Interpreterbau

Tokenizer, Parser, AST, EBNF, ...

Max - max@noppelmax.online

May 18, 2018

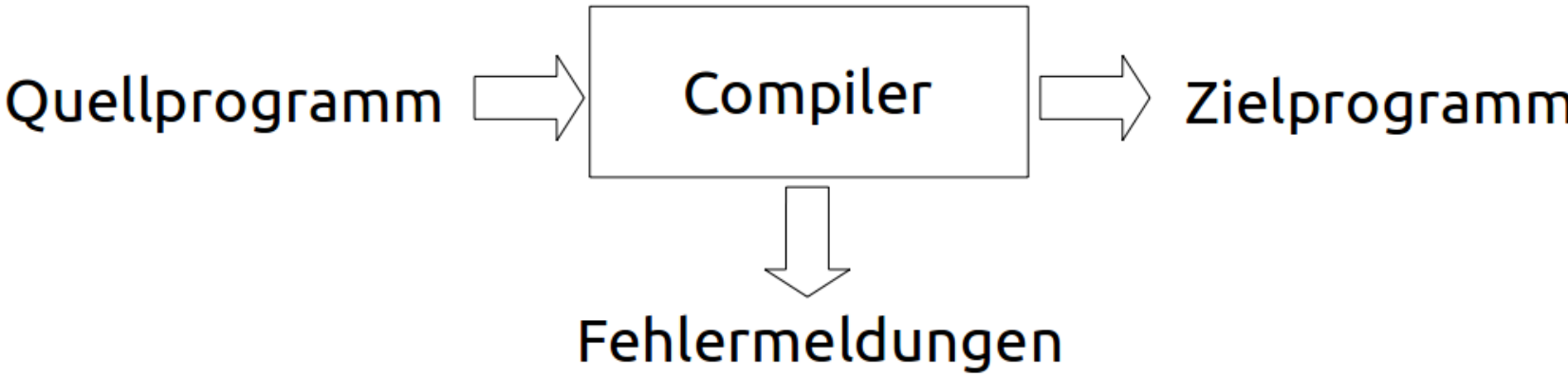
Übersicht

Was ist ein Compiler?

Compiler

Im Grunde ist ein Compiler ein Programm, das ein in einer bestimmten Sprache - der *Quell*sprache - geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache - der *Ziel*sprache - übesetzt.[1]

[1] *Compilerbau*, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1988



Was ist dann ein Interpreter?

Interpreter

Anstatt ein Zielprogramm als Übersetzung zu erzeugen, führt ein Interpreter die im Quellprogramm enthaltenen Operationen direkt aus. Ein Interpreter könnte z.B. für eine Zuweisung einen Baum [...] aufbauen und anschließend bei durchwandern die Operationen in den Knoten ausführen.[2]s

[2] *Compilerbau*, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1988

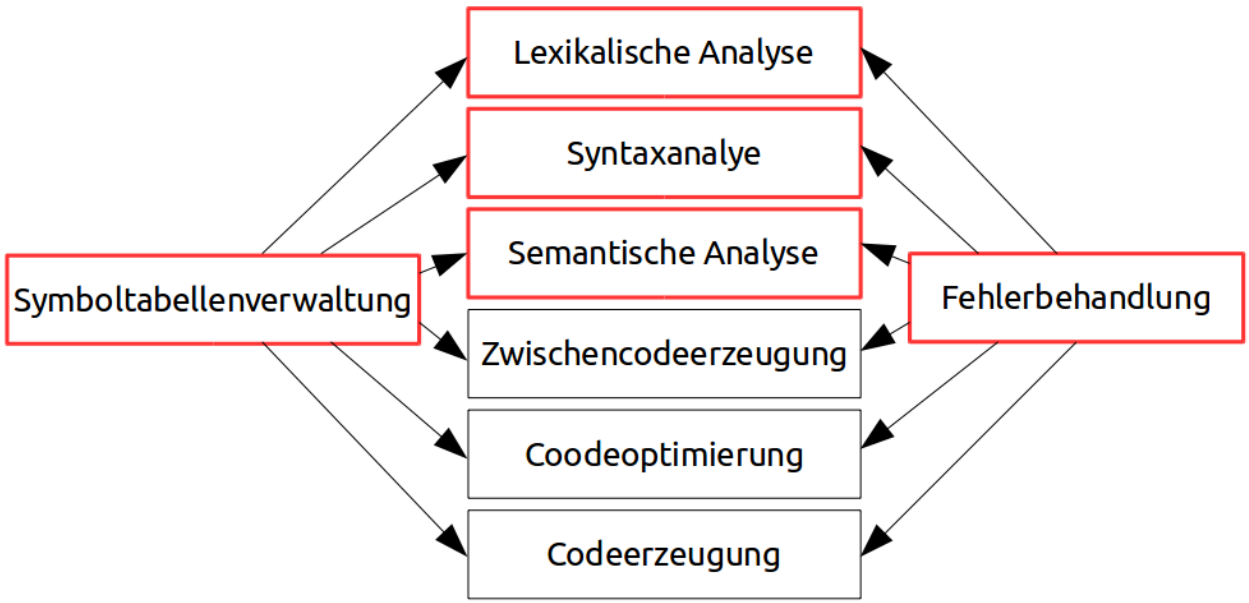
Eine ganze einfache Sprache

Addition und Subtraktion von zwei Zahlen.

Beispiel: `number + number - number`

Was müssen wir uns dazu ansehen?

Phasen



Übersicht

Kontextfreie Grammatiken

Beispiel aus C: **if** (Ausdruck) Anweisung **else** Anweisung

Kontextfreie Grammatiken

Beispiel aus C: **if** (Ausdruck) Anweisung **else** Anweisung

Das kann mit folgender `Produktion` beschrieben werden:

$stmt \rightarrow$ **if** ($expr$) $stmt$ **else** $stmt$

Kontextfreie Grammatiken

Beispiel aus C: **if** (Ausdruck) Anweisung **else** Anweisung

Das kann mit folgender `Produktion` beschrieben werden:

stmt $\rightarrow$ **if** (*expr*) *stmt* **else** *stmt*

if und () heißen *Terminale* oder *Symbole*

Variablen wie *expr* und *stmt* heißen *Nichtterminale*

Ein *Nichtterminal* wird als *Startsymbol* definiert

Ein String der keine *Terminale* enthält wird als *leerer String* bezeichnet und als ϵ geschrieben.

Kontextfreie Grammatiken

Alle Wörter die aus dem Startsymbol herleitbar sind (durch anwenden der Produktionen) bilden zusammen die von der Grammatik definierte *Sprache*.

Kontextfreie Grammatiken

Alle Wörter die aus dem Startsymbol herleitbar sind (durch anwenden der Produktionen) bilden zusammen die von der Grammatik definierte *Sprache*.

Schauen wir uns das die einfache Sprache von vorhin an:

Beispiel: `number + number - number`

Produktionen:

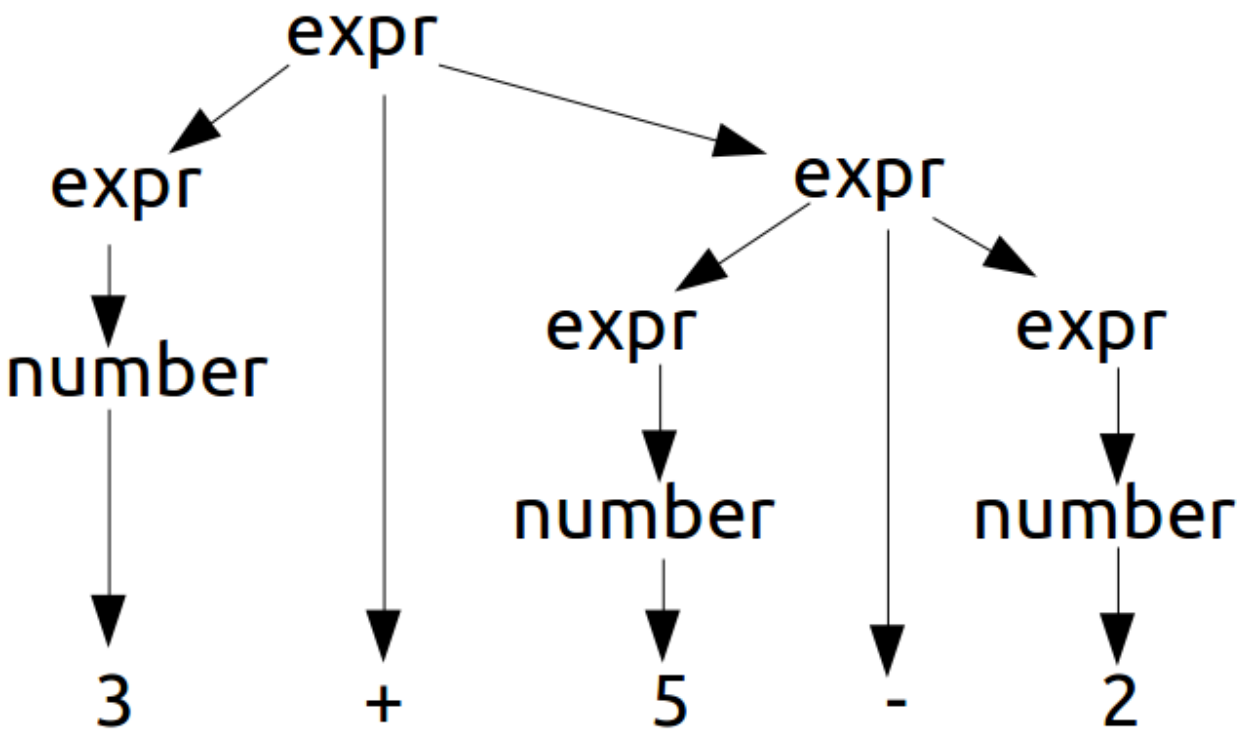
$$expr \rightarrow expr + expr \mid expr \text{ - } expr \mid number$$

$$number \rightarrow 0 \mid 1 \mid \ldots 9$$

Parsebaum

$$expr \rightarrow expr + expr \mid expr \text{ - } expr \mid number$$

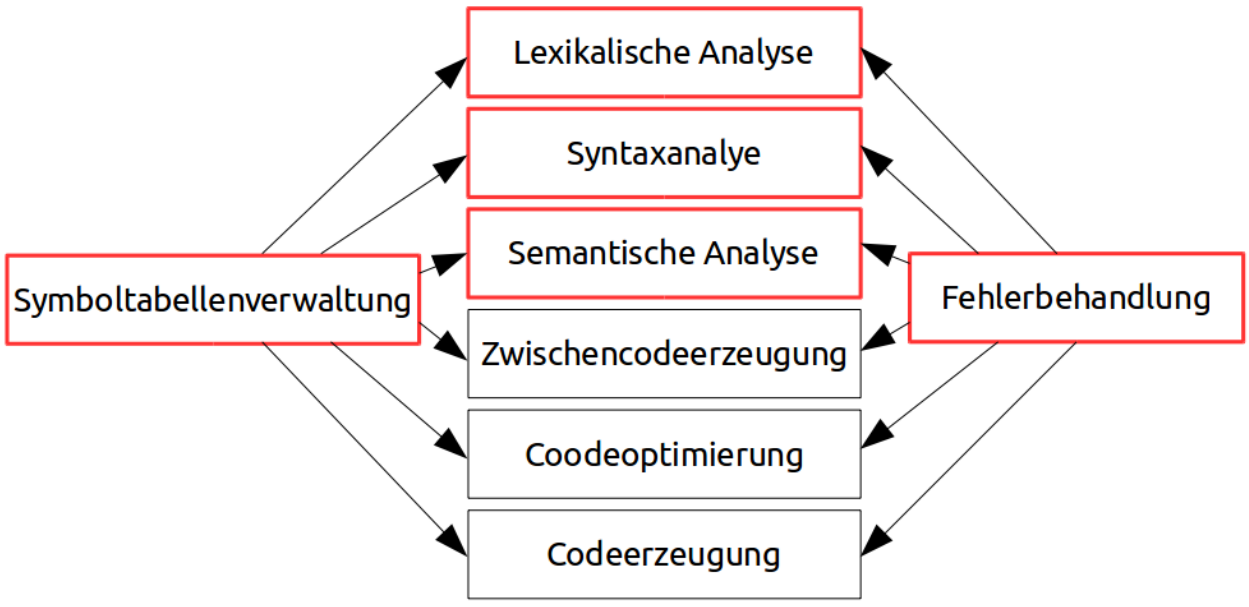
$$number \rightarrow 0 \mid 1 \mid \ldots 9$$



Phasen

Das wollen wir jetzt mal implementieren. Wie fangen wir an?

Phasen



Übersicht

Die Lexikalische Analyse

ist die erste Phase der Interpretation.

Der Eingabestring wird in Tokens eingeteilt

Whitespace und Kommentare werden dabei in der Regel ignoriert

Die Tokens entsprechen den *Terminalen*, können aber schon Werte enthalten. Wie beispielsweise bei Zahlen.

Beispiele

Beispiele:

`if` → T_IF

`!=` → T_NEQ

`(` → T_LPAREN

`foo` → ID('foo')

3 → INT(3)

Tokenizer/Scanner/Lexer

Aus `9 + 2 * ( 3 + 3 )` wird ein Stream aus Tokens:

`<9> <+> <2> <*> <(> <3> + <3> <)>`

Tokenizer/Scanner/Lexer

Aus `9 + 2 * ( 3 + 3 )` wird ein Stream aus Tokens:

`<9> <+> <2> <*> <(> <3> + <3> <)>`

In diesem einfachen Fall können wir das gewählte Token von einem Zeichen abhängig machen.

NEA/DEA[3]

Die konkreten Eingaben werden mit einem endlichen Automaten erkannt. Jeder akzeptierende Zustand stellt das jeweils erkannte Token dar.

[3](Nicht-)Deterministischer Endlicher Automat

NEA/DEA[3]

Die konkreten Eingaben werden mit einem endlichen Automaten erkannt. Jeder akzeptierende Zustand stellt das jeweils erkannte Token dar.

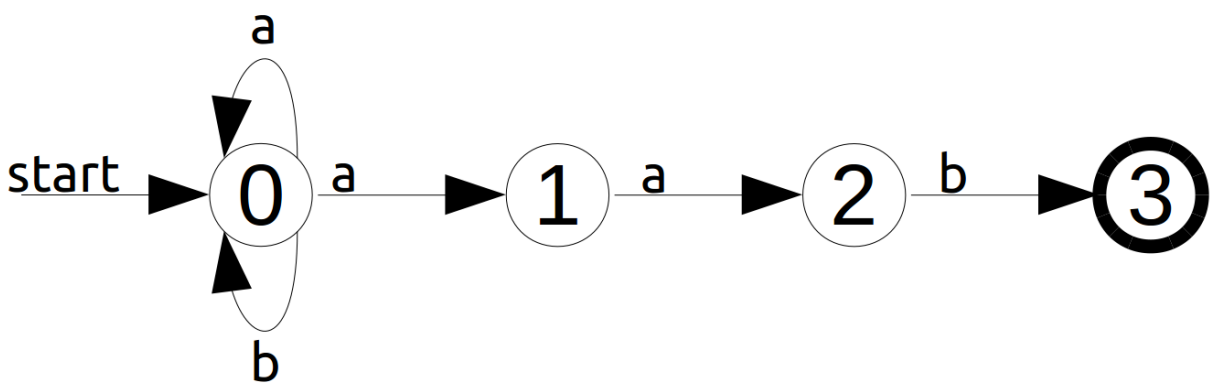
Wollen wir also beispielsweise die Sprache `(a|b)*abb` erkennen wollen.

[3](Nicht-)Deterministischer Endlicher Automat

NEA/DEA[3]

Die konkreten Eingaben werden mit einem endlichen Automaten erkannt. Jeder akzeptierende Zustand stellt das jeweils erkannte Token dar.

Wollen wir also beispielsweise die Sprache `(a|b)*abb` erkennen wollen.



[3](Nicht-)Deterministischer Endlicher Automat

Mehrdeutigkeiten

Es gibt im Allgemeinen mehrere Möglichkeiten eine Folgen in Tokens zu zerlegen.

Längest Übereinstimmung: Um dieses Problem zu umgehen wählt man die Regel mit der längsten Übereinstimmung. Das längste Präfix das zu einem der regulären Ausdrücke passt.

Regelprioriät: Wenn das nicht hilft gilt muss für die Regeln eine Priorität vergeben werden.

Lexergeneratoren

Es gibt Lexergeneratoren, die direkt aus eine Spezifikationsdatei einen Lexer/Tokenizer/Scanner generieren.

Das war eine ganze Zeit lang super hip. Inzwischen geht der Trend wieder zum selberschreiben.

Wenn euch das interessiert schaut euch: lex, flex, JLex, JFlex oder Ocamllex an.

Übersicht

Syntaxanalyse

Die hierachische Analyse heißt *Syntaxanalyse* oder engl. *parsing*. Ihre Aufgabe besteht darin, die Symbole des Quellprogramms zu grammatikalischen Sätzen zusammenzufassen, [...].[4]

[4] *Compilerbau*, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, 1988

Syntaxanalyse

Zweite Phase

Erkennen einer Produktionsregel der kontextfreien Grammatik in dem Tokenstrom.

Überprüfen ob die Eingabe synthaktisch Korrekt ist.

Umwandeln in Abstract Syntax Tree

Dieser Part des Interpreters wird Parser genannt.

Syntaxanalyzer/Parser

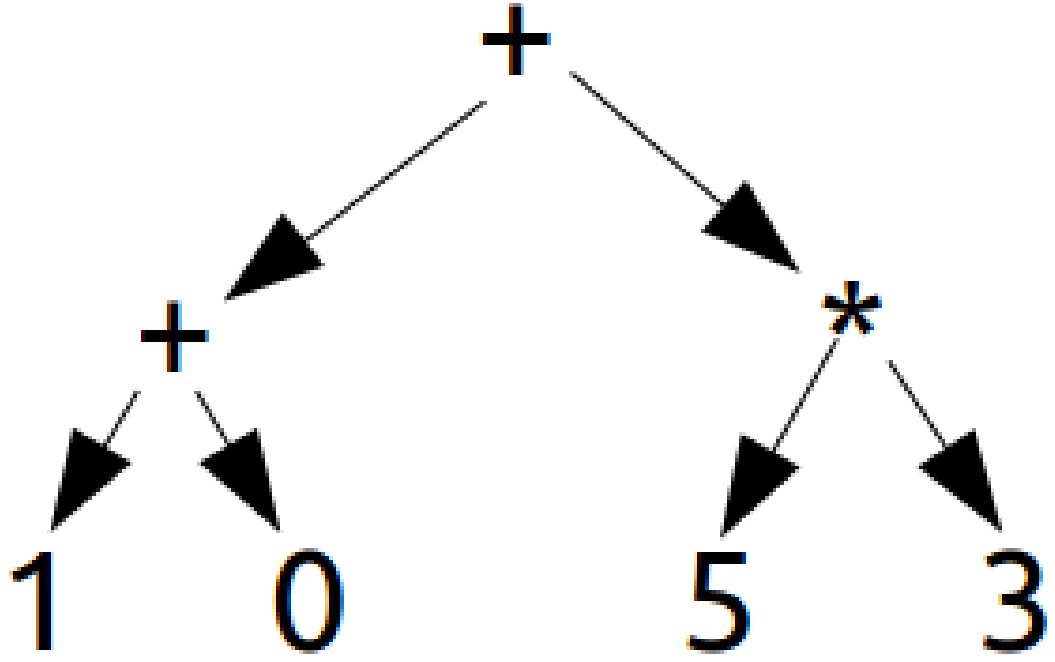
Möglichst linear in der Eingabegrößen. Dazu schränkt man sich auf spezielle Grammatiken ein (LL(1), LR(1) oder LALR(1)) für die spezielle Algorithmen verfügbar sind.

Auch für Parser gibt es Generatoren: yacc, bison, ML-yacc, JavaCUP, JavaCC, ANTLR

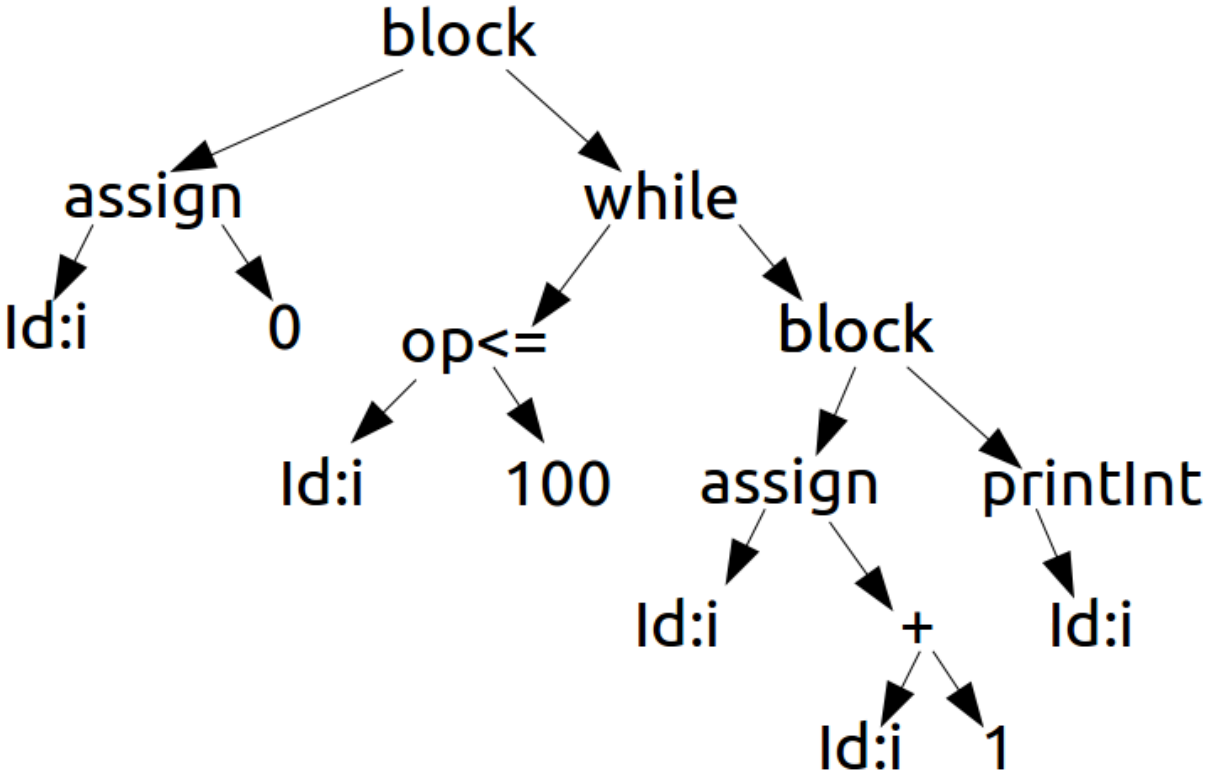
Abstract Syntax Tree

Der AST ist das Ergebnis der Syntaxanalyse. Der AST enthält Knoten für alle `Nichtterminale` der Grammatik. Die Knoten sind mit anderen Knoten baumartig angeordnet.

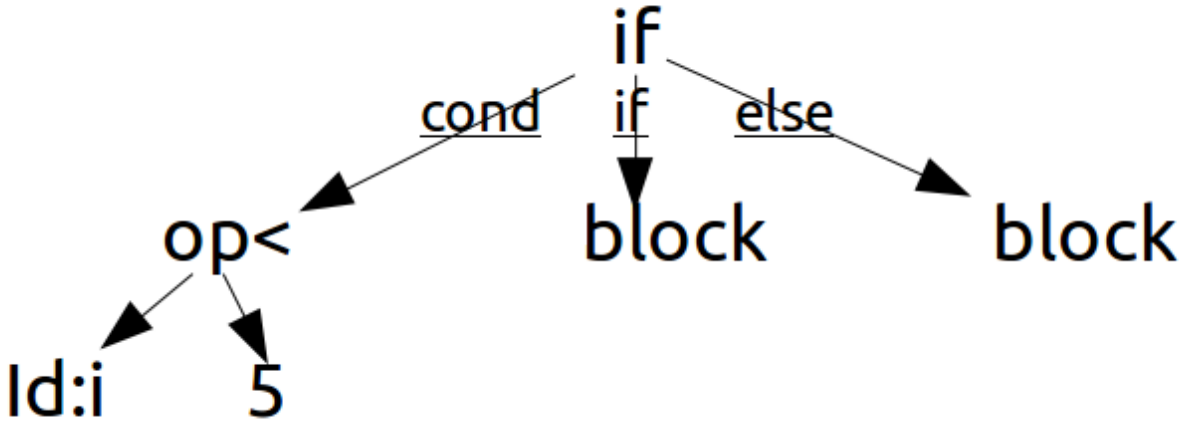
Abstract Syntax Tree



Abstract Syntax Tree



Abstract Syntax Tree



Übersicht

Main


```
Tokenizer* tokenizer = new Tokenizer(execString);
Parser* parser = new Parser(tokenizer);

Node* startnode = parser->parse();

int result = startnode->eval();

std::cout << result << std::endl;
```

Tokenizer


```
Token* Tokenizer::getNextToken() {
    Token* t;

    if (currentCharacter == '+') {
        readNextChar();
        return new PlusOpToken();

    } else if (currentCharacter == '-') {
        readNextChar();
        return new MinusOpToken();

    } else if (currentCharacter == '*') {
        readNextChar();
        return new MulOpToken();

    } else if (currentCharacter == '/') {
        readNextChar();
        return new DivOpToken();

    } else if (currentCharacter == '(') {
        readNextChar();
        return new LParenToken();
```

Parser::parseExpression


```
Node* Parser::parseExpression() {
    Node* node = parseTerm();

    if (currentToken == 0x0) {
        return node;
    }

    while (currentToken->getTT() == T_PLUS || currentToken->getTT() == T_MINUS) {
        TokenType lastTT = currentToken->getTT();
        loadNextToken();
        Node* newNode = parseTerm();
        if (lastTT == T_PLUS) {
            node = new ExpressionNode(node, newNode, ExpressionOperator::PLUS);
        } else if (lastTT == T_MINUS) {
            node = new ExpressionNode(node, newNode, ExpressionOperator::MINUS);
        }

        if( currentToken == 0x0 ){
            return node;
        }
    }

    return node;
}
```

Parser::parseFactor


```
Node* Parser::parseFactor() {

    Node* node;
    if (currentToken->getTT() == T_LPAREN) {
        accept(T_LPAREN);
        node = parseExpression();
        accept(T_RPAREN);
    } else if (currentToken->getTT() == T_NUMBER) {
        int value = ((NumberToken*) currentToken)->getValue();
        loadNextToken();
        node = new FactorNode(value);
    }

    return node;

}
```

FactorNode::eval


```
int FactorNode::eval() {
    return value;
}
```

ExpressionNode::eval

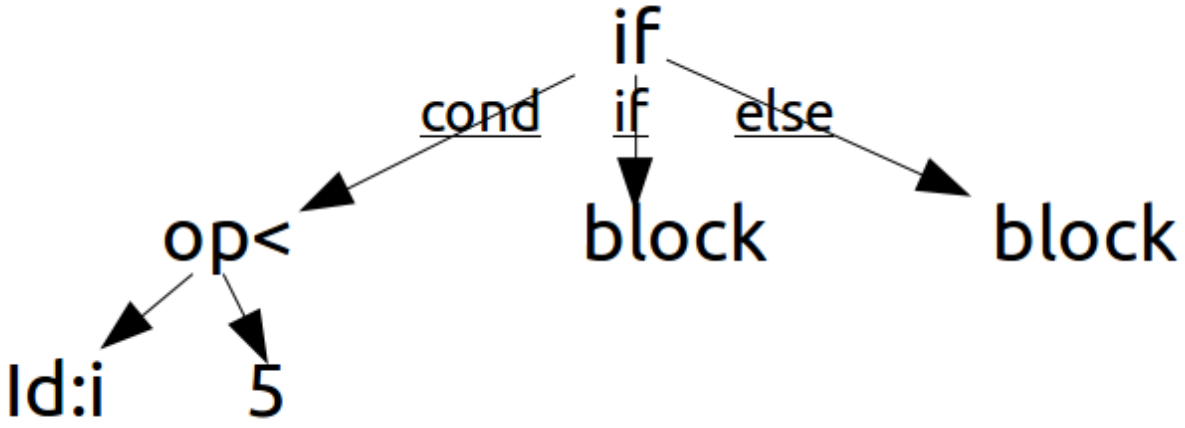

```
int ExpressionNode::eval() {

    if (op == ExpressionOperator::PLUS) {
        return left->eval() + right->eval();

    } else if (op == ExpressionOperator::MINUS) {
        return left->eval() - right->eval();

    } else {
        return 0;
    }
}
```

Übersicht

If-Bedingung

Natürlich kann man mit der EBNF auch eine IF-Bedingung abbilden.

If-Bedingung



While-Schleife

The body of the frame.

Variablen

The body of the frame.

Context

The body of the frame.

Übersicht

Fehlermeldungen - Was wollen?

Zeilenummer und Token bei dem ein Fehler aufgetreten ist

Möglichst genau welcher Fehler es ist

Wie kann der Fehler behoben werden

Nicht nur einen Fehler sondern möglichst viele aufeinmal anzeigen

Fehlermeldungen - Wie kommen wir dahin?

Zeilen und Spaltennummer im Tokens mitspeichern

Differenziert Fehler erkennen

Bei einem aufgetretenen Fehler eine Art Fehlerkorrektur machen und versuchen, weiter zuparsen

Danke euch. Das wars!

Übersicht

Weitergehende Fragestellungen

Codeerzeugung

Programmanalyse

Software-Sicherheitsanalyse

Codeoptimierung

Laufzeitmechanismen

Attributgrammatiken

LL(k), LR(k), LALR(k), LAG(k) Grammatiken

Mehr-Pass-Compiler

Präprozessoren

Fehlerquellen

Division durch Null

Nullpointer

Speichermanagement im Fehlerfall

...

Nicht-kontextfreie Grammatiken

Es gibt natürlich auch Sprachen die nicht durch kontextfreie Grammatiken definiert werden können und regelmäßig in Programmiersprachen verwendet werden. Einige Beispiele[5]:

$$L_1 = \{\omega\ c\ \omega\ |\omega \in (a|b)*\}$$

$$L_2 = \{a^n b^m c^n d^m | n \geq 1 \wedge m \geq 1\}$$

$$L_3 = \{a^n b^n c^n | n \geq 0\}$$

[5] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[6]:

$$L_1' = \{\omega\ c\ \omega^R | \omega \in (a|b)*\}$$

[6] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[6]:

$$L_1' = \{\omega\ c\ \omega^R | \omega \in (a|b)*\}$$

$S \rightarrow \mathrm{a}S\mathrm{a} \mid \mathrm{b}S\mathrm{b} \mid \mathrm{c}$

[6] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[7]:

$$L_2' = \{a^n b^m c^m d^n | n \geq 1 \wedge m \geq 1\}$$

[7] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[7]:

$$L_2' = \{a^n b^m c^m d^n | n \geq 1 \wedge m \geq 1\}$$

$S \rightarrow \mathrm{a}S\mathrm{d} \mid \mathrm{a}A\mathrm{d}$

$A \rightarrow \mathrm{b}A\mathrm{c} \mid \mathrm{bc}$

[7] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[8]:

$$L_2'' = \{a^n b^n c^m d^m | n \geq 1 \wedge m \geq 1\}$$

[8] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[8]:

$$L_2'' = \{a^n b^n c^m d^m | n \geq 1 \wedge m \geq 1\}$$

$S \rightarrow AB$

$A \rightarrow \mathrm{a}A\mathrm{b} \mid \mathrm{ab}$

$B \rightarrow \mathrm{c}B\mathrm{c} \mid \mathrm{cd}$

[8] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[9]:

$$L_3' = \{a^n b^n | n \geq 1\}$$

[9] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[9]:

$$L_3' = \{a^n b^n | n \geq 1\}$$

$S \rightarrow \mathrm{a}S\mathrm{b} \mid \mathrm{ab}$

[9] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Kontextfreie Grammatiken

Sehr ähnliche Sprachen sind aber kontextfrei[9]:

$$L'_3 = \{a^n b^n | n \geq 1\}$$

$S \rightarrow \mathrm{a}S\mathrm{b} \mid \mathrm{ab}$

Bemerkung: L'_3 kann nicht durch einen *Regulären Ausdruck* definiert werden.

[9] *Principles of Compiler Design*, Alfred V. Aho, Jeffrey D. Ullman, 1977

Weitere TechEvent Themen

Genetische Programmierung

Compilerbau- Theorie

Algorithmen für Zellularautomaten

Randomisierte Algorithmen

Datenschutz - von Anonymisierung bis Zugriffskontrolle

Binary Exploitation

Web Exploitation