

# Einführung in den Compilerbau

Niclas Kusenbach

LaTeX version:  SCHOUTER

## Table of Contents

### Contents

<b>1 Einführung</b>	<b>3</b>	3.2 Identifikation und Geltungsbereiche . . . . .	10
1.0.1 Wirkung und Bedeutung . . . . .	3	3.2.1 Blockstrukturen . . . . .	10
1.0.2 Motivation . . . . .	3	3.2.2 Die Identifikationstabelle (Symboltabelle) . . . . .	10
1.1 Aufbau eines Compilers . . . . .	3	3.3 Attribute und AST-Dekoration . . . . .	11
1.1.1 Syntaxanalyse . . . . .	3	3.4 Typprüfung (Type Checking) . . . . .	12
1.1.2 Kontextanalyse . . . . .	3	3.5 Implementierung: Das Visitor Pattern . .	12
1.1.3 Codeerzeugung . . . . .	3	3.6 Standardumgebung (Standard Environment)	12
1.1.4 Optimierung . . . . .	4	3.7 Typäquivalenz . . . . .	12
1.2 Syntax und Grammatik . . . . .	4	<b>4 Laufzeitorganisationen</b>	<b>13</b>
1.2.1 Begrifflichkeiten . . . . .	4	4.1 Einführung und Überblick . . . . .	13
1.3 (Mini-)Triangle . . . . .	4	4.2 Triangle Abstract Machine (TAM) . . . . .	13
1.3.1 Syntaxbäume . . . . .	4	4.2.1 Speicherbereiche und Register . . . . .	13
1.4 Kontextuelle Einschränkungen . . . . .	5	4.2.2 Instruktionen . . . . .	14
1.5 Semantik . . . . .	5	4.3 Datendarstellung (Repräsentation) . . . . .	14
1.5.1 Operationelle Sicht . . . . .	5	4.3.1 Prinzipien . . . . .	14
1.5.2 Beispiele . . . . .	5	4.3.2 Primitive Typen . . . . .	14
1.6 Zusammenfassung . . . . .	5	4.3.3 Zusammengesetzte Typen . . . . .	14
<b>2 Syntaktische Analyse/Lexparse</b>	<b>6</b>	4.4 Auswertung von Ausdrücken . . . . .	15
2.1 Compilerstruktur . . . . .	6	4.4.1 Stack-Maschine (z.B. TAM) . . . . .	15
2.2 Syntaxanalyse – Überblick . . . . .	6	4.4.2 Register-Maschine . . . . .	15
2.3 BNF und EBNF . . . . .	6	4.5 Speicherverwaltung (Stack) . . . . .	15
2.4 Grammatiktransformationen . . . . .	6	4.5.1 Arten von Variablen . . . . .	15
2.5 Parsing-Grundlagen . . . . .	7	4.5.2 Stack Frame (Activation Record) .	15
2.6 Top-Down Parsing . . . . .	7	4.5.3 Verkettung (Linking) . . . . .	15
2.7 Rekursiver Abstieg . . . . .	7	4.6 Routinen und Protokolle . . . . .	16
2.8 Starter- und Folgemengen . . . . .	7	4.6.1 Ablauf eines Aufrufs (TAM) . . . . .	16
2.8.1 Bottom-Up Parsing (Shift & Reduce)	8	4.6.2 Parameterübergabe . . . . .	17
2.9 AST (Abstract Syntax Tree) . . . . .	8	4.6.3 Funktionen als Parameter (Closures)	17
2.10 Scanner (Lexikalische Analyse) . . . . .	9	4.7 Heap-Speicherverwaltung . . . . .	17
2.11 Automatisierung . . . . .	9	4.7.1 Organisation . . . . .	17
2.12 Typische Fehler . . . . .	9	4.7.2 Probleme und Strategien . . . . .	17
2.13 Zusammenfassung – Wichtigste Punkte .	9	4.7.3 Garbage Collection (Automatische Speicherbereinigung) . . . . .	18
<b>3 Kontextuelle Analyse</b>	<b>10</b>	<b>5 Code-Generierung</b>	<b>19</b>
3.1 Einordnung und Ziele . . . . .	10	5.1 Einführung und Herausforderungen . . . . .	19

5.2	Code-Selektion und Code-Funktionen . . .	19
5.2.1	Beispiele für Code-Schablonen . . .	20
5.3	Implementierung mittels Visitor-Pattern .	21
5.3.1	Backpatching (Rückwärts-Einsetzen von Adressen) . . . . .	21
5.4	Speicherverwaltung und Adressierung . .	21
5.4.1	Verwaltung im AST (Runtime Entities) . . . . .	21
5.4.2	Adressvergabe . . . . .	21
5.4.3	Adressierung in verschachtelten Blöcken (Triangle) . . . . .	22
5.5	Prozeduren und Funktionen . . . . .	23
5.5.1	Deklaration . . . . .	23
5.5.2	Aufruf (Call) . . . . .	23
5.5.3	Parameterübergabe . . . . .	23
<b>6</b>	<b>Lexer/Parser-Generierung mit ANTLR</b>	<b>24</b>
6.1	Einführung und Grundlagen . . . . .	24
6.2	Struktur einer Grammatik . . . . .	24
6.2.1	Operatoren und Syntax . . . . .	24
6.3	Generierte Artefakte und Parse-Trees . .	24
6.4	Traversierung: Listener vs. Visitor . . .	25
6.4.1	1. Listener Pattern . . . . .	25
6.4.2	2. Visitor Pattern . . . . .	25
6.4.3	Labeling von Alternativen . . . . .	25
6.5	Fortgeschrittene Themen . . . . .	25
6.5.1	Assoziativität und Präzedenz . . . .	25
6.5.2	Dangling Else Problem . . . . .	26
6.5.3	Semantische Prädikate . . . . .	26
6.6	Fehlerbehandlung . . . . .	26

# 1 Einführung

## Was ist ein Compiler?

Ein **Compiler** ist die **Schnittstelle zwischen Programmiersprache und Maschine**. Er übersetzt menschenlesbaren Quellcode in maschinennahe Instruktionen.

- **Programmiersprachen:** gut handhabbar für Menschen (z.B. Java, C++)
- **Maschine:** optimiert auf Geschwindigkeit, Energieeffizienz, Fläche

### 1.0.1 Wirkung und Bedeutung

- Compiler beeinflussen direkt die **effektive Rechenleistung**.
- Beispiel: Unterschiedliche Compiler erzeugen unterschiedlich effizienten Code.
- Spezialisierte Prozessoren erfordern angepasste Compiler (DSPs, GPUs, FPGAs).

## Paralleles Rechnen

Trend von Ein-Prozessor-Systemen hin zu **Mehrkern- und heterogenen Systemen**.

- OpenMP – Mehrkern-CPUs
- CUDA – GPUs
- OpenCL – Kombination aus CPUs und GPUs

### 1.0.2 Motivation

- Compilerbau kombiniert Theorie, Architektur und Softwaretechnik.
- Zentrale Themen: Parsing, Codegenerierung, Optimierung.

## 1.1 Aufbau eines Compilers

### Compilerphasen

1. **Front-End:** Lexikalische, syntaktische und kontextuelle Analyse
2. **Middle-End:** Optimierung der Zwischendarstellung (IR)
3. **Back-End:** Codeerzeugung für Zielarchitektur

### 1.1.1 Syntaxanalyse

- Überprüfung der Syntaxregeln ⇒ **Abstrakter Syntaxbaum (AST)**

### 1.1.2 Kontextanalyse

- Variablenbindung, Typprüfung, Scope-Überprüfung
- Ergebnis: **Dekorierter AST (DAST)**

### 1.1.3 Codeerzeugung

- Zuweisung von Speicher, Übersetzung von AST zu Maschinencode

#### 1.1.4 Optimierung

- Ziel: effizienterer Code bei gleicher Semantik
- Beispiele:
  - **Constant Folding:**  $x = (2 + 3) * y \Rightarrow x = 5 * y$
  - **Common Subexpression Elimination**
  - **Strength Reduction**
  - **Loop-Invariant Code Motion**

### 1.2 Syntax und Grammatik

#### Syntax

Beschreibt die **Struktur korrekter Programme**.

#### Formalisierungsmethoden

- **Reguläre Ausdrücke (RE)** – beschreiben Tokens, aber nicht Programmsyntax.
- **Kontextfreie Grammatiken (CFG)** – Basis für Programmiersprachen.
- **BNF / EBNF** – Notation zur Beschreibung von CFGs.

#### 1.2.1 Begrifflichkeiten

- **Terminale:** konkrete Symbole
- **Nichtterminale:** syntaktische Kategorien
- **Produktionen:** Regeln der Grammatik
- **Startsymbol:** Ausgangspunkt der Herleitung

#### Mehrdeutigkeit

Eine Grammatik ist **mehrdeutig**, wenn ein Satz mehrere Ableitungsbäume hat. Für Compiler sind nur eindeutige CFGs sinnvoll.

### 1.3 (Mini-)Triangle

#### Mini-Triangle

- Pascal-artige Beispiel-Sprache
- Enthält Variablen, Konstanten, Schleifen, Bedingungen
- Keine Unterprogramme
- Beispielhafte CFG-Definitionen für:
  - **Command, Expression, Declaration, Type-denoter**

#### 1.3.1 Syntaxbäume

- **Konkrete Syntax:** enthält alle syntaktischen Details
- **Abstrakte Syntax:** reduziert auf semantisch relevante Struktur (AST)

## AST als IR

- **Vorteile:** maschinenunabhängig, gut für Analysen
- **Nachteile:** weniger geeignet für hardwarenahe Optimierungen

## 1.4 Kontextuelle Einschränkungen

### Geltungsbereiche (Scopes)

- Jede Variable muss **vor ihrer Verwendung** deklariert sein.
- Deklaration = *bindendes Auftreten*, Verwendung = *verwendendes Auftreten*.

### Typprüfung

- Jede Operation verlangt passende Operandentypen.
- Beispielregeln:

$E_1 > E_2$  : liefert bool, wenn  $E_1, E_2 : \text{int}$

$V := E$  : nur erlaubt, wenn Typen äquivalent

$\text{while } E \text{ do } C$  : nur erlaubt, wenn  $E : \text{bool}$

## 1.5 Semantik

### Semantik

Beschreibt die **Bedeutung von Programmen zur Laufzeit**.

### 1.5.1 Operationelle Sicht

- **Anweisungen:** verändern Zustand (z.B. Variablen, I/O)
- **Ausdrücke:** werden evaluiert und liefern Werte
- **Deklarationen:** binden Namen an Speicherbereiche

### 1.5.2 Beispiele

- **AssignCmd:**  $V := E$ 
  1. Evaluiere  $E \Rightarrow v$
  2. Weise  $v$  an Variable  $V$  zu
- **BinaryExp:**  $E_1 \text{ op } E_2$ 
  1. Evaluiere  $E_1, E_2 \Rightarrow v_1, v_2$
  2. Führe Operation  $\text{op}(v_1, v_2)$  aus

## 1.6 Zusammenfassung

- Compiler übersetzen Hochsprache → Maschinencode.
- Bestehen aus: Front-End, Middle-End, Back-End.
- Zentrale Themen: Syntax, Semantik, Typen, Optimierung.
- Mini-Triangle dient als Lehrsprache zur Umsetzung der Konzepte.

## 2 Syntaktische Analyse/Lexparse

### Übersetzung und Phasen

- **Syntaxanalyse** → Struktur des Programms (AST)
- **Kontextanalyse** → Bedeutungsprüfung (Typen, Gültigkeit)
- **Codegenerierung** → Übersetzung in Zielcode

### 2.1 Compilerstruktur

Ein-Pass-Compiler:

- Führt alle Phasen gleichzeitig aus
- Keine echte Zwischendarstellung (IR)
- Typisch für kleine Sprachen (z. B. Pascal)

Multi-Pass-Compiler:

- Arbeitet mit mehreren Durchgängen über Quelltext/IR
- Datenweitergabe über IR (AST)
- Bessere Modularität und Optimierung

### 2.2 Syntaxanalyse – Überblick

- **Scanner (Lexer)**: Wandelt Zeichenfolge → Tokenfolge
- **Parser**: Wandelt Tokenfolge → Abstract Syntax Tree (AST)
- Token = atomares Symbol des Quellprogramms

### Kontextfreie Grammatik (CFG)

Eine CFG ist ein 4-Tupel  $(N, T, P, S)$  mit:

- $N$ : Nichtterminale
- $T$ : Terminale
- $P$ : Produktionen
- $S$ : Startsymbol

### 2.3 BNF und EBNF

- **BNF**: Grundform zur Definition von Grammatiken
- **EBNF**: Erweiterung mit regulären Ausdrücken, optionalen und wiederholten Konstrukten
- Beispiel:  
Expression ::= primary-Expression (operator primary-Expression)\*

### 2.4 Grammatiktransformationen

- **Gruppierung**: Zusammenfassen gleicher LHS
- **Linksauklammern**: Gemeinsame Präfixe auslagern
- **Linksrekursion beseitigen**:  $N ::= X|NY \Rightarrow N ::= X(Y)^*$

- **Ersetzung von Nicht-Terminalen:** falls nur eine Regel existiert

## 2.5 Parsing-Grundlagen

### Parsing

Entscheidung, ob Eingabe zur Grammatik gehört und Aufbau des Syntaxbaumes.

- **Top-Down (z. B. rekursiver Abstieg):** Von Startsymbol zu Terminalen
- **Bottom-Up (z. B. Shift/Reduce):** Von Terminalen zur Wurzel

## 2.6 Top-Down Parsing

- Aufbau des Syntaxbaums von oben nach unten
- Expandiere jeweils das linke Nichtterminal
- LL(k): Grammatik, bei der mit  $k$  Lookahead-Tokens eindeutig entschieden werden kann
- **LL(1)** → wichtigster Fall für rekursiven Abstieg

## 2.7 Rekursiver Abstieg

### Rekursiver Abstieg

Jedes Nichtterminal erhält eine Prozedur `parseN()`, deren Aufrufstruktur dem Parsebaum entspricht.

- **accept(t)** prüft aktuelles Token
- **acceptIt()** akzeptiert aktuelles Token ohne Prüfung
- **currentToken:** vom Scanner geliefert

## 2.8 Starter- und Folgemengen

- $T$ : Menge der Terminals (Tokens, z.B. `id`, `+`)
- $N$ : Menge der Nichtterminale (Variablen, z.B. `Expression`)
- $\varepsilon$ : Das leere Wort (Epsilon)
- $\$$ : End-of-File Marker (`eof`)

### `starters[[X]]` (First-Menge)

Menge aller Terminals, mit denen ein aus  $X$  abgeleiteter String beginnen kann.

**Berechnung:**

- Ist  $X \in T$  (Terminal):  

$$\text{starters}[[X]] = \{X\}$$
- Ist  $X \in N$  (Nichtterminal) mit  $X \rightarrow Y_1 Y_2 \dots$ :
  - Füge  $\text{starters}[[Y_1]] \setminus \{\varepsilon\}$  hinzu.
  - Falls  $\varepsilon \in \text{starters}[[Y_1]]$ , füge auch  $\text{starters}[[Y_2]]$  hinzu (usw.).
- **Epsilon:** Falls  $X \rightarrow \varepsilon$  existiert, ist  $\varepsilon \in \text{starters}[[X]]$ .

## follow[[A]] (Folgemenge)

Menge aller Terminals, die in einer Satzform unmittelbar rechts von einem Nichtterminal  $A$  stehen können.

### Regeln:

1. **Startsymbol  $S$ :** Enthält immer das Ende-Zeichen (\$).
2. **Rechter Nachbar ( $B \rightarrow \alpha A \beta$ ):**  
Alles aus  $starters[[\beta]]$  (außer  $\epsilon$ ) kommt zu  $follow[[A]]$ .
3. **Eltern-Vererbung ( $B \rightarrow \alpha A$  oder  $\beta \Rightarrow^* \epsilon$ ):**  
Wenn  $A$  am Ende steht (oder  $\beta$  wegfallen kann), erbt  $A$  alles aus  $follow[[B]]$ .

Hinweis: Folgemengen enthalten niemals  $\epsilon$ , können aber \$ enthalten.

## LL(1)-Konfliktfreiheit

Eine Grammatik ist LL(1), wenn für jede Produktion mit Alternativen  $A \rightarrow \alpha | \beta$  gilt:

- **Disjunkte Starter (Auswahl-Konflikt):**  
Die Alternativen dürfen nicht mit demselben Terminal beginnen.  
 $starters[[\alpha]] \cap starters[[\beta]] = \emptyset$
- **Disjunkte Folge bei Epsilon (Nullable-Konflikt):**  
Falls  $\alpha \Rightarrow^* \epsilon$  (d.h.  $\alpha$  kann verschwinden), darf die Folgemenge von  $A$  keinen gemeinsamen Start mit  $\beta$  haben.  
 $starters[[\beta]] \cap follow[[A]] = \emptyset$

### 2.8.1 Bottom-Up Parsing (Shift & Reduce)

Beim Bottom-Up Parsing (z.B. LR-Parser) wird die Eingabe von links gelesen und versucht, sie auf das Startsymbol zu reduzieren.

#### Operationen:

- **Shift:** Schiebe das nächste Eingabe-Symbol auf den Stack.
- **Reduce:** Ersetze eine Symbolfolge oben auf dem Stack durch das entsprechende Nichtterminal (gemäß Grammatikregel).

**Beispiel-Trace (aus Übung):** Grammatik:  $S \rightarrow Ac$ ,  $A \rightarrow aA | b$ . Eingabe:  $aabc$

Stack	Eingabe	Aktion
$\epsilon$	$aabc$	Shift $a$
$a$	$abc$	Shift $a$
$aa$	$bc$	Shift $b$
$ab$	$c$	Reduce ( $A \rightarrow b$ )
$aaA$	$c$	Reduce ( $A \rightarrow aA$ )
$aA$	$c$	Reduce ( $A \rightarrow aA$ )
$A$	$c$	Shift $c$
$Ac$	$\epsilon$	Reduce ( $S \rightarrow Ac$ )
$S$	$\epsilon$	<b>Akzeptiert</b>

## 2.9 AST (Abstract Syntax Tree)

- Strukturierte Repräsentation des Programms
- Parser erzeugt AST-Knoten beim rekursiven Abstieg
- Jede Grammatikregel entspricht einer AST-Unterklasse

## AST-Aufbau

- **Abstrakte Basisklasse:** AST
- **Subklassen:** Command, Expression, Declaration, TypeDenoter
- Terminalknoten (z. B. Identifier, Operator) speichern tatsächlichen Text

## 2.10 Scanner (Lexikalische Analyse)

### Scanner

Wandelt Zeichen → Tokens anhand regulärer Ausdrücke (REs).

#### Aufgaben:

- Entfernt Whitespace, Kommentare
- Liefert Token(kind, spelling, position)
- Nutzt endlichen Automaten oder rekursiven Abstieg

#### Beispiel EBNF Mini-Triangle:

```
Identifier ::= Letter (Letter — Digit)*  
Integer-Literal ::= Digit Digit*  
Operator ::= + | - | * | / | < | > | =
```

## 2.11 Automatisierung

- Scanner-Generatoren: **JLex, JFlex**
- Parser-Generatoren: **ANTLR (LL\*), JavaCC (LL(k))**

## 2.12 Typische Fehler

- Linksrekursion nicht entfernt
- Linksausklammern vergessen
- Anfangs-/Folgemengen überschneiden sich → nicht LL(1)
- Schlüsselwörter nicht vom Identifier getrennt

## 2.13 Zusammenfassung – Wichtigste Punkte

- CFG-Grundlagen (BNF/EBNF)
- Transformationen: Gruppierung, Linksausklammern, Rekursionsbeseitigung
- LL(1)-Parsing und Bedingungen
- Rekursiver Abstieg: Struktur und Methoden
- AST-Struktur: Klassenhierarchie, Terminal- und Nichtterminalknoten
- Scanner: REs, endliche Automaten, Schlüsselworterkennung

## 3 Kontextuelle Analyse

### 3.1 Einordnung und Ziele

Die kontextuelle Analyse ist die Phase zwischen Syntaxanalyse (Parsing) und Code-Generierung. Während der Parser nur die grammatischen Korrektheit prüft (Kontextfreie Grammatik), prüft diese Phase Regeln, die vom Kontext abhängen.

- **Eingabe:** Abstrakter Syntaxbaum (AST).
- **Ausgabe:** Dekorierter AST (Knoten sind mit Typ- und Bindungsinformationen angereichert).
- **Aufgaben:**
  1. **Identifikation** (Identification): Zuordnung von Bezeichner-Verwendungen zu ihren Deklarationen (Geltungsbereiche prüfen).
  2. **Typprüfung** (Type Checking): Sicherstellen, dass Operatoren auf kompatible Typen angewendet werden.

### 3.2 Identifikation und Geltungsbereiche

#### 3.2.1 Blockstrukturen

Programmiersprachen definieren **Geltungsbereiche** (Scopes), in denen Bezeichner sichtbar sind.

##### Arten von Blockstrukturen

- **Monolithisch** (z.B. BASIC): Ein einziger globaler Scope. Keine Namensdopplungen erlaubt.
- **Flach** (z.B. FORTRAN): Trennung in Global und Lokal.
- **Verschachtelt** (z.B. Triangle, Java, Pascal): Beliebig tief Schachtelung von Blöcken.

##### Regeln für verschachtelte Strukturen (Nested Scopes):

1. Ein Bezeichner darf innerhalb eines Blocks nur **einmal** deklariert werden.
2. Ein benutzerdefinierter Bezeichner muss im aktuellen oder einem umschließenden (äußeren) Block deklariert sein.
3. **Verschattung (Hiding)**: Eine Deklaration in einem inneren Block verdeckt eine gleichnamige Deklaration in einem äußeren Block.

#### 3.2.2 Die Identifikationstabelle (Symboltabelle)

Die Symboltabelle (in den Folien **IdentificationTable**) ist die zentrale Datenstruktur, um Deklarationen zu verwalten und effizient abzurufen.

##### Problem naiver Ansätze:

- *Liste*: Lineare Suche ist zu langsam ( $O(n)$ ).
- *Einfache Map*: Kann keine Verschattung (gleicher Name in verschiedenen Scopes) abbilden.

**Effiziente Implementierung (Triangle-Ansatz):** Es wird eine Kombination aus Hash-Map und Stacks verwendet, um schnellen Zugriff ( $O(1)$ ) und Scope-Verwaltung zu kombinieren.

## Datenstrukturen der IdentificationTable

- `private Map<String, Stack<Attribute>> idents`  
Bildet Bezeichnernamen (String) auf einen Stapel von Attributen ab.
  - *Warum ein Stack?* Wenn Variable `x` global und lokal existiert, liegt die lokale (aktuelle) Definition oben auf dem Stack.
- `private Stack<List<String>> scopes`  
Verwaltet die Schachtelungsebenen. Jedes Element des Stacks ist eine Liste aller Bezeichner, die im *aktuellen Scope* deklariert wurden.
  - *Zweck:* Ermöglicht das schnelle Aufräumen (Löschen) aller Variablen eines Blocks, wenn dieser verlassen wird.

## Algorithmus der Scope-Operationen:

1. **Scope öffnen** (`openScope`):
  - Lege eine neue, leere Liste auf den `scopes`-Stack.
  - Markiert den Beginn eines neuen Blocks (z.B. bei `LetCommand`).
2. **Eintrag hinzufügen** (`enter(id, attr)`):
  - Hole den Attribut-Stack für `id` aus `idents` (erstelle ihn, falls nicht existent).
  - Pushe das neue `attr` auf diesen Stack (Verschattung aktiv).
  - Füge `id` zur Liste hinzu, die oben auf `scopes` liegt (damit wir wissen, dass `id` zu diesem Scope gehört).
3. **Eintrag abrufen** (`retrieve(id)`):
  - Suche `id` in `idents`.
  - Wenn vorhanden: Gib das oberste Element des Stacks zurück (tiefste/aktuellste Verschachtelungsebene).
  - Wenn Stack leer/nicht vorhanden: Bezeichner nicht deklariert → Fehler.
4. **Scope schließen** (`closeScope`):
  - Poppe die oberste Liste von `scopes` (Liste der lokalen Variablen).
  - Durchlaufe diese Liste: Für jeden String `id` darin, poppe das oberste Element vom entsprechenden Stack in `idents`.
  - *Effekt:* Die lokalen Deklarationen sind “vergessen”, vorherige (globale) Deklarationen liegen wieder oben auf den Stacks in `idents`.

## 3.3 Attribute und AST-Dekoration

Was genau wird in der Symboltabelle gespeichert?

- **Klassischer Ansatz:** Eigene Klasse `Attribute` mit Feldern für `Kind` (Var, Const, Proc) und `Type` (Int, Bool). Wird bei komplexen Typen (Arrays, Records) schnell unhandlich.
- **AST-Ansatz (Triangle):** Da im AST (genauer: im Deklarations-Teilbaum) bereits alle Infos stehen, speichert man in der Symboltabelle einfach **Referenzen auf die AST-Knoten**.

### Dekoration

Der Prozess der Kontextanalyse reichert den AST an:

- **Bei der Deklaration:** Der AST-Knoten der Deklaration wird in die Symboltabelle eingetragen.
- **Bei der Verwendung (Applied Occurrence):** Der AST-Knoten der Verwendung (z.B. `Identifier`) erhält einen Zeiger (`public Declaration decl`) auf den AST-Knoten seiner Deklaration.

## 3.4 Typprüfung (Type Checking)

- **Ziel:** Sicherstellen, dass Operationen mit validen Typen ausgeführt werden (z.B. `if` benötigt `Boolean`).
- **Vorgehen:** Bottom-Up Verfahren (von den Blättern zur Wurzel).
- **Statische Typisierung:** Findet zur Compile-Zeit statt. Jeder Ausdruck hat einen festen Typ.

Ablauf am Beispiel `n + 1`:

1. `IntLit (1)`: Typ ist direkt bekannt (`Integer`).
2. `Ident (n)`: Typ wird aus der Symboltabelle geholt (via Link zur Deklaration).
3. `BinaryExpr (+)`: Prüft, ob Operator `+` für `Integer × Integer` definiert ist und was der Rückgabetyp ist.

## 3.5 Implementierung: Das Visitor Pattern

Um die Logik der Kontextanalyse nicht in den AST-Klassen zu verstreuen, wird das **Visitor Pattern** verwendet. Dies trennt Datenstruktur (AST) von Algorithmus (Checker).

- **Prinzip:** `astNode.visit(visitor, arg)`.
- **Double Dispatch:** Der Knoten ruft zurück auf `visitor.visitSpecificNode(this, arg)`.
- **Vorteil:** Neue Analysen (z.B. CodeGen) können hinzugefügt werden, ohne AST-Klassen zu ändern.

**Spezialisierte Visitors in Triangle:** Statt eines einzigen Visitors werden spezialisierte Unterklassen verwendet, um Typsicherheit bei Rückgabewerten zu erhöhen:

- `ExpressionChecker`: Liefert `TypeDenoter` (da Ausdrücke einen Typ haben).
- `CommandChecker`: Liefert `Void` (Befehle haben keinen Typ).
- `DeclarationChecker`: Trägt Bezeichner in die Symboltabelle ein.

## 3.6 Standardumgebung (Standard Environment)

Sprachen haben vordefinierte Typen (`Integer`, `Boolean`) und Funktionen (`put`, `get`).

- Diese sind nicht Teil der Grammatik.
- **Lösung:** Vor dem Start der eigentlichen Analyse wird die Symboltabelle mit “künstlichen” Deklarationen befüllt (z.B. wird ein AST-Fragment für eine Konstante `true` erzeugt und eingetragen).

## 3.7 Typäquivalenz

Wann gelten zwei Typen als gleich?

### Äquivalenz-Arten

- **Strukturelle Äquivalenz** (Triangle): Typen sind gleich, wenn ihre Struktur identisch ist.  
Beispiel: `array 8 of Char` ist kompatibel mit `array 8 of Char`.
- **Namensäquivalenz** (Pascal, Ada): Typen sind nur gleich, wenn sie denselben Typnamen haben. Jede Typ-Definition erzeugt einen neuen, inkompatiblen Typ.

## 4 Laufzeitorganisationen

### 4.1 Einführung und Überblick

Die Laufzeitorganisation beschäftigt sich mit der Abbildung von abstrakten Strukturen einer Hochsprache (Variablen, Prozeduren, Objekte) auf die konkreten Ressourcen der Zielmaschine (Register, Speicher, Instruktionen).

Es existiert eine **Semantic Gap** zwischen den komplexen Konstrukten der Hochsprache (Arrays, Objekte, Methoden) und den primitiven Möglichkeiten der Hardware.

#### Aufgaben der Laufzeitorganisation

- Datendarstellung (Primitive Typen, Records, Arrays).
- Auswertung von Ausdrücken (Stack vs. Register).
- Speicherverwaltung (Global, Lokal/Stack, Heap).
- Routinen und Aufrufkonventionen (Parameterübergabe).

### 4.2 Triangle Abstract Machine (TAM)

Die TAM ist eine abstrakte Zielmaschine für Lehrzwecke. Sie basiert auf einer **Harvard-Architektur**, was bedeutet, dass Befehls- und Datenspeicher getrennt sind.

#### 4.2.1 Speicherbereiche und Register

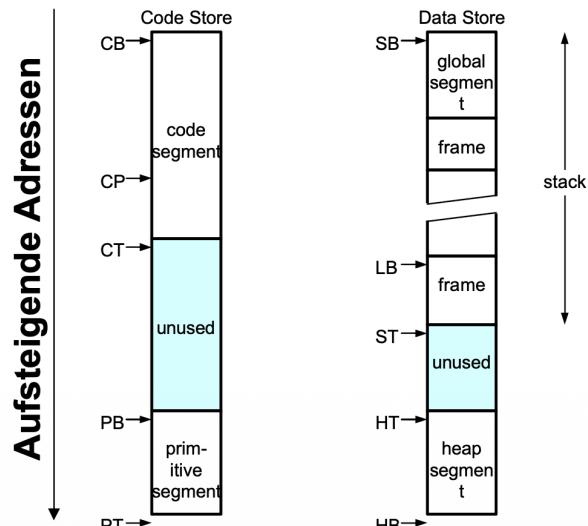
Die TAM nutzt verschiedene Register zur Adressierung der Speichersegmente.

**Instruktionsspeicher (Code Store)** Der Code-Speicher enthält das ausführbare Programm.

- **CB** (Code Base): Startadresse des Code-Segments (konstant).
- **CT** (Code Top): Endadresse des Code-Segments (konstant).
- **CP** (Code Pointer): Aktueller Befehlszähler (Instruction Pointer), zeigt auf den nächsten auszuführenden Befehl.
- **PB** (Primitive Base): Startadresse der Intrinsics (eingebaute Funktionen).
- **PT** (Primitive Top): Endadresse der Intrinsics.

**Datenspeicher (Data Store)** Der Datenspeicher ist in Stack und Heap unterteilt. In der TAM wachsen diese Bereiche aufeinander zu (siehe Speicherverwaltung).

- **SB** (Stack Base): Boden des Stacks (Start der globalen Variablen).
- **ST** (Stack Top): Aktuelles oberes Ende des Stacks.
- **HB** (Heap Base): Startadresse des Heaps (oberes Ende des Speichers).
- **HT** (Heap Top): Aktuelle Grenze des belegten Heaps.



- **LB** (Local Base): Zeiger auf den aktuellen *Stack Frame* (Beginn der lokalen Variablen der aktuellen Prozedur).

#### 4.2.2 Instruktionen

TAM-Instruktionen sind 32-bit breit und haben folgendes Format:

$$\text{Instruktion} = \underbrace{\text{op (4 Bit)}}_{\text{Opcode}} \mid \underbrace{\text{r (4 Bit)}}_{\text{Register}} \mid \underbrace{\text{n (8 Bit)}}_{\text{Größe}} \mid \underbrace{\text{d (16 Bit)}}_{\text{Displacement}}$$

Beispiel: LOAD (1) 3[ST] lädt ein Wort von der Adresse  $ST + 3$ .

### 4.3 Datendarstellung (Repräsentation)

Daten müssen im Speicher so abgelegt werden, dass sie effizient zugreifbar sind.

#### 4.3.1 Prinzipien

1. **Unverwechselbarkeit**: Unterschiedliche Werte sollten unterschiedliche Bitmuster haben.
2. **Einzigartigkeit**: Ein Wert wird immer gleich dargestellt.
3. **Konstante Größe**: Alle Werte eines Typs belegen gleich viel Platz.

#### Invariante der Datengröße

Es muss gelten:  $\text{size}[T] \geq \log_2(\#[T])$ , wobei  $\#[T]$  die Anzahl der unterschiedlichen Elemente in  $T$  ist.

#### 4.3.2 Primitive Typen

- **Boolean**: 1 Wort (16b in TAM). Werte: 00..00 (false), 00..01 (true). *Hinweis: In C/x86 oft nur 8 Bit.*
- **Char**: 1 Wort (16b), Unicode/ASCII.
- **Integer**: 1 Wort (16b), Zweierkomplement.

#### 4.3.3 Zusammengesetzte Typen

**Records (Verbundtypen)** Die Felder eines Records werden im Speicher nacheinander (sequenziell) abgelegt.

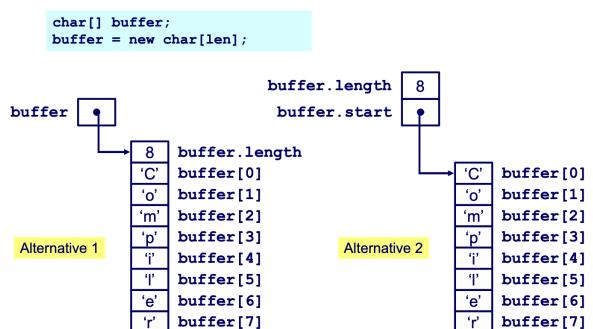
- **Adressierung**: Adresse des Records + Offset des Feldes.
- **Padding**: Viele Prozessoren verlangen eine Ausrichtung (Alignment) auf Wortgrenzen (z.B. 32-bit), was zu ungenutzten Lücken (Padding) führen kann. TAM adressiert wortweise, daher weniger Padding-Probleme, aber Platzverschwendungen bei Booleans.

#### Arrays (Felder)

- **Statische Arrays**: Größe zur Compile-Zeit bekannt. Elemente liegen direkt hintereinander.

$$address[me[i]] = address[me] + i \times \text{size}[Element]$$

- **Dynamische Arrays**: Größe erst zur Laufzeit bekannt.
- **Repräsentation**: Indirekt über einen **Deskriptor** (Dope Vector). Dieser enthält einen Zeiger auf die Daten (im Heap) und die aktuelle Größe.



**Variante Records (Disjoint Unions)** Ähnlich wie Records, aber die Komponenten überlagern sich im Speicher (Union in C). Ein *Type Tag* entscheidet, welche Interpretation gerade gültig ist. Die Größe richtet sich nach der größten Komponente.

## 4.4 Auswertung von Ausdrücken

Wie werden mathematische Ausdrücke wie  $a \times a + 2 \times a \times b$  berechnet?

### 4.4.1 Stack-Maschine (z.B. TAM)

Arbeitet nach dem **Post-Fix-Prinzip**. Operanden werden auf den Stack gelegt (LOAD), Operationen (ADD, MUL) nehmen die obersten Elemente, verrechnen sie und legen das Ergebnis zurück.

- **Vorteil:** Einfache Code-Generierung, keine Registerverwaltung nötig.
- **Nachteil:** Viele Speicherzugriffe, langsamer als Registermaschinen.

### 4.4.2 Register-Maschine

Berechnungen finden in schnellen CPU-Registern statt.

- **Vorteil:** Sehr schnell.
- **Nachteil:** Begrenzte Anzahl Register erfordert komplexe Zuteilungsstrategien (Register Allocation), wenn Zwischenergebnisse die Anzahl der Register übersteigen ("Spilling").

## 4.5 Speicherverwaltung (Stack)

Die Verwaltung des Speichers für Variablen hängt von ihrer Lebensdauer ab.

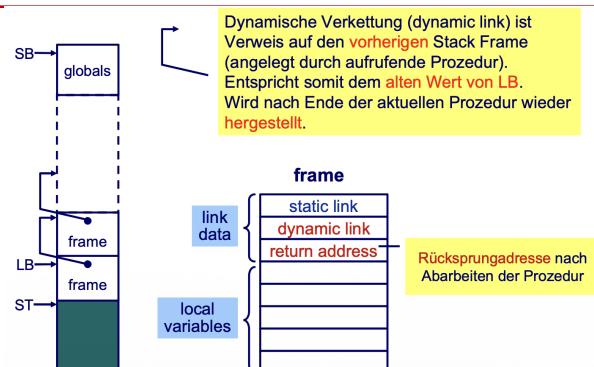
### 4.5.1 Arten von Variablen

1. **Globale Variablen:** Existieren über die gesamte Laufzeit. Adresse ist fest relativ zu *SB*.
2. **Lokale Variablen:** Existieren nur, solange der Block (Prozedur/Funktion) aktiv ist. Verwaltung über den **Stack**.
3. **Heap-Variablen:** Lebensdauer unabhängig vom Scope (siehe Abschnitt Heap).

### 4.5.2 Stack Frame (Activation Record)

Jeder Prozederaufruf erzeugt einen neuen Stack Frame. Dieser enthält:

- **Parameter:** Vom Aufrufer abgelegt.
- **Verwaltungsdaten (Link Data):** Static Link, Dynamic Link, Rücksprungadresse.
- **Lokale Variablen:** Innerhalb der Prozedur angelegt.
- **Zwischenergebnisse:** Für die Expression-Evaluation.



### 4.5.3 Verkettung (Linking)

Um auf Variablen zuzugreifen, werden zwei Arten von Links im Stack Frame gespeichert:

#### Dynamic Link (Dynamische Verkettung)

Zeigt auf den Stack Frame des **Aufrufers** (Caller). Entspricht dem alten Wert des *LB*-Registers. Dient dazu, beim Rücksprung (Return) den Stack-Kontext des Aufrufers wiederherzustellen.

## Static Link (Statische Verkettung)

Zeigt auf den Stack Frame der Prozedur, die die aktuelle Prozedur im Quelltext **umschließt** (textuelle/lexikalische Hierarchie). Dient dem Zugriff auf **nicht-lokale Variablen** in verschachtelten Prozeduren.

### Bestimmung des Static Link (SL)

Wenn Prozedur  $P$  (auf Ebene  $L_P$ ) eine Prozedur  $Q$  (auf Ebene  $L_Q$ ) aufruft:

- **Aufruf einer globalen Prozedur ( $L_Q = 0$ ):**  $SL = SB$ .
- **Aufruf einer eingebetteten Prozedur ( $L_Q > 0$ ):**
  - $Q$  ist direkt in  $P$  definiert ( $L_Q = L_P + 1$ ):  $SL = LB$  (aktueller Frame von  $P$ ).
  - $Q$  ist auf gleicher Ebene oder weiter außen ( $L_Q \leq L_P$ ): Man muss der statischen Kette von  $P$  folgen ( $k = L_P - L_Q + 1$  Schritte), um den korrekten Kontext zu finden.

*Display-Register:* Eine Alternative zur statischen Verkettung, bei der ein Array von Zeigern (Display) gepflegt wird, das für jede Schachtelungstiefe direkt auf den aktuellen gültigen Frame zeigt. Schnellerer Zugriff, aber aufwendigerer Prozeduraufruf.

## 4.6 Routinen und Protokolle

Das Zusammenspiel von Aufrufer (Caller) und Aufgerufenem (Callee) wird durch ein Protokoll (Calling Convention) geregelt.

### 4.6.1 Ablauf eines Aufrufs (TAM)

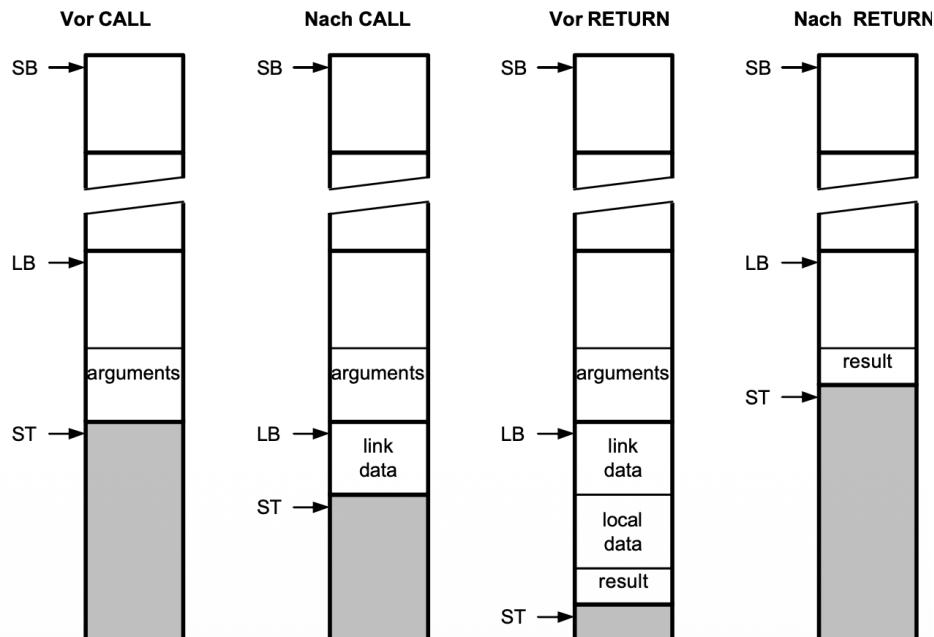


Figure 1: Vor/Nach Call/Return im Stack

#### 1. Vor CALL (Caller):

- Argumente (Parameter) werden auf den Stack gepusht (in TAM: in umgekehrter Reihenfolge, damit das erste Argument oben liegt oder direkt über  $LB$  adressierbar ist).

#### 2. CALL (Instruktion):

- Sichert *Static Link* (wird berechnet/übergeben).

- Sichert *Dynamic Link* (aktueller LB).
- Sichert *Return Address* (PC + 1).
- Setzt neuen *LB* auf den Beginn des neuen Frames.
- Sprung zur Code-Adresse der Routine.

### 3. In der Routine:

- Reserviert Platz für lokale Variablen (Inkrementiert *ST*).

### 4. RETURN (Callee):

- Entfernt lokalen Speicher und Verwaltungsdaten.
- Entfernt Argumente vom Stack.
- Legt Rückgabewert (Result) auf den Stack.
- Stellt alten *LB* und *ST* wieder her.
- Springt zurück.

## 4.6.2 Parameterübergabe

---

Parameter werden relativ zu *LB* mit **negativen Offsets** adressiert (da sie vor dem Frame-Start auf den Stack gelegt wurden). Lokale Variablen haben positive Offsets.

- **Call-by-Value:** Der Wert der Variable wird kopiert. Änderungen in der Prozedur haben keinen Effekt auf den Aufrufer.
- **Call-by-Reference (var):** Die *Adresse* der Variable wird übergeben. Die Prozedur arbeitet via Indirektion direkt auf dem Speicherplatz des Aufrufers. Änderungen sind global sichtbar.

## 4.6.3 Funktionen als Parameter (Closures)

---

Wenn eine Funktion *F* als Parameter übergeben wird, reicht die Startadresse nicht aus, da *F* Zugriff auf ihren statischen Kontext benötigt.

- Lösung: **Closure** (Funktionsabschluss).
- Repräsentation: Paar aus (**Code-Adresse**, **Static Link**).
- Aufruf: **CALLI** (Call Indirect) nutzt dieses Paar.

## 4.7 Heap-Speicherverwaltung

---

Der Heap dient für Daten, deren Lebensdauer nicht an den Block-Scope gebunden ist (z.B. verkettete Listen, Bäume).

### 4.7.1 Organisation

---

In der TAM (und vielen Systemen) wachsen Stack und Heap aufeinander zu. Wenn sie sich treffen → *Out of Memory*.

- **Allokation:** Suchen eines freien Blocks geeigneter Größe.
- **Deallokation:** Freigabe von Speicher.

### 4.7.2 Probleme und Strategien

---

- **Fragmentierung:** Durch unregelmäßiges Anlegen und Freigeben entstehen Lücken ("Löcher"), die zu klein für neue Objekte sind, obwohl in Summe genug Speicher frei wäre.
- **Freispeicherliste (Free List):** Liste (z.B. HF in TAM) verkettet alle freien Blöcke.
- **Kompaktierung:** Verschieben von belegten Blöcken, um Lücken zu schließen. Erfordert Aktualisierung aller Zeiger (schwierig!) oder Nutzung von *Handles* (Zeiger auf Zeiger).

#### **4.7.3 Garbage Collection (Automatische Speicherbereinigung)**

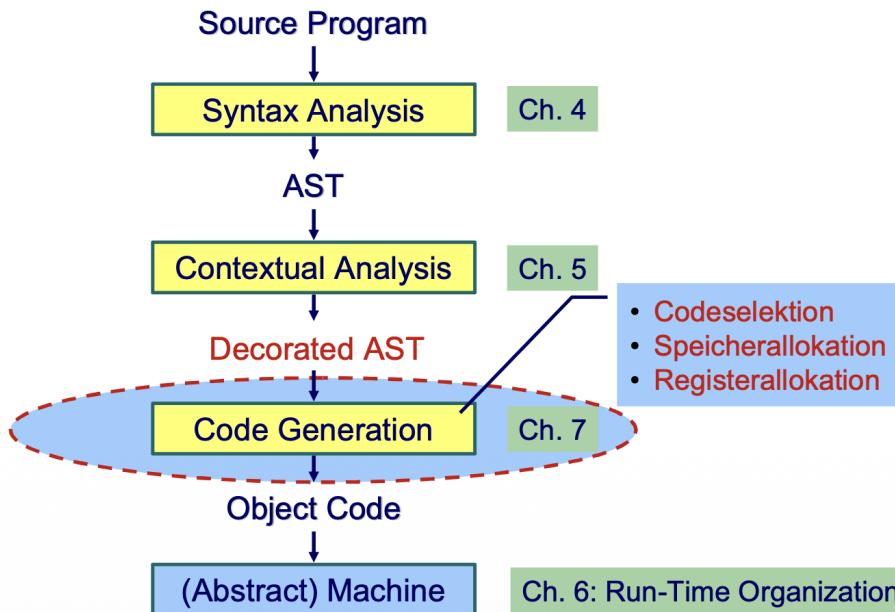
---

Verfahren, um nicht mehr erreichbaren Speicher automatisch freizugeben (z.B. in Java).

##### **Mark-and-Sweep Algorithmus**

1. **Mark (Markieren):** Gehe von allen Wurzeln (Register, Stack-Variablen) aus und verfolge alle Zeiger. Markiere jedes erreichte Objekt im Heap als “lebendig”.
2. **Sweep (Fegen):** Durchlaufe den gesamten Heap. Alle nicht markierten Objekte sind “Müll” und werden zur Freispeicherliste hinzugefügt. Markierungen werden für den nächsten Lauf zurückgesetzt.

# 5 Code-Generierung



## 5.1 Einführung und Herausforderungen

Die Code-Generierung ist der letzte Schritt im Compiler-Backend. Sie übersetzt den dekorierten Abstrakten Syntaxbaum (AST) in den Zielcode (hier: TAM-Maschinencode).

- **Abhängigkeit:** Dieser Schritt hängt sowohl von der **Semantik der Eingabesprache** als auch von der **Zielmaschine** ab. Das macht eine allgemeingültige Formulierung schwierig.
- **Semantik:** Erst in diesem Schritt erhält eine Zeichenkette (z.B. "4.2") ihre tatsächliche Bedeutung als Zahl (Integer 42) auf der Zielmaschine.
- **Ziel:** Erzeugung einer Instruktionsfolge, die semantisch äquivalent zum Quellprogramm ist.

Das Gesamtproblem wird in drei Teilprobleme zerlegt:

1. **Code-Selektion:** Abbildung von Phrasen des Quellprogramms auf Folgen von Maschineninstruktionen.
2. **Speicherallokation:** Zuweisung von Speicheradressen für Variablen und Verwaltung der Adressen (Buchführung).
3. **Registerallokulation:** Verwaltung der Register für Variablen und Zwischenergebnisse (bei der TAM als Stackmaschine weitestgehend irrelevant, da alles auf dem Stack passiert).

## 5.2 Code-Selektion und Code-Funktionen

Die Übersetzung erfolgt **induktiv**: Die Übersetzung des Gesamtprogramms wird aus den Übersetzungen der Einzelphrasen hergeleitet. Hierfür werden **Code-Funktionen** definiert, die durch **Code-Schablonen** (Templates) konkretisiert werden.

### Code-Funktionen

Eine Code-Funktion bildet eine Phrase (Teil des AST) auf eine Folge von Maschineninstruktionen ab.

Wichtige Code-Funktionen für Triangle/TAM:

- $run[P]$ : Führt das Programm  $P$  aus und hält dann an. Startet und endet mit leerem Stack.
- $execute[C]$ : Führt das Kommando  $C$  aus. Der Stack darf sich während der Ausführung ändern, muss aber am Ende wieder die gleiche Höhe haben (keine Netto-Stack-Änderung).
- $evaluate[E]$ : Wertet den Ausdruck  $E$  aus und legt das Ergebnis oben auf den Stack (Push result).
- $fetch[V]$ : Legt den Wert der Variablen/Konstanten  $V$  auf den Stack.
- $assign[V]$ : Nimmt einen Wert vom Stack und speichert ihn in die Variable  $V$ .
- $elaborate[D]$ : Verarbeitet eine Deklaration  $D$ . Reserviert Speicherplatz auf dem Stack für die deklarierten Variablen.

### 5.2.1 Beispiele für Code-Schablonen

---

#### 1. Sequentielle Ausführung ( $C_1; C_2$ )

$$execute[[C_1; C_2]] = \begin{cases} execute[[C_1]] \\ execute[[C_2]] \end{cases}$$

#### 2. Zuweisung ( $V := E$ )

$$execute[[V := E]] = \begin{cases} evaluate[[E]] & (\text{Berechnet Wert, legt auf Stack}) \\ assign[[V]] & (\text{Speichert Wert vom Stack in V}) \end{cases}$$

#### 3. If-Anweisung ( $\text{if } E \text{ then } C_1 \text{ else } C_2$ ) Hier werden Labels und Sprungbefehle benötigt.

$$execute[\text{if } E \text{ then } C_1 \text{ else } C_2] = \begin{cases} evaluate[[E]] \\ JUMP IF(0) L_{else} & (\text{Springe zu Else, wenn false/0}) \\ execute[[C_1]] \\ JUMP L_{fi} & (\text{Überspringe Else-Zweig}) \\ L_{else} : \\ execute[[C_2]] \\ L_{fi} : \end{cases}$$

#### 4. While-Schleife (Optimierung)

Eine naive Implementierung prüft die Bedingung am Anfang und springt am Ende zurück. Eine effizientere Variante (weniger Sprünge im “Steady State”) nutzt folgenden Aufbau:

$$execute[\text{while } E \text{ do } C] = \begin{cases} JUMP L_{check} \\ L_{loop} : \\ execute[[C]] \\ L_{check} : \\ evaluate[[E]] \\ JUMP IF(1) L_{loop} & (\text{Springe zurück, wenn true}) \end{cases}$$

*Anmerkung:* In der Vorlesung wurde auch die Standard-Variante mit  $L_{while}$  am Anfang und  $JUMP IF(0)$  zum Ende besprochen.

**Sonderfallbehandlung (Optimierung):** Anstatt generische Schablonen zu nutzen, können spezielle Muster erkannt werden:

- $i + 1$ : Statt ‘LOAD i’, ‘LOADL 1’, ‘CALL add’ → ‘LOAD i’, ‘CALL succ’.
- **Constant Inlining:** Wenn der Wert einer Konstanten zur Kompilierzeit bekannt ist, muss kein Speicherzugriff (‘LOAD’) erfolgen. Stattdessen wird der Wert direkt mit ‘LOADL’ in den Code eingebettet.

## 5.3 Implementierung mittels Visitor-Pattern

---

Der Code-Generator wird systematisch mit dem Visitor-Pattern implementiert. Verschiedene Encoder-Klassen (z.B. ‘CommandEncoder‘, ‘ExpressionEncoder‘) besuchen die entsprechenden AST-Knoten.

### 5.3.1 Backpatching (Rückwärts-Einsetzen von Adressen)

---

Ein Problem bei der Generierung von Kontrollstrukturen (wie ‘if‘ oder ‘while‘) sind Vorwärtssprünge, da die Zieladresse des Sprungs zum Zeitpunkt der Generierung des Sprungbefehls noch nicht bekannt ist (der Code dazwischen wurde noch nicht erzeugt).

#### Lösung: Backpatching

1. Erzeuge Sprunginstruktion mit einer “leeren” Zieladresse (z.B. 0).
2. Merke die Adresse dieser unvollständigen Instruktion.
3. Generiere den Code für den Rumpf/Block.
4. Sobald die Zieladresse erreicht ist (aktuelle Code-Adresse), **patche** die gemerkte Instruktion nachträglich mit der korrekten Adresse.

```
1 int jumpAddr = nextInstrAddr;
2 emit(Machine.JUMPIFop, 0, Machine.CBr, 0); // Dummy Ziel 0
3 // ... Generiere Code fuer Block ...
4 int targetAddr = nextInstrAddr;
5 patch(jumpAddr, targetAddr); // Trage echtes Ziel nach
```

*Pseudocode Backpatching*

## 5.4 Speicherverwaltung und Adressierung

---

### 5.4.1 Verwaltung im AST (Runtime Entities)

---

Der Code-Generator muss wissen, wo Variablen liegen oder welche Werte Konstanten haben. Diese Information wird in **Entitätsdeskriptoren** (‘RuntimeEntity‘) gespeichert und an die AST-Knoten gehängt.

- **KnownValue:** Für Konstanten mit bekanntem Wert. Speichert den Wert direkt.
- **KnownAddress:** Für Variablen/Konstanten mit bekannter Adresse. Speichert Größe und Adresse (Level, Displacement).
- **UnknownValue:** Wert wird erst zur Laufzeit berechnet (z.B. ‘const c x + 5‘).
- **UnknownAddress:** Für Referenzparameter (‘var‘-Parameter), deren Adresse erst zur Laufzeit bekannt ist.

### 5.4.2 Adressvergabe

---

Der Compiler führt Buch über den belegten Speicherplatz. Dies geschieht oft über einen Parameter in den Visitor-Methoden (z.B. ‘gs‘ für Global Space oder ‘frame‘).

- **Eingabe-Parameter:** Aktueller Offset / erste freie Adresse.
- **Rückgabewert:** Größe des durch die Deklaration zusätzlich belegten Speichers.

Vorteil der Rückgabe des Deltas: Beim Verlassen eines Blocks (Scope) muss keine globale Variable zurückgesetzt werden; die lokalen Änderungen “verfallen” automatisch.

## Globale Variablen

var	size	address
a	1	[0] SB
b	1	[1] SB
c	1	[2] SB

In TAM, echte Maschinen haben hier wahrscheinlich unterschiedliche Größen

## Verschachtelte Blöcke

let var a: Integer in begin ... let var b: Boolean; var c: Integer in ... let var d: Integer in ... end
---

var	size	address
a	1	[0] SB
b	1	[1] SB
c	1	[2] SB
d	1	[1] SB

d verwendet Platz von b wieder  
(anderer Geltungsbereich)

### 5.4.3 Adressierung in verschachtelten Blöcken (Triangle)

In Sprachen mit verschachtelten Prozeduren (wie Triangle, im Gegensatz zu Mini-Triangle) reicht eine einfache Adresse relativ zu 'SB' (Stack Base) nicht aus.

#### Adress-Tupel

Jede Variable wird durch ein Paar identifiziert: (**Schachtelungstiefe, Displacement**).

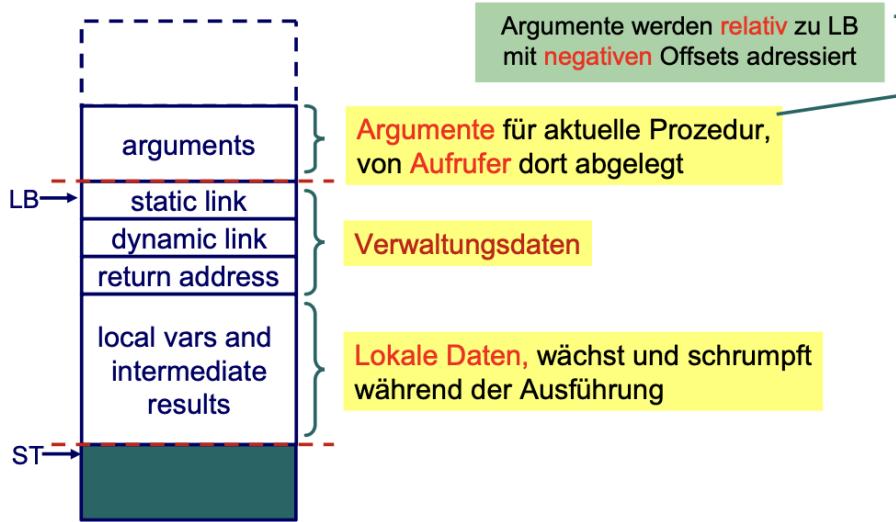
Der Zugriff erfolgt über **Display-Register** oder 'LB' (Local Base) / 'SB' (Stack Base):

**Algorithmus zur Wahl des Basisregisters:** Sei  $level_{decl}$  die Ebene, auf der die Variable deklariert wurde, und  $level_{current}$  die aktuelle Ausführungsebene.

- Wenn  $level_{decl} == 0$ : Benutze **SB** (Globale Variable).
- Wenn  $level_{decl} == level_{current}$ : Benutze **LB** (Lokale Variable im aktuellen Frame).
- Sonst: Benutze Display-Register oder Statische Verkettung, um den Frame zu finden.

$$r = L(level_{current} - level_{decl})$$

Oder über Display-Register direkt, sofern die Architektur diese unterstützt.



## 5.5 Prozeduren und Funktionen

### 5.5.1 Deklaration

Da der Code linear im Speicher liegt, aber Prozeduren nur bei Aufruf ausgeführt werden sollen, muss der Kontrollfluss um den Prozedurrumpf herumgeleitet werden.

Code-Schema für Deklaration ‘proc I() C’:

1. **JUMP g**: Überspringe den Prozedurcode.
2. **Label e**: Einstiegspunkt (Entry) der Prozedur.
3. **execute[C]**: Code des Rumpfes.
4. **RETURN**: Rückkehr zum Aufrufer.
5. **Label g**: Ziel des Sprungs von Schritt 1 (weiter im Hauptprogramm).

### 5.5.2 Aufruf (Call)

Beim Aufruf (‘CALL’) muss die **Statische Verkettung** (Static Link) korrekt gesetzt werden, damit die aufgerufene Prozedur auf Variablen der umgebenden Scopes zugreifen kann.

- Die Adresse der Routine ist im Deskriptor gespeichert (‘KnownRoutine’).
- Der ‘CALL’-Befehl erhält das korrekte Basisregister (siehe Adressierung), welches als Static Link in den neuen Stack Frame geschrieben wird.

### 5.5.3 Parameterübergabe

- **Aufrufer**: Legt die aktuellen Parameter (Argumente) auf den Stack.
- **Aufgerufener**: Greift auf Parameter relativ zu ‘LB’ mit **negativen Offsets** zu (da sie vor dem Link-Data-Bereich auf dem Stack liegen).
- **Var-Parameter**: Werden als ‘UnknownAddress’ behandelt. Es wird nicht der Wert, sondern die Adresse übergeben und bei ‘fetch’/‘assign’ genutzt.

# 6 Lexer/Parser-Generierung mit ANTLR

## 6.1 Einführung und Grundlagen

ANTLR (Another Tool for Language Recognition) ist ein Generator für Lexer und Parser. In der aktuellen Version 4 setzt es auf einen adaptiven *LL(\*)*-Algorithmus (auch *ALL(\*)* genannt).

### ANTLR v4 Eigenschaften

- **Adaptive *LL(\*)*:** Die Grammatik wird zur Laufzeit analysiert. Der Parser nutzt Heuristiken, um Konflikte und Mehrdeutigkeiten automatisch aufzulösen.
- **Direkte Linksrekursion:** Wird im Gegensatz zu klassischen LL-Parsern unterstützt (wird intern transformiert). Indirekte Linksrekursion ist weiterhin nicht erlaubt.
- **Trennung von Grammatik und Code:** Semantische Aktionen (eingebetteter Java-Code) werden vermieden. Stattdessen werden Parse-Trees automatisch erstellt und über *Listener* oder *Visitor* traversiert.

## 6.2 Struktur einer Grammatik

Eine ANTLR-Grammatikdatei (Endung ‘.g4’) definiert Lexer- und Parser-Regeln.

- **Parser-Regeln:** Beginnen mit einem **Kleinbuchstaben** (z.B. `stat`, `expr`). Sie definieren die syntaktische Struktur.
- **Lexer-Regeln (Token):** Beginnen mit einem **Großbuchstaben** (z.B. `ID`, `INT`). Sie definieren die lexikalischen Einheiten (Zeichenfolgen).

### 6.2.1 Operatoren und Syntax

Die Notation orientiert sich an EBNF (Extended Backus-Naur Form).

Operator	Bedeutung
<code>x y</code>	Konkatenation (x gefolgt von y)
<code>x   y</code>	Alternative (x oder y)
<code>x*</code>	0 oder mehr Wiederholungen (Kleene-Stern)
<code>x+</code>	1 oder mehr Wiederholungen
<code>x?</code>	0 oder 1 Mal (optional)
<code>~[a-z]</code>	Negation (alle Zeichen außer a-z)
<code>.*?</code>	<b>Non-Greedy Match:</b> Matcht so wenig Zeichen wie möglich (wichtig für Kommentare!)

### Non-Greedy Beispiel

Um beispielsweise `/* ... */` Kommentare korrekt zu parsen, darf der `*?`-Operator nicht gierig (greedy) sein, da er sonst bis zum allerletzten `*/` der Datei lesen würde.

Lösung: `'/*' .*? '*'` (stoppt beim ersten Vorkommen von `*/`).

## 6.3 Generierte Artefakte und Parse-Trees

ANTLR erzeugt aus der Grammatik Java-Klassen (oder andere Zielsprachen), die den Input in einen abstrakten Parse-Tree (AST) verwandeln.

### Wichtige Klassen:

- **Parser:** Überprüft die syntaktische Struktur.

- **Lexer:** Zerlegt den Input-Stream in Tokens.
- **ParserRuleContext:** Repräsentiert innere Knoten im Baum (Regeln). Speichert Start-/End-Tokens und Referenzen auf Kinder.
- **TerminalNode:** Repräsentiert Blätter im Baum (Tokens/Literale).

## 6.4 Traversierung: Listener vs. Visitor

---

ANTLR v4 bietet zwei Mechanismen, um den Parse-Tree zu verarbeiten.

### 6.4.1 1. Listener Pattern

---

Der Listener ist der Standardmechanismus. ANTLR generiert einen `ParseTreeWalker`, der den Baum mittels Tiefensuche (DFS) automatisch traversiert.

- **Funktionsweise:** Der Listener reagiert auf Ereignisse. Beim Betreten eines Knotens wird `enterX()` aufgerufen, beim Verlassen `exitX()`.
- **Vorteil:** Vollautomatisch, kein Boilerplate-Code für die Traversierung nötig.
- **Nachteil:** Keine Kontrolle über den Ablauf (Reihenfolge ist fix), keine Rückgabewerte aus Methoden, Kommunikation zwischen Knoten schwierig (oft über Instanzvariablen oder einen Stack).

### 6.4.2 2. Visitor Pattern

---

Muss explizit mit `-visitor` generiert werden. Hier steuert der Entwickler die Traversierung selbst.

- **Funktionsweise:** Man implementiert `visitX(Context ctx)`. Um Kinder zu besuchen, muss man explizit `visit(ctx.child)` aufrufen.
- **Vorteil:** Volle Kontrolle (Überspringen von Zweigen, geänderte Reihenfolge), Methoden können Werte zurückgeben (z.B. `Integer` für einen Taschenrechner).
- **Nachteil:** Man muss die Traversierung (den Aufruf von `visit` für Kinder) selbst schreiben.

#### Vergleich für die Prüfung

Nutzen Sie **Listener**, wenn Sie den gesamten Baum standardmäßig abarbeiten wollen (z.B. Code-Formatierung, einfache Übersetzung).

Nutzen Sie **Visitor**, wenn Sie Ergebnisse berechnen (Interpreter), den Kontrollfluss steuern oder kontextabhängig traversieren müssen.

### 6.4.3 Labeling von Alternativen

---

Damit der Visitor/Listener spezifische Methoden für Alternativen generiert (statt einer riesigen Methode mit 'if/else'), können Alternativen mit '#' benannt werden.

**Beispiel:**

```
expr : expr '*' expr # MulDiv
      | expr '+' expr # AddSub
      | INT          # Int
      ;
```

Erzeugt Methoden: `visitMulDiv`, `visitAddSub`, `visitInt`.

## 6.5 Fortgeschrittene Themen

---

### 6.5.1 Assoziativität und Präzedenz

---

In ANTLR v4 wird Operator-Präzedenz implizit durch die Reihenfolge der Alternativen bestimmt.

- **Präzedenz:** Regeln, die weiter oben stehen, binden stärker (haben höhere Priorität). Beispiel: ‘\*’ vor ‘+’ definieren.
- **Assoziativität:** Standard ist links-assoziativ ( $1 + 2 + 3 \rightarrow (1 + 2) + 3$ ). Rechts-Assoziativität (z.B. für Exponenten  $2^{3^4}$ ) wird mit `<assoc=right>` annotiert.

### 6.5.2 Dangling Else Problem

---

Das Problem der Mehrdeutigkeit bei ‘if expr then if expr then stat else stat’: Gehört das ‘else’ zum ersten oder zweiten ‘if’?

**Lösung in ANTLR:** ANTLR parst **greedy** (gierig). Es bindet das ‘else’ an das nächstmögliche (innerste) offene ‘if’. Dies entspricht dem Standardverhalten der meisten Programmiersprachen.

### 6.5.3 Semantische Prädikate

---

Manchmal reicht die Grammatik allein nicht aus (z.B. wenn Parsing von Laufzeitdaten abhängt).

**Syntax:** `{Bedingung}?`

Dies ist ein boolescher Ausdruck in der Zielsprache (Java).

- Ist die Bedingung `true`, wird die Alternative/Regel aktiviert.
- Ist sie `false`, wird die Alternative ignoriert (als ob sie nicht in der Grammatik stünde) und der Parser versucht eine andere Möglichkeit.

**Beispiel (Datenabhängiges Parsen):** Eine Zahl  $n$  gibt an, wie viele folgende Zahlen gelesen werden sollen.

```
sequence[int n] locals [int i = 1;]
  : ( {$i <= $n}? INT {$i++;} )* ;
```

Hier deaktiviert das Prädikat die Schleife, sobald  $i > n$  ist.

## 6.6 Fehlerbehandlung

---

ANTLR generiert Parser mit eingebauter Fehlerbehandlung.

- **Token ignorieren:** Wenn ein Token unerwartet ist, aber das darauf folgende passt.
- **Token einfügen:** Wenn ein Token fehlt, fügt der Parser ein fiktives Token ein, um weiterzumachen (Single Token Insertion).
- Ziel ist es, möglichst viele Fehler in einem Durchlauf zu finden (Error Recovery), statt beim ersten Fehler abzubrechen.