1 Einführung

Nutzen ComBau
- Programmiersprachen und Sprachkonzepte besser verstehen
- Sprachfeatures beurteilen können
- Konzepte in verwandten Bereichen einsetzen

1.1 Begriffe

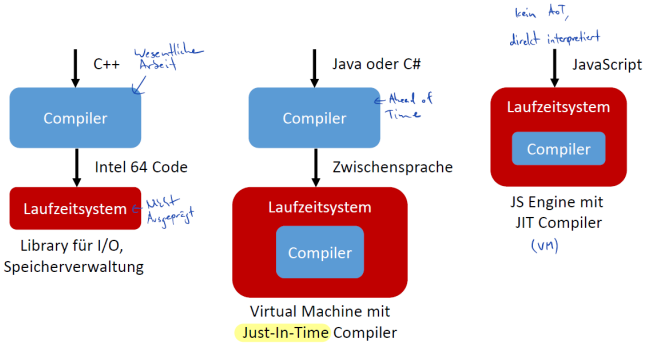
Compiler
- Transformiert Quellcode in Maschinencode

Runtime System
- Unterstützt die Programmausführung mit Software und Hardware Mechanismen

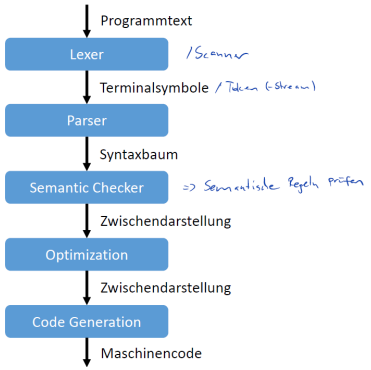
Syntax
- Definiert Struktur des Programms
- Bewährte Formalismen für Syntax

Semantik
- Definiert Bedeutung des Programms
- Meist in Prosa beschrieben

1.2 Architekturen



1.3 Aufbau Compiler



1.3.1 Lexer

Lexikalische Analyse, Scanner
- Zerlegt Programmtext in Terminalsymbole (Tokens)
- keine Tiefenstruktur

1.3.2 Parser

Syntaktische Analyse
- Erzeugt Syntaxbaum gemäss Programmstruktur
- Kontextfreie Sprache

1.3.3 Semantic Checker

Semantische Analyse
- Löst Symbole auf
- Prüft Typen und semantische Regeln

1.3.4 Optimization

- Wandelt Zwischendarstellung in effizientere um

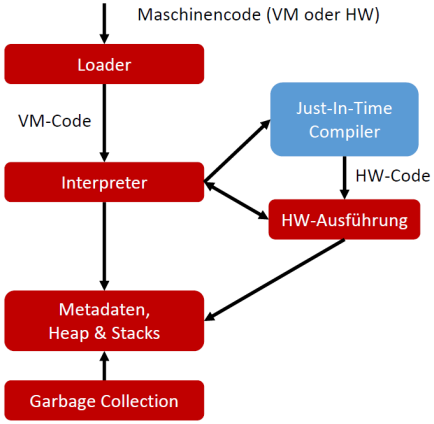
1.3.5 Code Generation

- Erzeugt ausführbarer Maschinencode

1.3.6 Zwischenarstellung

Intermediate Representation
- Beschreibt Programm als Datenstruktur (diverse Varianten)

1.4 Aufbau Laufzeitsystem



1.4.1 Loader

- Lädt Maschinencode in Speicher
- Veranlasst Ausführung

1.4.2 Interpreter

- Liest Instruktionen und emuliert diese in Software

1.4.3 JIT (Just-In-Time) Compiler

- Übersetzt Code-Teile in Hardware-Instruktionscode

1.4.4 HW-Ausführung (nativ)

- Lässt Instruktionscode direkt auf HW-Prozessor laufen

1.4.5 Metadaten, Heap + Stacks

- Merken Programminfos, Objekte und Prozeduraufrufe

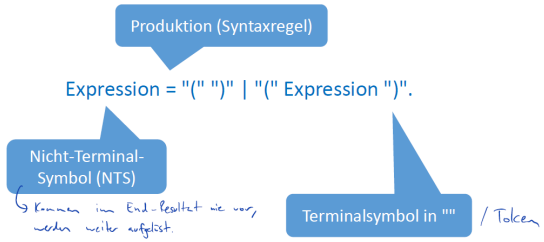
1.4.6 Garbage Collection

- Räumt nicht erreichbare Objecte ab

1.5 Syntax

1.5.1 EBNF

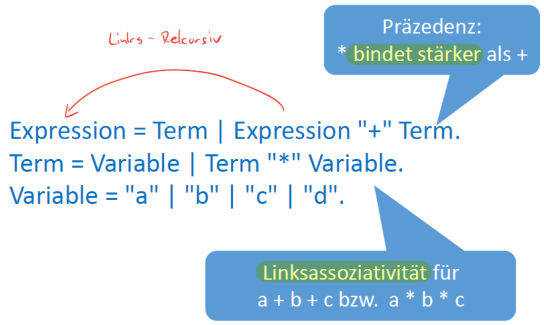
Extended Backus-Naur Form



	Beispiel	Sätze
Konkatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" oder "B"
Option	["A"]	leer oder "A"
Wiederholung	{ "A" }	leer, "A", "AA", "AAA", etc.

Runde Klammern für stärkere Bindung.
| bindet schwächer als andere Konstrukte.

1.5.2 Arithmetische Ausdrücke



2 Lexikalische Analyse

2.1 Lexer / Scanner

Endlicher Automat (DEA)
- Kümmert sich um die lexikalische Analyse
- Input: Zeichenfolge (Programmtext)
- Output: Folge von Terminalsymbolen (Tokens)

2.1.1 Aufgaben

- Fasst Textzeichen zu tokens zusammen
- Eliminiert Whitespaces
- Eliminiert Kommentare
- Merkt Positionen in Programmcode

2.1.2 Nutzen

Abstraktion
- Parser muss sich nicht um Textzeichen kümmern

Einfachheit
- Parser braucht Lookahead pro Symbol, nicht Textzeichen

Effizienz
- Lexer benötigt keinen Stack im Gegensatz zu Parser

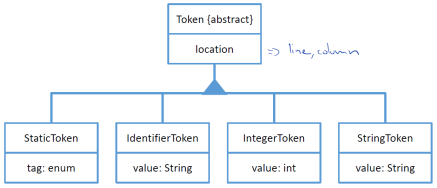
2.2 Tokens

Statisch (Keywords, Operationen, Interpunktion)
- *if*
- *else*
- *while*
- *
- *&&*
- *;*

Identifiers
- MyClass
- readFile
- name2

Zahlen
- 123
- 0xfe12
- 1.2e-3

Strings
- ”Hello”
- ””
- ”\n”



2.2.1 Lexem

- Spezifische Zeichenfolge, die einen Token darstellt
- z.B. *MyClass* ist ein Lexem des Tokens Identifier

2.2.2 Maximum Munch

- Lexer absorbiert möglichst viel in einem Token

2.3 Reguläre Sprachen

- Lexer unterstützt nur reguläre Sprachen
- **Regulär:** Als EBNF ohne Rekursion ausdrückbar

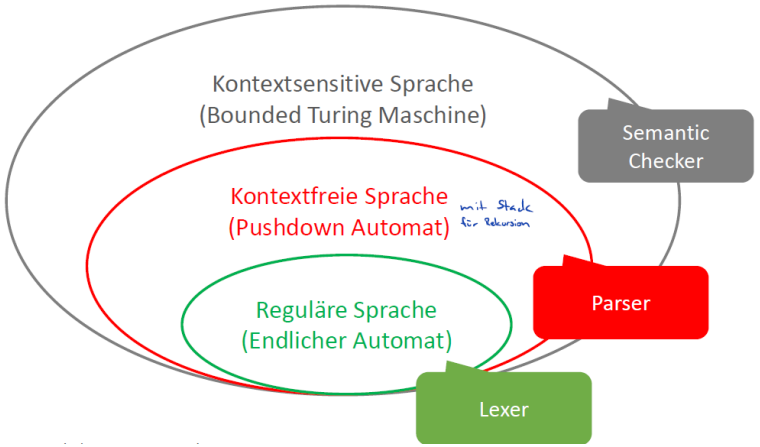
```
Integer = Digit { Digit }.
Digit = "0" | ... | "9".
```

Regulär
=> nicht rekursiv

```
Ausdruck = [ "(" Ausdruck ")" ].
```

Nicht regulär
=> rekursiv nötig

2.4 Chomsky Hierarchie



2.5 Lexer Gerüst

```
class Lexer {
    private final Reader reader;
    private char current; // one Character Lookahead
    private boolean end; // EOF

    private Lexer(Reader reader) {
        this.reader = reader;
    }

    public static Iterable<Token> scan(Reader reader) {
        return new Lexer(reader).readTokenStream();
    }

    // ...
}
```

2.6 Token Stream lesen

```
Iterable<Token> readTokenStream() {
    var stream = new ArrayList<Token>();
    readNext(); // One Character Lookahead
    skipBlanks();
    while (!end) {
        stream.add(readToken());
        skipBlanks();
    }
    return stream;
}
```

2.7 Lexer Kernlogik

```
Token readToken() {
    if(isDigit(current)) {
        return readInteger();
    }
    if(isLetter(current)) {
        return readName(); // Identifier / Keyword
    }
    return switch(current) {
        case '"': readString();
        case '+': readStaticToken(Tag.Plus);
        case '-': readStaticToken(Tag.Minus);
        case '/': readPotentialSlash();
    }
}
```

2.7.1 Static Token scannen

```
StaticToken readStaticToken(Tag tag) {
    readNext();
    return new StaticToken(tag);
}
```

2.7.2 Zahlen scannen

Beachten:
- Range Check (32 bit): Integer Overflow
- $Integer.MIN = Integer.MAX + 1$

```
IntegerToken readInteger() {
    int value = 0;
    while (!_end && isDigit(current)) {
        int digit = current - '0'; // char to int
        value = value * 10 + digit; // create decimal number
        readNext();
    }
    return new IntegerToken(value);
}
```

2.7.3 Identifier und Keywords scannen

```
Token readName() {
    String name = Character.toString(current);
    readNext();
    while(!end & (isLetter(current) || isDigit(current))) {
        name += current;
        readNext();
    }
    if(KEYWORDS.containsKey(name)) {
        return new StaticToken(KEYWORDS.get(name));
    }
    return new IdentifierToken(name);
}
```

2.7.4 String scannen

Beachten:
- Kein \t
- Kein \"
- Kein \n
- Keine mehrzeiligen Strings

```
StringToken readString() {
    readNext(); // Skip leading double Quote
    String value = "";
    while (!end && current != '"') {
        value += current;
        readNext();
    }
    if(end) {
        // Error: String not closed
    }
    readNext(); // Skip trailing double Quote
    return new StringToken(value);
}
```

2.7.5 Kommentare erkennen

```
StaticToken readPotentialSlash() {
    readNext();
    if(current == '/') {
        skipLineComment();
        // move on to next token
    } else if (current == '*') {
        skipCommentBlock();
        // move on to next token
    } else {
        return new StaticToken(Tag.Divide);
    }
}
```

3 Parser

Kontextfreie Sprache

- Kümmert sich um die syntaktische Analyse
- Input: Tokens (Terminalsymbole)
- Output: Syntaxbaum / Parse Tree

3.1 Aufgabe

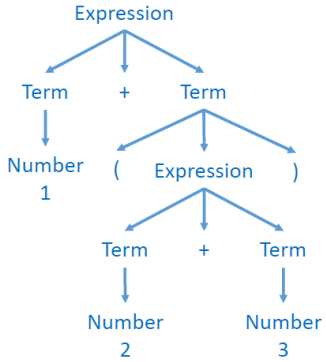
- Finde eindeutige Ableitung der Syntaxregeln, um einen gegebenen Input herzuleiten
- Analysiert die gesamte Syntaxdefinition (mit rekursiven Regeln)
- Erkennt, ob Eingabetext Syntax erfüllt
- Erzeugt Syntaxbaum

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

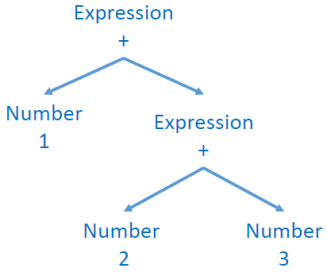
3.2 Parse Tree

- Concrete Syntax Tree
- Ableitung der Syntaxregeln als Baum wiedergespiegelt
- Kann generiert werden



3.2.1 Abstract Syntax Tree

- Unwichtige Details auslassen
- Struktur vereinfacht
- Für Weiterverarbeitung massgeschneidert
- Eigendesign nach Gusto des Compiler-Entwicklers
- Nur mit Selbstimplementation möglich



3.3 Parser Strategien

3.3.1 Top-Down

- Beginne mit Start-Symbol
- Wende Produktionen an
- Expandiere Start-Symbol auf Eingabetext
- $Expr- > Term + Term- > ...- > 1 + (2 - 3)$

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Number "+" Term
Number "+" "(" Expression ")"
Number "+" "(" Term "-" Term ")"
Number "+" "(" Number "-" Term ")"
Number "+" "(" Number "-" Number ")"

Top-Down

linksseitig expandieren

3.3.2 Bottom-Up

- Beginne mit Eingabetext
- Wende Produktionen an
- Reduziere Eingabetext auf Start-Symbol
- $Expr < -Term + Term < -... < -1 + (2 - 3)$

Input: 1 + (2 - 3)

Ableitung:
Expression
Term "+" Term
Term "+" "(" Expression ")"
Term "+" "(" Term "-" Term ")"
Term "+" "(" Term "-" Number ")"
Term "+" "(" Number "-" Number ")"
Number "+" "(" Number "-" Number ")"

Bottom-Up

rechtsseitig reduzieren

3.4 Recursive Descent

- Pro Nicht-Terminalsymbol eine Methode
  - Implementiert die Erkennung gemäss EBNF-Produktion
- Vorkommen eines Nicht-Terminalsymbols in Syntax
  - Aufruf der entsprechenden Methode
- Funktioniert bei rekursiven und nicht-rekursiven Produktionen

```
void parseExpression() {
    parseTerm();
    // ...
}
void parseTerm() {
    parseExpression();
    // ...
}
```

3.5 Parser Gerüst

```
public class Parser {
    private final Iterator<Token> tokenStream;
    private Token current; // One Token Lookahead

    private Parser(Iterable<Token> tokenStream) {
        this.tokenStream = tokenStream.iterator();
    }

    public static ProgramNode parse(Iterable<Token> stream) {
        return new Parser(stream).parseProgram(); // Aufbasierte Klasse
    }
}
```

3.5.1 Parser-Einstieg

$Program = Expression$

```
private ProgramNode parseProgram() {
    var classes = new ArrayList<ClassNode>();
    parseExpression();
    while (!isEnd()) {
        next();
        classes.add(parseClass());
    }
    return new ProgramNode(classes);
```

```
}
```

3.5.2 Expression

$Expression = Term(" + "|" - ")Term$

```
Expression parseExpression() {
    var left = parseTerm();
    while(is(Tag.PLUS) || is(Tag.MINUS)) {
        var op = is(Tag.PLUS) ? Operator.PLUS : Operator.MINUS;
        next();
        var right = parseTerm();
        var left = new BinaryExpression(op, left, right);
    }
    return left;
}
```

3.5.3 Term

$Term = Number|"("Expression")"$

```
Expression parseTerm() {
    if(isInteger()) {
        int value = readInteger();
        next();
        return new IntegerLiteral(value);
    } else if (is(Tag.OPEN_PARENTHESIS)) {
        next();
        var expression = parseExpression();
        if(is(Tag.CLOSE_PARENTHESIS)) {
            next();
        } else {
            error(); // missing closed parenthesis
        }
        return expression();
    } else {
        error(); // missing open parenthesis
    }
}
```

3.6 One Symbol Lookahead

Statement
$Assignment|IfStatement$

- Bestimme mögliche Terminalsymbole, die mit einer Produktion ableitbar sind (FIRST-Menge)
- Benutze FIRST zur Entscheidung der Alternative beim zielorientierten Parsen

```
void parseStatement() {
    if(isIdentifier()) { // FIRST(Assignment)
        parseAssignment();
    } else if(is(Tag.IF)) { // FIRST(IfStatement)
        parseIfStatement();
    } else {
        error();
    }
}
```

3.7 Technische Syntax-Umformung

- Falls 1 Lookahead nicht reicht

```
Statement = Assignment | Invocation
Assignment = Identifier "=" Expression
Invocation = Identifier "(" ")"
```

↓

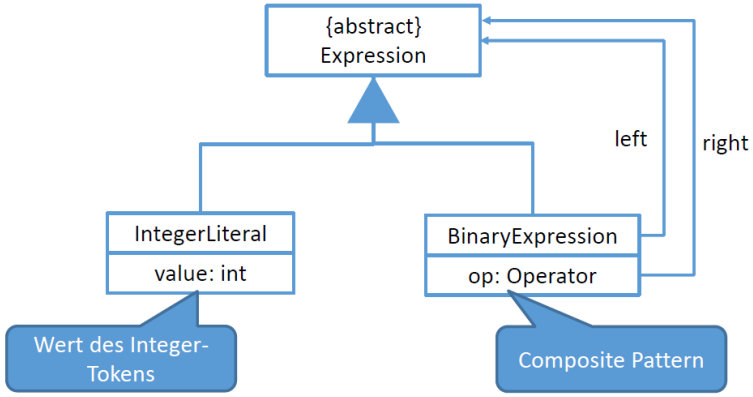
```
Statement = Identifier (AssignmentRest | InvocationRest)
AssignmentRest = "=" Expression
InvocationRest = "(" ")"
// Lookahead 1 reicht wieder
```

3.7.1 Code Beispiel

```
var parseStatement() {
    var identifier = readIdentifier();
    next();
    if (is(Tag.ASSIGN)) {
        parseAssignmentRest(identifier);
    } else if (is(Tag.OPEN_PARENTHESIS)) {
        parseInvocationRest(identifier);
    } else {
        error();
    }
}
```

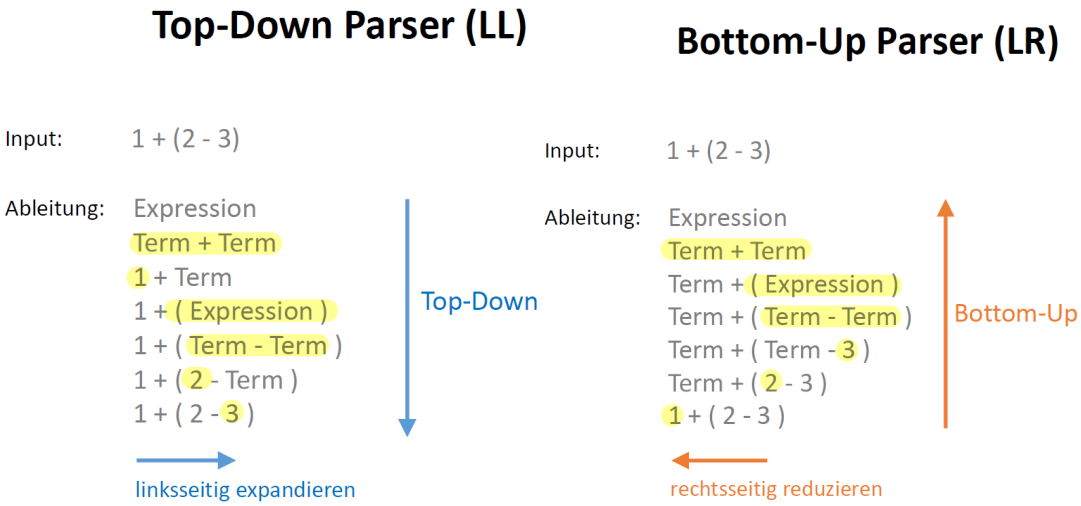
4 Parser Vertiefung

4.1 Abstrakter Syntaxbaum Design



4.2 Bottom-Up Parsing

- Mächtiger als LL (Top-Down) Parser
- Kann Linksrekursion behandeln



4.2.1 Ansatz

- Lese Symbole im Text ohne fixes Ziel
- Prüfe nach jedem Schritt, ob gelesene Folge Produktion entspricht
  - Wenn ja: Reduziere auf Syntaxkonstrukt (REDUCE)
  - Wenn nein: Lese weiteres Symbol im Text (SHIFT)
- Am Schluss bleibt Startsymbol übrig, sonst Syntaxfehler

4.2.2 Beispielablauf

Schritt	Erkannte Konstrukte	Rest der Eingabe
		1 + (2 - 3)
SHIFT	1	+ (2 - 3)
REDUCE	Term	+ (2 - 3)
SHIFT	Term +	(2 - 3)
SHIFT	Term + (	2 - 3)
SHIFT	Term + (2	- 3)
REDUCE	Term + (Term	- 3)
SHIFT	Term + (Term -	- 3)
SHIFT	Term + (Term - 3	)
REDUCE	Term + (Term - Term	)
REDUCE	Term + (Expression	)
SHIFT	Term + (Expression)	
REDUCE	Term + Term	
REDUCE	Expression	

4.2.3 Parser Tabelle - Vereinfacht

Erkannte Konstrukte	Regel
... Number	REDUCE Term
... Term + Term	REDUCE Expression
... "(" Expression ")"	REDUCE Term
Sonst	SHIFT

Suffix des Zustands entscheidet (Stack-Prinzip)

4.3 LR-Parser (Bottom-Up) Varianten

LR(0)
- Parse Tabelle ohne Lookahead erstellen
- Zustand reicht, um zu entscheiden

SLR(k) (Simple LR)
- Lookahead bei REDUCE, um Konflikt zu lösen
- Keine neuen Zustände

LALR(k) (Look-Ahead LR)
- Analysiert Sprache auf LR(0)-Konflikte
- Benutzt Lookahead bei Konfliktstellen mit neuen Zuständen

LR(k)
- Pro Grammatikschritt + Lookahead ein Zustand
- Nicht praxistauglich, zu viele Zustände

5 Semantic Checker

- Kümmert sich um die semantische Analyse
- Input: Syntaxbaum
- Output: Zwischendarstellung (Syntaxbaum + Symboltabelle)

5.1 Semantische Prüfung

Deklarationen
- Jeder Identifier ist eindeutig deklariert

Typen
- Typregeln sind erfüllt

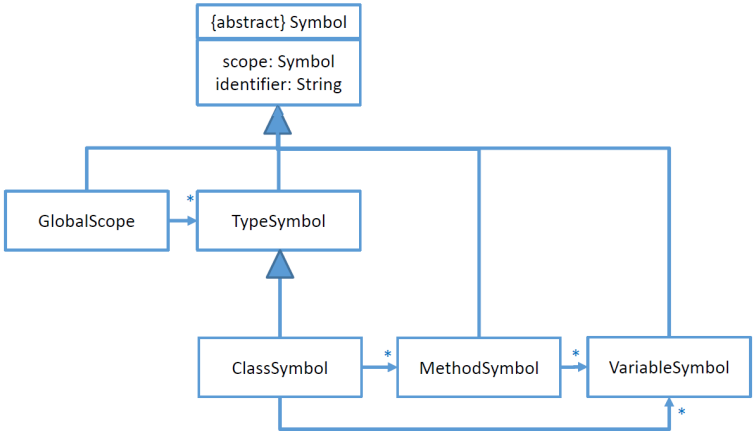
Methodenaufrufe
- Argumente und Parameter sind kompatibel

Weitere Regeln
- z.B. keine zyklischen Vererbung
- nur eine main() Methode

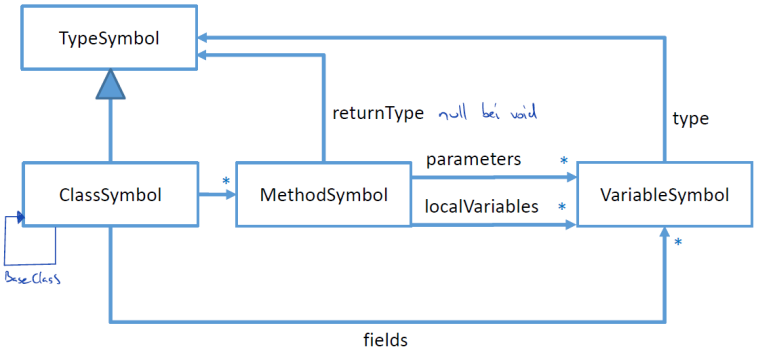
5.2 Symboltabelle

- Datenstruktur zur Verwaltung der Deklarationen
- Wiederspiegelt hierarchische Bereiche im Programm

5.2.1 Design



5.2.2 Detailliertere Beziehungen



5.2.3 Design Aspekte

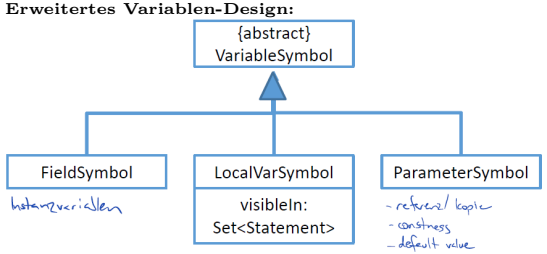
Typinfo für Variable-Symbol
- Zuerst unaufgelöst (Identifier)

Weitere Infos
- Klassen: Basisklasse

Lokale Variablen
- Deklarationsbereich merken (Statements)

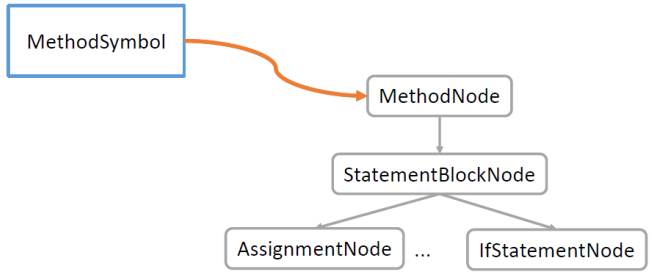
Erweitertes Typ-Design
- Klassen
- Basistypen (int, boolean, string)
- Arrays

Erweitertes Variablen-Design:



5.2.4 AST Verknüpfung

- Symboltabelle enthält Mapping Symbol → AST
- Für alle Deklarationen




```
node = symbolTable.getDeclarationNode(symbol)
```

5.3 Global Scope

- Mehrere Klassen im Programm

5.4 Shadowing

- Deklarationen in inneren Bereichen verdecken gleichnamige von äusseren Bereichen
- Hiding: Bei gleicher Member-Name bei Vererbung

5.5 Vorgehen

1. Konstruktion der Symboltabelle
2. Typen in Tabelle auflösen
3. Deklaration in AST auflösen
4. Typen in AST auflösen

5.5.1 1. Konstruktion der Symboltabelle

AST traversieren
- Beginne mit Global Scope
- Pro Klasse, Methode, Parameter, Variable: Symbol in übergeordnetem Scope einfügen
- Explizit und/oder mit Visitor

Forward-Referenzen → Typ-Namen und Designatoren noch nicht auflösen!
- Da vlt noch nicht alle Klassen in der Symboltabelle sind

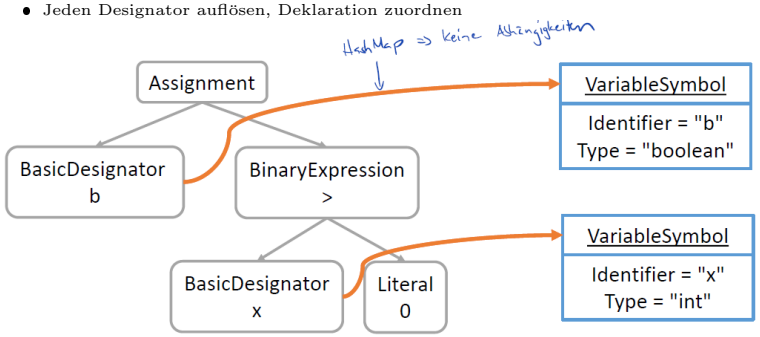
5.5.2 2. Typen in Tabelle auflösen

- Für Variablentype, Parametertyp, Rückgabetyp etc.
- Brauche Suche für Identifier auf Symboltabelle
  - Starte mit innerstem Scope
  - Suche stetig nach aussen ausbreiten
  - Zuletzt in Global Scope suchen, ansonsten nicht vorhanden

```
Symbol find(Symbol scope, String identifier) {
    if(scope == null) {
        return null; // nicht im global scope
    }
    for (Symbol declaration : scope.allDeclarations()) {
        if(declaration.getIdentifier().equals(identifier)) {
            return declaration;
        }
    }
    return find(scope.getScope(), identifier); // rekursiv in nächst höheren Bereich
}
```

5.5.3 3. Deklaration in AST auflösen

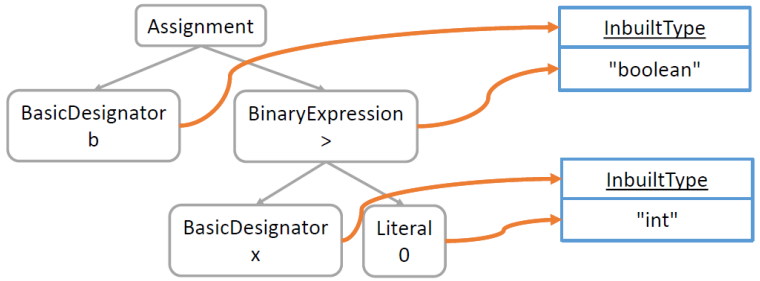
- Traversiere Ausführungscode in AST
- Jeden Designator auflösen, Deklaration zuordnen



5.5.4 4. Typen in AST bestimmen

Typ zu jeder Expression zuordnen

- Literal: definierter Typ
- Designator: Typ der Deklaration
- Unary/BinaryExpression: Resultat des Operators



Ablauf der Typenbestimmung:

- Post-Order-Traversierung
- AST am besten nicht erweitern sondern Maps in Symboltabelle verwenden

5.5.5 Typauflösung per Visitor

```
@Override
public void visit(BinaryExpressionNode node) {
    Visitor.super.visit(node); // post-order travers
    var leftType = symboltable.findType(node.getLeft());
    var rightType = symboltable.findType(node.getRight());
    // ...
    switch(node.getOperator()) {
        case PLUS -> {
            checkType(leftType, globalScope.getIntType());
            checkType(rightType, globalScope.getIntType());
            symboltable.fixType(node, globalScope.getIntType());
        }
        // ...
    }
}
```

5.6 Semantic Checks

- Alle Designatoren beziehen sich auf Variablen/Methoden
- Typen stimmen bei Operatoren
- Kompatible Typen bei Zuweisungen
- Argumentliste passt auf Parameterliste
- Bedingungen in if, while sind boolean
- Return Ausdruck passt
- Keine Mehrfachdeklaration
- Kein Identifier ist reserviertes Keyword
- Exakt eine main() Methode
- Array length is read-only
- Kein Exit ohne Return (ausser void)
- Lesen von unitialisierten Variablen
- Null-Dereferenzierung
- Ungültiger Array-Index
- Division by Zero
- Out of Memory bei new()

6 Code Generator

6.1 Aufgabe

Erzeugung von ausführbarem Maschinencode

- Input: Zwischendarstellung (Symboltabelle + AST)
- Output: Maschinencode

Mögliche Zielmaschinen

- Reale Maschine, z.B. intel 64, ARM Prozessor
- VM, z.B. JVM, .NET CLI

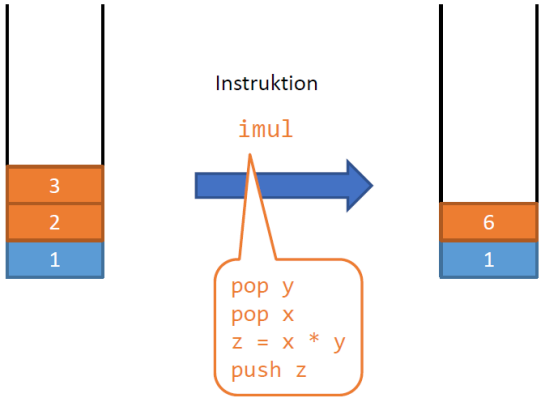
6.2 Unsere Zielmaschine

Kernkonzepte

- Virtueller Stack-Prozessor: keine Register
- Branch Instructions (Goto): Programmfluss steuern
- Metadaten

6.3 Stack Prozessor

- Instruktionen benutzen Auswertungs-Stack
- Keine Register wie auf echten Prozessoren



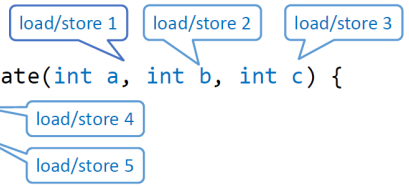
6.4 Auswertungs-Stack

- Jede Instruktion hat definierte Anzahl von Pop- und Push-Aufrufen
- Eigener Stack pro Methodenaufruf
- Stack hat unbeschränkte Kapazität

Instruktion	Bedeutung	Auswertungs-Stack
`ldc <const>`	Lade Konstante (int, boolean, string)	1 Push
`iadd`	Integer Addition	2 Pop, 1 Push
`isub`	Subtraktion	2 Pop, 1 Push
`imul`	Multiplikation	2 Pop, 1 Push
`idiv`	Integer Division	2 Pop, 1 Push
`irem`	Modulo	2 Pop, 1 Push
`ineg`	Integer Negation	1 Pop, 1 Push
`load <num>`	Lade parameter oder lokale Variable	1 Push
`store <num>`	Speichere Parameter oder lokale Variable	1 Pop

6.4.1 Load/Store Nummerierungen

- *this* Referenz: Index 0
- Danach, n **Parameters**: Index $1..n$
- Danach, m **lokale Variablen**: Index $n+1...n+m$



6.4.2 Compare-Instruktionen

Instruktion	Bedeutung
`cmpeq`	Compare Equal (verschiedene Typen)
`cmpne`	Compare Not Equal (verschiedene Typen)
`icmpgt`	Integer Compare Greater Than
`icmpge`	Integer Compare Greater Equal
`icmplt`	Integer Compare Less Than
`icmple`	Integer Compare Less Equal

Pop right, Pop left, Push boolean

6.4.3 Branch-Instruktionen

Instruktion	Bedeutung	Auswertungs-Stack
`goto <label>`	Branch (bedingungslos)	-
`if_true <label>`	Branch falls true	1 pop
`if_false <label>`	Branch falls false	1 pop

6.4.4 Metadaten

- Zwischensprache kennt alle Informationen zu
  - Klassen (Namen, Typen der Fields und Methoden)
  - Methoden (Namen, Parametertypen und Rückgabetyp)
  - Lokale Variablen (Typen)
- Kein direktes Speicherlayout festgelegt
- Nicht enthalten
  - Namen von lokalen Variablen und Parameter
  - Diese sind nur nummeriert

Verwendung:

- Speicherplatz-Allozierung
- Fehlermeldungen
- Funktionsaufrufe

6.5 Code Generierung

1. Traversiere Symboltabelle: Erzeuge Bytecode Metadaten
2. Traversiere AST pro Methode (Visitor): Erzeuge Instruktionen via Bytecode Assembler
3. Serialisiere in Output Format

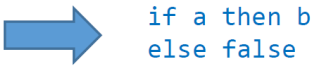
6.5.1 Backpatching

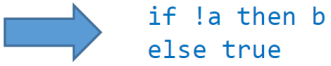
- Branch Offsets auflösen
- Label: relativer Instruktions-Offset ab Ende der aktuellen Branch-Instruktion

6.5.2 Template-Basierte Generierung

- Postorder-Traversierung: Kinder zuerst besuchen
- Jeweils Template für erkanntes Teilbaum-Muster anwenden

6.5.3 Short-Circuit Semantik

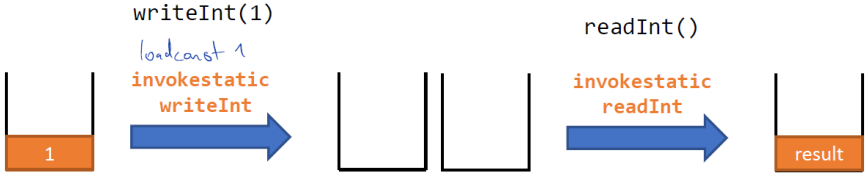




6.5.4 Methodenaufruf

Statisch

- Vordefinierte Methoden: *readInt*() etc.

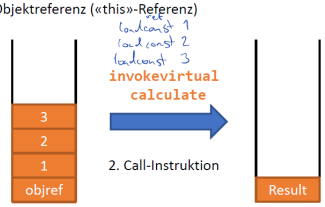


Virtuell

- Alle anderen Methoden

```
objref.calculate(1, 2, 3);
```

1. Argumente sind auf Stack (letzter zuoberst), zuunterst Objektreferenz («this»-Referenz)



2. Call-Instruktion

3. Call entfernt Argumente & Objektreferenz, legt Rückgabewert auf Stack (falls nicht void)

6.5.5 Parameter & Rückgabe

```
int sum(int x, int y) {
    return x + y;
}

load 1 // load param x
load 2 // load param y
iadd // x + y
ret // return from method (auch bei void, max 1 Wert auf Stack)
```