## Kompiliervorgang

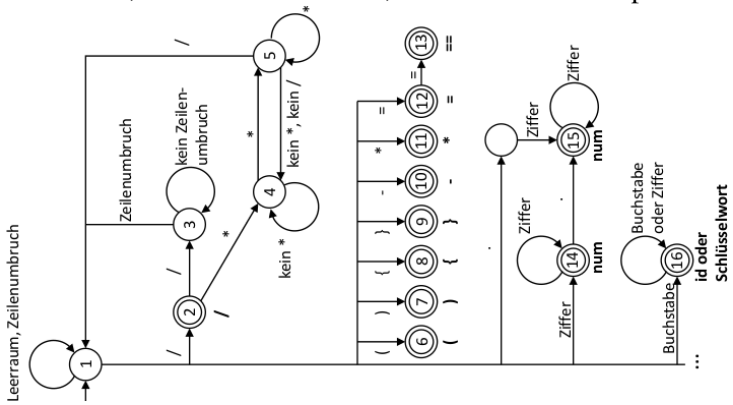
Legende: *optional* *Artefakt* → *Verarbeitung*→

*Zeichenstrom* → *Scanner (Lexikalische Analyse)*→
*Token-Strom* → *Parser*→
*Ableitungsbaum* → *AST Generierung & Semantische Analyse*→
*Abstract Syntax Tree (AST)* → *Zwischencode generieren/optimieren*→
*Zwischencode* → *Maschinencodegenerierung*→
*Maschinencode* ↔ *Maschinencodeoptimierung*

} Symboltabelle

### Scanner:

Erkennt Token für Parser (Schlüsselwörter, Bezeichner, Zahlen,. . . )
Whitespaces, Kommentare, Präprozessoranweisungen,.. beeinflussen Tokenerkennung
Implementierung: Angepasster DFA (Neustart nach Token, erkennt längstes Mögliches Token, Schlüsselworttabelle, Fehler wenn weder passende Kante noch Endzustand)




### Parser:

Rekonstruiert Ableitungsbaum (bzw. den reduzierten AST) gemäß der Grammatik der Programmiersprache.

- LL( k ) (Links-nach-Rechts Linksableitung); Vorschau von k Zeichen (v.a. k = 1)
  Parsen Top-Down:
  Start: Keller enthält Startwort
  Ende: Keller und Eingabe sind leer ($\varepsilon$)
  1. Predict: betrachte die vordersten k Zeichen und wähle **die** passende Regel aus der Grammatik.
  2. Match: Entferne übereinstimmende Terminale aus dem Keller und der Eingabe.
- LR (Links-nach-Rechts Rechtsableitung); mächtiger als LL Parser
  Parsen Bottom-Up:
  Start: Keller ist leer ($\varepsilon$)
  Ende: Startsymbol im Keller, Eingabe leer
  1. Shift: Lade nächstes Zeichen in den Keller
  2. Reduce: wende wenn möglich eine Regel der Grammatik an.

### Semantische Analyse:

Statische Bindungen (Bezeichner-Objekt → Typ-Objekt)⇒ Symboltabelle und AST
*Symboltabelle*: Sammelt Definitionen/Deklarationen von Objekten und damit auch:
Objektarten: Namensraum, Typ, Methode/Funktion, Parameter, Variable, Konstante
Bindungen: Typ, Adresse, Sichtbarkeit, innerer Gültigkeitsbereich
Gültigkeitsbereiche als Baumstruktur, Mehrdeutige Namen als eindeutiges Symbol

### AST:

Reduzierter Ableitungsbaum:

- keine Satzzeichen (desugaring)
- Operation Elternknoten, Operanden Kinder
- Verkettung der Anweisungen
- Deklarationen in Symboltabelle
- Namen verweisen auf die Symboltabelle

*Typprüfung*: Typen werden im AST propagiert

- Typprüfung
- Typinferenz (fehlende Typen in Symboltabelle eintragen)
- Auflösen von Überladungen und Literalen Konstanten
- implizite Konversionen erkennen
- generische Typen instanziieren

### Zwischencode:

Ableitungsbaum - Transpiler (Source-to-Source)
AST - "Lowering": Neue Konstrukte durch alte darstellen (z.B. Iteratoren)
Maschinenunabhängige Optimierung z.B. Function Inlining, Simple constant propagation, loop-unroll, ...

### Maschinencode:

Symboltabelle um Adressen erweitern (auch Stackpointer relative)
Registerallokation, Auswahl und Anordnung von Befehlen
*Maschinenabhängige Optimierung*: Architekturabhängige Befehle/Adressierungen
Cache Coherence
Keyhole-Optimierung: Folgen von Befehlen durch schnellere ersetzen (z.B. *4 durch shift-left 2)

### Linken:

statisch: Bibliotheken u. Laufzeitsystem => nach Kompilieren in die Binary gepackt
dynamisch: separate Bibliotheken; Linking Loader bindet vor Ausführung im RAM (dll)

### Laufzeitsystem:

Zur Ausführung nötiger Code (der Sprache) z.B. für:
Code-Verifikation, JIT, Exceptions, Garbage-Collector, Linken zur Laufzeit

## Speicherverwaltung

**Lebensdauer**:

*Global*: unbegrenzt

*Stack*: allokation/freigabe mit funktionsaufruf/rückgabe

*Heap*: Anlegen: Blöcke mit längenangaben, verkettet, werden nach speicher durchsucht => explizite reservierung/freigabe (Bei fehlern: Memory Leak/dangling reference)

**Zugriff**:

Symboltabelle (von Funktion/Struct) speichert offset

**Alignment**:

*primitive Typen*: Adresse muss durch Größe teilbar sein (z.B. Pointer: x8-Adressen)

*Zusammengesetzte Typen (Struct)*: Alignment vom größten Element;
padding auf nächste Alignte größe, auch am ende (z.B. int, bool => x4)
Struct/Class: Reihenfolge wird nicht verändert.
Frame: ggf. durch umordnen optimiert

**Größen in Byte**:

| Sprache | bool | char | int | float | double | Pointer(x64) |
|---|---|---|---|---|---|---|
| C++ | 1 | 1 | 4 | 4 | 8 | 8 |
| C# | 1 | 2 | 4 | 4 | 8 | 8 |
| Javascript | 1 | | | | 8 | 8 |

**RTTI**:

Dynamischer Typ (u.a. Vererbung) => Zeiger auf Typobjekt im Deskriptor
Typobjekt: Virtual Table (Adressen der **virtuellen** Methoden)
In C#/Java: zus.: Verweis auf Obertyp und andere Metadaten

## Stack

Aufbau Stackframe(x64 Windows): geregelt im ABI (Application Binary Interface):
Aufruf erzeugt neuen Stackframe (groß genung für alle parameter) und lädt parameter.
Aufbau:

Unterfunktionsframe,
**Achtung x16 Alignment**

| Shadowspace (wenn Funktionsaufruf)<br>**4** x 8 Byte |
|---|
| ...<br>Variablen (aligned)<br>ggf. Padding<br>... |
| Return Adress<br>(8 Byte) |

Aufruferframe,
**Achtung x16 Alignment**

**Parameterübergaberegister** :
RCX, RDX, R8, R9

**Rückgaberegister**:
RAX

**Caller saved**:
RAX, RCX, RDX, R8, R9, R10, R11
(Unterprogramm kann überschreiben)

**Callee preserved**:
RBX, RBP, RDI, RSI, RSP, R12...R15
(Unterprogram muss sichern(Prolog) und restaurieren (Epilog))

## Speicherfreigabe/Garbage Collection

**manuell**:

freigabe durch programmierer (C++: delete)

**Reference Counting**:

Zähler für daraufzeigende Pointer, löschen wenn zähler=0
Probleme: Zyklen sind selbsterhaltend
=> schwache pointer: Beeinflussen Counter nicht -> existenzprüfung!

*manuell*: Programmierer muss zähler selbst erhöhen/reduzieren (Bsp: COM)

*automatisch*: Smartpointer (shared_ptr / weak_ptr, unabhängig davon unique_ptr)
Implementierung (C++) Handle mit Zeiger, #ptr und #wptr

**Garbage Collection**:

*Wurzeln*: direkt verwendbare Objekte: Register,Stack,globale Variablen
Lebendige Objekte sind direkt oder indirekt von Wurzeln erreichbar, Rest Garbage
Vorraussetzung: Zeiger müssen durch Metadaten gefunden werden (Funktionen werden anhand von Rücksprungadresse identifiziert, Objekte mit RTTI)

*Copy GC /Scavenge/ Stop& Copy*: Halbierung Speicher in from-space (genutzt) und to-space (ungenutzt); Tausch der Rollen nach jedem Vorgang
1. Alle direkt von Wurzeln erreichbaren Objekte => in den to-space kopieren; Eintragen der neuen Adresse an alter Speicherstelle (wird in allen Zeigern darauf eingetragen)
2. Dasselbe mit allen von den Objekten im to-space erreichbaren Objekte. (Breitensuche)
Vorteil: Laufzeit linear zu Anzahl gültiger Objekte; kompaktifiziert
Nachteil: hälfte des Speichers nicht nutzbar

*Mark-Sweep-Compact*:
1. Tiefensuche lebendiger Objekte (von der Wurzel aus); Markierung im Mark-Word des Objekts oder globaler Tabelle; Forward-zeiger auf zukünftige Speicherstelle;
2. Korrigieren aller Zeiger (durch Forward-Zeiger)
3. Verschieben des Objekts auf den Forward-Zeiger, zurücksetzen Mark-Word
Aufwand: Mark-Phase linear zu anzahl lebendiger Objekte, Sweep-Compact linear zu Heap-Größe;

*Mark-Sweep*: Analog zu Mark-Sweep Compact ohne 3. Schritt;
Vorteil: keine Zeigerkorrekturen nötig
Nachteil: Fragmentierung des Speichers

*Mehrgenerationen GC*: kurzlebige temporäre Objekte, langlebige Geschäftsobjekte
=> Aufteilen des Heaps in Generationen (0-...); Nach überleben mehrerer GCs verschieben in höhere Generation; GC höherer Generationen nur wenn GC unterer Generationen nicht ausreichend. Gen 0: Copy GC; darüber Mark-Sweep(-Compact)
*Schreibbarriere:* Protokollieren der Zeiger Gen 1 auf Gen 0 pro Card (8kB Block) in Card Table (1Byte pro Karte). => Traversieren nicht durch Generation 1 sondern nur in den markierten Cards.
=> Markierung und Zeigerkorrektur schneller, Zeigerupdates langsamer

## Funktionen

*Statische Kette*: nichtlokale Variablen in darunterliegenden Stackframes, implementierung durch Pointer

*Closure*: wird bei übergabe/speichern von Funktion gebildet. (Lambda)
Besteht aus Funktionszeiger und zeiger auf Heap-Objekt mit den gefangenen Variablen
Löst upward Funarg problem(Verweis auf nicht mehr existierende stackframes)
Ggf. teil der statische kette
C#: Func <T1,. . . , Result>, Action<T>, Predicate<T>, Lambda
C++: 2 Varianten:
capture-by-copy: gefangene Variablen werden kopiert: [=](int a) {return a+4;}
capture-by-reference: Closure erhält pointer auf variablen: [&](int a) {return a+4;}

*Parameterübergabe*: Call by Value: Wert wird kopiert
"Call by Sharing": Call by Value einer Pointervariable ( => Swap nicht möglich)
Call by Reference: Zeiger auf übergebene Variable
Call by const Reference (c++): analog Call by reference, verbot schreibzugriff
Call by name: unausgewertete Übergabe durch implizite(explizite) Closures

*Methoden*: statische Methoden werden zu normalen Funktionen aufgelöst,
nicht-virtuelle Methoden zu Funktionen mit this-Pointer als 1. Parameter,
virtuelle Methoden werden zur Laufzeit über die virtual Table im Typobjekt ausgewählt.

## Konversionen

| ⇔ | |
|---|---|
| Explizit Conversion<br>Cast | Implizit Conversion<br>/Typanpassung /Coercion |
| Converting Cast:<br>Bits werden umgerechnet | Non-converting Cast:<br>Bitmuster bleibt |
| Narrowing:<br>keine Teilmenge (int->float) | Widening:<br>Zieltyp ist umfassender |
| Checked Conversion:<br>Prüfung zur Laufzeit, ggf. Exception | Unchecked Conversion:<br>ohne Prüfung |

## Arrays

*Adressrechnung*: a mit größe n1,n2,n3
a[i1,i2,i3] => a[0] + (i1 * n2* n3 + i2* n3 + i3) * sizeof(type);

*Prüfen Adressgrenzen*: C#: Deskriptor mit Längenangaben für jede Dimension

## Iteration

*Äußere Iteration*: Foreach schleife: von außen über die Collection

*Innere Iteration*: Collectionfunktion: Übergabe von Closure, ausführen für alle Elemente

*Generatoren*: Erzeugung von Iteratoren mit yield => nächster Einsprung erfolgt hinter dem letzten yield, kontext bleibt erhalten

*Tail-recursion*: wenn rekursion letzter schritt, dann goto statt funktionsaufruf
Allgemein: Tail Call Optimization: funktionsaufruf als letzter schritt => kein neuer Stackframe sondern nur den eigenen anpassen.

## Konstanten

*Compilezeitkonstanten*: feste Adressen / immediate Werte

*Laufzeitkonstanten*: (readonly,final)

*Immutable Typen*: bieten keine setter an, variablen können umverzeigert werden.

## Exceptions

Ersatz von Fehlercode-Rückgabewerten durch Exceptions;
Globale Tabellen ordnen Programmzählerbereiche entsprechenden Handlern zu;

*Stack unwinding*: Werfen einer Exception => Stack nach unten durchlaufen bis gehandelt

## Bindungen

*statisch*: zur Kompilezeit in Symboltabelle
z.B. int i = 32; Achtung: auch Typinferenz (var in C#)

*dynamisch*: zur Laufzeit im Speicher z.B. Werte, virtuelle Methoden, Typen
z.B. Javascript, dynamic in C#
=> Ducktyping