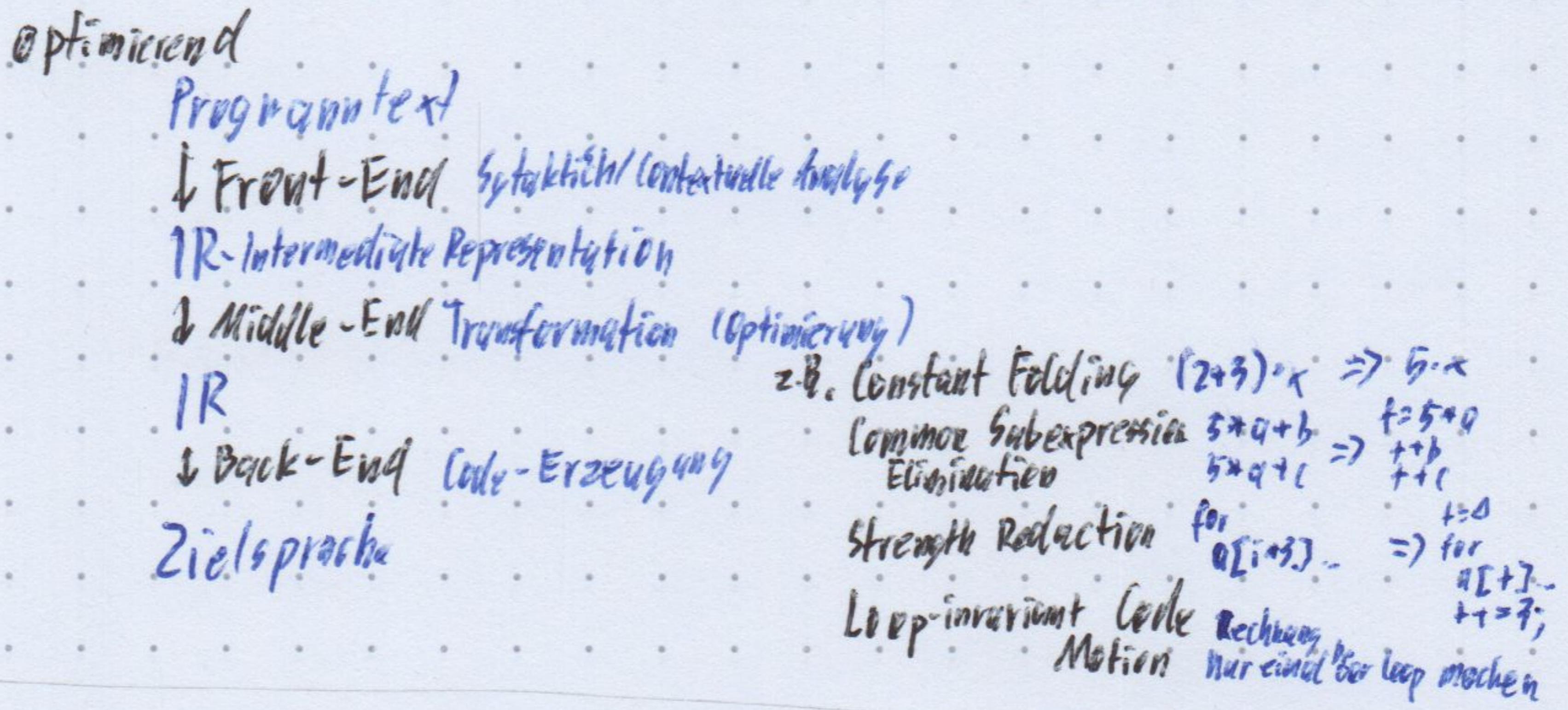
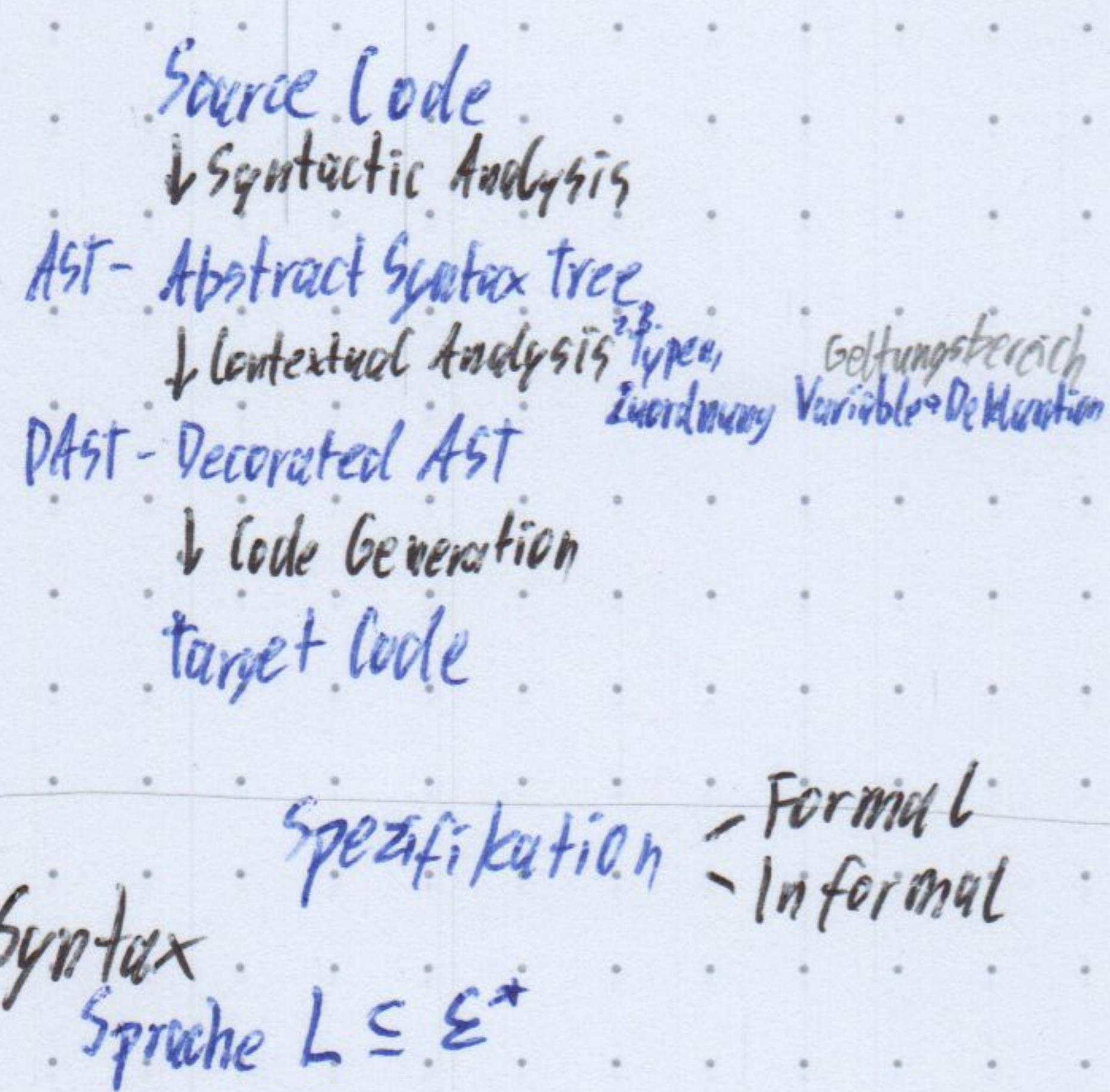


Phasen



Angabe:

scher. zu handhaben
 mathematische Mengennotation $L = \{x^n y^n \mid n \geq 0\} \cap a, b\}$ - schneller zu handhaben
 Reguläre Ausdrücke | für Alternativen clevere Werte
 * 0 oder mehrere Komma (...) zum Bräppieren - nur für reguläre Sprachen

für Syntax \rightarrow Kontextfreie Grammatiken (CFGs)

Menge Terminalsymbole T aus Alphabet

Nicht-Terminalsymbole N

Startsymbol S EN

Menge Produktionsregeln P

- BNF - Backus-Naur-Fom

Nicht-Terminal ::= Zeichenkette aus T und N

extended BNF (nicht regular expressions Bsp. Mini-Triangle slides 09 S. 48)

für sinnvolle praktische Anwendung eindeutig z.B. durch expression ::= primary Expression | Expression operator primary Expression

\Rightarrow Syntax Bäume geordneter Baum

markiert Blätter mit Terminalsymbolen

Knoten Nicht-Terminalsymbolen

Knoten N Kinder $x_1, \dots, x_n \Rightarrow N ::= x_1 \dots x_n$ EP

konkrete Syntax keiner Einfluss auf Semantik z.B. oder $Z := E$

\Rightarrow Abstraktes Syntax Tree (AST)

schlüsselwörter, Begrenzer irrelevanter Orientierung an Subphrase-Struktur

häufig High-level IR von Zielarchitektur unabhängig

+ weitreichende Analyse (Vervielfachung Anweisungen, Änderung Struktur)

- schlecht für Maschinenanalyse (Registerspezifische Befehlsfolgen)

N-Baum NEN als Wurzel

Anweisungen ausgeführt

Ausdrücke ausgewertet

Deklarationen Allokieren Initialisieren

Setzen fest
 Ändern Werte
 Ein-/Ausgabe

Kontextuelle Einschränkungen

sichtbarkeit von Bezeichnern
 Gültigkeitsbereiche (scope)

Zuordnung Verwendung \mapsto Bindung (Deklaration)

Typen

statische Typisierung zu Compilezeit
 dynamische Ausführung

jeder Wert Typ

operator Anforderungen Typ des Operanden
 Regeln für Typ des Ergebnisses

Kompilierung

dominiert compiler aufbau
Pass / Durchgang durch komplettes Programm
In Phasen - Transformationsschritte

z.B. Pascal
Ein-Pass \leftrightarrow Multi-Pass Compiler
+ Laufzeit
+ Speicher für große Projekte
- nicht für alle Sprachen

z.B. Java muss multi-Pass

+ Modularität, Flexibilität
+ Globale Optim.

Syntaktische Analysen

source

↓
Scanner
Token Stream

Parser
LAST

Token - atomares Symbol Quellprogramm
Keywords, Klammern, Identifier, keine Leerzeichen
Literale, Operatoren, Kommentare
Typspezif. sonst meist nur Token Typ interessant
Sourceposition = for debugging
nicht Spelling

Parsing - Erkennung Bestimmung Phrasen-Struktur
Erkennung ob Satz Grammatik
Eindimensionalität der Grammatik

Top-Down
rekursiver Abstieg
benötigt LL(k) Grammatik
↳ Links nach rechts
linkste Nicht-terminal
expandieren - Aktion
↑ Testen ob danach noch Zeichen

Bottom-Up
Aktionen

Shift - lese Zeichen k lege auf Stack
Reduce - tausche oberste Stack Elemente
durch RHS Produktions
teilweise Zustand erforderlich
z.B. schon gesehen jetzt Objekt

+ mächtiger als LL(k)
- komplexer, schlechter verständlich

Parse mit rekursivem Abstieg LL(1)

Scanner provides currentToken
accept(t)
acceptIf(t)

main() { parse(); checkEndOfFile(); }

parse(X) { für jedes Nicht-terminal

↳ keine Anwendung body

↳ accept(t);

P → parse(P);

PQ → parse(P);

parse(Q);

parse(P);

else if (currentToken t ∈ disset({P}))

parse(P);

else if (currentToken t ∈ disset({Q}))

parse(Q);

else syntax Fehler

LL(k) Parser

Annahme bis 1992
exponentieller Aufwand bei k>1

aber Worst-Case kann
in der Regel vermieden werden

z.B. ANTLR LL(k) bis LL(*)
Java LL(1)

+ gut automatisierbar

returns AST N - Attribute
Extends für alle Teile
AST Sonderfall Terminal-knoten
Spelling bei Blättern h)

P* → while (currentToken ∈ starterset({P})) {
 parse(P); // Unterscheidung
 links/rechts-assoziativ}

+ gut automatisierbar

z.B. JFlex/JFLex Scanner ähnelt Parser über Tiefen

Token Def. durch RE ohne Rekursion

als endlicher Automat mit
Transitioen mit Eingangssymbolen

Rekursiver Abstieg

currentChar: scan(): nächster Token
take(t) Scann(t) für jede Produktions
takel(t) $t \in X$

scanh() {
while (currentChar == whitespace, komma, ...)
 scanSeparator();
 currentSpelling;
 TokenKind currentKind = scanToken();
 return new Token(kind, spelling);

scanToken() {
switch (currentChar) {
case ' ': take();
 return TokenKind.ID;

if (spelling reserviertWord)
return ...

BRUNNEN

Trennung Identifier - Dez keyword während scan fixen

Syntaktische Analyse

Compiler Driver
↓
Syntactic Analysis
Contextual Analyser

Compiler Driver
↓
Syntactic Analyser Contextual Analyser Code Generator

Grammatik-Transformationen

Gruppierung $x ::= a \ x ::= b \Rightarrow ab$

Linksauflösung $X \mid X \mid Z \Rightarrow X(X \mid Z)$

Beseitigen Linkssanktion $N ::= X \mid NY \Rightarrow X Y^*$

Ersetzen von Nicht-terminalsymbolen wenn $N ::= X$ einzige
in LHS mit ϵ

aber nützlich zur Dokumentation

Links nach rechts legen
von rechts/rückwärts zusammenfassen
LR(k)-Technik

Theorie

starters [[EBNF-Ausdruck]] = Menge Terminalsymbole die an Anfang stehen können

starters[[ε]] = {ε}

+ = +

$X^* = \{ \text{starters}[X] \mid \text{falls } \epsilon \text{ aus } X \}$

$XY = \text{starters}[X] \cup \text{starters}[Y]$

$X \mid Y = \text{starters}[X] \cup \text{starters}[Y]$

$X^* = \text{starters}[X]$

follon [[EBNF-Ausdruck]] = Menge Token die in Grammatik stehensymbol → leere Menge darauf folgen können innerhalb RHS → Ausdruck rechts davon Ende/Nicht-terminal → rechte Seiten vorkommen Nichtterminal

$\text{dirset}[[\text{EBNF-Ausdruck}]] = \{ \text{starters}[X] \mid \text{falls } \epsilon \text{ herleitbar aus } X \}$

$\text{starters}[X] \cup \text{follon}[X]$

Für LL(1) muss gelten

$X \mid Y \text{ nichts zu } \epsilon \Rightarrow \text{starters}[X] \cap \text{starters}[Y] = \emptyset$

$\epsilon \text{ zu } \epsilon \Rightarrow \text{starters}[X] \cap \text{disset}[Y] = \emptyset$

$\Rightarrow \text{disset}[X] \cap \text{disset}[Y] = \emptyset$

$X^* \Rightarrow \text{starters}[X] \cap \text{follon}[X^*] = \emptyset$

if (spelling reserviertWord)
return ...

Kontextuelle Analyse
müssen deklariert sein
Bezeichner bekommen Bedeutung durch Kontext

Typ

Art Konstante, Variable, Prozedur, Funktion, Wert-Parameter,...
Sichtbarkeit
Anderes z.B. synchronized, static, volatile
Code Erzeugung: Adresse

Geltungsbereiche beschrieben durch Block Konstrukte
max 1 Deklaration pro Geltungsbereich

Blocks & Frakturen

Mono lithisch

↓ L0 1 globaler Block Bezeichner $\frac{m}{n}$ Eintrag
retrive(id)
enter(id, attr)

Flach

↓ Tiefen 2 (lokal, global) Bezeichner $\frac{m}{n}$ Eintrag
verschachtelter überlappende flache Blöcke $\frac{+}{+}$ openScope()
vermehr lokaler Kontext nach Block close

Verschachtelt

↓ beliebige Tiefe Bezeichner $\frac{m}{n}$ Einträge
Modellierung als Baum z.B. Impl. mit Stack
immer nur 1 Pfad sichtbar

tiefster
Eintrag
first

Typ - Einschränkung Interpretation Speicherbereich / Programmierkonstrukt

+ Fehlervermeidung
+ Laufzeitoptimierung

statische Typisierung

↳ Ausdrücke miss-typisiert
↳ statischer Typ-T ohne Evaluation bestimmt (Polymorphismus)

viele Sprachen haben auch dynamische Typprüfung

Implementierung kontextuelle Analyse

Versuche

Object-Oriented - Ansatz

AST-Subklassen

rekursive check, encode, prettyPrint function

param and return-type Object

+ easy adding

Types

Functional Ansatz - ignoriert one check method eingebauten Fallunterscheidung nach Klasse Dispatcher

+ easy adding functions

Visitor Pattern - neue Operationen ohne Klassenänderungen + vielen Operationen

public abstract class ASTF

 public abstract <RetTy, ArgTy> RetTy visit(Visitor<RetTy, ArgTy> v, ArgTy arg);

 public class XYZ extends ASTF

 public <RetTy, ArgTy> RetTy visit(Visitor<RetTy, ArgTy> v, ArgTy arg) {

 return v.visitXYZ(this, arg);

 }

 public interface Visitor<RetTy, ArgTy> {

 RetTy visitXYZ(XYZ x, ArgTy arg);

 }

 ↓ mehr Typsicherheit

 public abstract class VisitorBase implements Visitor {

 //impl alle Methoden, Exception werfend

BRUNNEN

Teilschritte

Identifikation - Zuordnung Bezeichner ~ Def

Symboltabelle für schnelles Abrufen

Eintragen

→ z.B. binärer Suchbaum, Hash-Tabelle (mit Stk)

Map<String, Stack<Attribute>> ident;

stack<List<String>> scopes;

Attribute Liste IDs

Wie Typ Speichern?

Explizit

Imperativenum

Objektorientiert:

- Lang

durch kontin.

viele Typen

auch eigen

→ Verweis auf TypDef. in AST

(AST-basierte Attribute)

Attribut

Typprüfung für statische

Bottom Up

Blätter Typ bekannt (literal in Bezeichner)

interne Knoten aus Typregeln und Typ Kinder

enthält Anforderungen

solange Bindung innerer ver Verwendung

zusammen in 1 Pass machbar z.B. Tiefensuche von links nach rechts

Ergebnis DAST

Expression
Type
Identifier
Declaration

- von Klassenstruktur abhängig
+ add Operationen
durchbricht Kapselung
- Param, Return-Type reicht offen

public class Checker implements Visitor<AST, AST> {

 public void check(Program prog) { prog.visit(this, null); }

//Impl of all Visitor Methods

public class Checker {

 check(Program prog)

private class CommandChecker extends VisitorBase {

 //Impl für alle Command Subklassen

Typsicher

Void/Kontext

B7

How to handle?

Standardumgebung

• atomare Typen → (Dummy) AST Deklaration
• ohne Definition

Typhagivolerz

strukturrell \Leftrightarrow primitive gleich
sonst Elementegleich (Vergleiche Dekl. Ast)

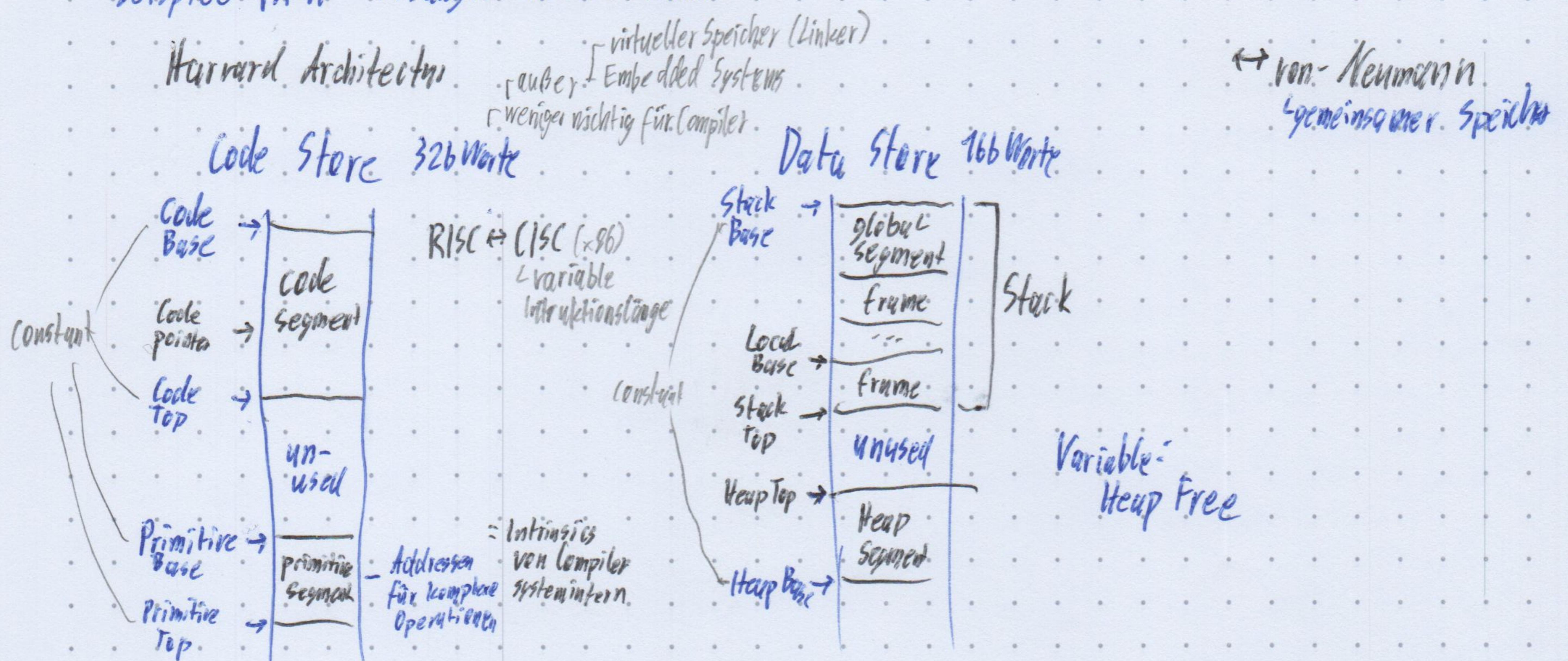
Über Namen \hookrightarrow gleicher AST verweist
 \hookrightarrow nur zu sich selbst

Laufzeitorganisation (Abstract) Machine

Darstellung abstrakter Strukturen

NUMA, COMA, ...

Beispiel TAN - Triangle Abstract Machine



Instruction

$\begin{cases} 4b \text{ op} \\ 4b \text{ register} \\ 8b \text{ Operandengröße} \\ \text{op} 16b \text{ Adressverschiebung} \end{cases}$

- 0 - LOAD(n), d[r] data of address
- 1 - " A address
- 2 - LOAD1(n) half address top piece
- 3 - LOADL d value d
- 4 - STORE(n), d[r] push to stack
- 5 - STORE1(n) pop address ← pop in word
- 6 - CALL(n), d[r], at d[r] address in n as link
- 7 - CALL1 pop closure (static links code address) call :+
In-var-ante
- 8 - RETURN(n), d
- 9 -

- 10 - PUSH d
- 11 - POP(n), d
- 12 - JUMP d[r]
- 13 - JUMP1
- 14 - JUMP1 F(n), d[r]
- 15 - HALT

Auswertung von Ausdrücken

zur Grunde liegende Instruktionen
oft 2 Operanden \Rightarrow Teil ausdrücke

Zwischen Ergebnisse?

Teinfach

Stack-Maschine \leftrightarrow Register-Maschine

post-fix Auswertung

replace top values with result

Load on stack, store from it

Register-Maschine

sehr schnell, im Prozessor

begrenzte Anzahl

nicht \forall Register für \forall Operationen

Registers as operands.

+ effizienter

- komplexer

Darstellung von Daten

Unverwechselbarkeit diff value \Rightarrow diff representation
Einzigartigkeit one value \Rightarrow always same repres.
Konstante Größe für einen Typ

bit pattern \rightarrow reinterpret
direct \rightarrow indirect
Effizient Zeiger konstante Größe
+ Typen mit variabler Größe

Primitive Typen - nicht weiter zerlegbar

$H[T] = \text{Anzahl diff elements}$

$\text{size}[T] = \min \text{ Bit to represent}$

$\text{size}[T] \geq \log_2(H[T])$

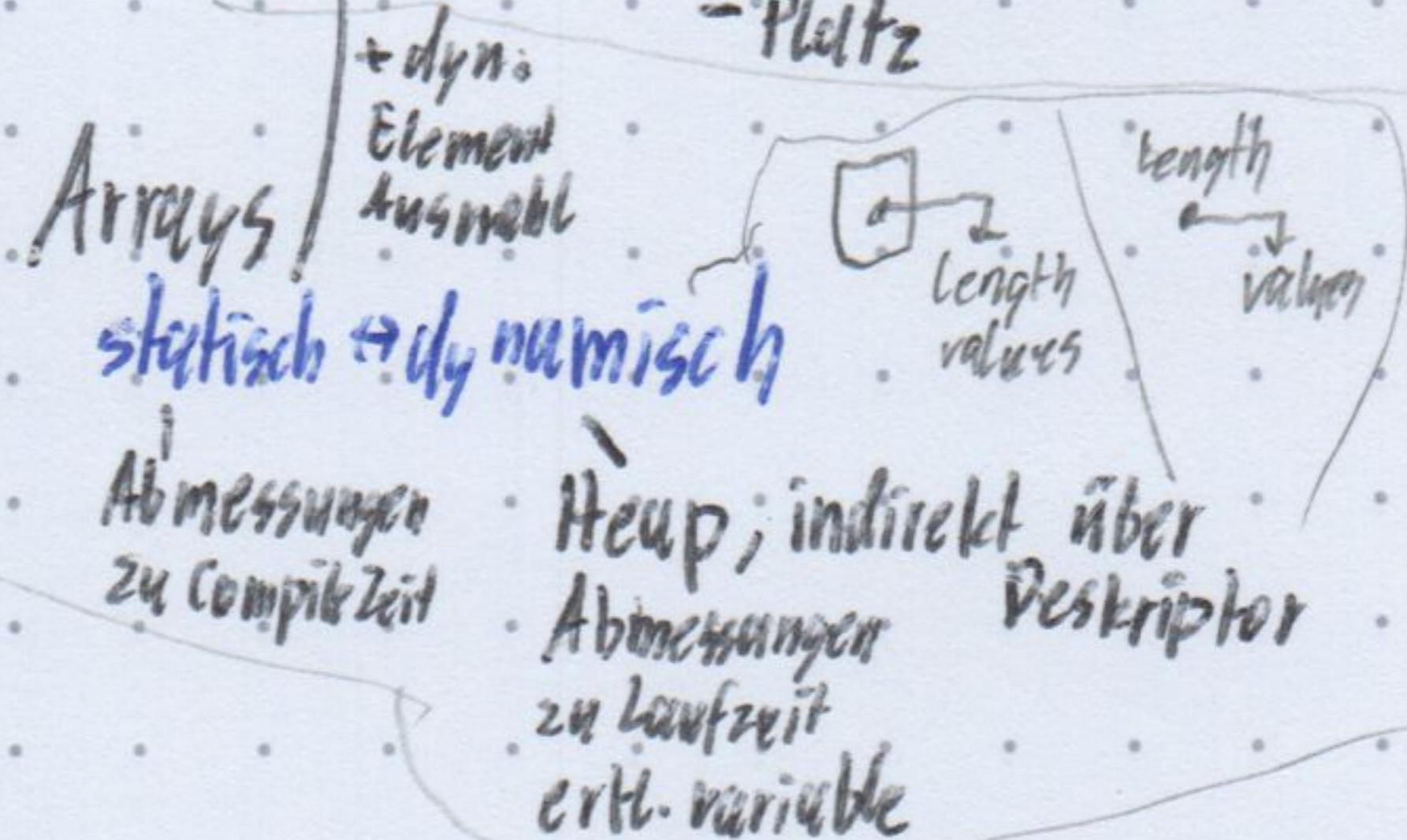
	x86	TAN
bool	8	16
char	8	16
int	16/32	16

Variant Records (disjoint unions)

K nur Untermenge aktiver Komponenten
 \Rightarrow Selektion durch Type Tag
disjunkte Teile übereinander legen

Rekursive Typen

\Rightarrow In der Regel nur über Zeiger



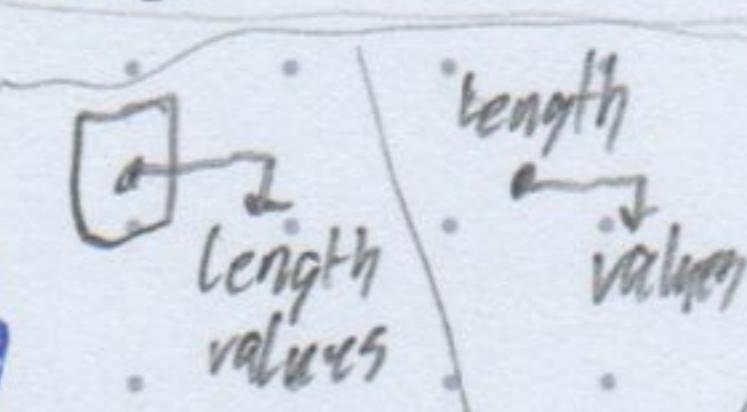
+ dyn. Elementauswahl

staticisch \leftrightarrow dynamisch

Abmessungen zu Compilezeit

Abmessungen zu Laufzeit

zu Laufzeit erteilt variable



Heap; indirekt über Descripteur

Abmessungen zu Laufzeit

erteilt variable

Speicherverwaltung

statisch für globale var
 ↳ über gesamte Laufzeit
Compiler berechnet Speicherbedarf

Verwaltung von Stapelspeicher

lokale var

 ↳ nur in Block hierarchisch verschachtelt

→ Stack Frame ^(Activation Record) pro Prozedur

 ↳ Verwaltungsdaten / Link Porte

 ↳ dynamic link (perious LB)

 ↳ return address

 ↳ local variables nur
 caller
 bekannt

 ↳ Aktuelle param

- dynamische Verkettung

 ↳ statische Verkettung - von wenigen Sprachen unterstützt

 ↳ Zugriff auf lexikalisch umschließende Prozedurarten (lokal vars)

 ↳ nicht-lokale vars

 ↳ static link contents (static link) umschließender stackframe

 ↳ (SV)

- kompliziert

- runtime overhead

 ↳ display registers für bis zu ... static links

 ↳ L1, L2, ...

 ↳ wenn nichts darüber = SB

Routinen Assemble-Aquivalent Prozeduren/Funktion

Routinenprotokoll / calling conventions oft von Betriebssystem in ABI vorgegeben

maschinenabhängig ↳ Application Binary Interface

TAM

call (reg) addrs
 ↳ Routine Wert für Stk.

return (n) of

 ↳ Anzahl param

 ↳ to remove from stack

 ↳ Anzahl Werte erg

für Stack Maschine oft:

1. caller legt param. auf stack \Rightarrow LB mit negativem Offset
2. Aufruf. (callee nutzt param.)
3. callee ~~pops~~ pushes param push result

aktuelle \sqsubseteq formale Param
 ↳ bei Aufruf Linientyp lizenziert

Sonderfall
 ↳ Methoden als Param

Übergebe Startadresse

für Aufruf Funktionsdeskriptor block (closure)

1. closure auf stack

2. Aufruf

 ↳ Startadresse

 ↳ statische Verkettung

 ↳ bestimmt bei Aufrufstelle

 ↳ als Tiefe referenz

Übergabe von Werten \Leftrightarrow Referenzen

 ↳ unabhängige Übergabe Adresse
 ↳ Kopie Routine nutzt Indirektion

meist selber
 ↳ Adressraum
wie Stack,
aber anderes
Ende Funktion?

Heap-Speicher
 ↳ Lebenszeit unabhängig von Gültigkeitsbereich
 ↳ Explizite Verwaltung (Teilweise automatisiert)

freien oder belegten Platz merken.
 ↳ HF

Verwaltung - Fragmentierung verhindern

- z.B. 1. Finde genau passenden in HF
 2. sonst größeren
 3. sonst vergreifen Heap
 4. sonst ent-df-methan

 ↳ kleinsten passenden & so das Rest

 ↳ Verschmelze benachbarte freie

 ↳ Kompaktieren

 ↳ alle Zeiger must change

 ↳ oder Poppelte Indirection über Handels

 ↳ change einfacher

Teilautomatisch

Garbage Collection

z.B. Mark-and-sweep

iterativ erreichbares finden, Rest frei

- ① Was Zeiger im Speicher
 ↳ Blöcke müssen ihre Größe kennen
 ↳ Zeiger mitten in Blöcke?

Code-Generierung

Speicherallokation

Registerallokation

Code-Selektion - Zuordnung Phrasen \mapsto Maschineninstruktionen
mehrere Möglichkeiten \Rightarrow Selektion

Code-Funktion für jede Phrasenart
name: Phrase \rightarrow Instructions*

Code-Schablonen

spezielle Phrasenform \rightarrow Maschineninstruktionen

Anwendung Code-Funktion

z.B.

execute[[...]] = Orientiert sich an Subphrasenstruktur
evaluate
...
run
fetch
assign
elaborate

oft spezielle Schablonen für optimierbare Sonderfälle

z.B. if

Inlining von Kästenfunktionen in Maschinencode

Implementierung

Visitor Pattern nach Code-Funktionen mit

backpatching - Nachbessern bereits generierter Code
z.B. Sprungadressen

vor elaborieren - Speicherort zuweisen
Ende wieder freigeben

un- \rightarrow bekannte Adressen/Göße Wert Entitätsdeskriptor

Lz.B. in DAST offset (Level statische Verkettung) / address

runtime \leftrightarrow compiletime

durch runtimeEntity attribut

Repräsentation Zielmaschine

class Instruction

class Machine \rightarrow defines const

class Encoder

emit(Instruction ...)

class NameEncoder extends VisitorBase<Void, Void>

// dadurch uneindeutige Schablonen

\rightarrow Hilfsmethoden in Encoder

statische Vergabe von Adressen
speichere belegtes/nächste freie Stelle

Stack-Verwaltung dynamisch

global var ~ offset [SB]

local LB

nicht lokal statisches Verkettungsregister
(merke Verschachtelungstiefe)

stackframe

- ANTLR v4
- nicht Klausurrelevant