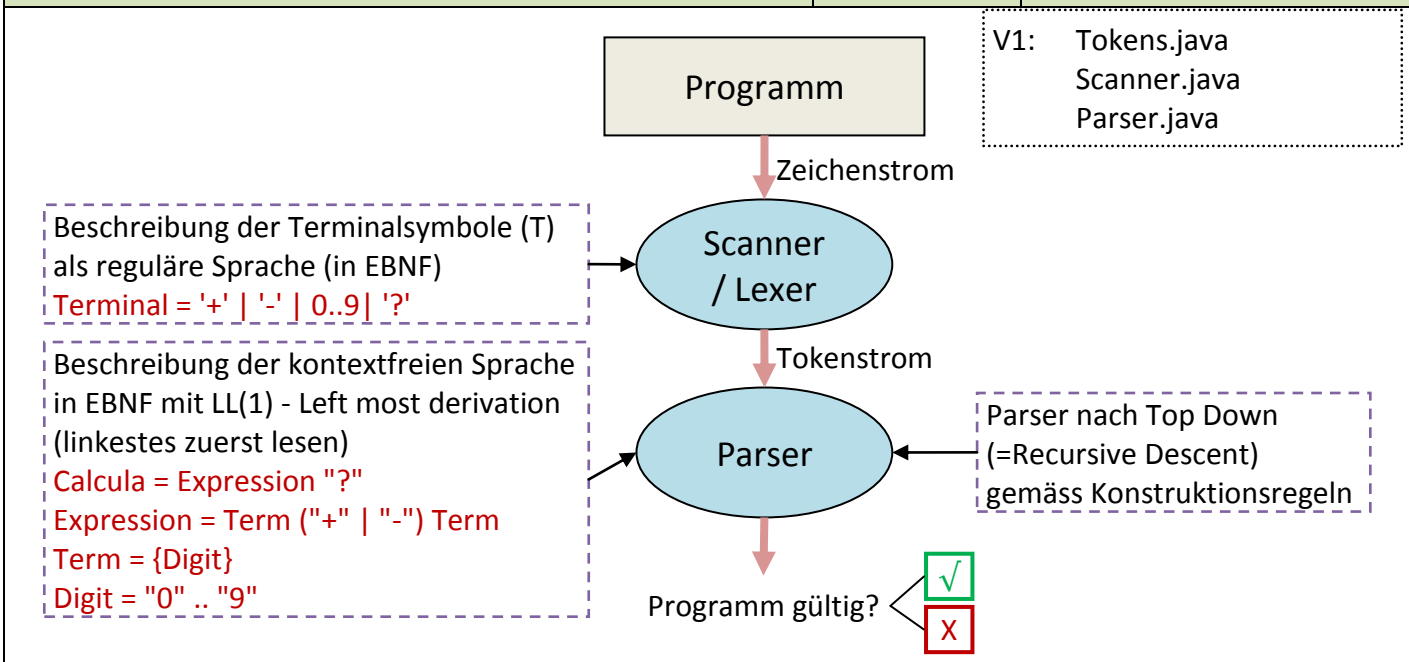
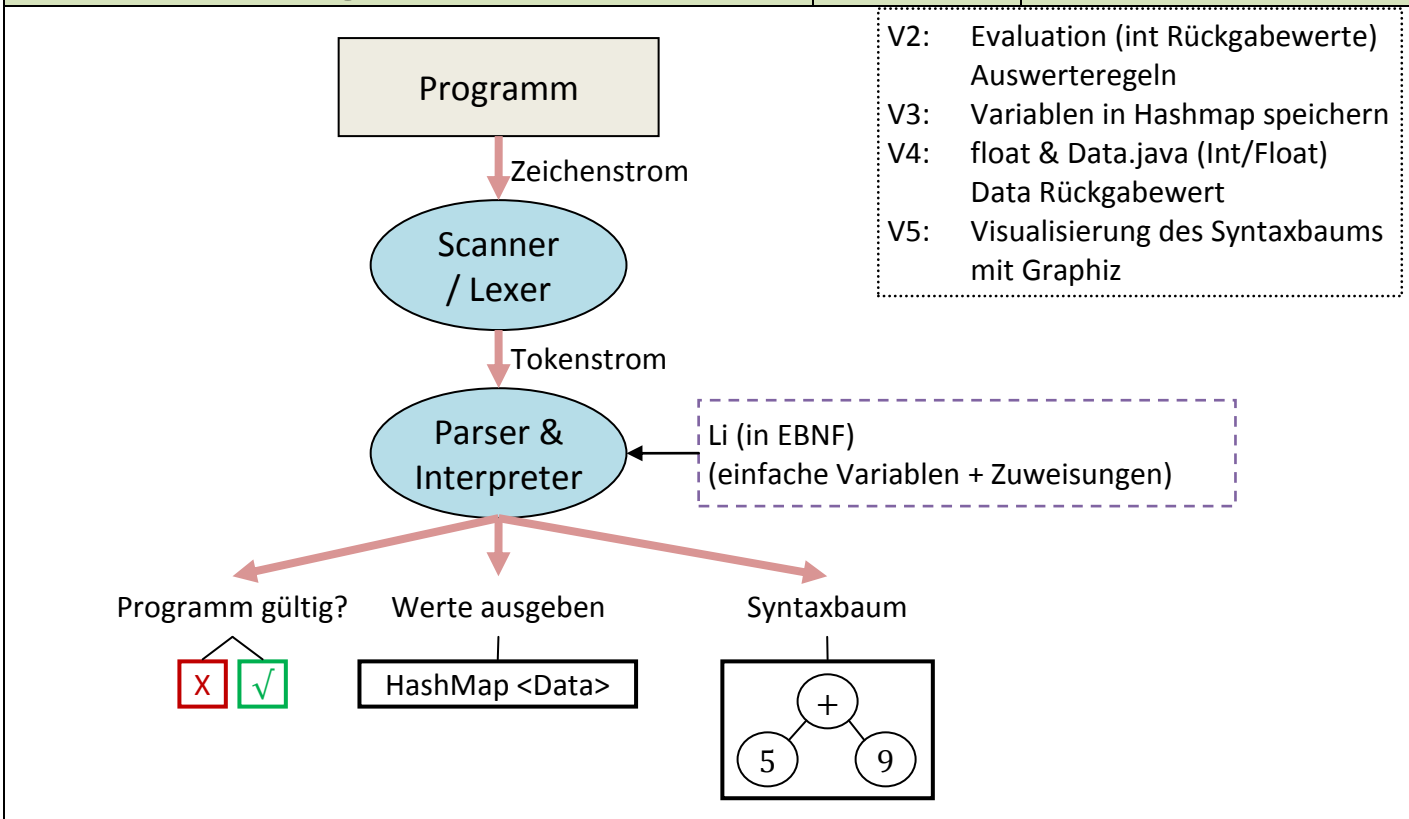
COMPILERBAU - ÜBERSICHT

1 Parser | Wirth 2,3,4.1 | 03 Syntaktische Analyse



Parser nach Top Down (=Recursive Descent) gemäss Konstruktionsregeln

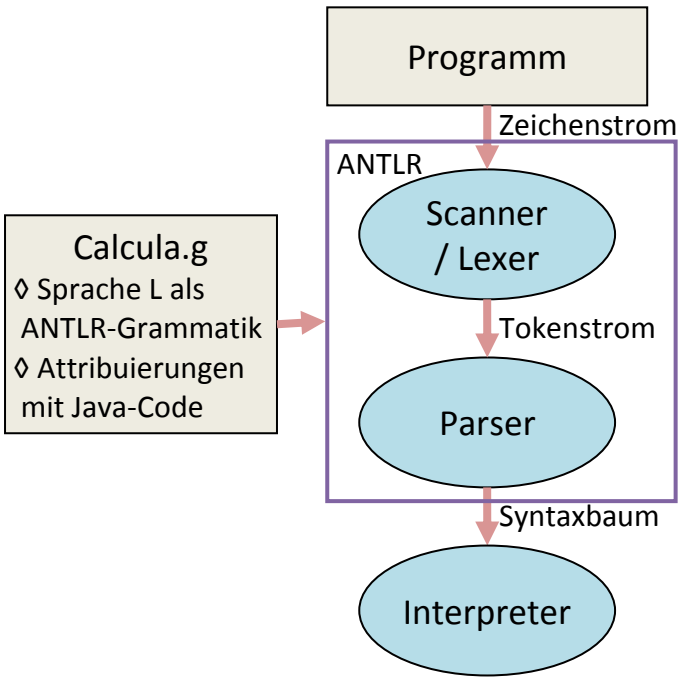
2 Attributierungen (Interpreter) | Wirth 5 | 04 Semantische Analyse



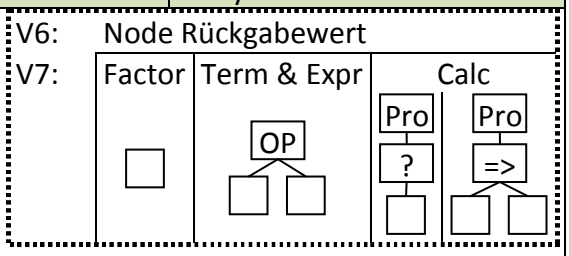
3 Syntaxtree

Wirth 5.3

05 Generator
06 Syntaxbäume

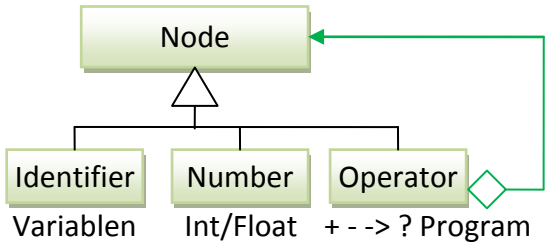


Pro = Program (=Hauptknoten)



bisher: DIY (Do it yourself)
neu: ANTLR Codegenerator

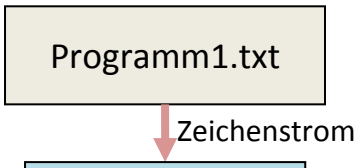
Datenstruktur des Syntaxbaumes (Kompositum)

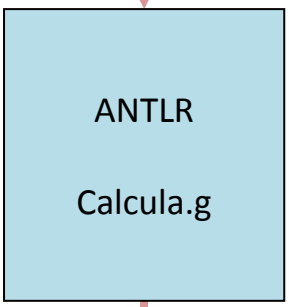


Repräsentation der Programmstruktur
Alle Aufgaben des Interpreters abgewickelt

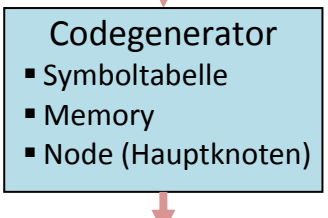
4 Typensystem (Arrays & Records)

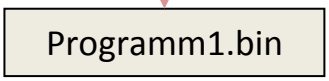
07 Typensystem U1





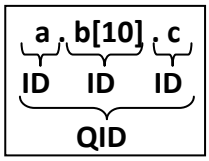
Symboltabelle
Syntaxbaum + Typen + Objekte





V8: Symboltabelle (Object)
- Namen
- Klassenzugehörigkeit
- Referenz auf Datentyp

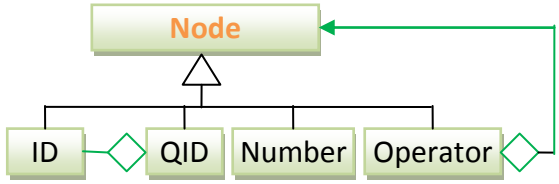
V9: Memorylayout
- Ort im Memory
- Grösse im Memory
- Node hat getTyp



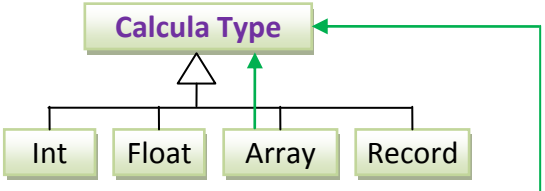
Berechnung von Konstanten Ausdrücken bleiben erhalten (Node evaluate)

Spracherweiterung "strong typing"
- Datentypdefinition
- Deklaration

package syntaxtree

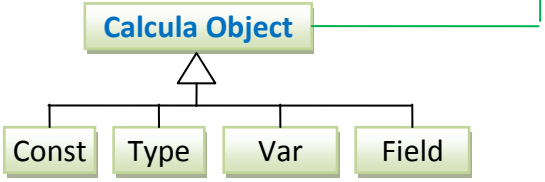


package typensystem typen



Record = Klasse in Java

package typensystem object



Type = für eigene Datentypen
Field = Variable einer Klasse

5 Codegenerator 1

07 Typensystem U2

1. Memorylayout für Variablen

Typen	Berechnung des Speicherplatzbedarfs "Bottom up" -> Type.size()
Variablen	Berechnung der (Basis)Speicheradresse mit Hilfe eines Memory Managers
QID / ID	Referenzieren eines Bereichs innerhalb des Variablenblocks Berechnung der Adressen (für Array-Indizes nur, wenn Index-Expressions statisch auswertbar ist)

2. Typisierung des Syntaxbaumes

	Wiederaufnahme der Typenregeln (siehe 2 Attributierungen) void-Typ

6 Codegenerator 2

Plattform & Expressions

08 Codegenerierung U1 bis U3

1. Zielplattform

Registerarchitektur
Befehlssatz (Register, Speicher, Kontrollflus))

2. Registersatz als Stack

-> Expressions mit Post-fix-Traversal bearbeiten

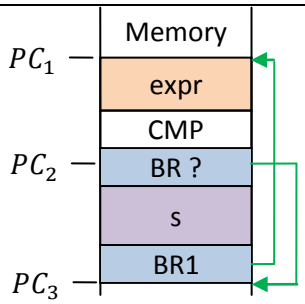
3. Bedingte Anweisungen & Schleifen

-> Umsetzung mit Sprüngen (Bxx-Befehle, Goto)

-> Fixup von Vorwärtssprung-Adressen

z.B. while:

WHILE expr DO **s** END



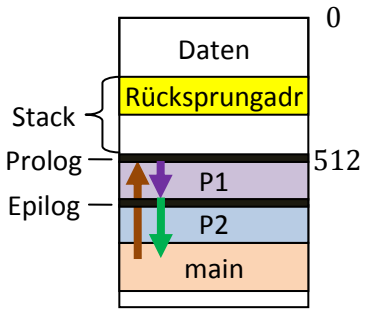
a) PC_1 merken
1) Code für expression
2) Platz für Sprung PC_2
3) Code für s
4) BR1 mit Sprung PC_1
5) BR? an PC_2 zuweisen (Sprung an Adresse PC_3)

4. Prozeduren (Procedure Call)

Hauptprogramm
| Prozedur P1 (formale Parameter)
| | lokale Variablen, Code
| Prozedur P2 (-------)
| |
| P1(-----)

Stack: Verwaltete Rücksprungadressen
Prolog = Speichern der Rücksprungadresse (PUSH)
Epilog = Schreibt die Rücksprungadresse (POP)
Aufrufkonvention der formalen Parameter

- CbV (Call by Value) – Java
- CbR (Call by Referenz) – C, C#, (.net)



Call:
1) Springe zur Startadresse der Prozedur P1
+ Rücksprungadr. merken

Code:
2) Code in P1 ausführen
3) Zurückspringen (und weiterfahren)