

Advanced Programming SS 2020: LLVM



<https://xkcd.com/303/>

```
# Compute Ackermann's function
# for two parameters a and b.
def ack(a, b)
    if a == 0 then
        b + 1
    else if b == 0 then
        ack(a - 1, 1)
    else
        ack(a - 1, ack(a, b - 1))

def run(a, b)
    ack(a, b)
```

- Parser

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator

```
// Get the precedence of the given operator.  
private static int GetOperatorPrecedence(Operator op)  
{  
    switch (op)  
    {  
        case Operator.Add: return 80;  
        case Operator.Subtract: return 80;  
        case Operator.Multiply: return 100;  
        case Operator.Divide: return 100;  
        case Operator.Equal: return 60;  
        case Operator.LowerThan: return 60;  
        case Operator.LowerThanEqual: return 60;  
        case Operator.GreaterThan: return 60;  
        case Operator.GreaterThanEqual: return 60;  
  
        default: throw new InvalidOperationException("Invalid operator type.");  
    }  
}
```

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == 4 + 5 * 6 - 7$$

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == ((4 + (5 * 6)) - 7)$$

Multiplikation

Addition

Subtraktion

Vergleich

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == (4 + ((5 * 6) - 7))$$

Multiplikation

Subtraktion

Addition

Vergleich

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Exkurs: Wie wird bei Gleichstand verfahren?
 - Linksassoziativ: $a \text{ op } b \text{ op } c \rightarrow (a \text{ op } b) \text{ op } c$
 - Rechtsassoziativ: $a \text{ op } b \text{ op } c \rightarrow a \text{ op } (b \text{ op } c)$
 - Beispiele?

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Exkurs: Wie wird bei Gleichstand verfahren?
 - Linksassoziativ: $a \text{ op } b \text{ op } c \rightarrow (a \text{ op } b) \text{ op } c$
 - Rechtsassoziativ: $a \text{ op } b \text{ op } c \rightarrow a \text{ op } (b \text{ op } c)$
 - Beispiele? Subtraktion (links), Potenzierung (rechts)
 - Wir parsen linksassoziativ!

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == ((4 + (5 * 6)) - 7)$$

Multiplikation


Addition

Subtraktion

Vergleich

- **Parser**

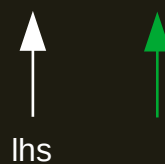
- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == 4 + 5 * 6 - 7$$


lhs

- **Parser**

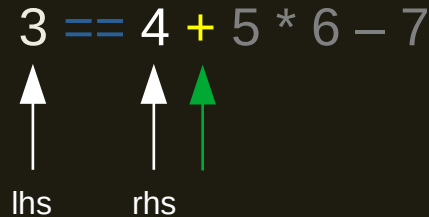
- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

$$3 == 4 + 5 * 6 - 7$$


lhs

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:

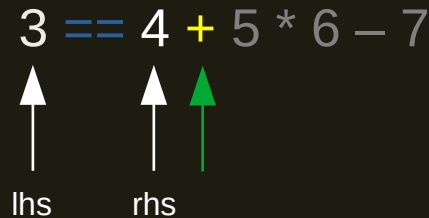


Fallunterscheidung:

Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:



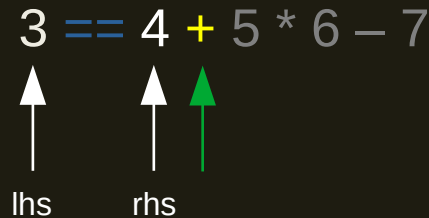
Fallunterscheidung:

Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

Wenn nein: Setze $lhs := \text{BinOp}(lhs, rhs, op)$, Rekursion

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:



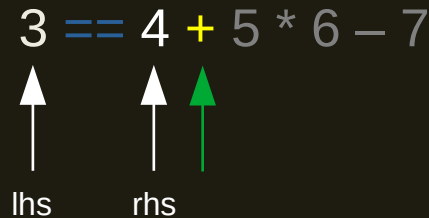
Fallunterscheidung:

Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

Wenn nein: Setze $lhs := \text{BinOp}(lhs, rhs, op)$,
Rekursion \rightarrow Tail-Rekursion in Schleife übersetzen

- **Parser**

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:



Fallunterscheidung:

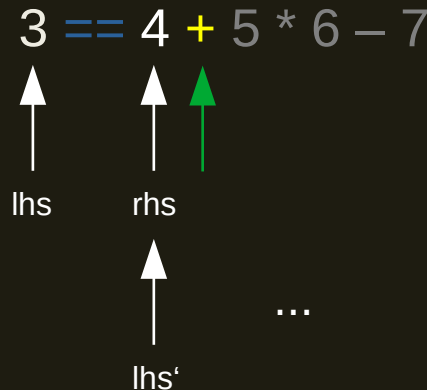
Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

Wenn nein: Setze $lhs := \text{BinOp}(lhs, rhs, op)$,
Rekursion \rightarrow Tail-Rekursion in Schleife übersetzen

Wenn ja: Rekursion mit *rhs* als *lhs*

- Parser

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:



Fallunterscheidung:

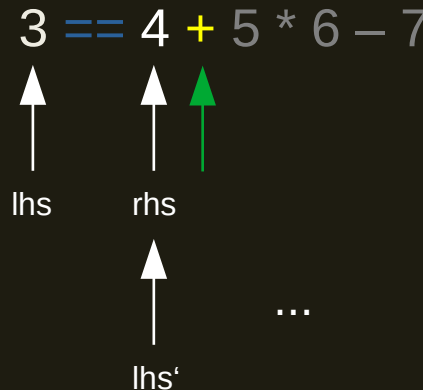
Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

Wenn nein: Setze $lhs := \text{BinOp}(lhs, rhs, op)$,
Rekursion \rightarrow Tail-Rekursion in Schleife übersetzen

Wenn ja: Rekursion mit *rhs* als *lhs*

- Parser

- Binäre Operationen: Operatorrangfolge (*Precedence*) muss beachtet werden!
- Spezifikation über *Precedence Table*: Je höher der Wert, desto stärker bindet der Operator
- Beispiel:



Fallunterscheidung:

Bindet der Look-Ahead-Operator stärker als der aktuelle Operator?

Wenn nein: Setze $lhs := \text{BinOp}(lhs, rhs, op)$,
Rekursion \rightarrow Tail-Rekursion in Schleife übersetzen

Wenn ja: Rekursion mit rhs als lhs
Problem, wenn Folgeoperator schwächer als der ursprüngliche Operator (hier $==$) \rightarrow Rekursion abbrechen

- **Parser**
 - Traversieren des AST, Operationen auf Elementen ausführen
 - Beispiele: `print()`, `generateCode()`, ...
 - AST-Hierarchie basiert auf Interfaces → dynamische Bindung / Polymorphie

- **Parser**

- Traversieren des AST, Operationen auf Elementen ausführen
- Beispiele: `print()`, `generateCode()`, ...
- AST-Hierarchie basiert auf Interfaces → dynamische Bindung / Polymorphie
- Problem: Heterogene Operationen überfrachten die AST-Klassen
- Verhalten sollte in separate Klassen ausgelagert werden: `class Printer`, `class CodeGenerator`, ...

- **Parser**

- Traversieren des AST, Operationen auf Elementen ausführen
- Beispiele: `print()`, `generateCode()`, ...
- AST-Hierarchie basiert auf Interfaces → dynamische Bindung / Polymorphie
- Problem: Heterogene Operationen überfrachten die AST-Klassen
- Verhalten sollte in separate Klassen ausgelagert werden: `class Printer`, `class CodeGenerator`, ...
- Überladung funktioniert hier nicht: Kein *Double Dispatch*!
- In C# existiert zu diesem Zweck das `dynamic`-Schlüsselwort.
- Hier stattdessen (da generell anwendbar): *Visitor-Pattern*

Compilerbau

- **Visitor-Pattern**

- *Double Dispatch* in einer *Single-Dispatch*-Sprache implementieren
- `Visitor`-Interface: Eine Methode pro Modell-Klasse



- Visitor-Pattern

- *Double Dispatch* in einer *Single-Dispatch*-Sprache implementieren
- Visitor-Interface: Eine Methode pro Modell-Klasse

```
public interface Visitor
{
    void VisitLiteralExpression(LiteralExpression expr);
    void VisitParameterExpression(ParameterExpression expr);
    void VisitBinaryOperatorExpression(BinaryOperatorExpression expr);
    void VisitCallExpression(CallExpression expr);
    void VisitConditionalExpression(ConditionalExpression expr);
    void VisitFunctionPrototype(FunctionPrototype prototype);
    void VisitFunctionDeclaration(FunctionDeclaration declaration);
    void VisitFunctionDefinition(FunctionDefinition definition);
}
```



- **Visitor-Pattern**

- *Double Dispatch* in einer *Single-Dispatch*-Sprache implementieren
- `Visitor`-Interface: Eine Methode pro Modell-Klasse
- Modell-Interface (hier: `Element`): Eine Methode (meist `accept()` genannt), die einen `Visitor` übernimmt und sich selbst an ihn übergibt.



- **Visitor-Pattern**

- *Double Dispatch* in einer *Single-Dispatch*-Sprache implementieren
- Visitor-Interface: Eine Methode pro Modell-Klasse
- Modell-Interface (hier: `Element`): Eine Methode (meist `accept()` genannt), die einen `Visitor` übernimmt und sich selbst an ihn übergibt.

```
public interface Element
{
    void Accept(Visitor visitor);
}
```

```
public class LiteralExpression: Expression
{
    // The value of the literal
    public double Value { get; private set; }

    internal LiteralExpression(double value)
    {
        this.Value = value;
    }

    public void Accept(Visitor visitor)
    {
        visitor.VisitLiteralExpression(this);
    }
}
```

- **Visitor-Pattern**

- *Double Dispatch* in einer *Single-Dispatch*-Sprache implementieren
- Visitor-Interface: Eine Methode pro Modell-Klasse
- Modell-Interface (hier: `Element`): Eine Methode (meist `accept()` genannt), die einen `Visitor` übernimmt und sich selbst an ihn übergibt.

```
public interface Element
{
    void Accept(Visitor visitor);
}
```

- Eine Klasse mit `Visitor`-Implementierung für jede Operation (ein `Printer-Visitor`, ein `CodeGen-Visitor`, ...)

```
public class LiteralExpression: Expression
{
    // The value of the literal
    public double Value { get; private set; }

    internal LiteralExpression(double value)
    {
        this.Value = value;
    }

    public void Accept(Visitor visitor)
    {
        visitor.VisitLiteralExpression(this);
    }
}
```