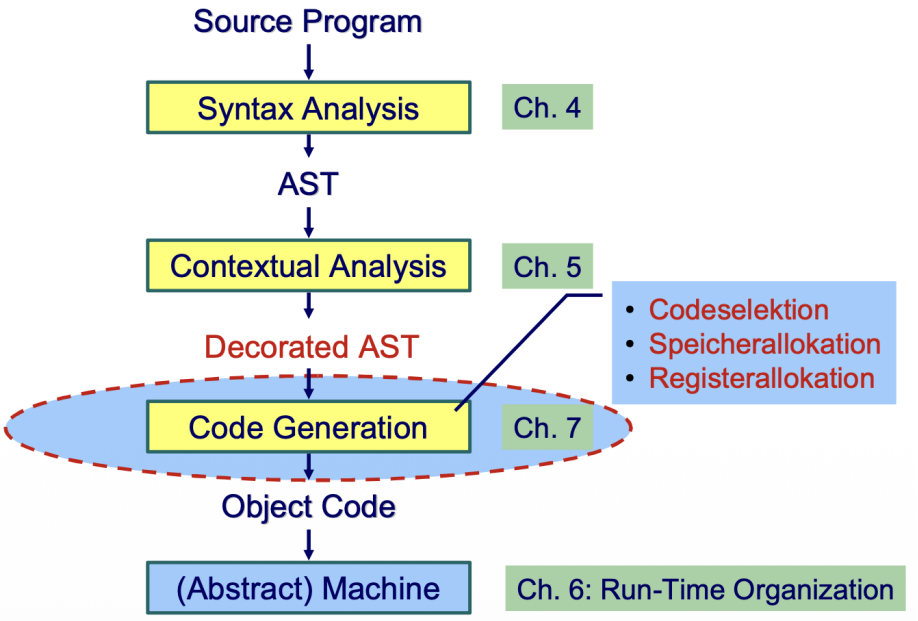
1 Code-Generierung



1.1 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. **Registerallokation:** Verwaltung der Register für Variablen und Zwischenergebnisse (bei der TAM als Stackmaschine weitestgehend irrelevant, da alles auf dem Stack passiert).

1.2 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.

1.2.1 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 ($if\ E\ then\ C_1\ else\ C_2$) Hier werden Labels und Sprungbefehle benötigt.

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

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:

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

Anmerkung: In der Vorlesung wurde auch die Standard-Variante mit L_{while} am Anfang und $JUMPIF(0)$ 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.

1.3 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.

1.3.1 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.

```
int jumpAddr = nextInstrAddr;
emit(Machine.JUMPIFop, 0, Machine.CBr, 0); // Dummy Ziel 0
// ... Generiere Code f r Block ...
int targetAddr = nextInstrAddr;
patch(jumpAddr, targetAddr); // Trage echtes Ziel nach
```

Pseudocode Backpatching

1.4 Speicherverwaltung und Adressierung

1.4.1 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.

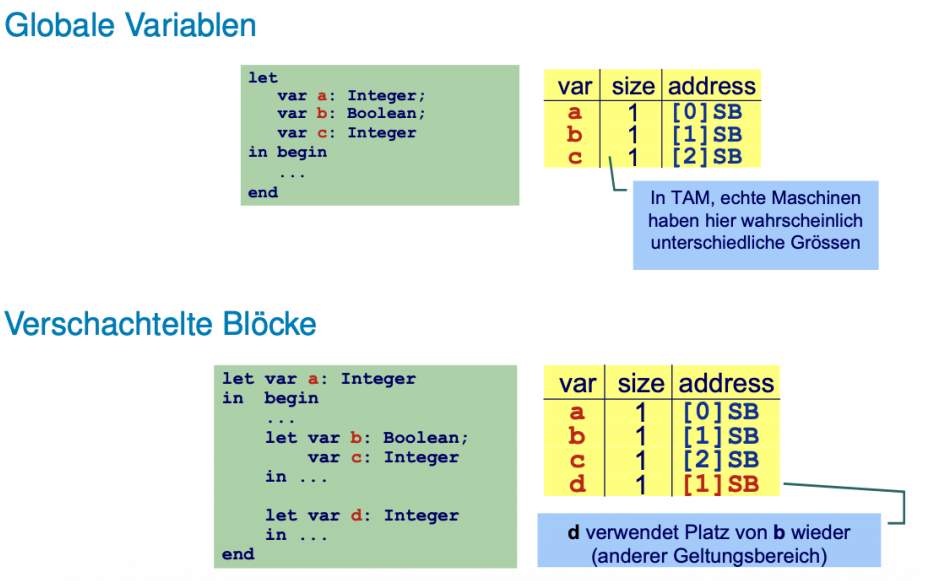
- **KnownValue:** Für Konstanten mit bekanntem Wert. Speichert den Wert direkt.
- **KnownAddress:** Für Variablen/Konstanten mit bekannter Adresse. Speichert Größe und Adresse (Level, Displacement).
- **UnknownValue:** Wert wird erst zur Laufzeit berechnet (z.B. ‘const c x + 5‘).
- **UnknownAddress:**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

1.4.2 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.



1.4.3 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)**.

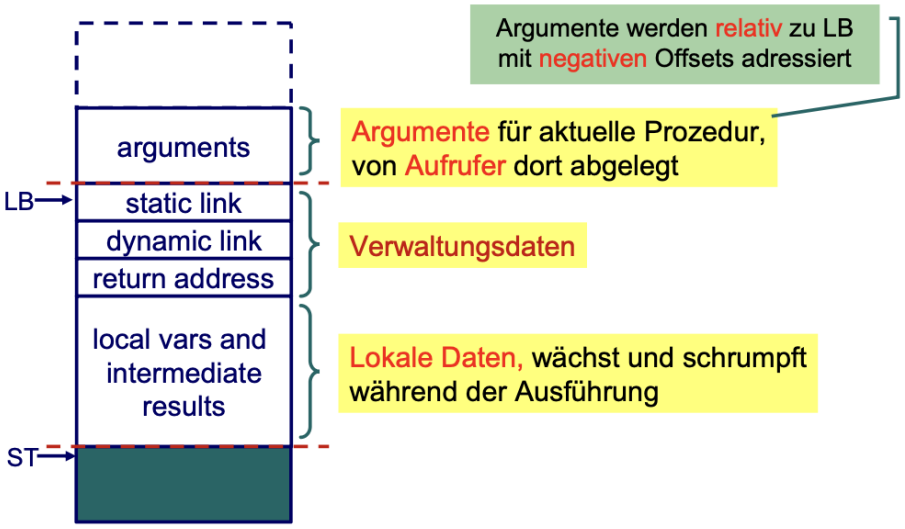
Der Zugriff erfolgt über **Display-Register** oder 'LB' (Local Base) / 'SB' (Stack Base):

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.



1.5 Prozeduren und Funktionen

1.5.1 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).

1.5.2 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.

1.5.3 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.