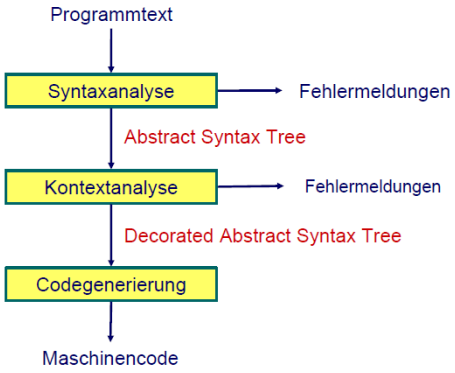
Compiler I

Kompilierung

Vom Quellcode … zum MaschinenCode



Entspricht häufig den Teilen der Sprachspezifikation

1. Syntax → Syntaxanalyse
2. Kontextuelle Einschränkungen → Kontextanalyse
3. Semantik →Codegenerierung

Ein-Pass Compiler

- Geht nur ein einziges Mal über das Programm
  - Baut i.d.R. keine Zwischendarstellung auf
- Führt alle Phasen gleichzeitig aus
- z.B.: häufig Pascal Compiler

Multi-Pass Compiler

- Geht mehrmals über das Programm
  - Programm = Quelltext oder Zwischendarstellung (IR)

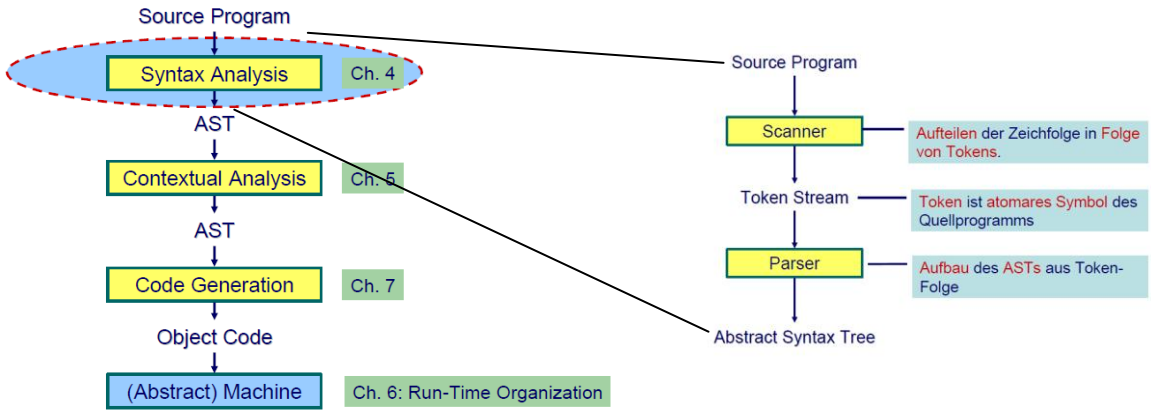
Vergleich Ein-/Multi-Pass Compiler

	Ein-Pass	Multi-Pass
Laufzeit	+	-
Speicher	+ (für große Programme)	+ (für kleine Programme)
Modularität	-	+
Flexibilität	-	+
Globale Optimierung	--	+
Eingabesprache	Nicht für alle	

Nur Möglich bei Sprachen, bei denen Bezeichner vor ihrer Verwendung deklariert werden

Syntaxanalyse

1. Pass, in 2 Phasen aufgeteilt



Scanner

Auch lexikalische Analyse oder Lexer genannt

Aufgabe

- Bilde Token aus Zeichen
- Entferne unerwünschte Leerzeichen, Zeilenvorschübe
- Führe Buch über Zeilennummer und Eingabedateinamen

Lässt sich mit regulären Ausdrücken (Automat) lösen.

Alternativ dazu:

Rekursiver Abstieg
Analog zum Parser

Normalerweise werden Kommentare übernommen, gibt aber auch Scanner, die in Kommentare reingucken (JavaDoc Generierung)

Lexikalische Grammatik in EBNF

```
Token ::= Identifier | Integer-Literal | Operator |
        ; | : | := | ~ | ( | ) | eot
Identifier ::= Letter (Letter | Digit)*
Integer-Literal ::= Digit Digit*
Operator ::= + | - | * | / | < | > | =
Separator ::= Comment | space | eol
Comment ::= ! Graphic* eol
```

Diese wird transformiert

```
Token ::= Letter (Letter | Digit)*
        | Digit Digit*
        | + | - | * | / | < | > | =
        | ; | : (=|ε) | ~ | ( | ) | eot
Separator ::= ! Graphic* eol | space | eol
```

Diese Transformierung ist nicht unbedingt nötig, macht es allerdings schneller, da es weniger Methoden gibt. Man kann nun aber nicht mehr zwischen Schlüsselwörtern und Bezeichner unterscheiden, dies muss während des Scannens repariert werden.

Implementierung

```
public Token scan() {
   // Get rid of potential separators before
   // scanning a token
   while (  (currentChar == '!')
         || (currentChar == ' ')
         || (currentChar == '\n' ) )
      scanSeparator();
   currentSpelling = new StringBuffer();
   currentKind = scanToken();
   return new Token(currentkind,
                    currentSpelling.toString());
}
```

```
private byte scanToken() {
  switch (currentChar) {
     case 'a': case 'b': ... case 'z':
     case 'A': case 'B': ... case 'Z':
       scan Letter (Letter | Digit)*
       return Token.IDENTIFIER;
     case '0': ... case '9':
       scan Digit Digit*
       return Token.INTLITERAL ;
     case '+': case '-': ... : case '=':
       takeIt();
       return Token.OPERATOR;
     ...etc...
}
```

```
case 'a': case 'b': ... case 'z':
case 'A': case 'B': ... case 'Z':
  takeIt();
  while (isLetter(currentChar)
        || isDigit(currentChar) )
    takeIt();
  return Token.IDENTIFIER;
```

```
public class Token {
...

        private static String[] tokenTable = new String[] {
        "<int>",    "<char>",    "<identifier>",    "<operator>",
        "array",    "begin",    "const",    "do",    "else",    "end",
        "func",    "if",    "in",    "let",    "of",    "proc",    "record",
        "then",    "type",    "var",    "while",
        ".",    "-",    ",",    ",",    ":=",    "~",    "(",    ")",    "[",    "]",    "{",    "}",    "",
        "<error>"  };

        private final static int firstReservedWord = Token.ARRAY,
                                 lastReservedWord  = Token.WHILE;
...
}
```

Parser

Parsen der Token Folge in einen Abstrakten Syntaxbaum (AST)

Token

- Token ist atomares Symbol des Programms
- Zeichen selbst i.d.R. uninteressant, Ausnahme:
  - Bezeichnernamen
  - Konstante Werte (Zahlen, Zeichen), sog. Literale

Der Parser ist zum aufbauen des AST nur an der Art des jeweiligen Tokens interessiert.

Nur wenige Tokens tauchen später im AST wirklich auf. Viele (z.B. Schlüsselwörter) bestimmen aber implizit die Struktur des AST.

Grammatiken

Kontextfreie Grammatiken (CFG) sind Spezifiziert durch ein Tupel von
(nicht Terminalsymbolen, Terminalsymbolen, Produktionen, Startsymbolen $\in$ nicht Terminalsymbolen)

Die Produktionen werden häufig in der Backus-Naur-Form (BNF) angegeben. Übersichtlicher ist es allerdings die Extended BNF (EBNF) zu benutzen. Diese dürfen auf der rechten Seite (RHS) sowohl BNF als auch Reguläre Ausdrücke enthalten

Transformation von Grammatiken

CFG kann transformiert werden, unter Beibehaltung der beschriebenen Sprache.
Dies ist sehr nützlich bei der Konstruktion von Parsern für CFGs

	Vor Transformation	Nach Transformation
Gruppierung	$S ::= X + S$ $S ::= X$ $S ::= \varepsilon$	$S ::= X + S \mid X \mid \varepsilon$
Linksausklammern	$S ::= XY \mid XZ$	$S ::= X\,(Y \mid Z)$
Linksrekursion	$N ::= X \mid NY$	$N ::= X(Y)*$
Ersetzung von Nicht-Terminalsymbolen	$N ::= X$	Wenn nur eine Produktion mit LHS N, dann in RHS allen Produktionen N durch X ersetzen n

Es können für den Menschen nützliche Informationen verloren gehen. Das Ergebnis ist allerdings für den Compieler besser

Terminologie

- *Erkennung*
  - Entscheidung, ob ein Eingabetext ein Satz der Grammatik G ist
- *Parsing*
  - Erkennung und zusätzlich Bestimmung der Phrasen-Struktur
    - z.B. durch konkreten/abstrakten Syntaxbaum
- *Eindeutigkeit*
  - Eine Grammatik ist eindeutig, falls jeder Eingabetext auf maximal eine Weise geparsed werden kann

Strategien

Beispiel: MicroEnglish

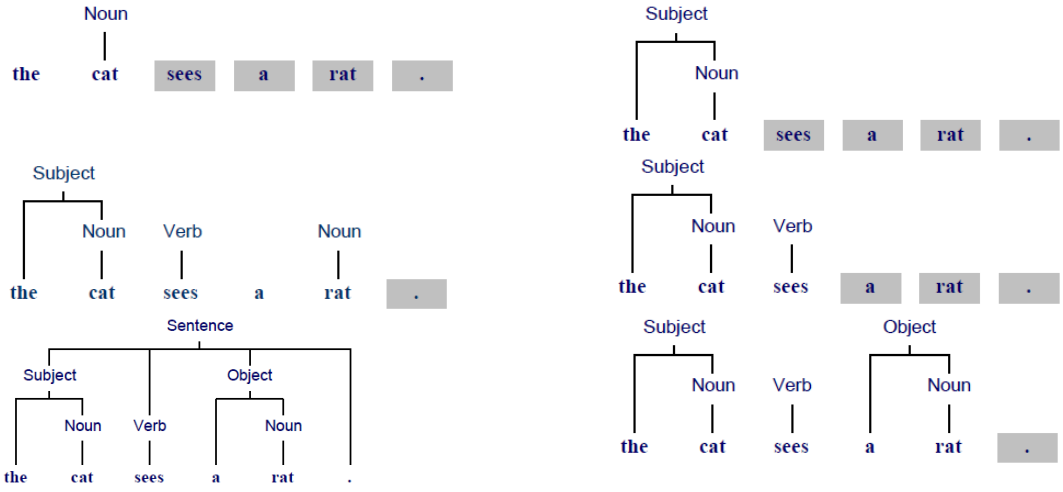
$$
\begin{array}{lll}
Sentence & ::= & Subject\ Verb\ Object\ . \\
Subject & ::= & I \mid a\ Noun \mid the\ Noun \\
Object & ::= & me \mid a\ Noun \mid the\ Noun \\
Noun & ::= & cat \mid mat \mid rat \\
Verb & ::= & like \mid is \mid see \mid sees
\end{array}
$$

Bottom-Up
Untersucht EingabeText Zeichenweise, von links nach rechts und baut Syntaxbaum von unten nach oben auf.

Aktionen

- Shift
  - Lese Zeichen ein und lege es auf Stack
- Reduce
  - Erkenne ein Nicht-Terminal LHS der Produktion p
    - Zusätzlich: Oberste Elemente des Stapels müssen RHS von p entsprechen. Ersetze durch LHS von p (auf den Stack legen)
    - Ende wenn Startsymbol S erreicht und Eingabetext komplett gelesen

Beispiel:



Schwierigkeiten beim Bottom-Up

Wie entscheiden, welche Produktion beim Zusammenfassen gewählt werden soll, wenn es mehrere Möglichkeiten gibt?

Lösung: Nicht nur bekannte Zeichen betrachten, sondern auch Zustand einbeziehen.

Arbeitet mit LR(k)-Technik

- L: Lese Eingabetext von links nach rechts
- R: Fasse die am weitesten rechts stehenden Terminal-Symbole zusammen und baue von unten auf

Top-Down
Rekursiver Abstieg

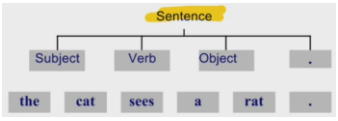
Untersuche Eingabetext Token weise, von links nach rechts
Baue Syntaxbaum von oben nach unten auf

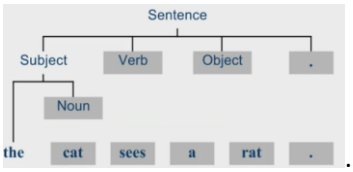
- Von Start-Nicht-Terminalsymbol in der Wurzel …
- … zu den Terminalsymbolen in den Blättern

Aktionen

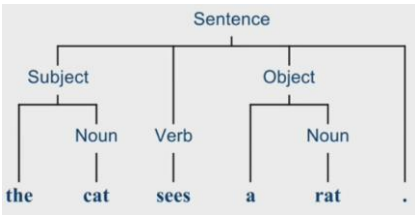
- Expandiere jeweils das am Weitesten links gelegene Nicht-Terminal N durch die Anwendung einer Produktion N ::= X
- Wähle Produktion aus durch betrachten der nächsten n Zeichen (Token) (LookAhead) des Eingabetextes (hier n = 1)
- Falls keine Produktion auf Zeichen passt → Fehler
- Ende wenn Eingabetext komplett gelesen und keine nicht expandierten Nicht-Terminale mehr existieren

Beispiel:

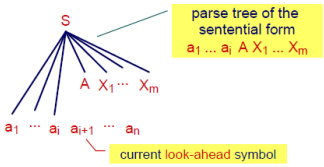




…



Hintergrund



Falls es möglich ist …

- … bei Betrachtung der nächsten k Zeichen (Tokens) des Textes
- … immer die richtige Produktion zu finden

… dann ist die Grammatik LL(k)

- L: Lese EingabeText von links nach rechts,
- L: Leite immer vom am weitesten Links stehenden Nicht-Terminal ab

Konstruktion von LL(k) Grammatik kann mühsam sein.
Durch Transformation kann die Lesbarkeit erschwert werden.

Implementierung
Top-Down Parser

Die Struktur des konkreten Syntaxbaumes entspricht dem Aufrufmuster von sich wechselseitig aufrufenden Prozeduren. Für jedes Nicht-Terminal XYZ existiert eine Parse-Prozedur parseXYZ, die genau dieses Nicht-Terminal parst.

Sentence ::= Subject Verb Object .

```
protected void parseSentence() {
   parseSubject();
   parseVerb();
   parseObject();
   accept(".");
}
```

accept(t) prüft, ob aktuelles Token das erwartete Token t ist.

Subject ::= I | a Noun | the Noun

```
protected void parseSubject() {
   if (currentToken matches "I") {
      accept("I");
   } else if (currentToken matches "a") {
      accept("a");
      parseNoun();
   } else if (currentToken matches "the") {
      accept("the");
      parseNoun();
   } else
      report a syntax error
}
```

Die Methode muß immer anhand von currentToken die passende Alternative auswählen können.

Ablauf einer parseN Methode

- Bei Eintritt enthält currentToken eines der Token, mit denen N beginnen kann
- … sonst wäre eine andere Parse-Methode aufgerufen worden (oder Syntaxfehler liegt vor)

Ablauf einer accept(t) Methode

- Bei Eintritt muss currentToken = t sein
- … sonst Syntaxfehler
- Bei Austritt enthält currentToken das auf t folgende Token

Entwicklung von Parsern mit rekursivem Abstieg

1. Formuliere Grammatik (CFG) in EBNF
    a. Eine Produktion pro Nicht-Terminal
    b. Beseitige IMMER Linksrekursion
    c. Klammer gemeinsame Teilausdrücke nach links aus wo möglich
2. Erstelle Klasse für den Parser mit
    a. protected Variable currentToken
    b. Schnittstellenmethoden zum Scanner
        i. accept(t) und acceptIT() [z.B. bei let muss nicht auf „let“ überprüft werden]
    c. Public Methode parse welche …
        i. Erstes Token via Scanner aus dem Eingabetext liest
        ii. Die Parse-Methode des Start Nicht-Terminals S der CFG aufruft
3. Implementiere protected Parsing Methoden
    a. Methode parseN für jedes NichtTerminalsymbol N

LL(k) Grammatik

$starters[[X]]$

$$starters[[ab]] = \{a\}$$
$$starters[[a|b]] = \{a, b\}$$
$$starters[[(re) * set]] = \{r, s\}$$

$follow[[X]]$

$$N \quad ::= \quad XY$$

$$X ::= a \mid b$$
$$Y ::= c \mid d$$

$$follow[[N]] = \{\ \}$$
$$follow[[X]] = \{c, d\}$$
$$follow[[Y]] = \{\ \}$$

Eine Grammatik G ist LL(1), wenn gilt:

- Falls G X|Y enthält und sich weder X noch Y zu Epsilon ableiten lassen:
  $starters[[X]] \cap starters[[Y]] = \emptyset$
- Falls G X|Y enthält und sich beispielsweise Y zu Epsilon ableiten lässt:
  $starters[[X]] \cap (starters[[Y]] \cup follow[[X|Y]] = \emptyset$
- Falls G X* enthält :
  $starters[[X]] \cap follow[[X*]] = \emptyset$

Man kann manche nicht LL(1) Grammatiken in LL(1) Grammatiken transformieren

Beispiel: Wenn G nicht LL(1), aber Schema trotzdem angewandt

```
single-Command  ::= V-name := Expression
                  | Identifier ( Expression )
                  | if Expression then single-Command
                        else single-Command
                  | ...
```

starters[[V-name := Expression]] = starters[[V-name]]
= { Identifier }
starters[[Identifier (Expression)]] = { Identifier }
starters[[if Expression then ...]] = { if }

```
private void parseSingleCommand () {
   switch (currentToken.kind) {
   case Token.IDENTIFIER: {
      parseVname();
      accept(Token.BECOMES);
      parseExpression(); }
      break;
   case Token.IDENTIFIER: {
      parseIdentifier();
      accept(Token.LPAREN);
      parseExpression();
      accept(Token.RPAREN)
   }
```

➔

Beispiel: Fehler wenn Linksausklammern vergessen

```
single-Command ::= Identifier := Expression
                 | Identifier ( Expression )
                 | if Expression then single-Command
                       else single-Command
```

starters[[Identifier := Expression]] = { Identifier }

starters[[Identifier (Expression)]] = { Identifier }

Beispiel: Fehler wenn Linksrekursion nicht eliminiert

```
Command ::= single-Command
          | Command ; single-Command
```

starters[[single-Command]]
= { Identifier, if, while, let, begin }
starters[[Command ; single-Command]]
= { Identifier, if, while, let, begin }

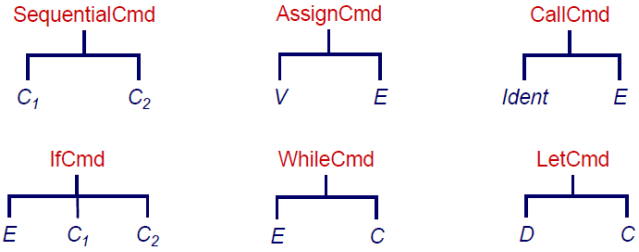
```
private void parseCommand () {
   switch (currentToken.kind)
   case Token.IDENTIFIER:
   case Token.IF:
   case Token.WHILE:
   case Token.LET:
   case Token.BEGIN:
      parseSingleCommand();
      break;
   case Token.IDENTIFIER:
   case Token.IF:
   case Token.WHILE:
   case Token.LET:
   case Token.BEGIN: {
      parseCommand();
      accept(Token.SEMICOLON)
      parseSingleCommand();
      }
      break;
```

Abstrakte Syntaxbäume

Unser bisheriger Parser baut mit seinem rekursiven Abstieg implizit einen Syntaxbaum auf, dieser wird allerdings noch nicht gespeichert. Die rekursiven Aufrufe werden benutzt, um den Baum aufzubauen

Der AST basiert auf der nicht transformierten Grammatik von Triangle

```
Program       ::= Command                                  Program
Command       ::= Command ; Command                        SequentialCmd
                | V-name := Expression                     AssignCmd
                | Identifier ( Expression )                CallCmd
                | if Expression then single-Command         IfCmd
                      else single-Command
                | while Expression do single-Command        WhileCmd
                | let Declaration in single-Command         LetCmd
Expression    ::= Integer-Literal                          IntegerExpr
                | V-name                                   VnameExpr
                | Operator Expression                      UnaryExpr
                | Expression Operator Expression           BinaryExpr
V-name        ::= Identifier                               SimpleVname
Declaration   ::= Declaration ; Declaration                SeqDecl
                | const Identifier ~ Expression            ConstDecl
                | var Identifier : Type-denoter            VarDecl
Type-denoter  ::= Identifier                               SimpleTypeDen
```



Repräsentation des AST in der Java Implementierung

```
abstract class Command
   extends AST { ... }
```

```
Command
    ::= Command ; Command                    SequentialCmd
    |   V-name := Expression                 AssignCmd
    |   Identifier ( Expression )            CallCmd
    |   if Expression then single-Command    IfCmd
            else single-Command
    |   while Expression do single-Command   WhileCmd
    |   let Declaration in single-Command    LetCmd
```

```
public class SequentialCmd extends Command {
   public Command c1, c2;
   ...
}
public class AssignCmd extends Command {
   public Vname      v;
   public Expression e;
   ...
}
public class CallCmd extends Command {
   public Identifier i;
   public Expression e;
   ...
}
public class IfCmd extends Command {
   public Expression e;
   public Command    c1, c2;
   ...
```

etc.

Die AST Subklassen haben auch entsprechende Konstruktoren zur korrekten Initialisierung der Objekte.

Aufbau des AST

EBNF

Command ::= single-Command (; single-Command)∗

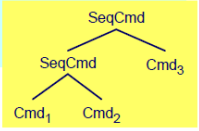
Wird vom Parser verwendet

AST

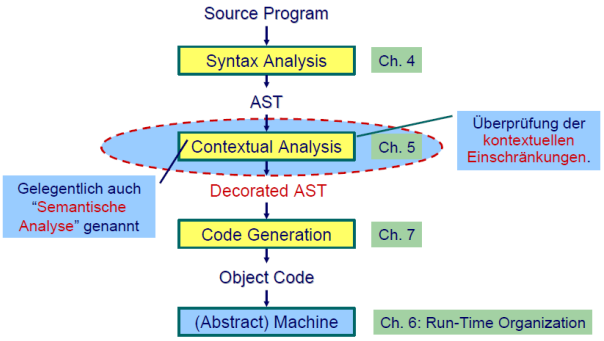
Command ::= Command ; Command SequentialCmd

Wird für den AST verwendet

```
protected Command parseCommand() {
   Command c1AST = parseSingleCommand();
   while (currentToken.kind == Token.SEMICOLON) {
      acceptIt();
      Command c2AST = parseSingleCommand();
      c1AST = new SequentialCmd(c1AST, c2AST);
   }
   return c1AST;
}
```

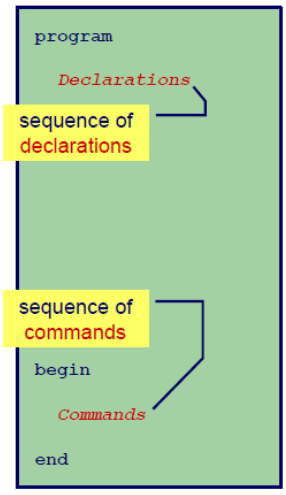


Kontextanalyse (Semantische Analyse)

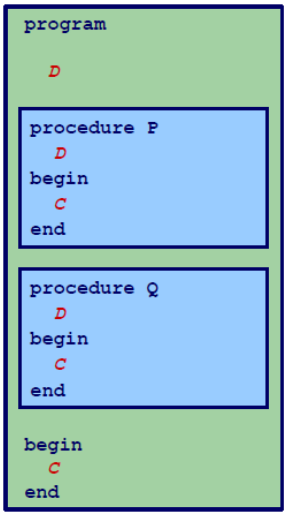


Geltungsbereiche und Symboltabellen

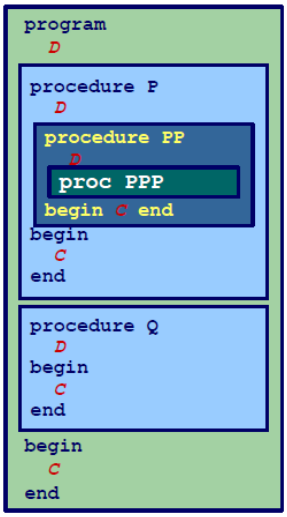
Monolithische Blockstruktur



- Charakteristika
  - Nur ***ein*** Block
  - Alle Deklarationen gelten ***global***
- Regeln für Geltungsbereiche
  - Bezeichner darf nur genau ***einmal*** deklariert werden
  - Jeder benutzer Bezeichner ***muß*** deklariert sein
- Symboltabelle
  - Für jeden Bezeichner genau ***ein*** Eintrag in der Symboltabelle
  - Abruf von Daten muß schnell gehen (binärer Suchbaun. Hash-Tabelle)

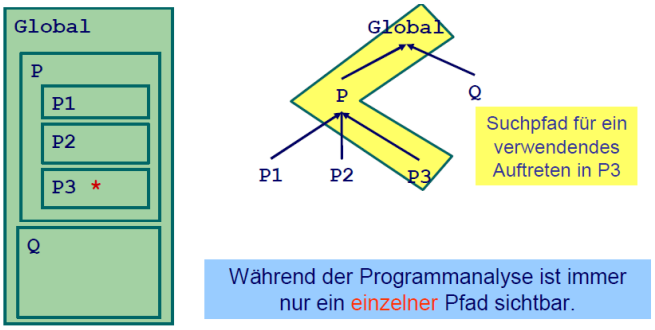


- Charakteristika
  - ***Mehrere*** überlappungsfreie Blöcke
  - Zwei Geltungbereiche: ***Global*** und ***Lokal***
- Regeln für Geltungsbereiche
  - Global deklarierte Bezeichner dürfen nicht global redeklariert werden
  - Lokal deklarierte Bezeichenr dürfen nicht im selben Block redeklariert werden
  - Jeder benutzte Bezeichner muss global oder lokal zu seiner Verwendungstelle deklariert sein
- Symboltabelle
  - Bis zu ***zwei*** Einträge für jeden Bezeichner (global und loal)
  - Nach Bearbeiten eines Blockes müssen lokale Deklarationen verworfen werden
- Beispiel: FORTRAN



- Charakteristika
  - Blöcke ineinander verschachtelt
  - Beliebige Schachtelungstiefe der Blöcke
- Regeln für Geltungsbereiche
  - Kein Bezeichner darf mehr als einmal innerhalb eines Blockes deklariert werden
  - Kein Bezeichner darf verwendet werden, ohne dass er lokal oder in den ***umschließenden*** Blöcken deklariert wurde
- Symboltabelle
  - ***Mehrere*** Einträge je Bezeichner möglich
  - Aber maximal ein Paar (Verschachtelungstiefe, Bezeichner)
  - Schneller Abruf des Eintrages mit der größten Verschachtelungstiefe
- Beispiele: Pascal, Modula, Ada, Java, …

Struktur der Geltungsbereiche
Für Sprachen mit verschachtelter Blockstruktur



Implementierung der Symboltabelle

- Verkettete Liste und lineare Suche
  - In Triangle verwendet
- Hash-Tabelle (effizienter)
- Stack aus Hash-Tabellen

Attribute

Wofür werden Attribute gebraucht?

Mindestens für

- Überprüfung der Regeln für Geltungsbereiche von Deklarationen
  - Bei geeigneter Implementierung der Symboltabelle: Einfaches Abrufen reicht, da alle Regeln bereits in Datenstruktur realisiert
- Überprüfung der Typregeln
  - Erfordert Abspeicherung von Typinformationen
- Code-Erzeugung
  - Benötigt später z.B. Adresse der Variable im Speicher

Speicherung von Attributen

Imperativer Ansatz (Explizite Speicherung)

- Ok für sehr einfache Sprachen

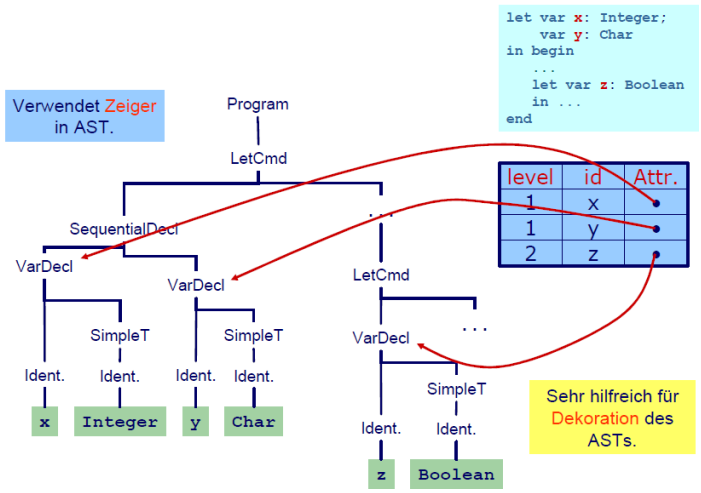
- Bei Komplexeren nicht möglich

Objektorientierter Ansatz (explizite Speicherung)

Funktioniert, wird aber bei realistischer Sprache sehr leicht unhandlich

Bisher wurden Kombinationen wie zum Beispiel Arrays und Records (und deren Kombination) nicht betrachtet. Dies mit Expliziten Strukturen kann leicht sehr komplex werden

Im AST stehen bereits alle Daten, also den Attributen einfach einen Verweis auf die ursprüngliche Definition (im AST) eintragen



Identifikation

1. Schritt der Kontextanalyse

- Beinhaltet den Aufbau geeigneter Symboltabelle
- Hat als Aufgabe den Verwendungen von Bezeichnern zu ihren Definitionen zu ordnen
- Durch Pass über den AST realisierbar

Aber besser kombiniert mit nächstem Schritt

Typprüfung

Was ist ein Typ?

- „Eine Einschränkung der möglichen Interpretationen eines Speicherbereiches oder eines anderen Programmkonstrukts“
- Eine Menge von Werten

Warum Typen benutzen?

- Fehlervermeidung: Verhindere eine Art von Programmierfehlern („eckiger Kreis“)
- Laufzeitoptimierung: Bindung zur Compile-Zeit erspart Entscheidung zur Laufzeit

Muss man immer Typen verwenden?

- Nein, viele Sprachen kommen ohne aus
    - Assembler, Skriptsprachen, LISP, …

Typüberprüfung

- Bei statischer Typisierung ist jeder Ausdruck E entweder
  - Misstypisiert, oder
  - Hat einen statischen Typ T, der ohne Evaluation von E bestimmt werden kann
- E wird bei jeder (fehlerfreien) Evaluation den statischen Typ T haben
- Viele moderne Programmiersprachen bauen auf statische Typprüfung auf
  - OOP-Sprachen haben aber auch dynamische Typprüfung zur Laufzeit (Polymorphismus)

Generelles Vorgehen

1 Berechne oder leite Typen von Ausdrücken her
   a. Aus den Typen der Teilausdrücke und der Art der Verknüpfung
2 Überprüfe, das Typen der Ausdrücke Anforderungen aus dem Kontext genügen

Bottom-Up Verfahren

- Typen an den Blättern des AST sind bekannt
  - Literale (Direkt aus Knoten)
  - Variablen (Aus Symboltabelle)
  - Konstanten (Aus Symboltabelle)
- Typen der internen Knoten her leitbar aus
  - Typen der Kinder
  - Typregeln für die Art der Verknüpfung im Ausdruck

Algorithmus für Kontextanalyse

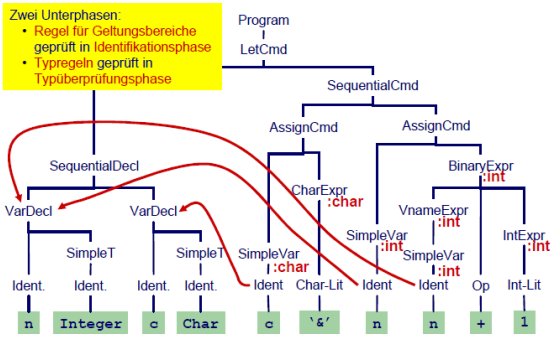
Kombiniere Identifikation und Typprüfung in einem Pass

Dies Funktioniert, solange Bindung immer vor Verwendung

Mögliche Vorgehensweise

- Tiefensuche von links nach rechts durch AST
- Dabei sowohl Identifikation und Typüberprüfung
- Speichere Ergebnisse durch Dekorieren des ASTs
  - Hinzufügen weiterer Informationen

Gewünschtes Ergebnis:




```
public abstract class Expression extends AST {
    // Every expression has a type
    public Type type;
    ...
}
```

```
public class Identifier extends Token {
    // Binding occurrence of this identifier
    public Declaration decl;
    ...
}
```

Um die zusätzlichen Informationen speichern zu können, müssen einige AST Knoten um zusätzliche Instanzvariablen erweitert werden.

Implementierung

Implementierung geht am besten über das Visitor-Pattern

Program	`visitProgram`	• return `null`
Command	`visit..Cmd`	• return `null`
Expression	`visit..Expr`	• dekoriere ihn mit seinem Typ • return Typ
Vname	`visitSimpleVname`	• dekoriere ihn mit seinem Typ • setze Flag, falls Variable • return Typ
Declaration	`visit..Decl`	• trage alle deklarierten Bezeichner in Symboltabelle ein • return `null`
TypeDenoter	`visit..TypeDenoter`	• dekoriere ihn mit seinem Typ • return Typ
Identifier	`visitIdentifier`	• prüfe ob Bezeichner deklariert ist • verweise auf bindende Deklaration • return diese Deklaration
Operator	`visitOperator`	• prüfe ob Operator deklariert ist • verweise auf bindende Deklaration • return diese Deklaration

Durch das ausnutzen von Overloading kann man die visitXYZ Methoden im Visitor alle in visit umbenennen.

Standartumgebung

Wo kommen Definitionen von Integer, Char, … und putint, getint, +, -, * her?

Diese müssen vorliegen, damit der Algorithmus richtig funktionieren kann.

Sie müssen also vorher definiert sein.

- Einlesen von Definitionen aus Quelltext
  - Ada, Haskell, VHDL
- Direkt im Compiler implementieret
  - Pascal, teilweise C, Java, …
  - Triangle

Geltungsbereich der Standardumgebung

- Ebene 0: Um gesamtes Programm herum oder
- Ebene 1: Auf Ebene der globalen Deklarationen im Programm

In Triangle

Idee: Trage Deklaration vorher direkt in AST ein (als Sub-AST)

Ohne konkrete Realisierung

- Die Konkrete Realisierung ist zu dem Zeitpunkt egal
- Bei der Code Generierung werden diese Konstrukt als Sonderfälle betrachtet

Typäquivalenz

In MiniTriangle einfach, da nur primitive Typen vorhanden.

Triangle ist komplizierter: Arrays, Records, benutzerdefinierte Typen

```
type T1 ~ record n: Integer; c: Char end;
type T2 ~ record c: Char; n: Integer end;

var t1 : T1; var t2 : T2;

if t1 = t2 then ...

Legal?
```

```
type Word ~ array 8 of Char;

var w1 : Word;
var w2 : array 8 of Char;

if w1 = w2 then ...

Legal?
```

Struktur nicht äquivalent, Name nicht äquivalent Struktur äquivalent, Namen nicht Äquivalent

Liegt in Entscheidung des Sprachdesigners.

Möglichkeiten:

- Strukturelle Typäquivalenz
  - Primitive Typen:
    - Müssen identisch sein
  - Arrays:
    - Äquivalenter Typ für Elemente,
    - gleiche Anzahl
  - Records:
    - Gleiche Namen für Elemente,
    - Äquivalenter Typ für Elemente,
    - gleiche Reihenfolge der Elemente
- Typäquivalenz über Namen
  - Jedes Vorkommen eines nicht-primitiven Typs (selbstdefiniert, Array, Record) beschreibt einen neuen und einzigarten Typ, der nur zu sich selbst äquivalent ist

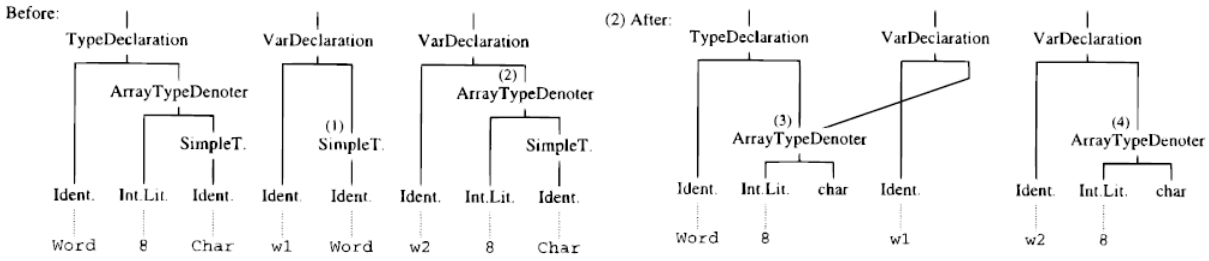
Aufgrund der Komplexität von dieser Sache in Triangle reicht die Klasse Type nicht aus

Idee: Wir verweisen auf die Typbeschreibung im AST

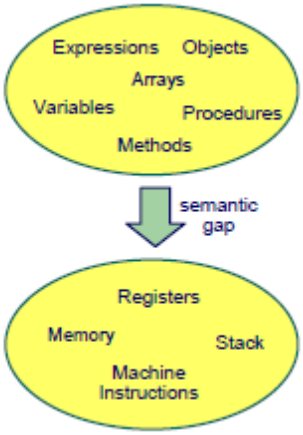
Vorgehen:

1. Ersetze in Kontextanalyse alle Typenbezeichner durch Verweise auf Sub-ASTs der Typdeklaration
2. Führe Typprüfung durch strukturellen Vergleich der Sub-ASTs der Deklaration durch

Beispiel:



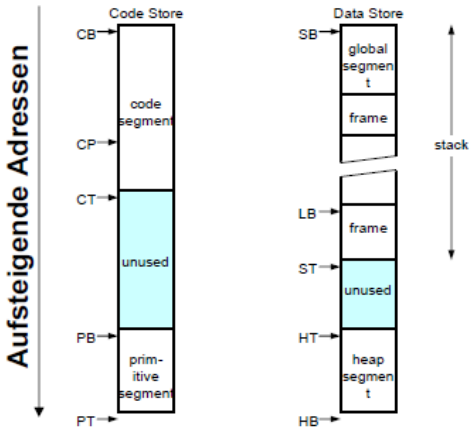
Run-Time Organization



- Compiler übersetzt Hochsprachenprogramm in Äquivalentes Maschinenprogramm
- Laufzeitorganisation beschreibt Darstellung von abstrakten Strukturen der Hochsprache auf Maschinenebene
- Instruktionen und Speicherinhalte

Wichtige Aspekte

- **Datendarstellung** der Werte jedes Typs der Eingabesprache
- **Auswertung von Ausdrücken** und Handhabung von Zwischenergebnissen
- **Speicherverwaltung** verschiedener Daten: Global, lokal und Heap
- **Routinen** zur Implementierung von Prozeduren, Funktionen und ihre Datenübergabe
- **Erweiterung auf OO-Sprachen** Objekte, Methoden, Klassen und Vererbung



TAM (Triangle Abstract Maschine)

Harward-Architektur (vs. Von Neuman-Architektur)

- Datenspeicher: 16b Worte
- Instruktionsspeicher 32b Worte

Die Adressierung der Adressbereich erfolgt über CPU Register (ansonsten Stackmaschine)

Adressierung des Instruktionsspeichers

Programm	CB	Code Base (konstant)
	CT	Code Top (konstant)
	CP	Code Pointer (variabel)
Intrinsics	PB	Primitive Base (konstant)
	PT	Primitive Top (konstant)

Adressierung des Datenspeichers

Stack	SB	Stack Base (konstant)
	ST	Stack Top (variabel)
Heap	HB	Heap Base (konstant)
	HT	Heap Top (variabel)
	HF	Heap Free (variabel)

TAM Instruktion

- op, 4b; Art der Instruktion
- r, 4b; Registernummer
- n, 8b; Operandengröße in Worten (nicht nur)
- d, 16b; Adressverschiebung (displacement, offset)

TAM Befehlssatz

Op.	Mnem.	Effect
0	LOAD(n) d[r]	Fetch an n-word object from the data address and push it onto the stack
1	LOADA d[r]	Push the data address onto the stack
2	LOADI(n)	Pop a data address from the stack, fetch an n-word object from that address, push it onto the stack
3	LOADL d	Push the one-word literal value d onto the stack
4	STORE(n) d[r]	Pop an n-word object from the stack, and store it at the data address
5	STOREI(n)	Pop an address from the stack, then pop an n-word object from the stack and store it at that address
6	CALL(n) d[r]	Call the routine at the code address using the address in register n as the static link
7	CALLI	Pop a closure (static link and code address) from the stack, then call the routine
8	RETURN(n) d	Return from the current routine; pop an n-word result from the stack, then pop the topmost frame, then pop d words of arguments, then push the result back
9	–	(unused)
10	PUSH d	Push d words (uninitialised) onto the stack
11	POP(n) d	Pop an n-word result from the stack, then pop d more words, then push the result back on the stack
12	JUMP d[r]	Jump to code address
13	JUMPI	Pop a code address from the stack, then jump to that address
14	JUMPIF(n) d[r]	Pop a one-word value from the stack, then jump to code address if and only if that value equals n
15	HALT	Stop execution of the program

TAM Intrinsics

Primitive

„Magische“ Adressen im Programmspeicher

Führen bei Aufruf als Routine komplexe Operationen aus

Addr.	Mnemo.	Arg.	Res.	Effect
...				
2[PB]	not	t	t'	t' = !t
...				
8[PB]	add	i1, i2	i'	i' = i1 + i2
...				
15[PB]	ge	i1, i2	t'	Set t'=true iff i1 ≥ i2
...				
26[PB]	putint	i	–	Write an integer whose value is i

Darstellung von Daten

- Unverwechselbarkeit Unterschiedliche Werte sollen unterschiedliche Darstellungen haben
  - Klappt nicht immer (duale Gleitkommadarstellung reeller Zahlen)
- Einzigartigkeit Ein Wert wird immer auf die gleiche Weise dargestellt
- Konstante Größe Alle Werte eines Typs belegen dieselbe Menge an Speicherplatz
- Art der Darstellung
  - Direkt Wert einer Variable x kann direkt adressiert werden
    - Effizienter Zugriff
  - Indirekt Wert einer Variable x muss über einen Zeiger bzw. Handle adressiert werden
    - Dynamische Arrays
    - Rekursive Typen
    - Objekte

Primitive Typen

#[T] : Anzahl unterschiedlicher Elemente in T

Size[T]: minimaler Speicherbedarf (in Bit) zur Darstellung eines Wertes aus T

Es muss immer gelten size[T] >= log2(#[T])

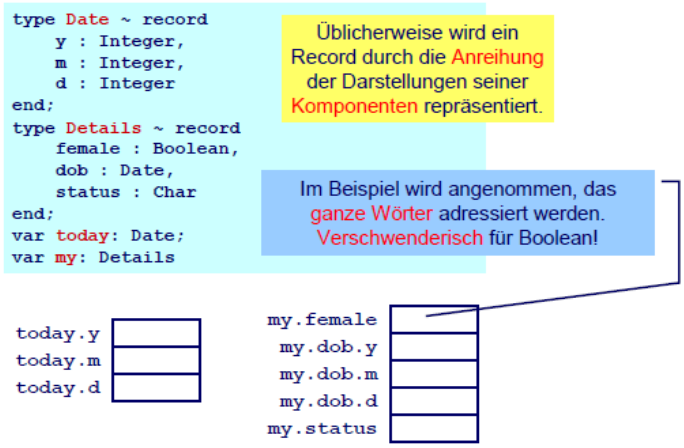
	#[T]	size[T]	Darstellung
Boolean	2	≥ 1	0 and 1
Integer	2^16 or 2^32	16 / 32	2-complement
Char	2^8 or 2^16	8 / 16	ASCII/Unicode
float	infinite	32 / 64	approximation

In der TAM	
Boolean	16b
Char	16b
Integer	16b

Da in TAM Wort adressiert

Bei Triangle brauchen wir für Records und co etwas mehr.

Records



Speicherbedarf und Adressierung

```
type Date = record
    y : Integer,
    m : Integer,
    d : Integer
    end;
var today: Date;
```

- size[Date] = 3 * size[Integer] = 3 Worte
- address[today.y] = address[today]
- address[today.m] = address[today] + size[Integer]
- address[today.d] = address[today] + 2*size[Integer]

Viele reale Prozessoren haben Anforderungen an Adressausrichtung der Daten

- Beispiel: Es können nur 32b Worte als Einheit adressiert werden
- Ist schneller, als größere Freiheit zu ungeschützten

Darstellung von Records im Speicher kann ineffizient werden

- Unter Platzgesichtspunkten (wenn optimal ausgerichtet)
- Unter Laufzeitgesichtspunkten (wenn optimal gepackt)

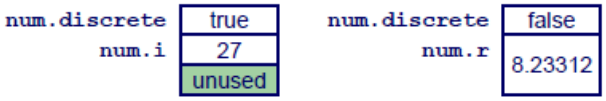
Variante Records

Ähnlich einer Record, aber zu einem Zeitpunkt existiert immer nur eine Untermenge von Komponenten. (Wurde zur Typumwandlung benutzt, illegal ;-))

Selektion durch *type tag*

```
type Number =
   record
      case (discrete:Boolean) of
          true: (i: Integer);
          false: (r: Real)
      end;
var num: Number
```

size[Number] = size[Boolean] + max(size[Integer], size[Real])
address[num.acc] = address[Number]
address[num.i] = address[Number] + size[Boolean]
address[num.r] = address[Number] + size[Boolean]



Arrays

- Zusammengesetzter Typ
- Besteht aus einem oder mehreren Werten eines Typs
  - Unterschied zu Record
- Zugriff über Index (Beginnt bei 0)

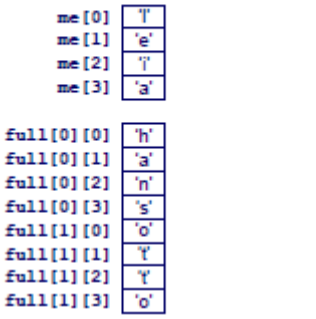
Statische Arrays

haben feste, zur Compile-Zeit bekannte Abmessungen

```
type Name = array 6 of Char;
var me : Name;
```

- size[Name] = 6 * size[Char] = 6 Worte
- address[me[0]] = address[me]
- address[me[1]] = address[me] + 1 * size[Char]
- address[me[i]] = address[me] + i * size[Char]

```
type Name = array 4 of Char;
var  me: Name;
var  full: array 2 of Name
```



Dynamische Arrays

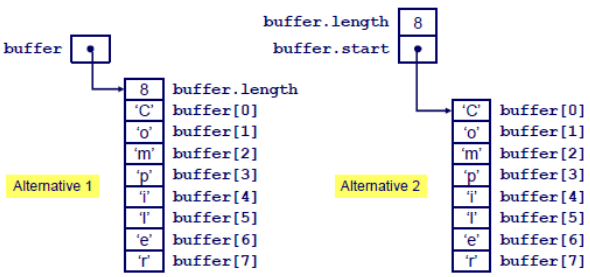
haben zur Laufzeit variable Abmessungen

Indirekte Darstellung über Deskriptor

- Adresse des ersten Elements
- Abmessung

Speicher wird zur Laufzeit angefordert (→ Heap)

```
char[] buffer;
buffer = new char[len];
```



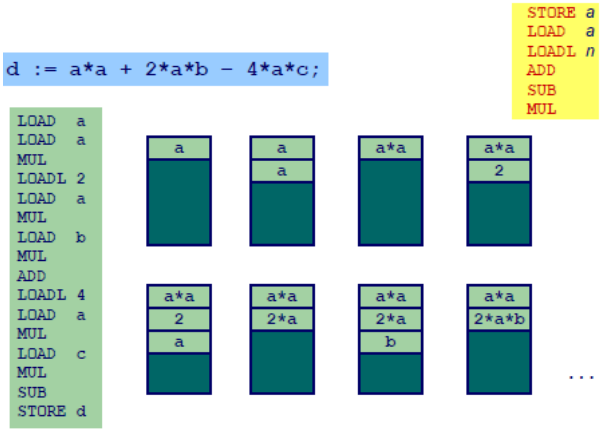
Rekursive Typen

Referenzieret sich selbst in seiner eigenen Definition

In der Regel nur über Zeiger

```
class IntList {
  int head;
  IntList tail;
}
```

Auswertung von Ausdrücken



Instr.	*Meaning*
STORE *a*	**Pop** the top value off the stack and **store** it at **address** *a*.
LOAD *a*	**Fetch** a value from **address** *a* and push it on to the stack.
LOADL *n*	**Push** the **literal value** *n* onto the stack.
ADD	**Replace** the **two** top values on the top by their **sum**.
SUB	**Replace** the **two** top values on the top by their **difference**.
MUL	**Replace** the **two** top values on the stack by their **product**.

Register-Maschine

Sehr schnelle Speicherelemente direkt im Prozessor

- Für Zwischenergebnisse
- In der Regel 8/16/32/64b breit
- Begrenzte Anzahl, üblicherweise 4..32 direkt verwendbar (in wirklichkeit gibt es viel mehr)

Nicht immer so allgemeingültig verwendbar, häufig Einschränkungen

- Nur bestimmte Register für bestimmte Operationen
- Nicht alle Arten von Operanden für alle Operationen

Code für Registermaschine ist effizienter, aber Kompilierung ist komplexer.

- Verwaltung (Allokation) von Registern
- Speichere Zwischenergebnisse in Registern
- Problem: Endlich viele Register! Was wenn Ausdruck komplizierter (zu viele Zwischenergebnisse)=

Speicherverwaltung

Globale Variablen: Existieren über gesamte Programmlaufzeit

- Compiler kann bereits Speicherbedarf jeder Variable berechnen
- Damit kann jeder Variable passender Speicher zugewiesen (alloziert) werden
- Nun bekannt: Adresse jeder Variable im Speicher

Bündige Anreihung

Lokale Variablen:

- Ist im Inneren eines Blockes definiert
    - Prozedur, Funktion, Let
- Existiert nur während der Block aktiv ist
- Hat eine Begrenzte Lebensdauer

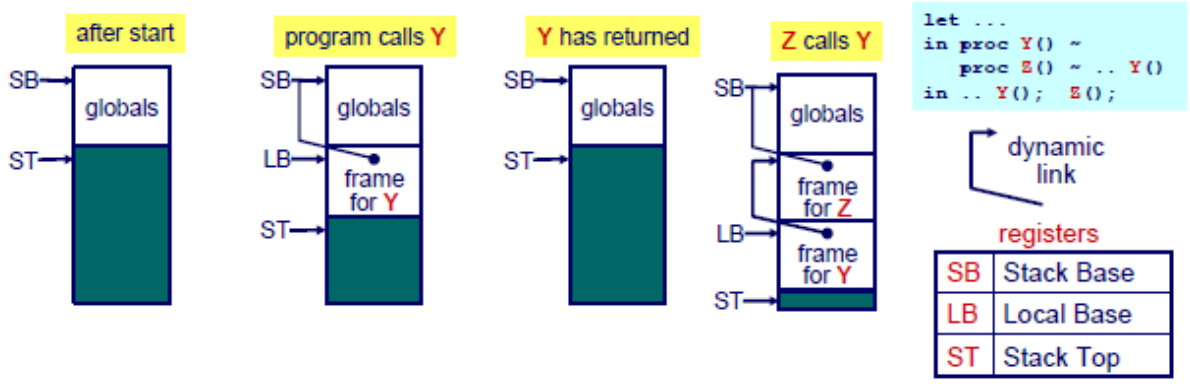
Wichtig: eine Prozedur kann gleichzeitig mehrfach aktiv sein (rekursion)

Organisationsstruktur: Stack Frame

Jede Prozedur hat einen Stack Frame

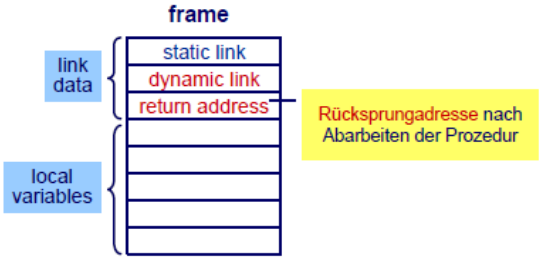
- Lokale Variablen
- Verwaltungsdaten
- Aktuelle Parameter

Wird bei Prozeduraufruf angelegt und nach Prozedurende abgebaut



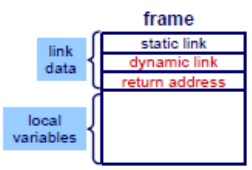
Verwaltungsdaten (3 Worte)

- return adress
  - Rücksprungadresse
- Dynamic link
  - Vorheriger LB (Verkettung der Frames)
- Static link



Beispiel: Adressierung von Variablen

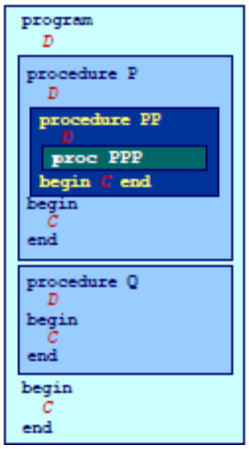
```
let
   var a: array 3 of Char;
   var b: Boolean;
   var c: Char;
in
   proc Y() ~
   let var d: Integer;
       var e: Integer
   in  ...

   proc Z() ~
   let var f: Integer
       var g: Char;
   in
      ... Y(); ...
in begin
   ... Y(); ...; Z(); ...
end
```



Wegen der Verwaltungsdaten (3 Worte) beginnen die lokalen Variablen erst bei Adresse 3 im Stack Frame

var	size	address
a	3	0 [SB]
b	1	3 [SB]
c	1	4 [SB]
d	1	3 [LB]
e	1	4 [LB]
f	1	3 [LB]
g	1	4 [LB]

Statische Programhierarchie



Mit LB und SB nicht lösbar. Daher Static Link benötigt

Variablen Umschließender Blöcke liegen auf jeden Fall auf dem Stack, die Frage ist nur wo? Man müsste sich irgendwie hochhangeln.

Statische Verkettung

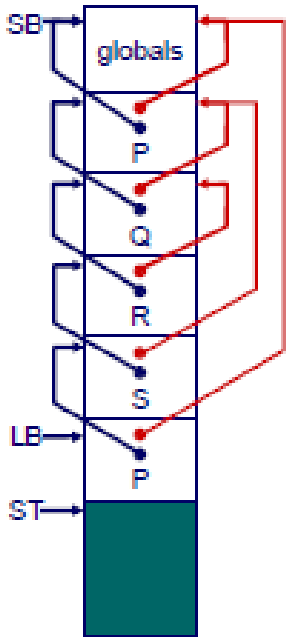
- Verweis auf Frame der im Programmtext umschließenden Prozedur
- Unterschied dynamische Verkettung
  - Hier Verweis auf Frame der a**ufrufenden** Prozedur
- Dient Zugriff auf **nicht-lokale** Variablen

„Nicht-Lokale Variablen werden nicht von allen Sprachen unterstützt“

Contents(LB) = Umschließender StackFrame

Contents(Contents(LB)) = noch weiter außenliegender Stack Frame

```
let
  ...
  proc P() ~
  let
    ...
    proc S() ~
    let ...
    in ... P(); ...

    proc Q() ~
    let
      ...
      proc R() ~
      let
      in ... S()

    in ... R(); ...

  in ... Q(); ...

in ... P(); ...
```



Statische Verkettung von R erlaubt Zugriff auf Variablen von Q. Statische Verkettung von Q erlaubt Zugriff auf Variablen von P.

Dynamische Verkettung ≠ Statische Verkettung

Hochhangeln würd durch Hardware realisiert (in TAM)

Sogenanntes Display

display registers	SB		Zeigt auf Frame mit globalen Variablen
	LB		Zeigt auf oberste Frame R
	L1	contents(LB)	Zeigt auf Frame R' umschließend R
	L2	contents(L1)	Zeigt auf Frame R'' umschließend R'
	L3	contents(L2)	Zeigt auf Frame R''' umschließend R''
	L4	contents(L3)	Zeigt auf Frame R'''' umschließend R'''
	. . .	...	...

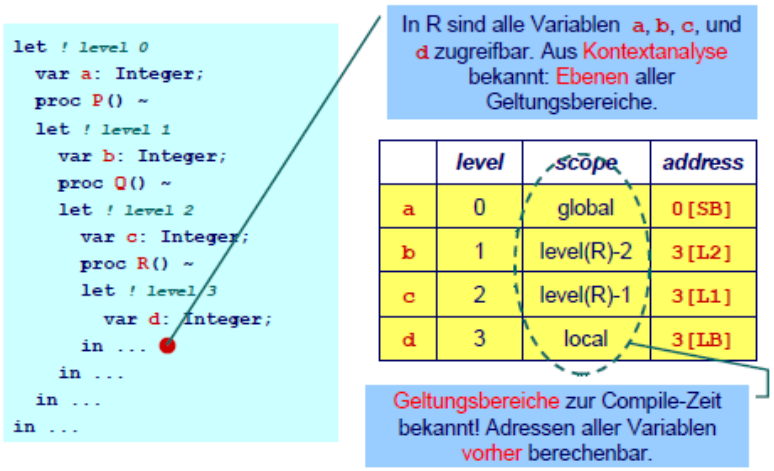
Bei TAM maximal bis L7

Das bedeutet nicht, dass Rekursion beschränkt ist, sonder man kann maximal Prozedurdefinitionen bis zur Stufe 7 definieren.

Die Display Berechnung ist per Hand sehr umständlich, daher Hardware.

Bestimmung statische Verkettung

Aus der Kontextanalyse ist die Ebene aller Geltungsbereiche bekannt.



Routine

R sei Routine deklariert auf Ebenen l, dann gilt für die statische Verkettung (SV)

- Wenn l = 0 (R ist globale Routine)
  - SV = SB; R seiht statisch nur globale Variablen
- Wenn l>0 (R ist eingeschachtelt deklariert)
  - SV = LB vor Aufruf
    - Wenn Aufruf von R aus Ebene l erfolgt
  - SV = L1 vor Aufruf
    - Wenn Aufruf von R aus Ebene l+1 erfolgt
  - SV = L2 vor Aufruf
    - Wenn Aufruf von R aus Ebene l+2 erfolgt
  - … (bis L7 in TAM)

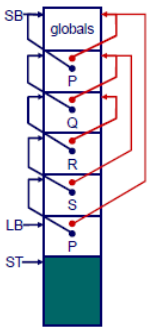
Anlegen von SV an Aufrufstelle

Nur der Aufrufer kennt seine Ebene

In Triangle/TAM: Parameter für CALL-Instruktion

Beispiel:

```
let
  ...
  proc P() ~
  let
    ...
    proc S() ~
    let ...
    in ... P(); ...

    proc Q() ~
    let
      ...
      proc R() ~
      let
      in ... S()

    in ... R(); ...

  in ... Q(); ...
in ... P(); ...
```



S() deklariert auf $l = 1$, Aufruf auf $l = 3$
$\rightarrow$ L2 verwenden

```
CALL (L2) S
```

Zusammenfassung

- Kompliziertere Kompilierung
- Auch Laufzeitoverhead durch statische Verkettung
  - Kompliziertere Funktionsaufruf
  - Erhöhter Speicherbedarf

Es Lohnt sich nicht!

Verteilung bei Pascal

- 49 % Global
- 49 % Lokal
- 2 % Nicht-Lokal

Routinen

Assembler-Äquivalent von Prozeduren und Funktionen einer Hochsprache.

Wichtige Maschineninstruktionen

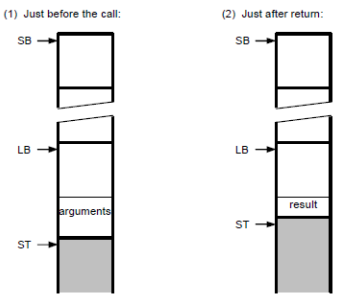
- CALL r lege nächste Programmzeigeradresse auf Stapel und Springe auf Adresse r
- RETURN Nehme einen Wert vom Stapel und Springe dorthin

Wichtige Aspekte

- Aufruf einer Routine und Übergabe von Parametern
- Rückkehr von einer Routine und Rückgabe eines Ergebnisses
- Verwaltung von statischen Verkettungen

In Form eines Protokolls definiert (maschinenabhängig) [auch calling conventions]

Für Stack häufig



- Aufrufer legt Parameter auf Stapel (Reihenfolge?)
  - Bei uns: links zuerst, dann nach links
- Routine wird aufgerufen und benutzt Parameterwerte
- Aufgerufene Routine nimmt Parameter vom Stael und ersetzt sie durch Rückgabewerd

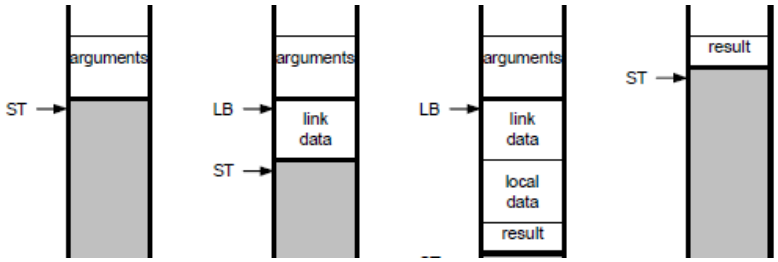
Dadurch beliebig viele Parameter übergebbar

Schwierig: Unterschiedliche Anzahl von Parameter an eine Routine übergeben.
Geht nicht ohne weiteres.

In der TAM

Relevante TAM Instruktionen

CALL (reg) addr	Ruft Routine an Adresse **addr** auf, verwendet den Wert in **reg** als statische Verkettung bei der Anlage eines neuen Frame
RETURN (n) d	Sichert **n** Worte als Ergebnis vom Stack, entfernt den aktuellen Frame und **d** Parameter, setzt Ausführung nach Aufrufstelle fort, legt Ergebnis oben auf dem Stack ab.

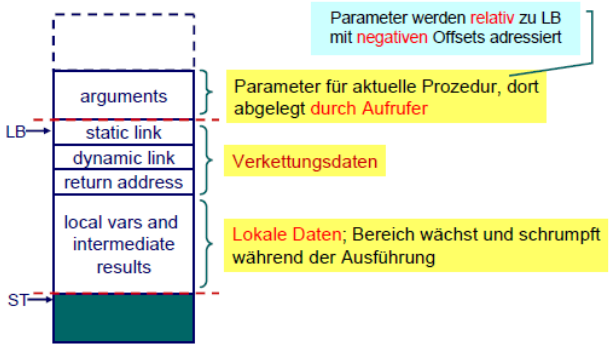


Parameter (Argumente) zum Datenaustausch

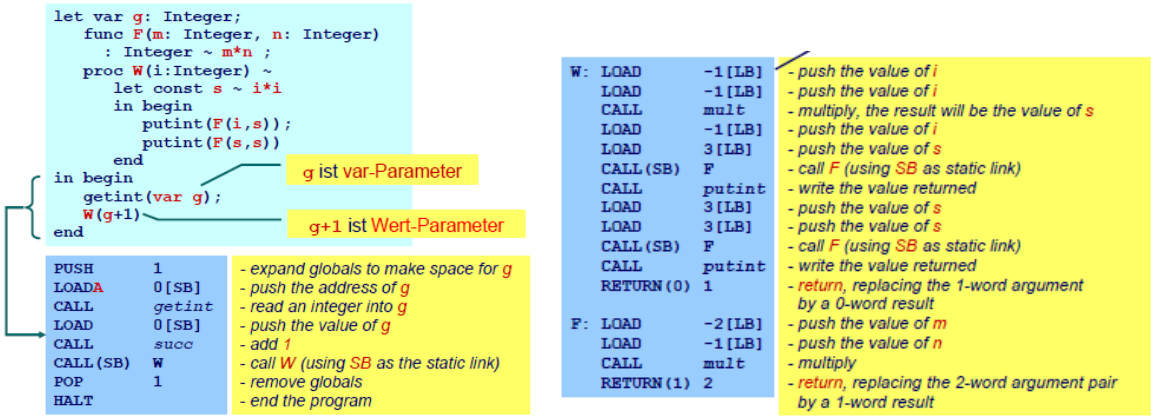
- Aktuelle Parameter verwendet von Aufrufer bei Aufruf der Prozedur
- Formale Parameter innerhalb der Prozedur verwenden
    - Verhalten sich innerhalb oder Prozedur wie lokale Variablen
- Eins-zu-eins Zuordnung von aktuellen und formalen Parameter

Übergabe als

- Call by Value
- Call by Referenz (var bei Deklaration und Aufruf der Prozedur)
    - Übergabe der Adresse der Variable (als Zeiger)



Beispiel: Call-by Reference



Sonderfall: Prozedur/Funktion als Parameter (func)

```
let
   func twice(func doit(Integer x): Integer, i: Integer): Integer ~
       doit(doit(i));
   func double(Integer d) ~ d*2;
   var x: Integer
in begin
   x := twice(double, 10);
end
```

- Repräsentiere Funktion durch Paar (Startadresse, statische Verkettung
- Soggenannte Closure oder Funktionsdeskriptor
- Aufruf dann über Closure
- TAM: Lege Closure auf Stack, dann CALLI zum Aufruf

Heap-Speicher

- Bisher Lebenszeit von Variablen gebunden an Geltungsbereiche
  - Auch verschachtelt (statische Verkettung)
- Häufig: Lebenszeit unabhängig von Geltungsbereichen
- Beispiel: Datenstrukturen wie Listen, Bäume, etc.
  - Struktur lebt unabhängig von Prozedure/Funktion

Dafür ist eine andere Speicherverwaltung als Stack

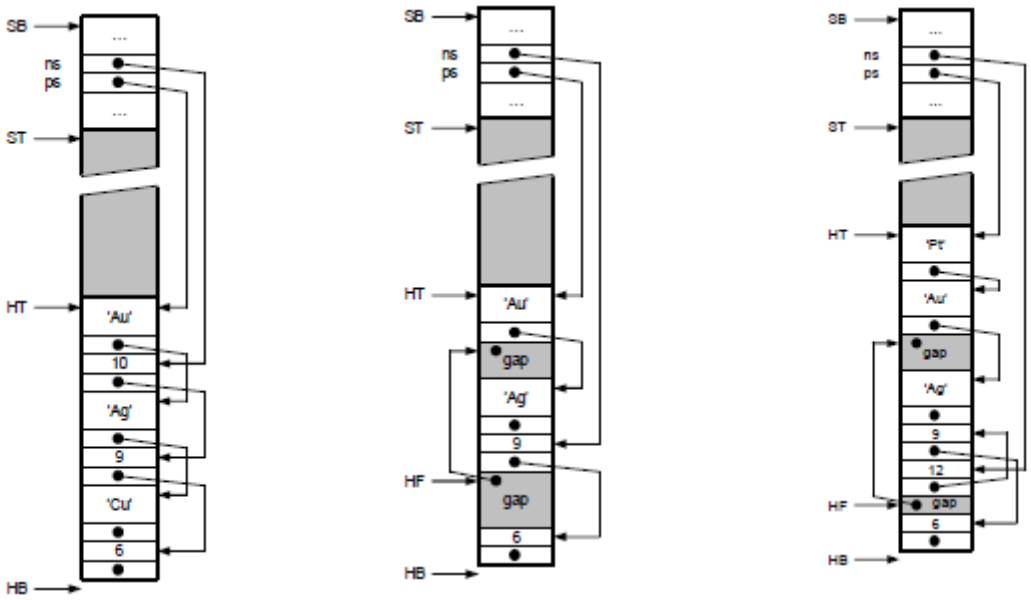
- Nachteil: Explizite Verwaltung durch Programm erforderlich
  - Pascal, C, C++
- Teilweise Automatisierung möglich
  - Java, Lisp, Smalltalk
  - GarbageCollector

Heap i.d.R im selben Speicher wie im Stack

- Stack wächst und schrumpft bei Blockeintritt/-austritt
- Heap wächst bei Anlegen neuer Variablen, schrumpft(?) bei Freigabe
- Heap und Stack wachsen aufeinander zu
  - Normalerweise: Stack wächst nach oben, Heap unten
  - Bei Tam: Stack wächst nach unten, Heap Oben

Beispiel: Heap

Elemente Löschen Neue Elemente hinzufügen



Einfügen neuer Elemente

- Bei nächten Einfügen ersten freien Platz verwenden
    - Problem: Kann zu vielen kleineren Löcher in der Heap kommen (Fragmentierung)
        - Heap wächst weiter
- Einfach Oben anhängen
        - Heap wird immer größer und nie kleiner
- Andere Ansatz
    - Finde genau passenden freien Speicherblock (Freie Bereiche verkettet)
    - Finde größeren freien Speicher in HF und benutzte ihn teilweise
    - Vergrößere Heap in Richtung Stack
    - Falls nicht möglich: out-of-memory

Fragmentierung bekämpfen

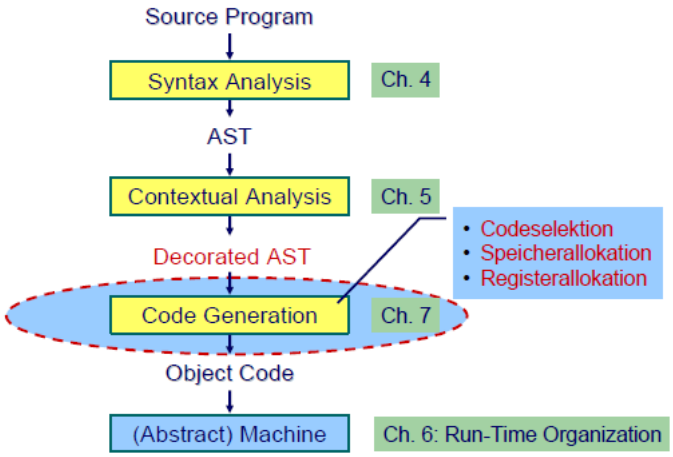
- Verwende immer kleinsten passenden freien Speicherblock (immer sinnvoll?)
    - Es können sehr viele kleine Lücken entstehen
    - Verschmelzen benachbarter freier Speicherblöcke
    - Kampaktierende Heap (verzweifelt!)
        - Alles Zusammenschieben
        - Problem: Alle Zeiger im Programm müssen aktualisiert werden
        - Teillösung: Doppelte Indirektion über Handles
            - These: „Es gibt kein Problem in der Informatik, was sich nicht durch hinzufügen einer weiteren Indirektionsebene beheben lässt“
            - Realisiert als Zeiger auf Zeiger
            - Programm operiert mit Handles, werden nicht beeinflusst
            - Zeiger innerhalb von Handles werden durch Kompaktierung aktualisiert

Teilautomatische Speicherverwaltung

Automatische Freigabe von nicht mehr benutzten Speicher

- Garbage Collection
- Viele verschiedene Ansätze
- Einfacher Ansatz
    - Kennzeichne alle Elemente auf Heap als nicht erreichbar
    - Gehe nun alle Variablen durch (auf Heap und Stack!)
    - Falls Zeiger: Markiere referenzierten Heap-Block als erreichbar
    - Trage alle unerreichbaren Speicherblock in HF-Liste ein
    - Problem:
        - Wie „Zeiger“ erkennen
            - Zeiger besonders kennzeichnen
        - Heap-Blöcke müssen ihre Größe kennen
        - Was, wenn Zeiger mitten in Heap-Block hinein?

Code-Generierung



Wir erzeugen direkt Maschinencode
Das erste Mal kommt Semantik ins Spiel:
Codegenerierung befasst sich mit Semantik der Eingabesprache

```
let
   var x: integer;
   var y: integer
in begin
   y := 2;
   x := 7;
   printint(y);
   printint(x);
end
```

```
PUSH      2
LOADL     2
STORE(1)  1[SB]
LOADL     7
STORE(1)  0[SB]
LOAD(1)   1[SB]
CALL      putint
LOAD(1)   0[SB]
CALL      putint
HALT
```

Gleiche Semantik für Quellprogramm und Zielprogramm (Ausnahmen)

- Abhängig von Eingabesprache
  - Syntaktische Analyse
  - Kontextanalyse
- Abhängig von Eingabesprache und Zielmaschine
  - Codegenerierung

Schwierig allgemein zu formulieren

Unterprobleme

- Code-Selektion
  - Ordnet Phrasen aus Quellprogramm Folgen von Maschineninstruktionen zu
- Speicherallokation
  - Weist jeder Variablen Speicherplatz zu und führt über diesen Buch
- Registerallokation
  - Verwaltet Registerverwendung für Variablen und Zwischenergebnisse (nicht in TAM)

Code Selektion

Semantik der Programmiersprache

- In der Regel auf Phrasenebene beschrieben
- Expressions, Commands, Declarations

Vorgehensweise

Induktives Herleiten der Übersetzung des gesamten Programms aus Übersetzungen von Einzelphrasen

Problem: Mehrere semantisch korrekte Übersetzungen für eine Phrase. Welche konkrete Instruktionsreihenfolge auswählen?

- Code-Selektion

Code-Funktion

Bildet Phrase auf Instruktionsfolge ab.

Wird definiert durch

Code-Schablone

Ordnet jeder speziellen Form einer Phrase eine Definition in Form von Maschineninstruktionen oder Anwendungen von Code-Funktionen zu.

Wichtig: Eingabesprache muss vollständig durch Code-Schablone abgedeckt werden.

Code-Funktion

$execute: Command \rightarrow Instruction*$	Anweisungsfolge C1;C2 Semantik: führe erst C1 aus, dann C2	$execute[\![C1; C2]\!] = execute[\![C1]\!]$ $execute[\![C2]\!]$
	Zuweisung I := E	$execute[\![I := E]\!]$ $= evaluate[\![E]\!]$ $store\ a, mit\ a = Adresse\ von\ I$

Beispiel:

Anweisungsfolge `f := f*n; n := n-1`

execute [[`f := f*n; n := n-1`]] =

execute [[`f := f*n`]]
execute [[`n := n-1`]] =

evaluate [[`f*n`]]
`STORE f`

evaluate [[`n - 1`]]
`STORE n` =

```
LOAD f
LOAD n
CALL mult
STORE f
LOAD n
CALL pred
STORE n
```

Aufbau einer Code-Funktion orientiert sich an Subphrasenstruktur

$$f_P\,[[\,\ldots Q \ldots R \ldots\,]] =$$
$$\ldots$$
$$f_Q\,[[\,Q\,]]$$
$$\ldots$$
$$f_R\,[[\,R\,]]$$
$$\ldots$$

Die Reihenfolge ist nicht unbedingt zwingend

Code-Spezifikation

- Sammlung aller
  - Code-Funktionen
  - Code-Schablonen
- Muss Eingabesprache vollständig überdecken

Basierend auf Abstrakter Syntax:

```
Program ::= Command                          Program
Command  ::= V-name := Expression            AssignCommand
           | Identifier ( Expression )        CallCommand
           | Command ; Command               SequentialCommand
           | if Expression then Command      IfCommand
             else Command
           | while Expression do Command     WhileCommand
           | let Declaration in Command      LetCommand
```

run : Program → Instruction*
execute : Command → Instruction*
evaluate : Expression → Instruction*
fetch : V-name → Instruction*
assign : V-name → Instruction*
elaborate : Declaration → Instruction*

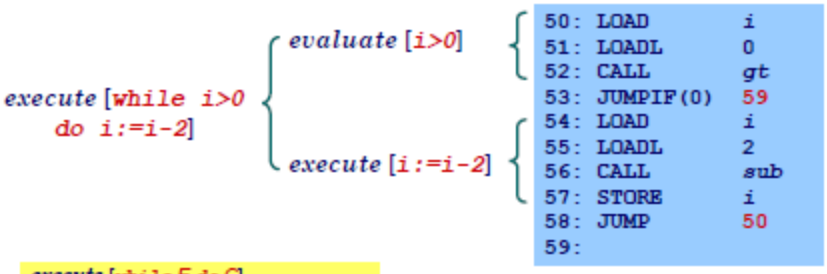
class	*code function*	*effect of the generated code*
Program	***run P***	Run the program P and then halt, starting and finishing with an empty stack.
Command	***execute C***	Execute the command C, possibly updating variables, but neither expanding nor contracting the stack.
Expression	***evaluate E***	Evaluate the expression E, pushing its result on the stack top, but having no other effects.
V-name	***fetch V***	Push the value of the constant or variable named V on the stack.
V-name	***assign V***	Pop a value from the stack top, and store it in the variable named V.
Declaration	***elaborate D***	Elaborate the declaration D, expanding the stack to make space for any constants and variables declared therein.

Run	Anweisungsfolge	Zuweisung
run [C] = *execute* [C] HALT	*execute* [C_1 ; C_2] = *execute* [C_1] *execute* [C_2]	*execute* [V := E] = *evaluate* [E] *assign* [V]
Bedingte Anweisung	**Schleife**	**Deklaration**
execute [if E then C_1 else C_2] = *evaluate* [E] JUMPIF(0) Else *execute* [C_1] JUMP Lfi Else: *execute* [C_2] Lfi:	*execute* [while E do C] = Lwhile: *evaluate* [E] JUMPIF(0) Lend *execute* [C] JUMP Lwhile Lend:	*execute* [let D in C] = *elaborate* [D] *execute* [C] POP(0) s POP nur, wenn s>0 (zusätzlicher Speicher alloziert wurde)

Finden Sinnvoller Lablenamen schwierig. Label müssen unterschiedlich sein.

Beispiel: Code-Schablone

```
while i > 0 do i := i - 2
```




```
execute [while E do C]
  = Lwhile: evaluate [E]
            JUMPIF(0)   Lend
            execute [C]
            JUMP        Lwhile
      Lend:
```

Code-Schablonen für Ausdrücke

Integer-Literal

```
evaluate[IL] =
    LOADL v          ; v is the value of IL
```

Variable

```
evaluate[V] =
    fetch V
```

Unärer Operator

```
evaluate[O E] =
    evaluate E
    CALL p           ; p is the address of the routine corresponding to O
```

Binärer Operator

```
evaluate[E1 O E2] =
    evaluate E1
    evaluate E2
    CALL p           ; p is the address of the routine corresponding to O
```

Code-Schablonen für Deklarationen

Konstante

```
elaborate[const I ~ E] =
    evaluate E       ; ... and decorate the tree
```

- Beachte: Legt berechneten Wert auf Stack ab! Optimierung möglich:
  - Setze Wert der Konstante direkt in Maschinencode ein
  - Dann leere Schablone

Variable

```
elaborate[var I : T] =
    PUSH size(T)     ; ... and decorate the tree
```

Deklarationsfolge

```
elaborate[D1; D2] =
    elaborate D1
    elaborate D2
```

Wenn 10 mal hintereinander eine Variable Deklariert wird, steht 10x PUSH z.B. 1, das könnte man optimieren.

Ab jetzt Mini-Triangle, keine lokalen, nicht-lokalen Variablen

Lesen

```
fetch[I] =
    LOAD d[SB] ;  d is the address of I
```

Schreiben

```
assign[I] =
    STORE d[SB] ;  ditto
```

Beispiel:

```
execute[let const n ~ 7; var i : Integer in i := n*n]


=   elaborate[const n ~ 7; var i : Integer]
    execute[i := n*n]


=   elaborate[const n ~ 7]
    elaborate[var i : Integer]
    evaluate[n*n]
    assign[i]


=   LOADL 7
    PUSH 1
    LOAD n
    LOAD n
    CALL mult
    STORE i
    POP(0) 2
```

Optimierung: const n → Inlining

Spezialisierte Schablonen für Sonderfälle

Allg. Schablone

```
LOAD    i
LOADL   1
CALL    add
```

Spez. Schablone

```
LOAD    i
CALL    succ
```

Effizienterer Code für "+1".

Analoges Vorgehen für Inlining von Konstanten

Inlining von Konstanten in Maschinen-Code

Konstante I mit statischem Wert $v = \text{valueOf}(IL)$

```
fetch[I] =
    LOADL  v  ;  ... v retrieved from DAST

elaborate[const I ~ IL] =
                ;  ... just decorate the tree
```

Optimiert:

```
execute[let const n ~ 7; var i : Integer in i := n*n] =
    elaborate[const n ~ 7; var i : Integer]
    execute[i := n*n]

=   elaborate[const n ~ 7]
    elaborate[var i : Integer]
    evaluate[n*n]
    assign[i]

=
    PUSH 1
    LOADL 7
    LOADL 7
    CALL mult
    STORE i
    POP(0) 2
```

Tippfehler: Pop(0) 1 richtig

Implementierung

Systematischer Aufbau

Orientiert sich direkt an Code-Funktion

Code-Funktionen beschreiben rekursiven Algorithmus zur Traversierung von DAST

Wieder bewährtes Visitor-Entwurfsmuster verwenden

Im Maschinencode gibt es keine LABELS nur Adressen

Backpatching

Const b ~ 10 ! direkt einfügen mit LOADL 10

Const y ~ 365 + x ! speicherplatz erstellen und berechnen

Geht in Triangle: