CompB Snippets

Compilerbau HS 2014

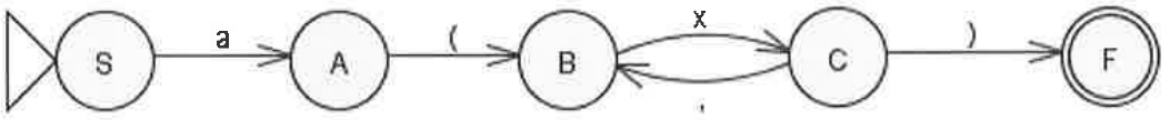
Thomas Charrière, Emanuel Duss, Marcel Loop & Lorenz Wolf

2015-01-19 22:40

Inhaltsverzeichnis

1 Diverse Theorien

1.1 Automatentheorie



Resultierende Grammatik in der Erweiterten Backus-Nauer Form (EBNF [1]):

- S $\rightarrow$ aA
- A $\rightarrow$ (B
- B $\rightarrow$ xC
- C $\rightarrow$)F | ,B
- F $\rightarrow \varepsilon$

[1] ISO/IEC 1497:1996 (E)

1.2 SLR Parsetabelle

2.8 Bottom-Up Parsing – Aufbau der SLR Parsertabelle (15 Punkte)

Gegeben:

Grammatik

0) $S' \rightarrow S$

1) $S \rightarrow A$

2) $A \rightarrow aB$

3) $A \rightarrow a$

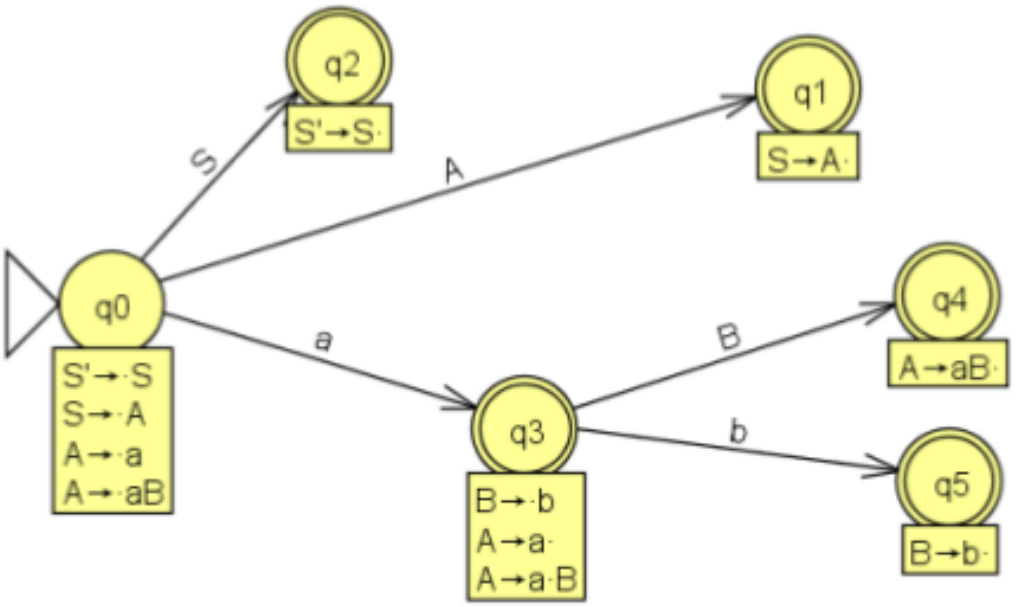
4) $B \rightarrow b$

First und Follow Sets

Symbol	First Set	Follow Set
S	{ a }	{ $ }
A	{ a }	{ $}
B	{ b }	{ $ }

Bestimmen Sie die Parsertabelle zu diesem Automaten: ergänzen Sie die Tabelle.

Zustand	a	b	$	A	B	S
0	s3			1		2
1			r1			
2			acc			
3		s5	r3		4	
4			r2			
5			r4			



2 Nicht LL(1) Grammatik

Folgende Grammatik ist nicht LL(1), da (1) Fist-Set von S und A sind nicht disjunkt (sie haben Gemeinsame Elemente a, b, c) und (2) die Grammatik ist linksrekursiv, da direkt auf S wieder ein S folgen kann. [2]

- S → A | S'A
- A → a|b|c

3 Lexer

3.1 Einfacher Lexer

Scann nach `new`:

```
public TokenInfo getToken() {
    ...
    // ... else
    if (ch == 'n') { // n
        buffer.append(ch);
        nextCh();
        if (ch == 'e') { // e
            buffer.append(ch);
            nextCh(); // hier muss man noch nach einem 'w' pruefen!!!
            if (!isLetter(ch)&&!isDIgit(ch)&&ch!='_'&&ch!='$') {
                return new TokenInfo(TokenKind.NEW, line);
            }
        }
        else if (ch == 'u') { // ca 10 Zeilen analog zu oben: u,l,l
            buffer.append(ch);
            nextCh();
            if (ch == 'l') {
                buffer.append(ch);
                nextCh();
                if (ch == 'l') {
                    buffer.append(ch);
                    nextCh();
                    if (!isLetter(ch) && !isDIgit(ch) &&
                    ch != '_' && ch != '$') {
                        return new TokenInfo(TokenKind.NULL,line);
                    }
                }
            }
        }
        while (isLetter(ch) || isDIgit(ch) || ch == '_' || ch == '$') {
            buffer.append(ch);
            nextCh();
        }
        return new TokenInfo(TokenKind.IDENTIFIER, line);
    }
    // ... else
}
```

Mit speicherung in einer Token Tabelle:

[2] © dieses wunderschönen Satzes by Sonnenbühl 11 WG :)

```
private void loadReserved() {
    reserved.put("abstract", TokenKind.ABSTRACT);
    reserved.put("boolean", TokenKind.BOOLEAN);
    // ...
    reserved.put("while", TokenKind.WHILE);
}
public TokenInfo nextToken() {
    if (isLetter(ch) || ch == '_' || ch == '$') {
        buffer = new StringBuffer();
        while (isLetter(ch) || isDIgit(ch) || ch == '_' || ch == '$') {
            buffer.append(ch);
            nextCh();
        }
        String identifier = buffer.toString();
        // RESERVED
        if (reserved.containsKey(identifier)) {
            return new TokenInfo(reserved.get(identifier), line);
        }
        // IDENTIFIER: eine Zeile Java Code (Hinweise: siehe unten)
        else {
            return new TokenInfo(TokenKind.IDENTIFIER, identifier, line);
        }
    }
    return null;
}
```

3.2 Lexer: `Lexer.java`

```
package lexer;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

import symbolTabelle.Token;

public class Lexer {
        private char ch = ' ';
        private BufferedReader input;
        private String line = "";
        private int lineno = 1;
        private int col = 1;
        private final String letters = "abcdefghijklmnopqrstuvwxyz"
                        + "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        private final String digits = "0123456789";
        private final char eolnCh = '\n';
        private final char eofCh = '\004';

        private final boolean showInput=false;

        public Lexer(String fileName) { // source filename
                try {
                        input = new BufferedReader(new FileReader(fileName));
                } catch (FileNotFoundException e) {
```

```
                System.out.println("Datei nicht gefunden: " + fileName);
                System.exit(1);
        }
}

private char nextChar() {
        // Return: naechstes Zeichen
        if (ch == eofCh)
                error("EOF erreicht");
        col++;
        if (col >= line.length()) {
                try {
                        line = input.readLine();
                } catch (IOException e) {
                        System.err.println(e);
                        System.exit(1);
                } // try
                if (line == null) // EOF
                        line = "" + eofCh;
                else {
                        // Debug Info
                        if (showInput)
                        System.out.println("[Programmzeile] "+ lineno + ":\        " + line)
                        lineno++;
                        line += eolnCh;
                } // if line
                col = 0;
        } // if col
        return line.charAt(col);
}


public Token next() { // Diese Methode wird vom Parser aufgerufen
        do {
                if (isLetter(ch)) { // ident oder keyword
                        String spelling = concat(letters + digits);
                        return Token.keyword(spelling);
                } else if (isDigit(ch)) { // int oder float literal
                        String number = concat(digits);
                        if (ch != '.') // int Literal
                                return Token.mkIntLiteral(number);
                        number += concat(digits);
                        return Token.mkFloatLiteral(number);
                } else
                        switch (ch) {
                        case ' ':
                        case '\        ':
                        case '\r':
                        case eolnCh:
                                ch = nextChar();
                                break;

        // Das war das TODO an der Uebung
                        case '/': // divide oder comment
                                /* Falls das nächste Zeichen noch ein'/' ist
                                 * dann ist die Zeile ein Kommentar
```

```
                 * und kann überlesen werden!
                 */
                ch = nextChar();
                if (ch != '/')
                        return Token.divideTok;
                // comment
                do {
                        ch = nextChar();
                } while (ch != eolnCh);
                ch = nextChar();
                break;

        case '\'': // char literal
                char ch1 = nextChar();
                nextChar(); // get '
                ch = nextChar();
                return Token.mkCharLiteral("" + ch1);

        case eofCh:
                return Token.eofTok;

        case '+':
                ch = nextChar();
                return Token.plusTok;
        case '-':
                ch = nextChar();
                return Token.minusTok;
        case '*':
                ch = nextChar();
                return Token.multiplyTok;
        case '(':
                ch = nextChar();
                return Token.leftParenTok;
        case ')':
                ch = nextChar();
                return Token.rightParenTok;
        case '{':
                ch = nextChar();
                return Token.leftBraceTok;
        case '}':
                ch = nextChar();
                return Token.rightBraceTok;
        case ';':
                ch = nextChar();
                return Token.semicolonTok;
        case ',':
                ch = nextChar();
                return Token.commaTok;

        case '&':
                check('&');
                return Token.andTok;
        case '|':
                check('|');
                return Token.orTok;
```

```
                        case '=':
                                return chkOpt('=', Token.assignTok, Token.eqeqTok);
                        case '<':
                                return chkOpt('=', Token.ltTok, Token.lteqTok);
                        case '>':
                                return chkOpt('=', Token.gtTok, Token.gteqTok);
                        case '!':
                                return chkOpt('=', Token.notTok, Token.noteqTok);

                        default:
                                error("Ungueltiges Zeichen " + ch);
                        } // switch
        } while (true);
} // next

private boolean isLetter(char c) {
        return (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z');
}

private boolean isDigit(char c) {
        return (c >= '0' && c <= '9');
}

private void check(char c) {
        ch = nextChar();
        if (ch != c)
                error("Unueltiges Zeichen! Erwartet: " + c);
        ch = nextChar();
}

private Token chkOpt(char c, Token one, Token two) {
        ch = nextChar();
        if (ch != c)
                return one;
        ch = nextChar();
        return two;
}

private String concat(String set) {
        String r = "";
        do {
                r += ch;
                ch = nextChar();
        } while (set.indexOf(ch) >= 0);
        return r;
}

public void error(String msg) {
        System.err.print(line);
        System.err.println("Fehler: Spalte " + col + " " + msg);
        System.exit(1);
}

static public void main(String[] args) {
        /*
         * Source File Quellcode erst einmal ausgeben!
```

```
                 */
                String fName = "src/programme/hello.cpp";
                try {
                        BufferedReader fRead = new BufferedReader(new FileReader(fName));
                        String zeile = null;
                        while ((zeile = fRead.readLine()) != null) {
                                System.out.println("[scr]" + zeile);
                        }
                        System.out.println();
                        fRead.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                if (args.length == 1) {
                        fName = args[0];
                }
                Lexer lexer = new Lexer(fName);
                Token tok = lexer.next();
                while (tok != Token.eofTok) {
                        System.out.println(tok.toString());
                        tok = lexer.next();
                }
        }
}
```

3.3 Token: `Token.java`

```
package symbolTabelle;

public class Token {
        private static final boolean DEBUG = false;

        /*
         * in der enum TokenType stehen alle reservierten Worte
         * vor dem Symbol für EOF.
         */
        private static final int KEYWORDS = TokenType.Eof.ordinal();

        private static final String[] reserved = new String[KEYWORDS];
        private static Token[] token = new Token[KEYWORDS];

        /*
         * jetzt folgen die einzelnen Token Objekte
         */
        public static final Token eofTok = new Token(TokenType.Eof, "<<EOF>>");
        public static final Token boolTok = new Token(TokenType.Bool, "bool");
        public static final Token charTok = new Token(TokenType.Char, "char");
        public static final Token elseTok = new Token(TokenType.Else, "else");
        public static final Token falseTok = new Token(TokenType.False, "false");
        public static final Token floatTok = new Token(TokenType.Float, "float");
        public static final Token ifTok = new Token(TokenType.If, "if");
        public static final Token intTok = new Token(TokenType.Int, "int");
        public static final Token mainTok = new Token(TokenType.Main, "main");
        public static final Token trueTok = new Token(TokenType.True, "true");
        public static final Token whileTok = new Token(TokenType.While, "while");
```

```
public static final Token leftBraceTok = new Token(TokenType.LeftBrace, "{");
public static final Token rightBraceTok = new Token(TokenType.RightBrace,"}");
public static final Token leftBracketTok = new Token(TokenType.LeftBracket,"[");
public static final Token rightBracketTok = new Token(TokenType.RightBracket, "]");
public static final Token leftParenTok = new Token(TokenType.LeftParen, "(");
public static final Token rightParenTok = new Token(TokenType.RightParen,")");
public static final Token semicolonTok = new Token(TokenType.Semicolon, ";");
public static final Token commaTok = new Token(TokenType.Comma, ",");
public static final Token assignTok = new Token(TokenType.Assign, "=");
public static final Token eqeqTok = new Token(TokenType.Equals, "==");
public static final Token ltTok = new Token(TokenType.Less, "<");
public static final Token lteqTok = new Token(TokenType.LessEqual, "<=");
public static final Token gtTok = new Token(TokenType.Greater, ">");
public static final Token gteqTok = new Token(TokenType.GreaterEqual, ">=");
public static final Token notTok = new Token(TokenType.Not, "!");
public static final Token noteqTok = new Token(TokenType.NotEqual, "!=");
public static final Token plusTok = new Token(TokenType.Plus, "+");
public static final Token minusTok = new Token(TokenType.Minus, "-");
public static final Token multiplyTok = new Token(TokenType.Multiply, "*");
public static final Token divideTok = new Token(TokenType.Divide, "/");
public static final Token andTok = new Token(TokenType.And, "&&");
public static final Token orTok = new Token(TokenType.Or, "||");

private TokenType type;
private String value = "";

private Token(TokenType t, String v) {
        type = t;
        value = v;
        if (t.compareTo(TokenType.Eof) < 0) {
                /* Eintragen der reservierten Wörter
                 * in das Array token[]
                 */
                // Debug Info: reserviertes Wort
                if (DEBUG) System.out.println("IF reserviertes Wort: "+ t.ordinal()+ " ; "+
                int ti = t.ordinal();
                reserved[ti] = v;
                token[ti] = this;
        }
}

public TokenType type() {
        return type;
}

public String value() {
        return value;
}

public static Token keyword(String name) {
        char ch = name.charAt(0);
        // Keywords werden klein geschrieben
        if (ch >= 'A' && ch <= 'Z') {
                // das muss ein Identifier sein
                // Debug Info
                if (DEBUG) System.out.println("[IDENTIFIER] "+ name);
```

```
                return mkIdentTok(name);
        }
        // Kleinbuchstaben: könnte Keyword sein
        for (int i = 0; i < KEYWORDS; i++) {
                if (name.equals(reserved[i])) {
                        // Debug Info
                        if (DEBUG) System.out.println("[KEYWORD] "+ name);
                        return token[i];
                }
        }
        // Kleinbuchstabe aber kein Keyword
        // Debug Info
        if (DEBUG) System.out.println("[IDENTIFIER] "+ name);
        return mkIdentTok(name);
    }

    public static Token mkIdentTok(String name) {
        return new Token(TokenType.Identifier, name);
    }

    public static Token mkIntLiteral(String name) {
        return new Token(TokenType.IntLiteral, name);
    }

    public static Token mkFloatLiteral(String name) {
        return new Token(TokenType.FloatLiteral, name);
    }

    public static Token mkCharLiteral(String name) {
        return new Token(TokenType.CharLiteral, name);
    }

    public String toString() {
        if (type.compareTo(TokenType.Identifier) < 0)
                return value;
        return type + "\        " + value;
    }

    public static void main(String[] args) {
        System.out.println("[Token]Ausgabe von EOF und while Tab und ...");
        System.out.println("Keywords: "+ KEYWORDS);
        System.out.println(eofTok);
        System.out.println(whileTok);
        System.out.println(mkIdentTok("alfa"));
        System.out.println(mkCharLiteral("beta"));
        System.out.println(mkIntLiteral("15"));
        //
        TokenType[] tt = TokenType.values();
        System.out.println("[TokenTypes] " + tt.length);
        for (TokenType t : tt) {
                System.out.println(t);
        }
    }
}
```

3.4 Token Typen: `TokenType.java`

```
package symbolTabelle;

/*
 * Die Reihenfolge ist so, dass jedes reservierte Wort < EOF ist.
 */

public enum TokenType {
        Bool,
        Char,
        Else,
        False,
        Float,
        If,
        Int,
        Main,
        True,
        While,
        Eof,
        LeftBrace, RightBrace,
        LeftBracket, RightBracket, LeftParen, RightParen,
        Semicolon, Comma, Assign, Equals, Less, LessEqual,
        Greater, GreaterEqual, Not, NotEqual, Plus, Minus,
        Multiply, Divide, And, Or, Identifier, IntLiteral,
        FloatLiteral, CharLiteral
}
```

4 Parser

Grundlegende Aufgabe ist die Überprüfung der Syntax eines Programmes mithilfe einer kontextfreien Grammatik.

Jedes Nichtterminalsymbol entspricht einer Funktion im Parser.

4.1 Bottom Up Parser

- s=shift=Zustand angeben
- r=reduce=Grammatik-Regel angeben

4.2 Top Down Parser: `Parser.java`

```
package parser;

import lexer.Lexer;
import symbolTabelle.Token;
import symbolTabelle.TokenType;
// Parser.java
// Recursive descent parser

public class Parser_Lsg {

        Token token; // aktueller Token
```

```
Lexer lexer;
String funcId = "main";

public Parser_Lsg(Lexer ts) {
        lexer = ts;
        token = lexer.next(); // erster Token
}

private String match(TokenType t) {
        String value = token.value();
        if (token.type().equals(t))
                token = lexer.next();
        else
                error(t);
        return value;
}

private void error(TokenType tok) {
        System.err.println("Syntax error: expecting: " + tok + "; saw: "
                        + token);
        System.exit(1);
}

private void error(String tok) {
        System.err.println("Syntax error: expecting: " + tok + "; saw: "
                        + token);
        System.exit(1);
}

public Program program() {
        // Program --> void main ( ) '{' Declarations Statements '}'
        TokenType[] header = { TokenType.Int, TokenType.Main,
                        TokenType.LeftParen, TokenType.RightParen };
        for (int i = 0; i < header.length; i++)
                // bypass "int main ( )"
                match(header[i]);
        match(TokenType.LeftBrace);
        Declarations decpart = declarations();
        Block body = statements();
        match(TokenType.RightBrace);
        return new Program(decpart, body);
}

private Declarations declarations() {
        // Declarations --> { Declaration }
        Declarations ds = new Declarations();
        while (isType()) {
                declaration(ds);
        }
        return ds;
}

private void declaration(Declarations ds) {
        // Declaration --> Type Identifier { , Identifier } ;
        Type t = type();
        while (!token.type().equals(TokenType.Semicolon)) {
```

```
                String id = match(TokenType.Identifier);
                ds.add(new Declaration(id, t));
                if (token.type().equals(TokenType.Comma))
                        match(TokenType.Comma);
        }
        match(TokenType.Semicolon);
}

private Type type() {
        // Type --> int | bool | float | char
        Type t = null;
        if (token.type().equals(TokenType.Int))
                t = Type.INT;
        else if (token.type().equals(TokenType.Bool))
                t = Type.BOOL;
        else if (token.type().equals(TokenType.Float))
                t = Type.FLOAT;
        else if (token.type().equals(TokenType.Char))
                t = Type.CHAR;
        else
                error("int | bool | float | char");
        token = lexer.next(); // pass over the type
        return t;
}

private Statement statement() {
        // Statement --> ; | Block | Assignment | IfStatement | WhileStatement
        Statement s = new Skip();
        if (token.type().equals(TokenType.Semicolon)) // Skip
                token = lexer.next();
        else if (token.type().equals(TokenType.LeftBrace)) { // Block
                token = lexer.next();
                s = statements();
                match(TokenType.RightBrace);
        } else if (token.type().equals(TokenType.If)) // IfStatement
                s = ifStatement();
        else if (token.type().equals(TokenType.While)) // WhileStatement
                s = whileStatement();
        else if (token.type().equals(TokenType.Identifier)) // Assignment
                s = assignment();
        else
                error("Illegal statement");
        return s;
}

private Block statements() {
        // Block --> '{' Statements '}'
        Block b = new Block();
        while (!token.type().equals(TokenType.RightBrace)) {
                b.members.add(statement());
        }
        return b;
}

private Assignment assignment() {
        // Assignment --> Identifier = Expression ;
```

```
        Variable target = new Variable(match(TokenType.Identifier));
        match(TokenType.Assign);
        Expression source = expression();
        match(TokenType.Semicolon);
        return new Assignment(target, source);
}

private Conditional ifStatement() {
        // IfStatement --> if ( Expression ) Statement [ else Statement ]
        match(TokenType.If);
        Expression test = expression();
        Statement thenbranch = statement();
        Statement elsebranch = new Skip();
        if (token.type().equals(TokenType.Else)) {
                token = lexer.next();
                elsebranch = statement();
        }
        return new Conditional(test, thenbranch, elsebranch);
}

private Loop whileStatement() {
        // WhileStatement --> while ( Expression ) Statement
        match(TokenType.While);
        match(TokenType.LeftParen);
        Expression test = expression();
        match(TokenType.RightParen);
        Statement body = statement();
        return new Loop(test, body);
}

private Expression expression() {
        // Expression --> Conjunction { || Conjunction }
        Expression e = conjunction();
        while (token.type().equals(TokenType.Or)) {
                Operator op = new Operator(token.value());
                token = lexer.next();
                Expression term2 = conjunction();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression conjunction() {
        // Conjunction --> Equality { && Equality }
        Expression e = equality();
        while (token.type().equals(TokenType.And)) {
                Operator op = new Operator(token.value());
                token = lexer.next();
                Expression term2 = equality();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression equality() {
        // Equality --> Relation [ EquOp Relation ]
```

```
        Expression e = relation();
        while (isEqualityOp()) {
                Operator op = new Operator(token.value());
                token = lexer.next();
                Expression term2 = relation();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression relation() {
        // Relation --> Addition [RelOp Addition]
        Expression e = addition();
        while (isRelationalOp()) {
                Operator op = new Operator(token.value());
                token = lexer.next();
                Expression term2 = addition();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression addition() {
        // Addition --> Term { AddOp Term }
        Expression e = term();
        while (isAddOp()) {
                Operator op = new Operator(match(token.type()));
                Expression term2 = term();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression term() {
        // Term --> Factor { MultiplyOp Factor }
        Expression e = factor();
        while (isMultiplyOp()) {
                Operator op = new Operator(match(token.type()));
                Expression term2 = factor();
                e = new Binary(op, e, term2);
        }
        return e;
}

private Expression factor() {
        // Factor --> [ UnaryOp ] Primary
        if (isUnaryOp()) {
                Operator op = new Operator(match(token.type()));
                Expression term = primary();
                return new Unary(op, term);
        } else
                return primary();
}

private Expression primary() {
        // Primary --> Identifier | Literal | ( Expression )
```

```
        // | Type ( Expression )
        Expression e = null;
        if (token.type().equals(TokenType.Identifier)) {
                Variable v = new Variable(match(TokenType.Identifier));
                e = v;
        } else if (isLiteral()) {
                e = literal();
        } else if (token.type().equals(TokenType.LeftParen)) {
                token = lexer.next();
                e = expression();
                match(TokenType.RightParen);
        } else if (isType()) {
                Operator op = new Operator(match(token.type()));
                match(TokenType.LeftParen);
                Expression term = expression();
                match(TokenType.RightParen);
                e = new Unary(op, term);
        } else
                error("Identifier | Literal | ( | Type");
        return e;
}

@SuppressWarnings("incomplete-switch")
private Value literal() {
        String s = null;
        switch (token.type()) {
        case IntLiteral:
                s = match(TokenType.IntLiteral);
                return new IntValue(Integer.parseInt(s));
        case CharLiteral:
                s = match(TokenType.CharLiteral);
                return new CharValue(s.charAt(0));
        case True:
                s = match(TokenType.True);
                return new BoolValue(true);
        case False:
                s = match(TokenType.False);
                return new BoolValue(false);
        case FloatLiteral:
                s = match(TokenType.FloatLiteral);
                return new FloatValue(Float.parseFloat(s));
        }
        throw new IllegalArgumentException("should not reach here");
}

private boolean isAddOp() {
        return token.type().equals(TokenType.Plus)
                        || token.type().equals(TokenType.Minus);
}

private boolean isMultiplyOp() {
        return token.type().equals(TokenType.Multiply)
                        || token.type().equals(TokenType.Divide);
}

private boolean isUnaryOp() {
```

```
                return token.type().equals(TokenType.Not)
                                || token.type().equals(TokenType.Minus);
        }

        private boolean isEqualityOp() {
                return token.type().equals(TokenType.Equals)
                                || token.type().equals(TokenType.NotEqual);
        }

        private boolean isRelationalOp() {
                return token.type().equals(TokenType.Less)
                                || token.type().equals(TokenType.LessEqual)
                                || token.type().equals(TokenType.Greater)
                                || token.type().equals(TokenType.GreaterEqual);
        }

        private boolean isType() {
                return token.type().equals(TokenType.Int)
                                || token.type().equals(TokenType.Bool)
                                || token.type().equals(TokenType.Float)
                                || token.type().equals(TokenType.Char);
        }

        private boolean isLiteral() {
                return token.type().equals(TokenType.IntLiteral) || isBooleanLiteral()
                                || token.type().equals(TokenType.FloatLiteral)
                                || token.type().equals(TokenType.CharLiteral);
        }

        private boolean isBooleanLiteral() {
                return token.type().equals(TokenType.True)
                                || token.type().equals(TokenType.False);
        }

        public static void main(String args[]) {
                String fName = "src/programme/hello.cpp";
                System.out.println("Begin parsing... " + fName);
                Parser_Lsg parser = new Parser_Lsg(new Lexer(fName));
                Program prog = parser.program();
                prog.display(); // display abstract syntax tree
        } // main

} // Parser

/*
Begin parsing... src/programme/hello.cpp

Program (abstract syntax):
  parser.Declarations:
    Declarations = {<c, char>, <i, int>}
  parser.Block:
    parser.Assignment:
      parser.Variable: c
      parser.CharValue: h
    parser.Assignment:
      parser.Variable: i
```

```
      parser.Binary:
        parser.Operator: +
        parser.Variable: c
        parser.IntValue: 3
*/
```

4.3 Parser: Rekursiver Abstieg

- A → xCB
- B → z
- B → Ax
- C → yBz
- C → ε

```
xz (gültig)
xyzzzzzzzzzz(gültig)
xxxzxx (gültig)
xxxxxyyyzzzz (error)
```

```
public class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void ntA() throws IOException {
        match('x');
        ntC();
        ntB();
    }

    void ntB() throws IOException {
        if(lookahead == 'z') {
            match('z');
        } else {
            ntA();
            match('x');
        }

    }

    void ntC() throws IOException {
        if(lookahead == 'y') {
            match('y');
            ntB();
            match('z');
        }
    }

    void match(int t) throws IOException {
        if(lookahead == t) {
            lookahead = System.in.read();
        } else
            throw new Error("Syntax Fehler");
```

```
    }

    public static void main(String[] args) throws IOException {
        Parser parser = new Parser();
        parser.ntA();
        System.out.write('\n');
    }
}
```

5 Abstract Syntax

5.1 Visitor

5.1.1 Exp.java

```
public abstract class Exp {
    public abstract int accept(Visitor v);
}
```

5.1.2 MinusExp.java

```
public class MinusExp extends Exp {
   public Exp e1, e2;
   public MinusExp(Exp a1, Exp a2) {
       e1 = a1;
       e2 = a2;
   }
   public int accept(Visitor v) {
      return v.visit(this);
   }
}
```

5.1.3 PlusExp.java

```
public class PlusExp extends Exp {
     public Exp e1, e2;
     public PlusExp(Exp a1, Exp a2) {
         e1 = a1;
         e2 = a2;
     }
     public int accept(Visitor v) {
        return v.visit(this);
     }
}
```

5.1.4 IntegerLiteral.java

```
public class IntegerLiteral extends Exp {
     public String f0;
     public IntegerLiteral(String lit) {
         f0 = lit;
     }
```

```
    public int accept(Visitor v) {
        return v.visit(this);
    }
}
```

5.1.5 Interpreter.java

```
public class Interpreter implements Visitor  {

    public static void main(String []args){
       Interpreter inter = new Interpreter();

       PlusExp plusExp =
       new PlusExp(
           new MinusExp(
               new IntegerLiteral("100"),
               new IntegerLiteral("94")
           ),
           new IntegerLiteral("10")
       );

       int result = inter.visit(plusExp);

       System.out.println("Result: " + result);
    }



    public int visit(PlusExp n) {
        return n.e1.accept(this) + n.e2.accept(this);
    }
    public int visit(MinusExp n) {
        return n.e1.accept(this) - n.e2.accept(this);
    }
    public int visit(IntegerLiteral n) {
        return Integer.parseInt(n.f0);
    }
}
```

5.1.6 Visitor.java

```
public interface Visitor {
   public int visit(PlusExp n);
   public int visit(MinusExp n);
   public int visit(IntegerLiteral n);
}
```