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

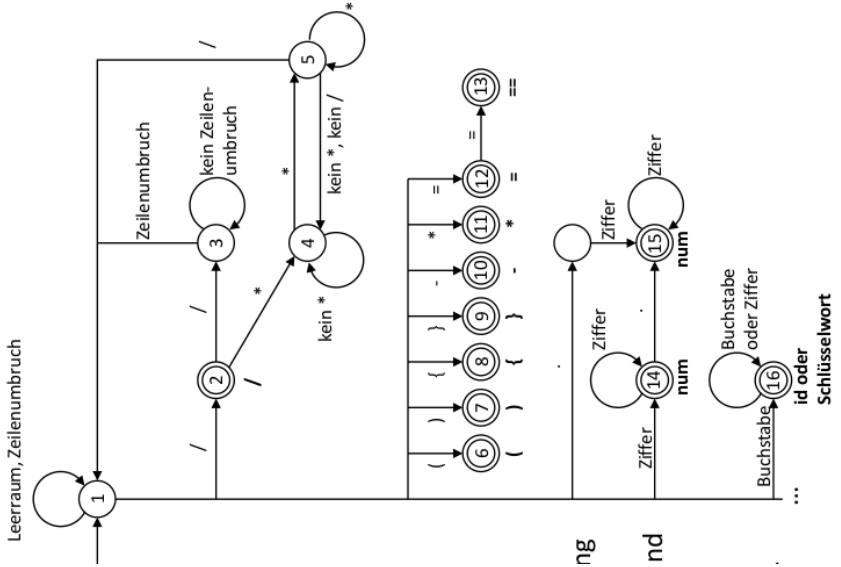
Kompiliervorgang

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:

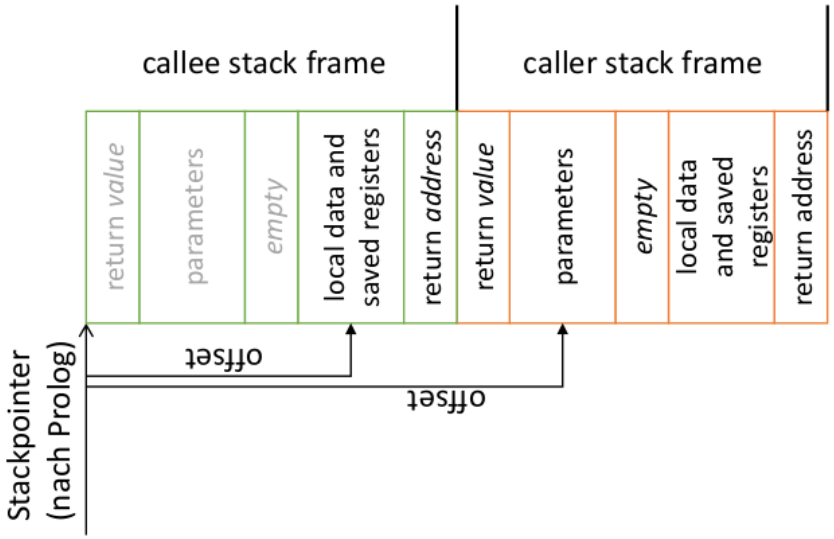
- keine Satzzeichen (desugaring)
- Operation Elternknoten, Operanden Kinder
- Verkettung der Anweisungen
- Deklarationen in Symboltabelle
- Namen verweisen auf die Symboltabelle

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
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):
Aufrufer erzeugt neuen Stackframe (groß genung für alle parameter) und lädt parameter.



Aufbau:

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