1 Einführung

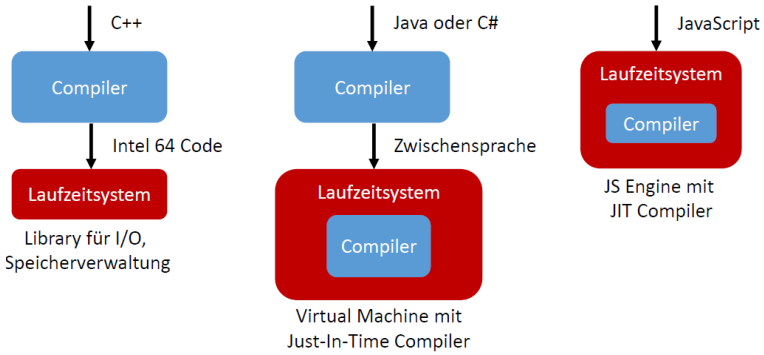
Wieso Compilerbau?

- Sprachkonzepte verstehen
- Einschränkungen und Kosten von Sprachfeatures beurteilen können
- Konzepte in verwandten Bereichen einsetzen: Converter, Analysen, Entwicklertools, Algorithmen

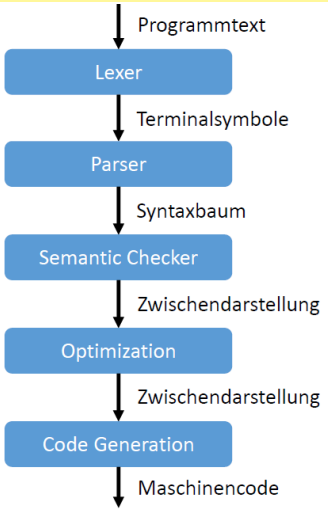
Compiler: Transformiert Quellcode einer Programmiersprache in ausführbaren Maschinencode.

Runtime System: Unterstützt die Ausführung mit software- und Hardware-Mechanismen

1.1 Architekturen

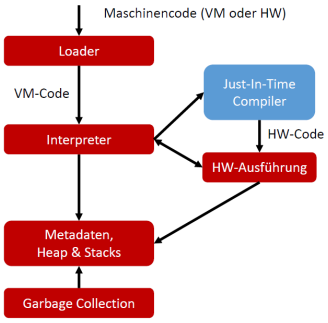


1.2 Aufbau Compiler



- Lexer (Lexikanische Analyse, Scanner)
  - Zerlegt Programmtext in Terminalsymbole (Tokens)
- Parser (Syntaktische Analyse)
  - Erzeugt Syntaxbaum gemäss Programmstruktur
- Semantic Checker (Schemantische Analyse)
  - Löst Symbole auf, prüft Typen und semantische regeln
- Optimization (Optional)
  - Wandelt Zwischendarstellung in effizientere um
- Code Generation
  - Erzeugt ausführbaren maschinencode
- Zwischendarstellung (Intermediate Representation)
  - Beschreibt Programm als Datenstruktur (diverse varianten)

1.3 Aufbau Laufzeitsystem



- Loader
  - Lädt maschinencode in Speicher, veranlasst Ausführung
- Interpreter
  - Liest Instruktionen und emuliert diese in Software
- JIT Compiler
  - Übersetzt Code-Teile in hardware-Instruktionscode
- HW-Ausführung (nativ)
  - Lässt Instruktionscode direkt auf HW-Prozessor laufen
- Metadaten, heap + Stacks
  - Merken programminfos, Objekte und Prozeduralaufrufe
- Garbage Collection
  - Räumt nicht erreichbare Objekte ab

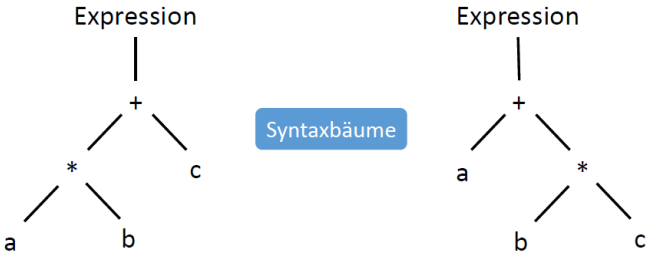
1.4 EBNF (Extended Backus Naur Form)

Definition einer Programmiersprache: Syntax (mittels Regeln/Formeln), Semantik (meist in Prosa)

1.4.1 EBNF Konstrukte

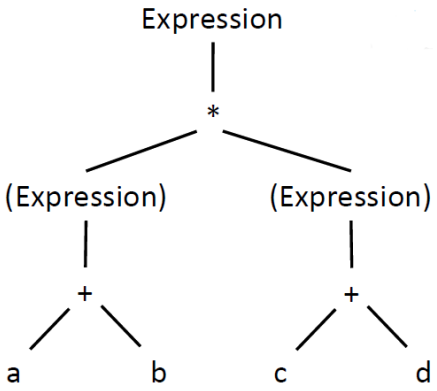
	Beispiel	Sätze
Konkatenation	"A" "B"	"AB"
Alternative	"A" | "B"	"A" oder "B"
Option	["A"]	leer oder "A"
Wiederholung	{ "A" }	leer, "A", "AA", "AAA", etc.

1.4.2 Arithmetische Adrücke



Expression = Term | Expression "+" Term.
Term = Variable | Term "*" Variable.
Variable = "a" | "b" | "c" | "d".

1.4.3 Explizite Klammerungen



Expression = Term | Expression "+" Term.
Term = Factor | Term "*" Factor.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".

Expression = Term { "+" Term }.
Term = Factor { "*" Factor }.
Factor = Variable | "(" Expression ")".
Variable = "a" | "b" | "c" | "d".

2 Lexikanische Analyse

→ **Kümmert sich um die lexikanische Analyse**

Input: Zeichenfolge (Programmtext)

Output: Folge von Terminalsymbolen (Tokens)

Aufgaben:

- Fasst Textzeichen zu Tokens zusammen
- Eliminiert Whitespaces und Kommentare
- Merkt Position in Programmcode für Fehlermeldung/Debugging

Nutzen:

→ Erleichtert spätere syntaktische Analyse (Parser)

- Abstraktion: Parser muss sich nicht um Textzeichen kümmern
- Einfachheit: Parser braucht Lookahead pro Symbol, nicht Textzeichen
- Effizienz: Lexer benötigt Stack im Gegensatz zu Parser

2.1 Tokens

- **Statisch:** Keywords, Operationen, Interpunktion
  - if, else, while, *, &&, ;
- **Identifiers**
  - MyClass, readFile, name2
- **Zahlen**
  - 123, oxfe12, 1.2e-3
- **Strings**
  - "Hello!", "", "01234", "\n"
- **Evt weitere**
  - Einzelne Characters wie 'a', '0'

2.2 Reguläre Sprachen

Regulär: Als EBNF ohne Rekursion ausdrückbar!!

```
Integer = Digit { Digit }.
Digit = "0" | ... | "9".
Ausdruck = [ "(" Ausdruck ")" ].
```

Regulär

Nicht regulär

```
Integer = Digit [ Integer ].
```

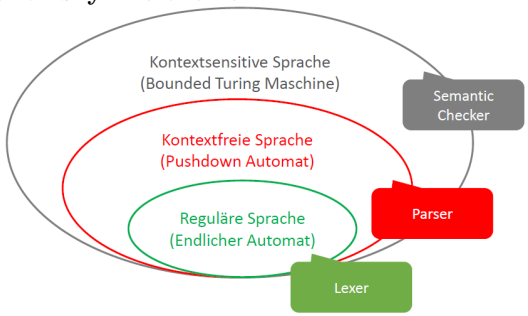
Umformung in äquivalente Syntax

```
Integer = Digit { Digit }.
```

keine Rekursionen

Regulär

Chomsky Hierarchie



2.3 Identifier

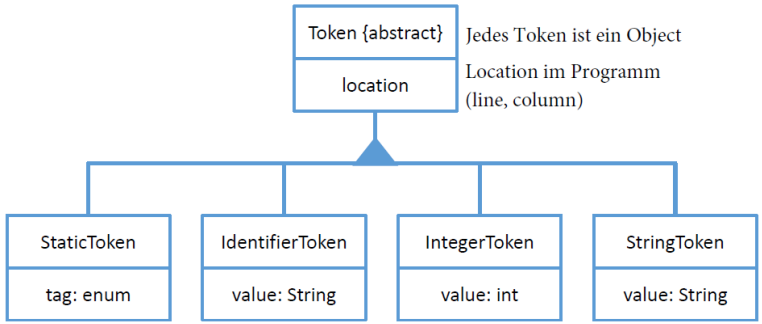
```
Identifier = Letter { Letter | Digit }.
Letter = "A" | ... | "Z" | "a" | ... | "z".
Digit = "0" | ... | "9".
```

- Bezeichner von Klassen Methoden, Variablen etc.
- Beginnt mit Buchstabe, danach Ziffern erlaubt
- (Java unterstützt auch Underscores, wir nicht)

2.4 Sonstiges

- **Maximum Munch:** Lexer absorbiert möglichst viel in einem Token
- **Whitespaces:** Von Lexer übersprungen, trennt Tokens, Tokens evt auch ohne Whitespac getrennt
- **Von Lexer übersprungen**
  - Blockkommentare: Nicht schachtelbar, weil sonst nicht mehr regulär
  - Zeilenkommentar: Bis Newline

2.5 Token-Model



2.6 Implementation

2.6.1 Tags für statische Tokens

Tipp: Reservierte Typnamen (void, boolean, int, string) und Werte (null, true, false) als Identifier im Lexer verarbeiten.

```
public enum Tag {
    CLASS,ELSE ,IF ,RETURN ,WHILE, ...
    AND, OR, PLUS, MINUS, SEMICOLON, ...
}
```

2.6.2 Lexer Gerüst

```
class Lexer {
    private final Reader reader;
    private char current; // One character lookahead
    private boolean end;

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

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

2.6.3 Token Stream lesen

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

2.6.4 Lexer Kernlogik

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

3 Parser Einführung

→ **Kümmert sich um die syntaktische Analyse**
Input: Folge von Terminalsymbolen (Tokens)
Output: Syntaxbaum/Parse Tree

Kontextfrei: Parser beschränkt sich auf kontextfreie Sprachen (in EBNF beschreibbar). Kontextabhängige Aspekte wie Boolean lassen sich nicht addieren etc. übernimmt der Semantic Checker
Aufgaben:

- Finde eindeutige Ableitung der Syntaxregeln, um einen gegebenen Input herzuleiten
- Analysiert die gesamte Syntaxdefinition
- Erkennt, ob Eingabetext Syntax erfüllt oder nicht
- Eindeutige Ableitung erwünscht
- Erzeugt Syntaxbaum

3.1 Concrete vs. Abstract Syntax Tree

Concrete: Ableitung der Syntaxregeln als Baum widerspiegelt
Abstract: Unwichtige Details auslassen, Struktur vereinfachen und für Weiterverarbbeitung massschneidern
→ Beides sind mögliche Intermediate Representations
→ Generierter Parser kann Concrete Syntax Tree liefern
→ Selbst implementierter Parser kann Abstract Syntax Tree liefern

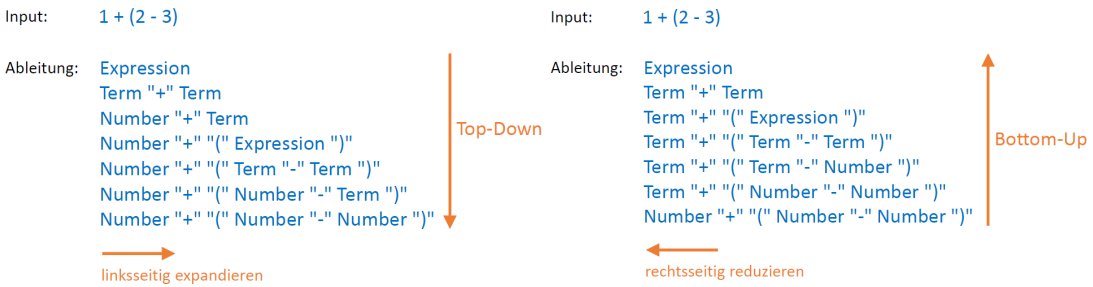
3.2 Parser Strategien

3.2.1 Top-Down

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

3.2.2 Buttom-Up

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



3.2.3 Recoursive Descent

- Pro Nicht-Terminalsymbol eine Methode
- Funktioniert bei rekursiven und nicht-rekursiven Produktionen

Diskusion:

- Recoursive Descent ist **Top-Down Parser**
  - Implizierter Stack durch Methodenaufrufe
  - Entspricht Push-Down Automat
- Zielorientierte Satzzerlegung (Predictive Direct)
  - Immer klar, welche Produktion genommen wird, Bevorzugte Vorgehensweise
- Anderer Ansatz: Backtracking
  - Falls unklar welche Produktion zu nehmen ist: Wähle Produktion aus, bei Syntaxfehler Undo und nächste probieren

3.3 Implementation

3.3.1 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();
    }
}
```

3.3.2 Parser Einstieg

```
private ProgramNode parseProgram() {
    var classes = new ArrayList<ClassNode>();
    try {
        while (!isEnd()) {
            next();
            classes.add(parseClass());
        }
    } catch(IllegalArgumentException e) {
        error(e.getMessage());
    }
    return new ProgramNode(location, classes);
}
```

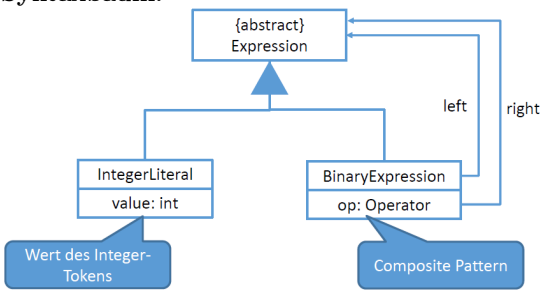
3.4 Parsen mit längerem Lookahead

```
// Statement = Assignment | Invocation.
// Assignment = Identifier -Ëxpression.
// Invocation = Identifier "("")".

// Umwandeln zu:
// Statement = Identifier (AssignmentRest | InvocationRest).
// AssignmentRest = -Ëxpression.
// Invocationrest = "("")".

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

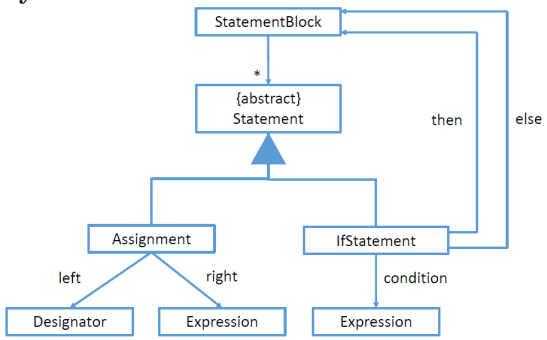
4 Parser Vertiefung

4.1 Syntaxbaum

Syntaxbaum:



Syntaxbaum:



4.2 Designfragen

- Abstrakt vs. konkret
  - Abstract Syntax Tree bei Eigendesign
  - Concrete Syntax Tree bei generiertem Parser
- Weitere Expression-Subklassen
  - UnaryExpression (z.B. für -3 oder +4)
  - Andere Literal-Typen (z.B. boolean, string)
  - Designator (z.B. für x oder y[0].z)
- Source Code Positionen merken
  - Für fehlermeldungen und Debugging
  - Von Lexer-Symbolstrom übernehmen

4.2.1 Term parsen

```
// 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();
        }
        return expression();
    } else {
        error();
    }
}
```

4.3 Syntaxfehler-Behandlung

- Weitermachen bei Fehler → Neuen Einstiegspunkt suchen
- Hypothesen nötig
  - Interpunktionsfehler sind häufig (z.B. fehlendes Semikolon)
  - Vergessener Operator ist selten (z.B. fehlendes Plus)
- Häufige Fehlerarten
  - Fehlendes Symbol wie Semikolon oder Klammer → ignorieren
  - Falsches Symbol wie falscher Klammertyp → ersetzen

Nicht erkannte Fehler: Müssen vom Semantic Checker geprüft werden

- Inkompatible Typen
- Anzahl Argumente ungleich Anzahl Parameter
- Nicht deklarierte Variablen/Methoden
- Ungültige Operanden

5 Semantische Analyse

6 Code Generierung

7 Virtual Machine

8 Objekt Orientierung

9 Typ Polymorphismus

10 Garbage Collection 1

11 Garbage Collection 2

12 JIT Compiler

13 Code Optimierung