced werden.

Das System muss jederzeit die ***Positionen aller Pointer kennen***, da es diese aktualisieren muss. Da in C/C++ Zahlen zu Pointern und umgekehrt gecasted werden können, kann ein Compacting GC hier nicht realisiert werden.

12.7. INKREMENTELLER GC

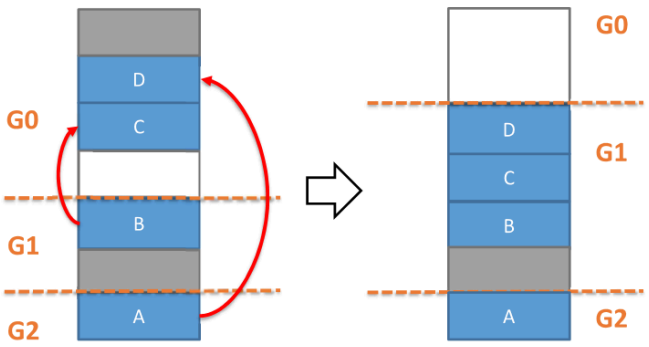
Ein ***Nachteil*** des «Stop-the-World»-GCs ist, dass die Programmausführung währenddessen ***pausiert*** werden muss. Dies ist für gewisse Anwendungsfälle nicht akzeptabel. In solchen Fällen werden ***inkrementelle GCs*** eingesetzt; diese sollen quasi-parallel zum Mutator laufen, indem sie den Mutator ***mehrmals kurz unterbrechen*** und in diesen ihre Arbeit ***inkrementell verrichten***.

12.7.1. Arten von Inkrementellen GCs

Generational GC

Junge Objekte werden ***schneller freigegeben***, weil diese tendenziell eine ***kurze Lebenserwartung*** haben *(Zeitspiegelungsheuristik)*. Es gibt 3 Generationen, deren Grenzen jeweils durch einen Pointer markiert werden:

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



Wird nun G0 aufgeräumt, werden alle ***nicht-markierten Elemente in G0 aufgeräumt***, und zusätzlich alle Referenzen von G1/G2, welche auf G0 zeigen. Die verbleibenden Blöcke werden ***eine Altersstufe nach oben***, zu G1, ***geschoben***.

Wenn eine alte Generation aufgeräumt wird, müssen auch die neueren mit aufgeräumt werden, um ***zyklischen Garbage***, welcher mehrere Generationen umspannt, zu erfassen. Dazu muss das System ***Reference-Writes*** in alten Generationen erkennen können. Dazu müssen ***Write Barriers*** in G1/G2 erstellt werden.

- ***Software-based write barriers:*** (JIT) Compiler, Loader, und/oder Interpreter fügen mit jedem Schreiben eines Pointers zusätzlichen Code ein, um potentielle Root Set-Kandidaten zu registrieren
- ***Hardware-based write barriers:*** Virtual Memory Management. Die betroffenen Memory Pages sind read-only, dadurch gibt es beim Schreiben einen Page Fault. Der Interrupt-Handler prüft dann auf das Schreiben von Referenzen

Partitioned GC

Heap wird in Partitionen zerlegt. Das Markieren wird ***nebenläufig*** durchgeführt, und in einer zweiten «Stop the World»-Phase werden verpasste Updates nachgeführt. Die Mark Phasen erstellen für jede Partition ein eigenes Root Set, welches alle Referenzen von anderen in diese Partition führen, das ***Remembered Set***. Damit diese Referenzen während der Mark Phase nicht verändert werden, sind auch hier `write` Barriers notwendig. Danach weiss der GC, wie viel Garbage jede Partition besitzt.

Der GC ***fokussiert*** zuerst auf ***Partitionen*** mit ***viel Garbage***. Die ***verbleibenden Objekte*** werden dann in eine neue Partition verschoben, was eine Anpassung der Referenzen wie im Moving GC notwendig macht. Die vorherigen Partitionen können nun in Gänze ***gelöscht*** werden. Zyklischer Garbage zwischen Partitionen benötigt jedoch immer noch einen vollständigen GC.

13. JIT COMPILER

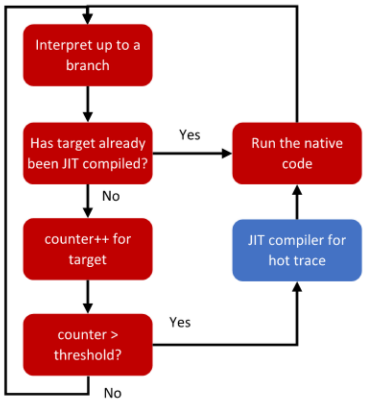
Der JIT-Compiler wandelt ***Bytecode*** direkt in ***nativen Code*** für den jeweiligen Prozessor um. So kann die Ausführung direkt gestartet werden, ohne dass noch eine Interpretation durch die Runtime stattfinden muss. Das führt zu einer ***viel effizienteren*** Ausführung. Macht jedoch nur für kritische Teile bzw. ***Hot Spots*** Sinn, welche ***oft ausgeführt*** werden, weil die JIT-Compilation ***ebenfalls Zeit kostet*** und sich nur lohnt, wenn der entsprechende Code ***oft ausgeführt*** wird *(z.B. Loops, oft aufgerufene Methoden)*.

Beispiel für Hotspot (Loop)

```
begin: load 1
       load 2
       icmplt
       if_false end
       load 1
       ldc 1
       iadd
       store 1
       br begin
end:   ...
```

13.1. PROFILING

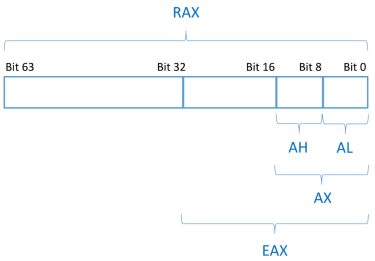
Der Interpreter ***zählt*** die Ausführung von gewissen Code-Teilen, um ***Hot Spots zu erkennen***. Wenn der Zähler einen bestimmten ***Schwellenwert*** überschreitet, veranlasst der Interpreter den JIT-Compiler für dieses Codestück. Ein ***Loop*** wird beispielsweise an ***jeder Jump-Instruktion*** auf Ausführung geprüft.



Ein einmal ***JIT-kompilierter Codeteil*** wird nach der Kompilierung ***gecached*** und beim erneuten Erreichen des Codes ausgeführt. JIT-Code, welcher noch viel häufiger ausgeführt wird, kann auch nach und nach mit immer stärkeren JIT-Optimierungen rekompiliert werden. Dazu werden wie beim regulären Code Profiling-Zähler in den Code eingefügt. Die Technik, zusätzliche Logik zum kompilierten Code hinzuzufügen, nennt sich ***Code Instrumentation***. Wir implementieren einen JIT-Compiler, welcher nur auf Methoden-Ebene arbeitet.

13.2. INTEL 64 ARCHITEKTUR

Instruktionen auf der x64-Prozessorarchitektur benutzen ***Register***, im Gegensatz zu unserer Stack-basierender VM. Es gibt ***14 allgemeine Register*** für Ganzzahlen, wir verwenden diese aber auch, um Booleans zu speichern:
`RAX, RBX, RCX, RDX, RSI, RDI, R8, R9, ..., R15`



Spezielle Register

RSP: Stack Pointer, ***RBP:*** Base Pointer, ***RIP:*** Instruction Pointer *(kann nicht explizit geschrieben / gelesen werden)*, diverse Floating Point Register

Die Intel x64-Architektur unterstützt auch ***Zugriffe auf die kleineren Register***, welche Teil der grösseren sind *(RAX 64-bit, EAX, 32-bit, AX 16-bit, AH/AL obere/untere 8-bit)*. Es wäre sogar performanter, die 32-bit SmallJ Strings in die 32-bit grossen Register zu speichern, der Einfachheit halber greift aber der JIT-Compiler nur auf die vollen 64-bit Register zu.

13.2.1. Instruktionen

Instruktion	*Bedeutung*
`ADD RAX, RBX`	`RAX += RBX`
`SUB RAX, RBX`	`RAX -= RBX`
`IMUL RAX, RBX`	`RAX *= RBX`
`IDIV RBX`	`RDX` muss vorher 0 sein bei nicht-negativem `RAX`, ansonsten −1. `RAX ≠ RBX, RDX = RAX % RBX`
`CDQ`	Vorzeichenbehaftete Konvertierung von `RAX` to `RDX:RAX` *(Convert to quad word: `RDX` wird zu 0 bzw. −1 je nach Vorzeichen von `RAX`)*

`IDIV`-Instruktion

Die ***`IDIV`-Instruktion*** ist auf mehrere Arten speziell: Sie führt ***gleichzeitig*** eine Division und eine Modulo-Operation durch. Die beiden Register `RDX:RAX` werden zusammen als 128-bit Zahl durch den Operand der Instruktion geteilt. Ist die Zahl ***positiv***, muss `RDX` 0 sein, ansonsten 1. Dies kann mit der ***`CDQ`-Instruktion*** gelöst werden, welche diesen Wert je nach Zahl in `RAX` setzt. Somit kann der JIT-Compiler `IDIV` ***vorbereiten***.

`IDIV` benutzt die fest vorgegebenen Register ***`RAX`*** für das Resultat der Division und ***`RDX`*** für das Ergebnis von Modulo. Dieses implizite Schreiben von Registern nennt man ***Register Clobbering***, da es die vorherigen Werte darin zerstört. Der Aufrufer muss daher ***sicherstellen***, dass die vorherigen Werte in diesen Registern ***gespeichert*** werden und sie, nachdem das Resultat auf den Evaluation Stack gepushed wurde, ***wiederherstellen***.

```
# Move current RAX/RDX values into
# registers previously checked as
free
MOV R8, RAX
MOV R9, RDX
CDQ          # Prepare RDX before
IDIV
IDIV RBX     # RAX = RDX:RAX / RBX
# Division in RAX, Modulo in RDX
# Push result on stack, restore RAX/
RDX
```

```
// IDIV implementation in JIT
Reserve(X64Register.RDX); // Free RDX
Force(X64Register.RAX, Pop());
var divisor = Pop();
_assembler.Cdq();
_assembler.Idiv(divisor);
Push(x64Register.RAX);
Release(x64Register.RDX);
Release(divisor);
```

Beispiel `(x - 1) / 3`

VM Bytecode

```
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 für IDIV vorbereiten
IDIV RBX      # RAX = (x - 1) / 3
# Resultat ist in RAX
```

13.2.2. Register-Allokation

Um die ***passenden Register*** für die Code-Fragmente zu bestimmen, müssen ***zwei Aspekte*** berücksichtigt werden:

- Die Registerwahl hängt von ***vorherigen Instruktionen*** ab und davon, wo diese Instruktionen die Werte platziert haben.
- Instruktionen können Register ***belegen*** und ***freigeben***.

Die Register-Allokation kann auf zwei Ebenen erfolgen:

- ***Lokale Register-Allokation:*** Nur für den Evaluation Stack. Jeder Eintrag des Stacks wird auf ein Register abgebildet. Der Register Stack entspricht dem Evaluation Stack. Pro übersetzte Bytecode-Instruktion wird der Stack nachgeführt.
- ***Globale Register-Allokation:*** Auch lokale Variablen und Parameter werden in Registern gespeichert. Die Abfrage der Variablen wird dadurch deutlich schneller, jedoch hat es nur eine begrenzte Anzahl Register, deshalb kann diese Allokation nur bedingt genutzt werden. `int`-Parameter werden oft als Register übergeben *(Windows C Calling Convention: RCX, RDX, R8, R9 – UNIX-like: RDI, RSI, RDX, RCX, R8, R9)*.

Bei komplexen Evaluation Stacks oder zu grosser Globaler Register-Allokation können die freien Register ausgehen, sogenannter ***Register Pressure***. Als Gegenmittel kann ***Stack Spilling*** angewendet werden: Registerwerte werden temporär auf dem Stack gespeichert. Alternativ kann auch globale Register-Allokation temporär ausgesetzt werden und die Werte werden an ihrem «normalen» Platz gespeichert. Wir setzen diese Massnahmen aber in unserem JIT-Compiler nicht um.

13.2.3. Intel Branches

Auf dem Intel-Prozessor basieren ***bedinge Verzweigungen*** *(Branches)* auf einem ***Bedingungscode***, der sich aus einem ***vorangegangenen Vergleich*** durch die `CMP`-Instruktion ergibt. Im Beispiel rechts werden zunächst ***`RAX` und `RBX` verglichen***. Wenn die Werte ***gleich*** sind, springt der Prozessor zur Zielposition, andernfalls fährt er mit der nächsten Anweisung fort. *(Zur Demonstration hier mit Label, in Bytecode mit relativem Jump Offset, siehe «Ausführung des Interpreter-Loops» (Seite 19))*

```
        CMP RAX, RBX
        JE target
        ....
target:
```

Instruktionen

Instruktion	*Bedeutung*
`CMP reg1, reg2`	Compare, speichert das Resultat in `RFLAGS`-Register
`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 equal, `reg1 ⩽ reg2`
`JMP label`	Unconditional Jump

Branch Code Template

Der ***Bytecode*** gibt ***vor*** dem Sprung an, welche Condition zum Sprung führt. Der ***Native Intel Code*** gibt die Condition jedoch erst ***beim*** Sprung an.

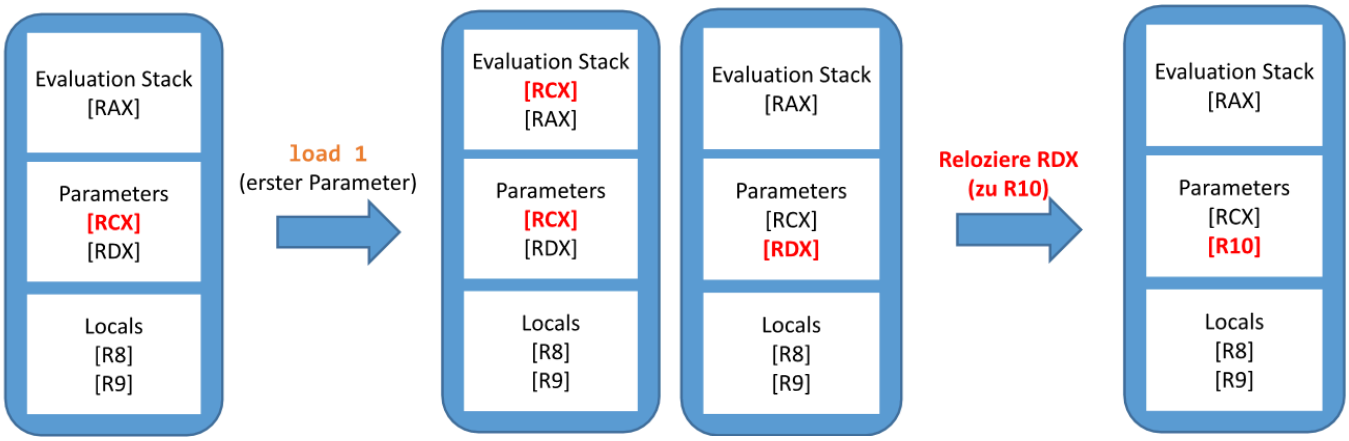
```
icmplt                  ⇒       CMP <left_register>, <right_register>
if_true <target>                JL <offset>
```

13.2.4. Register Buchhaltung

Wird ***globale Register-Allokation*** verwendet, muss nachverfolgt werden, in welchen Registern sich welche Variablen befinden. Die Werte in den Registern können die Inhalte von Parametern/lokalen Variablen oder Werte vom Evaluation Stack sein. Im ***Allocation Record*** wird aufgezeichnet, wo sich ein Wert befindet *(Evaluation Stack, Register oder Lokale Variable)*.

Soll beispielsweise der ***erste Parameter*** geladen werden, muss gar nicht die Instruktion `load 1` ausgegeben werden, sondern es kann einfach der ***Allocation Record angepasst*** werden. Wird die Windows Calling Convention verwendet, befindet sich der erste Parameter in ***`RCX`***, dieses Register muss nun einfach zuoberst auf den Evaluation Stack gepusht werden. Dies gilt ebenfalls für jeden Parameter und jede Variable, welche in einem Register gespeichert werden.

Die gleiche Technik kann auch dazu verwendet werden, um ***Register zu verschieben*** *(z.B. für `IDIV`)*. Es wird nur der Speicherort innerhalb des Allocation Records geändert. Dadurch wird das Zurückverschieben des Werts ebenfalls hinfällig.



In unserer JIT-Implementation haben wir verschiedene Helper-Methoden für den Umgang mit Registern:

Operation	*Effekt*
`acquire()`	Reserviere ein beliebiges freies Register
`release(reg)`	Gebe Register frei, falls nicht für lokale Variablen oder Parameter benutzt
`reserve(reg)`	Reserviere ein spezifisches Register (mittels Relokation falls nötig)

13.3. JIT COMPILER IMPLEMENTATION

In einem ***Switch Case*** werden alle Bytecode-Instruktionen abgearbeitet und die entsprechenden Assembly-Instruktionen ausgegeben. Viele Bytecode-Instruktionen können 1:1 gemapped werden. ***Unser JIT-Compiler*** ist limitiert auf simple Methoden und unterstützt der Einfachheit halber Methoden mit diesen Inhalten nicht: Methodenaufrufe, Systemaufrufe *(Heapallokation, Virtual Calls, Type Tests/Casts)*, Felderzugriff, Arrays, `this`-Referenzzugriff, Strings, nur bis zu 4 Parameter, limitierte Rückgabetypen *(nur `void`, `int`, `bool`)*.

```
switch (opCode) {
  ...
  case LDC:   var target = Acquire(); // get any free register
              var value = (int)instruction.Operand; // needs similar implementation for bools
              assembler.MOV_RegImm(target, value); // Move register immediate (variable)
              Push(target);
              break;
  case LOAD:  var index = (int)instruction.Operand;
              reg = IsParameter(index) ? allocation.Parameters[index - 1] : ... ;
              Push(reg);
              break;
  case STORE: var index = (int)instruction.Operand;
              var target = IsLocal(index) ? allocation.Locals[index-1-nofParams] : ...;
              var source = Pop();
              assembler.MOV_RegReg(target, source); // move register-to-register
              Release(source);
              break;
  case ISUB:  var operand2 = Pop();
              var operand1 = Pop();
              var result = Acquire();
              assembler.MOV_RegReg(result, operand1);
              assembler.SUB_RegReg(result, operand2);
              Release(operand1);
              Release(operand2);
              Push(result);
              break;
  ...
}
```

13.3.1. Ausführen von nativem Code in .NET

Der Code des JIT-Compilers muss sich in einer speziellen ***Virtual Page*** befinden, welche eine explizite ***Code-Ausführungserlaubnis*** besitzt. Diese ist nicht Teil des .NET-Heaps. Der JIT-Compiler umgeht einen ***Sicherheitsmechanismus***, dass Code in Pages nicht vom Prozessor ausgeführt werden darf *(erschwert Remote Code Execution)*. In .NET wird der Code in einen Delegate verpackt *(`Marshal.GetDelegateForFunctionPointer<>()`)*. Dieser führt den Code dann ausserhalb der .NET Runtime aus, was auch bedeutet, dass der JIT-Code ***nicht*** mit Standard-IDE-Mittel debugged werden kann.

13.3.2. Inkonsistente Allokation bei Branches

Wenn ***mehrere Branches*** zum ***gleichen Ziel*** führen, kann es passieren, dass verschiedene Registerzuweisungen verwendet werden. Diese ***nicht eindeutige Allokation*** kann ***schwerwiegende Probleme*** verursachen. Deshalb müssen die Allokationen mit ***`MatchAllocation()` vor jedem Branch abgeglichen*** werden. Die ***Speicherorte*** der Werte werden damit an die Orte angepasst, die von der Location nach dem Sprung erwartet werden.
Diese Verschiebung soll ausgeführt werden, wenn…

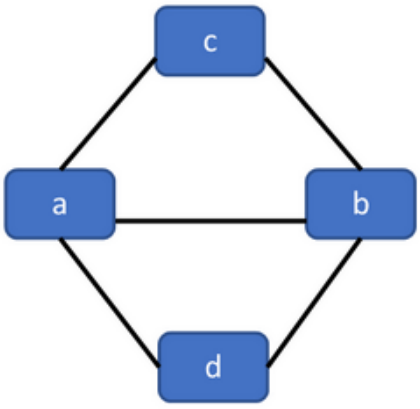
- ein ***Conditional Branch*** bevorsteht *(`if_true, if_false`)*
- ein ***Unconditional Branch*** bevorsteht *(`goto`)*
- an ***jedem Start eines Branches***, der durch die vorherige Instruktion erreicht werden kann *(Start von `if, while` etc.)*

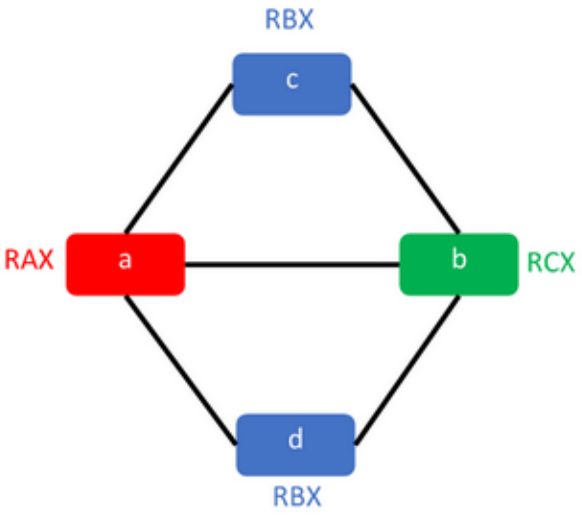
13.4. OPTIMIERTE GLOBALE-REGISTER-ALLOKATION

Da es ***nur 14 Register*** gibt, sollten wir genau abwägen, welche Variablen in diese gespeichert werden sollten. Um den ***maximalen Nutzen*** der globalen Register-Allokation zu erhalten, sollten ***zwei Aspekte*** beachtet werden: Die ***Anzahl Aufrufe*** einer Variable sowie ihre ***Lifetime***. Ersteres lässt sich leicht mit einem Usage Counter feststellen, so werden z.B. Variablen in Loops eher ausgewählt.

Für die Erkennung der Lifetime benötigen wir einen ***Register Interference Graph***, um festzustellen, ob eine Variable noch gültig ist. Falls nicht, kann sie durch eine andere lebende Variable ersetzt werden. Der Graph stellt alle Variablen dar und zieht jeweils eine Kante zwischen zwei Variablen, wenn sie zum selben Zeitpunkt leben, also überlappende Lifetimes besitzen.

```
a = 1;
b = 2;
c = a + b;
writeInt(c);
d = a - b;
writeInt(d);
writeInt(b);
writeInt(a);
```





Sobald zwei Variablen ***keine Kante*** haben, können sie sich ein Register teilen, da sie nie zum selben Zeitpunkt leben. Da wir so wenig Register wie möglich verwenden wollen, haben wir ein ***Graph Coloring Problem***, ein NP-Problem, für welches ***kein optimaler Algorithmus*** existiert. Es existiert jedoch ein ***pragmatischer Algorithmus***, welcher akzeptable Resultate liefert:
Für k verschiedene Register entferne alle Nodes mit weniger als k Kanten und weise sie dem nächsten freien Register zu. Die verbleibenden Nodes können nicht in k Register untergebracht werden. Deswegen wählen wir einen der verbleibenden Nodes und legen ihn auf den Stack *(Register Spilling)*. Dann wird dieser Node entfernt und der Algorithmus fährt weiter.

13.5. JIT ASSEMBLER & LINKER

Der JIT benötigt ebenfalls einen ***Assembler*** und einen ***Linker***. In unserem JIT-Compiler werden die Instruktionen nach der Intel 64 Architektur-Spezifikation encodiert. Der Linker patched die ***Adressen im Instruktionscode***, z.B. das Linken von Adressen zu statischen Variablen oder Methodenaufrufziele, die bereits zur JIT-Kompilierzeit bekannt sind. Diese ***statischen Adressen*** können dann direkt in die entsprechenden Instruktions-Operanden eingefügt werden. Ebenfalls fügt der Linker aneinanderliegende JIT-Blöcke zusammen, damit nicht mehr unnötig zum Interpreter zurückgesprungen werden muss.

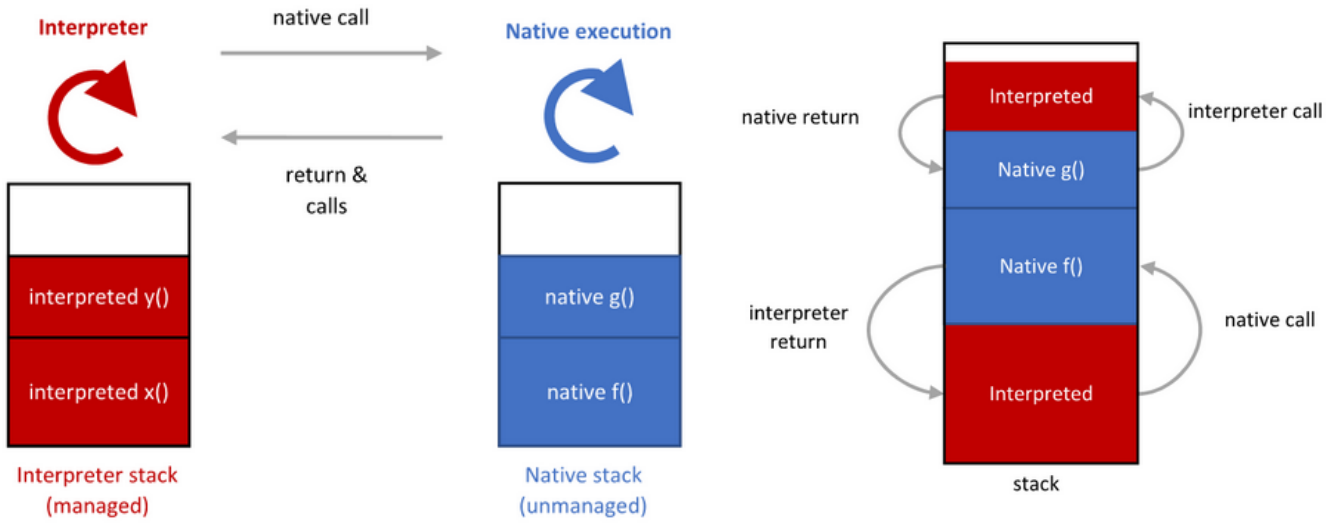
In unserer SmallJ-VM verwenden wir nur einen simplen Assembler ohne Linker, da wir weder statischen Variablen noch Methodenaufrufe innerhalb von JIT-Code unterstützen.

13.6. NATIVE STACK-VERWALTUNG

Während der ***Interpreter*** einen ***managed Stack*** verwendet, benötigt der ***JIT-Code*** einen ***nativen Stack***, welcher aus einem eigenen Speicherblock besteht, auf welchen der ***Base- und Stack-Pointer zeigen***.

Eine ***effizientere Vorgehensweise*** ist es, wenn beide Code-Arten sich denselben Stack teilen. Dafür müsste der Interpreter-Code aber ebenfalls einen nativen Stack verwenden, wie in Abschnitt «Unmanaged Call Stack» (Seite 21) beschrieben. Der ***Interpreter*** und ***JIT-Code*** verwenden dann ***abwechselnd*** diesen Stack, wobei beide Komponenten ihren spezifischen Activation Frames reservieren. Wird von einer Methode returnt, wird automatisch in der ***richtigen VM-Komponente*** fortgefahren.

Dadurch wird auch der ***Wechsel*** zwischen dem Interpreter und dem JIT-Code vereinfacht: Das System kann einfach den obersten Activation Frame auf dem Stack ersetzten und mit der jeweiligen Komponente fortfahren. Diese Technik wird ***On-Stack-Replacement (OSR)*** genannt.



13.7. ZUSÄTZLICHE JIT-FUNKTIONEN

JIT-Compiler in anderen Sprachen müssen auch ***System Calls*** *(Heapallokation, Virtual Calls, Type Tests/Casts)* handhaben können. Auch sollte der GC ***nativen Code*** aufräumen. Dazu kann der JIT-Compiler weitere Deskriptoren generieren, welche die Orte von Pointern auf dem Call Stack und in Registern aufzeichnen.