Grundlagen

- Universalität (muss Turing mächtig sein)
- Implementierbarkeit: Korrekte Programme müssen ausgeführt werden können.
- Syntax: Form (Anordnung von Zeichen, Ausdrücken...)
- Semantik: Bedeutung (Verhalten)
- Pragmatik: Zweck (wie, von wem, wozu wird Sprache verwendet)

Kategorien

imperativ:

- prozedural z.B. C
- objektorientiert z.B. Java, C#
- Skriptsprachen (interpretierbar, dyn. typisiert) z.B. JavaScript, Python

deklarativ:

- funktional z.B. LISP, Scala
- logisch z.B. Prolog
- Domain Specific Language (DSL) z.B. SQL, XAML

Kompilierzeitpunkt

Früh:
:Vorteil: Leistungsfähigere Geräte, 1x Übersetzen spart Ressourcen
aber: Information über Zielplattform nötig **Spät**:
:Vorteil: geht immer, einfacherer Compiler/Interpreter, besserer Code möglich da vollständige Information über Hardware, OS, Bibliotheken etc.

- AOT (Ahead-of-Time): Entwicklung, Server/Store, Installation

- JIT (Just-in-Time): Direkt vor/während Ablauf
  Ablauf: AOT in Zwischencode für hypothetische Maschine (Aufwändige Schritte wie Syntaxanalyse, Typprüfung)
  JIT von Zwischencode in optimierten Maschinencode => Performanter als Interpreter, Leichtere Validierung, i.d.R. verzögerter Start

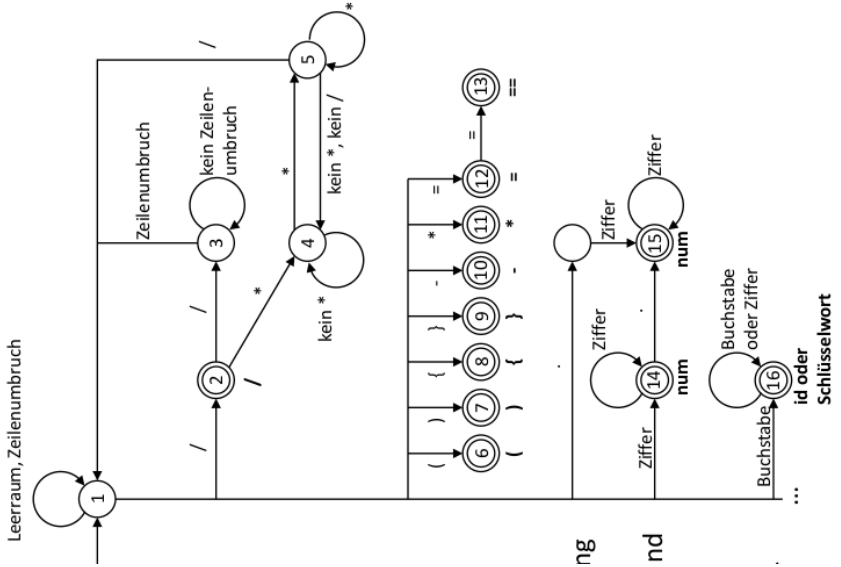
Kompiliervorgang

Zeichenstrom
→ *Scanner (Lexikalische Analyse*→
Token-Strom
→ *Parser*→

Ableitungsbaum
→ *AST Generierung & Semantische Analyse*→
Abstract Syntax Tree (AST)
→ *Zwischencode generieren/optimieren*→
Zwischencode
→ *Maschinencodegenerierung*→
Maschinencode
→ *Maschinencodeoptimierung*→

} arbeiten mit Symboltabelle

Scanner:
Erkennt Token für Parser (Schlüsselwörter, Bezeichner, Zahlen,...)
Zeilenumbrüche, 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 (ε)
  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 (ε)
  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)
⇒ erzeugt 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)
Zwischencode - Maschinenunabhängige Optimierung z.B. Function Inlining, Simple constant propagation, loop-unroll, ...
Maschinencode - Maschinenabhängige Optimierung
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 => werden in die Binary gepack
dynamisch: Bibliotheken/Laufzeitsystem sind seperat und werden vor ausführung vom Linking Loader im RAM gebunden
Laufzeitsystem:
Zur Ausführung nötiger Code (der Sprache) z.B. für:
Code-Verifikation, JIT, Exceptions, Garbage-Collector, Linken zur Laufzeit
Exceptions
Ersatz von Fehlercode-Rückgabewerten durch Exceptions;
Implementierung: Tabellen im globalen datensegment ordnen Programmzählerbereiche entsprechenden Handlern zu;
Stack unwinding: nach werfen der Exception wird stack nach unten durchlaufen bis fehler behandelt **Bindungen**
Namensbindung, Typbindung, Wertbindung, Adressbindung
anonym vs Namensbindung
statisch (zur Kompilezeit in Symboltabelle) vs dynamische Bindung (zur Laufzeit im Speicher z.B. Werte, virtuelle Methoden)
scope
lexikalischer Scope: bindung an den umgebenden Block.
freie Variablen: keine lokale bindung (nicht in diesem Block) Funktionen sind Closures wenn alle freien Variablen nicht-lokal gebunden sind
Speicherverwaltung

Lebensdauer:
Global: unbegrenzt
Stack: allokation/freigabe mit funktionsaufruf/rückgabe
Heap: explizite reservierung/freigabe (Bei fehlern: Memory Leak/dangling reference)

Stack:

Aufbau Stackframe(x64 Windows): geregelt im ABI (Application Binary Interface):
Aufruf erzeugt neuen Stackframe (groß genung für alle parameter) und lädt parameter.
Aufbau:
(16bit aligned):

Shadowspace (8 Byte)
Shadowspace (8 Byte)
Shadowspace (8 Byte)
Shadowspace (8 Byte)
... Variablen (aligned) ggf. Leerräume ...
Return Adress (8 Byte)

Parameterübergaberegister: RCX, RDX, R8, R9

Nächster Frame,
Achtung x16 Alignment
Rückgaberegister: RAX
Caller saved: RAX, RCX, RDX, R8, R9, R10, R11
(werden potentiell vom Aufrufer überschrieben)
Callee preserved:(RIP), RBX, RBP, RDI, RSI, RSP, R12...R15
(müssen vom Unterprogram gesichert(im Prolog) und restauriert(im Epilog) werden

Heap:

Blöcke mit längenangaben, verkettet, werden nach ausreichend speicher durchsucht

Umgebung bei lokalen Funktionen:

Statische Kette:: nichtlokale Variablen in darunterliegenden Stackframes, verfolgen von entsprechenden Pointern
Closure:: wird bei übergabe/speichern von Funktion gebildet. Besteht aus Funktionszeiger und zeiger auf Heap-Objekt mit den gefangenen Variablen Löst upward Funarg problem(Verweis auf nicht mehr existierende stackframes) Bilden auch statische kette

Kontrollfluss

Iteratoren:

Äußere Iteration: Foreach schleife: von außen über die Collection *Innere Iteration*: Map/Reduce Funktionen: Collection wendet Closure auf alle Elemente an
Generatoren: Erzeugung von Iteratoren mit yield => nächster Einsprung erfolgt hinter dem letzten yield, kontext bleibt erhalten **Schleife vs Rekursion**:
optimierung: 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.

Konversionen

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

Konversionen

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

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

Enum vs Symbol
Enum: statische Typen;
Symbole: dynamische Typen: werden nicht deklariert, bilden 1 Typ z.B. Symbol.for("Monday") in JS6
Collections::
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#: neben RTTI wird Deskriptor mit Längenangaben für jede Dimension angelegt
Konstanten:
Compilezeitkonstanten: feste Adressen / immediate Werte
Laufzeitkonstanten: (readonly,final)
Immutable Typen: bieten keine setter an, variablen können umverzeigert werden.
Speicherverwaltung

Alignment: :
primitive Typen: Adresse muss durch Größe teilbar sein (z.B. Pointer nur an x8-Adressen)
Zusammengesetzte Typen (Struct): richtet sich nach größtem Element (z.B. int, bool => x4)
Struct/Class: Reihenfolge wird nicht verändert.
Frame: durch umordnen optimiert
Zugriff:
Symboltabelle (von Funktion/Struct) speichert offset
RTTI:
Dynamischer Typ (u.a. Vererbung) => Zeiger auf Typobjekt im Deskriptor (eindeutige Kennung)
Typobjekt enthält virtual Table (adressen der methoden)
Heapspeicherverwaltung (löschen)
Probleme: Memory Leak, Dangling References, Garbage **manuell**:
freigabe durch programmierer (C++: delete)
Reference Counting:
Zähler für daraufzeigenden Pointer, löschen wenn =0
Probleme: Zyklen sind selbsterhaltend
=> schwache pointer: Beeinflussen Counter nicht
Varianten:
manuell: Programmierer muss erhöhen/reduzieren (Bsp: COM)
automatisch: Smartpointer (shared_ptr / weak_ptr, unabhängig davon unique_ptr)
Implementierung: 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, ansonsten 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)
Ablauf:
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); tausch der Rollen beim nächsten Vorgang
Vorteil: Laufzeit linear zu anzahl gültiger Objekte; kompaktifiziert
Nachteil: hälfte des Speichers nicht nutzbar
Mark-Sweep-Compact: Ablauf:
1. Tiefensuche lebendiger Objekte (von der Wurzel aus); Markierung im Mark-Word oder Tabelle; Forward-zeiger auf zukünftige Speicherstelle;
2. Korrigieren aller Zeiger
3. Verschieben 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