Der Compiler: Eine Einführung für den Amateur

Berliner Linux User Group

Benjamin Stürz

2024-03-01

Outline

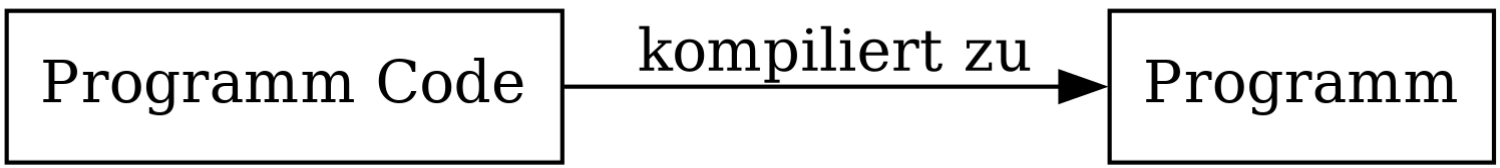
Wer bin ich?

- Benjamin Stürz <benni@stuerz.xyz>
- Hobby Compiler Entwickler
- Aus Berlin
- Studiere Informatik an der TH Brandenburg
- OpenBSD User
- `https://stuerz.xyz/Compiler.pdf`

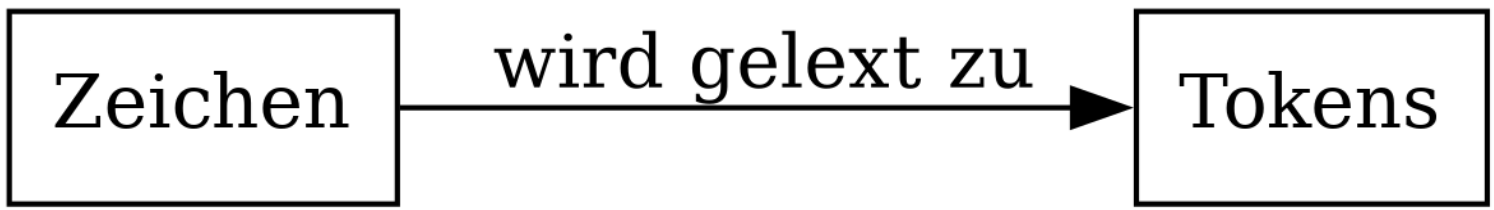
Die Sprache

```
fn main() {
    let x = 0;
    while x < 10 {
        print x;
        x = x + 1;
    }
    return 0;
}
```

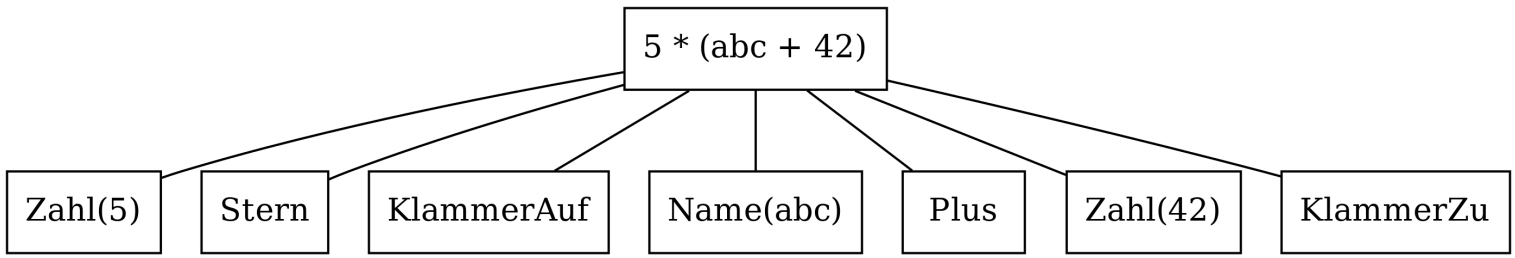
Was ist ein Compiler?



Lexer - Theorie



Lexer - Tokens



Lexer - Regeln I

```
Zahl            [0-9]+
Name            [a-zA-Z_][a-zA-Z_0-9]*
```

Lexer - Regeln II

```
Plus              "+"
Minus             "-"
Stern             "*"
Slash             "/"
Gleich            "="
GleichGleich      "=="
KlammerAuf        "("
KlammerZu         ")"
GeschwKlammerAuf  "{"
GeschwKlammerZu   "}"
Kleiner           "<"
KleinerGlech      "<="
Größer            ">"
GrößerGleich      ">="
Nicht             "!"
NichtGleich       "!="
Semikolon         ";"
```

Lexer - Regeln III

```
Return          "return"
Let             "let"
Print           "print"
Fn              "fn"
If              "if"
Else            "else"
While           "while"
```

Lexer - Implementation - Token

```
enum token_type {
    TK_Zahl,                // Zahl(x)
    TK_Name,                // Name(x)

    TK_Plus,                // "+"
    TK_Minus,               // "-"
    TK_KlammerAuf,          // "("
    TK_KlammerZu,           // ")"
    // Restliche Operatoren...

    TK_Print,               // "print"
    TK_Return,              // "return"
    // Restliche Schlüsselworter...

    TK_EOF,                 // End of File
};

struct token {
    enum token_type type;
    union {
        int zahl;
        char *name;
    };
};
```

Lexer - Implementation - Übersicht

```
struct token lex(void) {
    struct token tk;
    int ch;

    // Leerzeichen überspringen
    // Zeichen einlesen

    if (isdigit(ch)) {
        // Zahl lexen
    } else if (isalpha(ch) || ch == '_') {
        // Name oder Schlüsselwort lexen
    } else switch (ch) {
        // Operatoren
    default:
        // Fehler
    }

    return tk;
}
```

Lexer - Implementation - Leerzeichen

```
do {
    ch = getchar();
} while (isspace(ch));
```

Lexer - Implementation - Zahlen

```
if (isdigit(ch)) {
    int wert = 0;

    do {
      wert = wert * 10 + (ch - '0');
      ch = getchar();
    } while (isdigit(ch));

    ungetc(ch, stdin);

    tk.type = TK_Zahl;
    tk.zahl = wert;
}
```

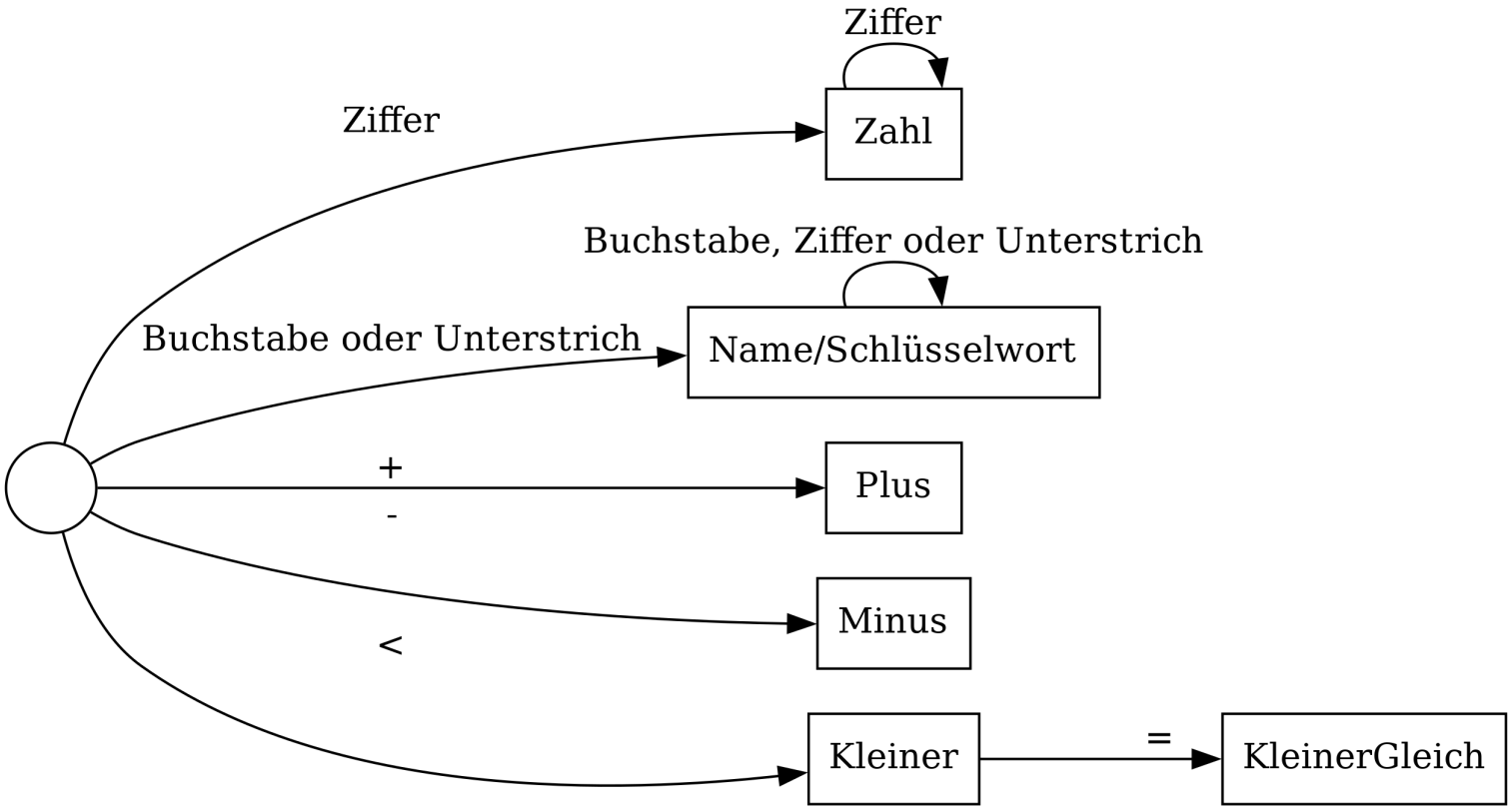
Lexer - Implementation - Namen & Schlüsselwörter

```
if (isalpha(ch) || ch == '_') {
    size_t len = 0, cap = 10;
    char *s = malloc(cap + 1);

    do {
        if (len == cap) {
            cap *= 2;
            s = realloc(s, cap + 1);
        }
        s[len++] = ch;
        ch = getchar();
    } while (isalnum(ch) || ch == '_');

    ungetc(ch, stdin);
    s[len] = '\0';

    if (strcmp(s, "print") == 0) {
        tk.type = TK_Print;
    } else if (strcmp(s, "return") == 0) {
        tk.type = TK_Return;
    // Weitere Schlüsselwörter...
    } else {
        tk.type = TK_Name;
        tk.name = s;
    }
}
```

Lexer - Implementation - Operatoren

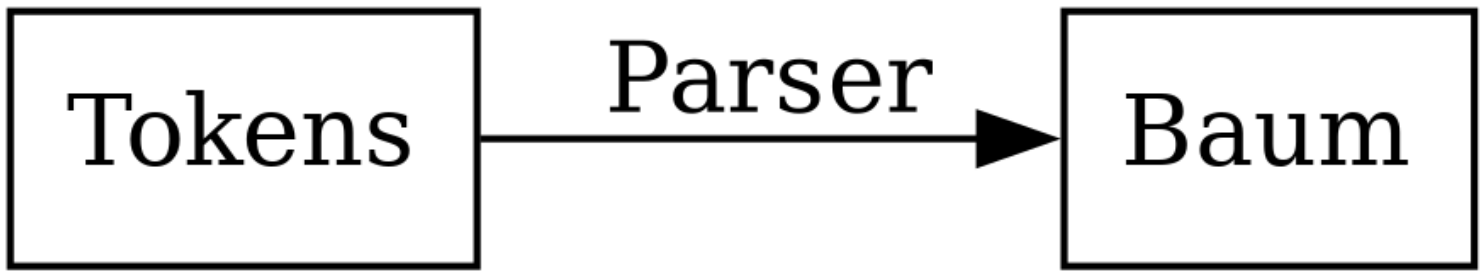
```
switch (ch) {
case EOF: tk.type = TK_EOF;                    break;
case '+': tk.type = TK_Plus;                   break;
case '-': tk.type = TK_Minus;                  break;
case '*': tk.type = TK_Stern;                  break;
case '/': tk.type = TK_Slash;                  break;
case '(': tk.type = TK_KlammerAuf;             break;
case ')': tk.type = TK_KlammerZu;              break;
case '{': tk.type = TK_GeschwKlammerAuf;       break;
case '}': tk.type = TK_GeschwKlammerZu;        break;
case ';': tk.type = TK_Semikolon;              break;
case '<':
    ch = getchar();
    if (ch == '=') {
        tk.type = TK_KleinerGleich;
    } else {
        tk.type = TK_Kleiner;
        ungetc(ch, stdin);
    }
    break;
// TODO: '>', '>=', '!', '!=', '=', '=='
default:
    errx(1, "invalid input: '%c'", ch);
}
```

Lexer - Zusammenfassung

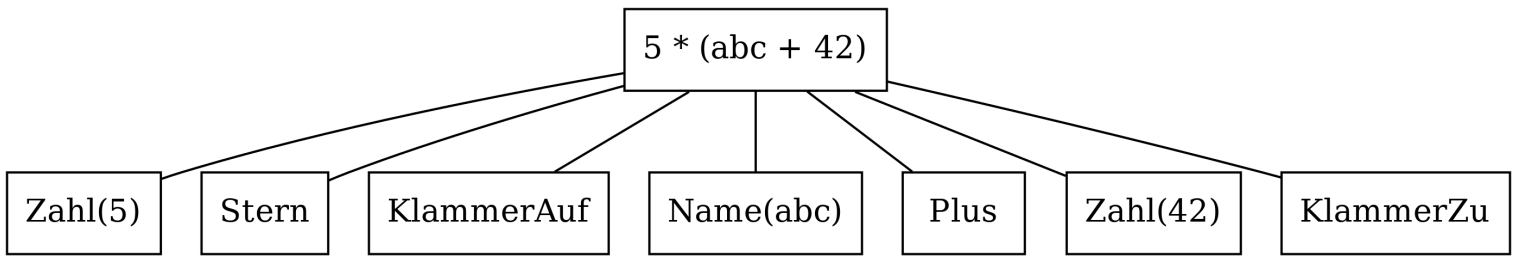



```
// Die einzelnen Typen unserer Token.
enum token_type;
// Die Tokens selbst
struct token;
// Die Funktion, die uns Tokens zurückgibt.
struct token lex(void);
```

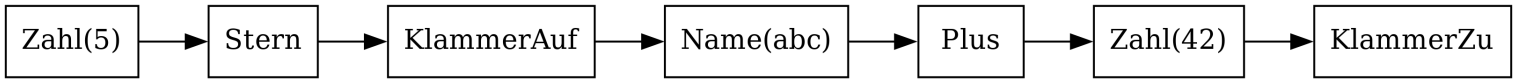
Parser - Theorie I

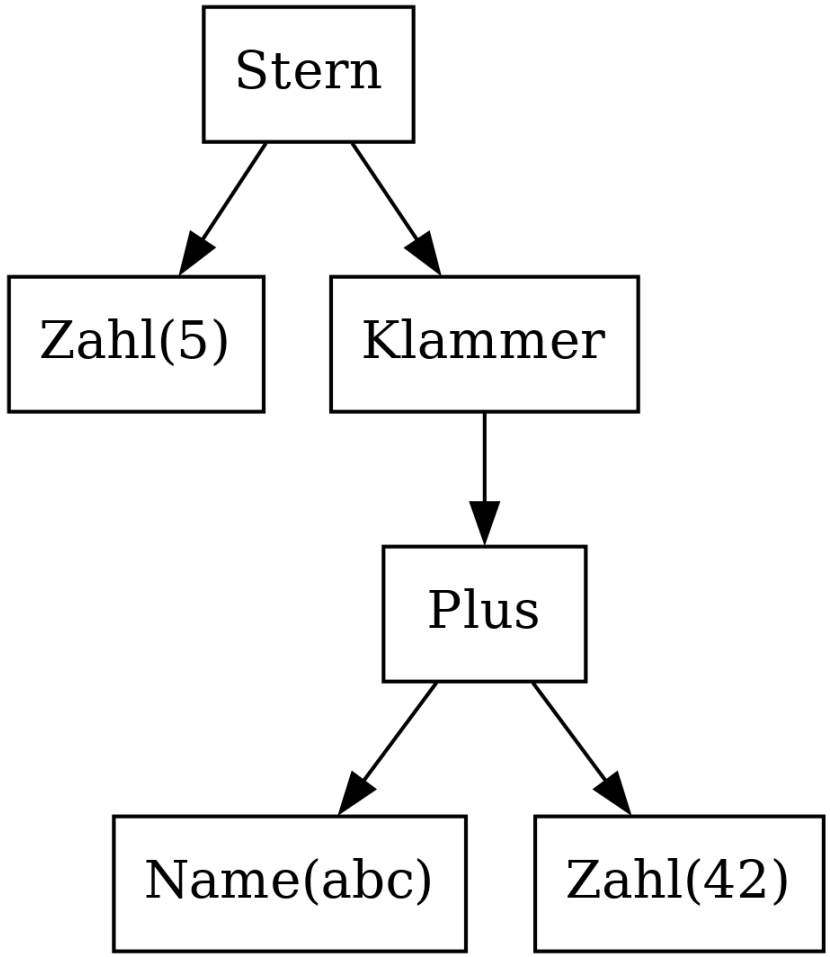


Parser - Theorie II



Parser - Theorie III

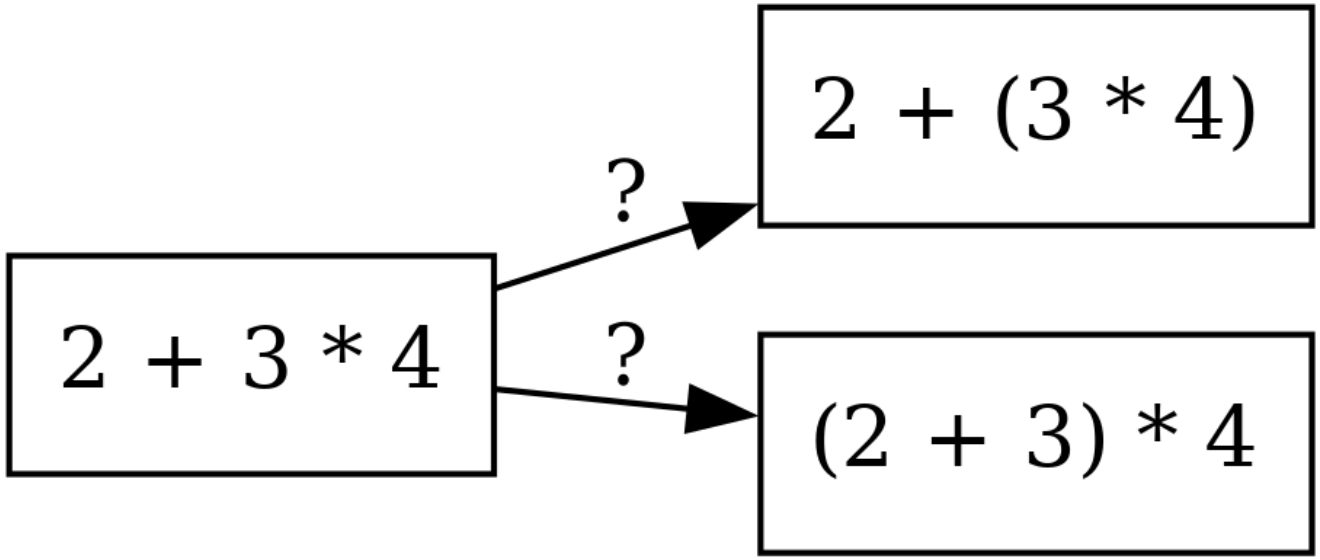


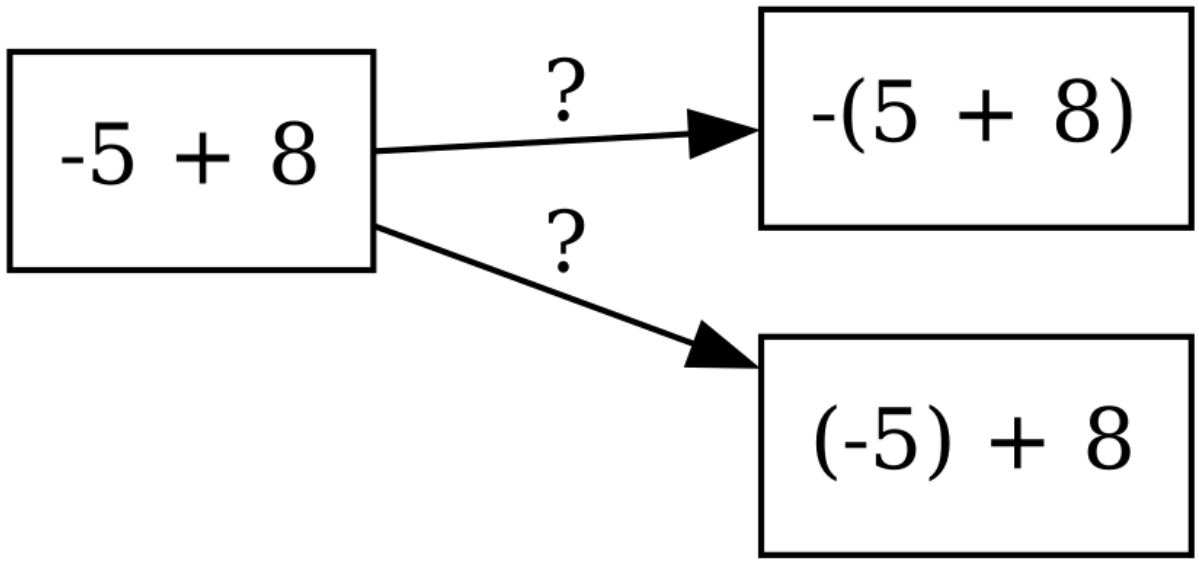


Parser - Regeln I

```
File       = Function+ EOF

Function   = "fn" Name "(" ")" Block

Block      = "{" Statement* "}"

Statement  = "print" Expression ";"
           | "return" Expression ";"
           | "let" Name "=" Expression ";"
           | Name "=" Expression ";"
           | "if" Expression Block ("else" Block)?
           | "while" Expression Block
           | Block

Expression = Zahl
           | Name
           | "(" Expression ")"
           | "-" Expression
           | Expression "+" Expression
           | Expression "-" Expression
           | Expression "*" Expression
           | Expression "/" Expression
           | Expression "==" Expression
           | Expression "!=" Expression
           | Expression "<"  Expression
           | Expression "<=" Expression
           | Expression ">"  Expression
           | Expression ">=" Expression
```

Parser - Regeln II

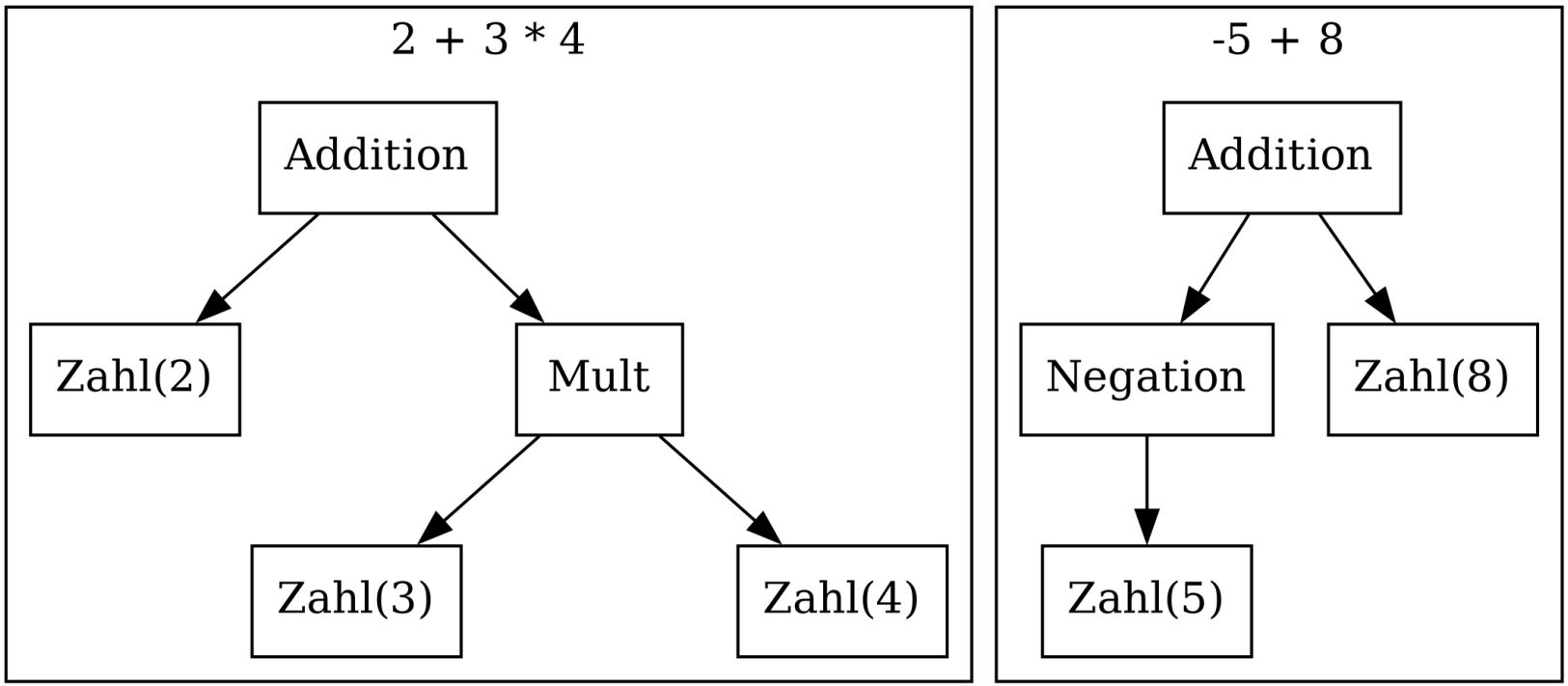




Parser - Regeln III

```
Expression = Comparison

Comparison = Addition "==" Addition
           | Addition "!=" Addition
           | Addition "<"  Addition
           | Addition "<=" Addition
           | Addition ">"  Addition
           | Addition ">=" Addition
           | Addition

Addition   = Addition "+" Mult
           | Addition "-" Mult
           | Mult

Mult       = Mult "*" Unary
           | Mult "/" Unary
           | Unary

Unary      = "-" Unary
           | Atom

Atom       = Zahl
           | Name
           | "(" Expression ")"
```

Parser - Regeln IV



Parser - Regeln V

```
File       = Function+ EOF

Function   = Fn Name KlammerAuf KlammerZu Block

Block      = GeschwKlammerAuf Statement* GeschwKlammerZu

Statement  = Print Expression Semikolon
           | Return Expression Semikolon
           | Let Name Gleich Expression Semikolon
           | Name Gleich Expression Semikolon
           | If Expression Block (Else Block)?
           | While Expression Block

Expression = Addition RelOp Addition
           | Addition

RelOp      = GleichGleich | NichtGleich | Kleiner | KleinerGleich | Größer | GrößerGleich

Addition   = Addition (Plus | Minus) Mult
           | Mult

Mult       = Mult (Stern | Slash) Unary
           | Unary

Unary      = Minus Unary
           | Atom

Atom       = Zahl
           | Name
           | KlammerAuf Expression KlammerZu
```

Parser - Datentypen - File, Function, Block

```
struct file {
    struct function **funcs;
};

struct function {
    char *name;
    struct block *block;
};

struct block {
    struct statement **stmts;
};
```

Parser - Datentypen - Statement

```
enum statement_type {
    ST_Print,
    ST_Return,
    ST_Let,
    ST_Assign,
    ST_If,
    ST_While,
};

struct statement {
    enum statement_type type;
    union {
        struct expression *expr; // print, return
        struct {
            char *name;
            struct expression *value;
        } let;
        struct {
            struct expression *condition;
            struct block *true_case;
            struct block *false_case; // optional
        } ifst; // if statement
        struct {
            struct expression *condition;
            struct block *block;
        } whst; // while statement
    };
};
```

Parser - Datentypen - Expression

```
enum expression_type {
    EX_Zahl,
    EX_Name,
    EX_Klammer,
    EX_Negation,
    EX_Addition,
    EX_Subtraktion,
    EX_Multiplikation,
    EX_Division,
    EX_Gleich,
    EX_Ungleich,
    EX_Kleiner,
    EX_KleinerGleich,
    EX_Größer,
    EX_GrößerGleich,
};

struct expression {
    enum expression_type type;
    union {
        int zahl;
        char *name;
        struct expression *expr; // Klammer, Negation
        struct {
            struct expression *links;
            struct expression *rechts;
        } binär;
    };
};
```

Parser - Implementation - Hilfsfunktionen

```
#define new(T) ((T *)malloc(sizeof(T)))

bool has_token;
struct token peekd;

struct token peek(void) {
    if (!has_token) {
        peekd = lex();
        has_token = true;
    }
    return peekd;
}

struct token next(void) {
    if (has_token) {
        has_token = false;
        return peekd;
    } else {
        return lex();
    }
}

struct token expect(enum token_type type) {
    struct tk = next();
    if (tk.type != type)
        errx(1, "syntax error");
    return tk;
}
```

Parser - Implementation - Atom

```
Atom = Zahl
     | Name
     | KlammerAuf Expression KlammerZu
```

```
struct expression *atom(void) {
    struct expression *e = new(struct expression);
    struct token tk;

    switch (tk.type) {
    case TK_Zahl:
        e->type = EX_Zahl;
        e->zahl = tk.zahl;
        break;
    case TK_Name:
        e->type = EX_Name;
        e->name = tk.name;
        break;
    case TK_KlammerAuf:
        e->type = EX_Klammer;
        e->expr = expression();
        expect(TK_KlammerZu);
        break;
    default:
            errx(1, "expected atom");
    }
    return e;
}
```

Parser - Implementation - Unary

```
Unary = Minus Unary
      | Atom
```

```
struct expression *unary(void) {
    if (peek().type == TK_Minus) {
        struct expression *e = new(struct expression);
        next();
        e->type = EX_Negation;
        e->expr = unary();
        return e;
    } else {
        return atom();
    }
}
```

Parser - Implementation - Mult

```
Mult = Mult (Stern | Slash) Unary
     | Unary
```

```
struct expression *mult(void) {
    struct expression *links = unary();

    while (peek().type == TK_Stern || peek().type == TK_Slash) {
        struct expression *e;

        if (peek().type == TK_Stern) {
            e->type = EX_Multiplikation;
        } else {
            e->type = EX_Division;
        }
        next();

        e->binär.links = links;
        e->binär.rechts = unary();
        links = e;
    }

    return links;
}
```

Parser - Implementation - Addition

```
Addition = Addition (Plus | Minus) Mult
         | Mult
```

```
struct expression *addition(void) {
    struct expression *links = mult();

    while (peek().type == TK_Plus || peek().type == TK_Minus) {
        struct expression *e;

        if (peek().type == TK_Plus) {
            e->type = EX_Addition;
        } else {
            e->type = EX_Subtraktion;
        }
        next();

        e->binär.links = links;
        e->binär.rechts = mult();
        links = e;
    }

    return links;
}
```

Parser - Implementation - Expression

```
RelOp      = GleichGleich | NichtGleich | Kleiner | KleinerGleich | Größer | GrößerGleich
Expression = Addition RelOp Addition
           | Addition
```

```
struct expression *expression(void) {
    struct expression *e, *links;
    enum expr_typ type;

    links = addition();

    switch (peek().type) {
    case TK_GleichGleich:  type = EX_Gleich;        break;
    case TK_NichtGleich:   type = EX_Ungleich;      break;
    case TK_KleinerGleich: type = EX_KleinerGleich; break;
    case TK_GrößerGleich:  type = EX_GrößerGleich;  break;
    case TK_Kleiner:       type = EX_Kleiner;       break;
    case TK_Größer:        type = EX_Größer;        break;
    default:               return links;
    }
    next();

    e = new(struct expression);
    e->type = type;
    e->binär.links = links;
    e->binär.rechts = addition();
    return e;
}
```

Parser - Implementation - Statement

```
Statement = Print Expression Semikolon
          | Return Expression Semikolon
          | Let Name Gleich Expression Semikolon
          | Name Gleich Expression Semikolon
          | If Expression Block (Else Block)?
          | While Expression Block
```

```
struct statement *statement(void) {
    struct statement *st = new(struct statement);
    struct token tk = next();

    switch (tk.type) {
    case TK_Print:
        st->type = ST_Print;
        st->expr = expression();
        expect(TK_Semikolon);
        break;
    case TK_If:
        st->type = ST_If;
        st->ifst.condition = expression();
        st->ifst.true_case = block();
        if (peek().type == TK_Else) {
            next();
            st->ifst.false_case = block();
        } else {
            st->ifst.false_case = NULL;
        }
        break;
    // ...
    }
    return st;
}
```

Parser - Implementation - Block

```
Block = GeschwKlammerAuf Statement* GeschwKlammerZu
```

```
struct block *block(void) {
    struct block *b = new(struct block);
    size_t len = 0, cap = 10;

    b->stmts = calloc(cap + 1, sizeof(struct statement *));
    expect(TK_GeschwKlammerAuf);

    while (peek().type != TK_GeschwKlammerZu) {
        if (len == cap) {
            cap *= 2;
            b->stmts = reallocarray(b->stmts, cap + 1, sizeof(struct statement *));
        }

        b->stmts[len++] = statement();
    }

    b->stmts[len] = NULL;
    expect(TK_GeschwKlammerZu);
    return b;
}
```

Parser - Implementation - Funktion

```
Function = Fn Name KlammerAuf KlammerZu Block
```

```
struct function *function(void) {
    struct function *f = new(struct function);

    expect(TK_Fn);
    f->name = expect(TK_Name).name;
    expect(TK_KlammerAuf);
    expect(TK_KlammerZu);
    f->block = block();

    return f;
}
```

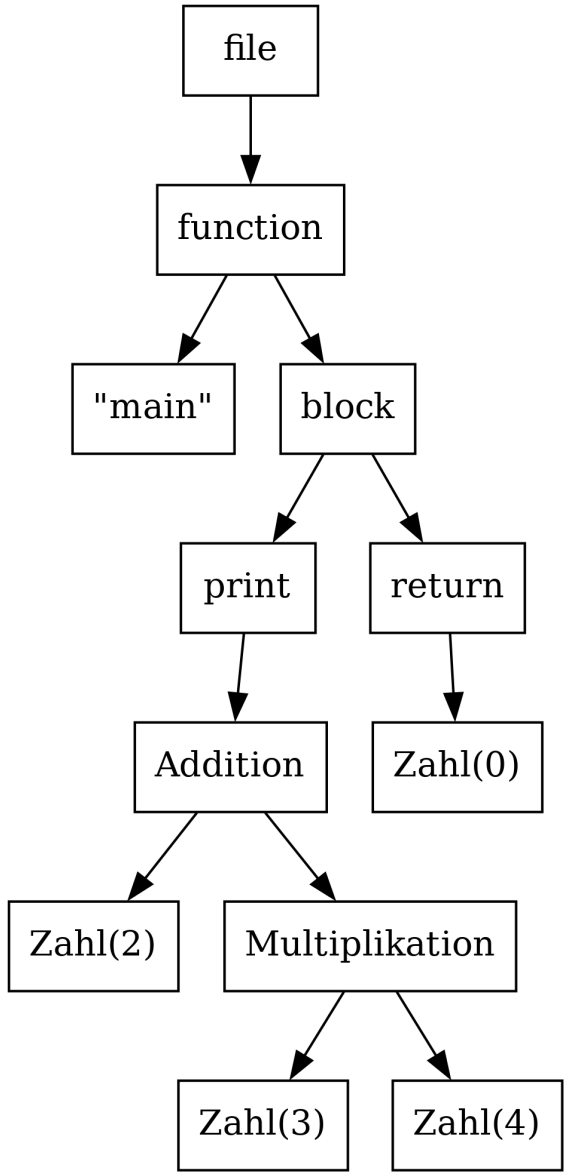
Parser - Implementation - File

```
File = Function+ EOF
```

```
struct file *parse(void) {
    struct file *f = new(struct file);
    size_t len = 0, cap = 10;

    f->funcs = calloc(cap + 1, sizeof(struct function *));

    do {
        if (len == cap) {
            cap *= 2;
            f->funcs = reallocarray(f->funcs, cap + 1, sizeof(struct function *));
        }
        f->funcs[len++] = function();
    } while (peek().type != TK_EOF);

    f->funcs[len] = NULL;

    return f;
}
```

Parser - Zusammenfassung

```
fn main() {
    print 2 + 3 * 4;
    return 0;
}
```



Parser - Was tun mit dem Baum?

- Ausführen => Interpreter
- Code Generieren => Compiler
  - Eine andere Sprache? => Transpiler
  - Assembler?
  - IR?

IR - Theorie

- Intermediate Representation
- liegt irgendwo zwischen Assembler und C
- Beispiele: LLVM IR, Java Bytecode, ...

IR - Warum?

- Einfachere Portierbarkeit auf andere Platformen
- Extra Optimisierungen

IR - Beispiel

```
fn main() {
    print 2 + 3 * 4;
    return 0;
}
```

```
fn main() {
    iload 2
    iload 3
    iload 4
    imul
    iadd
    iprint
    iload 0
    ireturn
}
```

- Java

```
fn main() {
    $1 = 2;
    $2 = 3
    $3 = 4;
    $4 = imul $2, $3;
    $5 = iadd $1, $4;
    print $5;
    $6 = 0;
    ret $6;
}
```

- GCC
- LLVM

IR - Generation Expressions

```
void gen_expr(const struct expr *e) {
    switch (e->type) {
    case EX_Zahl:
        printf("iload %s\n", e->zahl);
        break;
    case EX_Klammer:
        gen_expr(e->expr);
        break;
    case EX_Addition:
        gen_expr(e->binär.links);
        gen_expr(e->binär.rechte);
        printf("iadd\n");
        break;
    // ...
    }
}
```

IR - Generation Labels

```
if condition {
    print 42;
}
```

wird zu

```
  // Code für condition

  jmp-if-not .L1
  iload 42
  iprint

.L1:
  // Weiterer Code
```

CodeGen

```
fn main() {
    print 42;
}
```

=>

```
section .rodata
fmt: db "Zahl: %d", 10, 0

section .text
default rel
global main
extern printf
main:
        push rbp
        lea rdi, [fmt]
        mov rsi, 42
        xor eax, eax
        call printf
        pop rbp
        ret
```

Optimisierungen - Einführung

TODO: Schnelle Grafik

Optimisierungen - Peephole Optimizations

```
$0 = ...;
$1 = 2;
$2 = imul $0, $1;
```

=>

```
$0 = ...;
$1 = 1;
$2 = ishl $0, $1;
```

Optimisierungen - Dead Code Elimination

```
if 1 == 1 {
    print 1;
} else {
    print 0;
}
```

=>

```
if 1 == 1 {
    print 1;
}
```

Optimisierungen - Constant Folding

```
print x + 3 * 4;
```

=>

```
print x + 12;
```

Optimisierungen - Viel mehr...

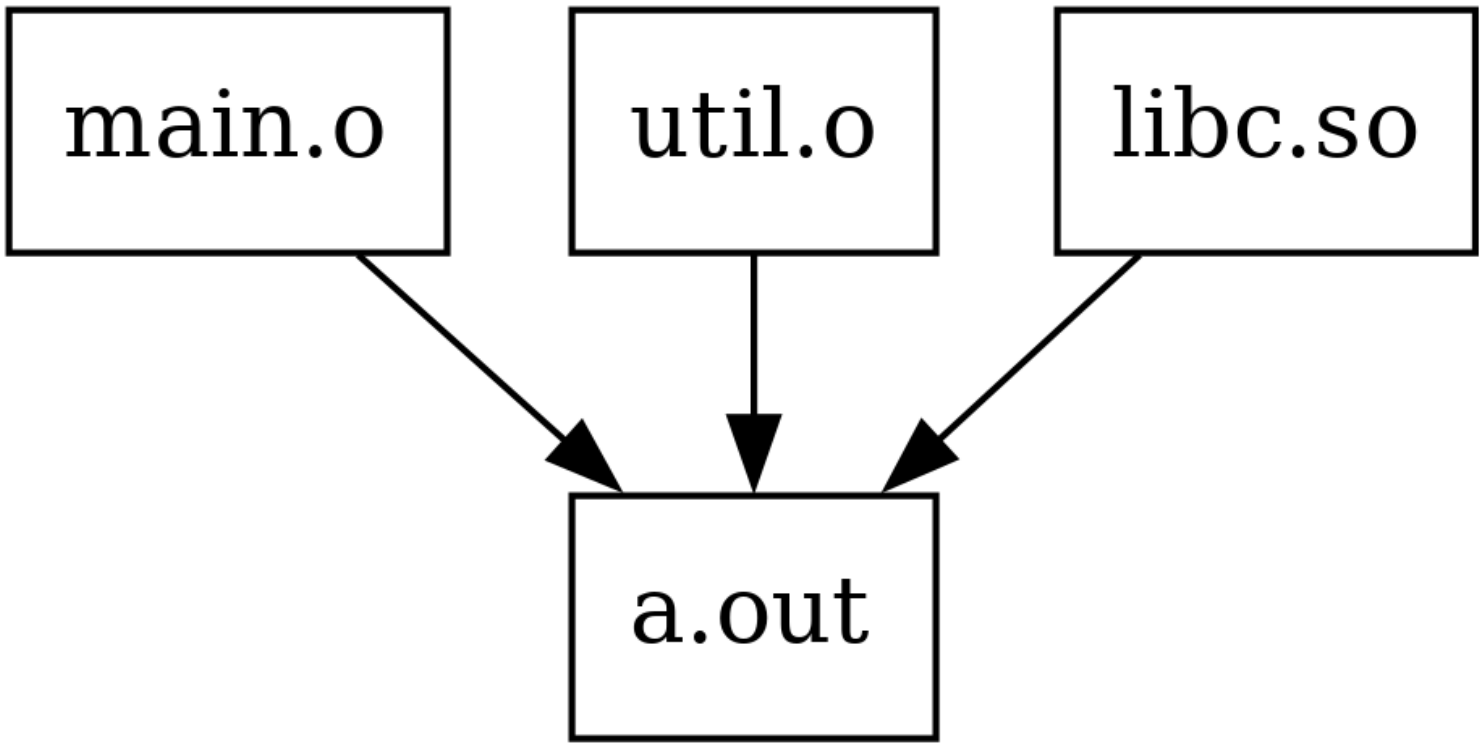
V • T • E	**Compiler optimizations**	[hide]
Basic block	Peephole optimization • Local value numbering
Loop	Automatic parallelization • Automatic vectorization • Induction variable • Loop fusion • Loop-invariant code motion • Loop inversion • Loop interchange • Loop nest optimization • Loop splitting • Loop unrolling • Loop unswitching • Software pipelining • Strength reduction
Data-flow analysis	Available expression • Common subexpression elimination • Constant folding • Dead store elimination • Induction variable recognition and elimination • Live-variable analysis • Use-define chain
SSA-based	Global value numbering • Sparse conditional constant propagation
Code generation	Instruction scheduling • Instruction selection • Register allocation • Rematerialization
Functional	Deforestation • Tail-call elimination
Global	Interprocedural optimization
Other	Bounds-checking elimination • Compile-time function execution • Dead-code elimination • Expression templates • Inline expansion • Jump threading • Partial evaluation • Profile-guided optimization
Static analysis	Alias analysis • Array-access analysis • Control-flow analysis • Data-flow analysis • Dependence analysis • Escape analysis • Pointer analysis • Shape analysis • Value range analysis

Assembler

```
$ nasm -felf64 -o main.o main.asm
```

Linker

```
$ cc -o program main.o util.o
```



Resourcen

- Compilerbau Bücher
- Wikipedia
- godbolt.org