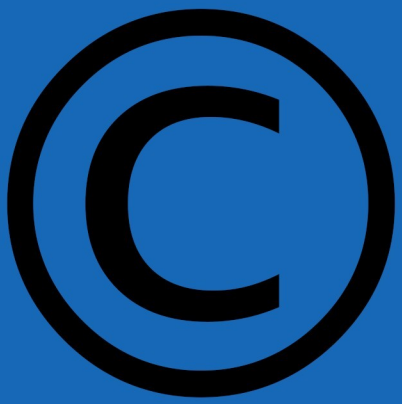
Sven Eric Panitz

Lehrbriefe
C für Java-Programmierer



abstrakter Syntaxbaum

Ein abstrakter Syntaxbaum

Sven Eric Panitz

28. Juni 2023

Inhaltsverzeichnis

1 Einleitung

In dieser Aufgabe sollen Sie einen kleinen Interpreter schreiben, für eine einfache Programmiersprache (LL für little language).

Die Sprache LL ist gegeben als abstrakter Syntaxbaum. Es gibt dabei folgende Ausdrücke:

- Zahlenliterale
- Variablen
- Funktionsaufrufe
- Zuweisung an eine Variable
- Binäre Operatoren
- if-else Ausdrücke
- while-Schleifen

Für alle diese stehen structs zur Verfügung. Diese werde in einer union vereinigt. Ein allgemeiner Ausdruck ist schließlich ein struct aus dieser union und einem enum Wert, der anzeigt, welche Daten in der union gerade gespeichert sind.

Ein LL Programm besteht aus einem Array von Funktionsdefinitionen.

Einer LL-Funktion stehen 100 Variablen zur Verfügung, die durchnummeriert sind und in einem Array, der die Umgebung, in der ausgewertet wird darstellt.

Der einzige Datentyp, auf dem LL arbeitet, ist `long int`.

Sie sollen in dieser Aufgabe folgende Funktionen implementieren:

- Konstruktorfunktionen für alle Datentypen.
- Destruktorfunktionen für alle Datentypen.
- die Funktion `eval` für alle Datentypen.

Gegeben ist auch ein kleiner Parser, der für ein Programm den Syntaxbaum erstellt.

Hier ein kleines LL Programm, das die Syntax exemplarisch zeigt:

fac.ll

```
fun doppelt(X42){
  2 * X42
}

fun fac(X1){
  if (X1<=0){1}else{X1*fac(X1-1)}
}

fun fib(X1){
  if(X1<=1){
    X1
  }else{
    fib(X1-2)+fib(X1-1)
  }
}

fun facIt(X2){
```

```
  X1=1;
  while(X2){
    X1=X1*X2;X2=X2-1
  };
  X1
}
```

Funktionen werden mit dem Schlüsselwort `fun` eingeleitet. Variablen haben die Form `X`n, wobei n eine natürliche Zahl kleiner 100 ist.

2 Syntaxbaum

Ein Programm wird als ein Baum dargestellt. Ein Baumknoten stellt ein Programmkonstrukt dar und die Kinder des Baumknotens, einzelne Bestandteile des Programmskonstrukts. So hat der Knoten für eine while-Schleife ein Kind mit der Schleifenbedingung und ein Kind für den Schleifenrumpf.

2.1 Operatoren

Zunächst definieren wir einen Aufzählungstyp für die zweistelligen Operatoren unserer Programmiersprache.

SyntaxTree.h

```
#ifndef LL__H
#define LL__H

typedef enum{
  ADD, SUB, MULT, DIV, MOD, LT, GT, LE, GE, EQ, NEQ, AND, OR, SEQ
  }  BinOp;
```

Wir haben die gängigen arithmetischen Operatoren, Vergleichsoperatoren, logischen Operatoren und schließlich mit `SEQ` einen Operatore der Sequenz von zwei Ausdrücken.

2.2 Typen der Baumknoten

Ein weiterer Aufzählungstyp definiert die unterschiedlichen Baumknotenarten, die wir benötigen.

SyntaxTree.h

```
typedef enum {
  BIN_OP, VAR, LIT, IF_ELSE, WHILE, ASSIGN, FUN_CALL
} ExprType;
```

2.3 Baumknoten für Ausdrücke

Der zentrale Typ wird der allgemeine Typ sein, der alle Baumknoten, die ein Programmkonstrukt darstellen, vereint. Diesen Typ nennen wir `Expr`.

SyntaxTree.h

```
struct Ex;
typedef struct Ex Expr;
```

Der Typ `Expr` entspricht in unserer Architektur im Vergleich zu Java, einer Schnittstelle. Diese wird nun von den einzelnen Typen für Programmkonstrukte umgesetzt.

2.3.1 Binäre Operatoren

Ein Ausdruck, der eine Anwendung eines binären Operators auf die Operanden darstellt, braucht den Operator und den linken und rechten Operanden. Das wird doch folgende Struktur ausgedrückt.

SyntaxTree.h

```
typedef struct{
  BinOp op;
  Expr* left;
  Expr* right;
} BinExpr;
```

2.3.2 Variablen

Eine Variable in unserer Sprache hat eine interne ganze Zahl als Nummer.

SyntaxTree.h

```
typedef struct {
  unsigned int varNr;
}VariableExpr;
```

2.3.3 Zahlenliterale

Ein Zahlenliteral hat immer genau eine Zahl als Wert:

SyntaxTree.h

```
typedef struct {
  long int value;
} LiteralExpr;
```

2.3.4 Zuweisungen

In einer Zuweisung wird einer der nummerierten Variablen der Wert eines Ausdrucks zugewiesen:

SyntaxTree.h

```
typedef struct {
  unsigned int varNr;
  Expr* rhs;
}AssignExpr;
```

2.3.5 Bedingungen

Der Bedingungsausdruck hat immer einen Zweig für den *else*-Fall:

SyntaxTree.h

```
typedef struct{
  Expr* cond;
  Expr* ifCase;
  Expr* elseCase;
} IfElseExpr;
```

2.3.6 Schleifen

Schleifen haben eine Bedingung und einen Rumpf:

SyntaxTree.h

```
typedef struct {
  Expr* cond;
  Expr* body;
} WhileExpr;
```

2.3.7 Funktionsaufrufe

Am wahrscheinlich komplexesten ist die Struktur, die Funktionsaufrufe definiert. Sie enthält den Namen der Funktion, eine interne Nummer dieser Funktion, einen Array von Ausdrücken für die übergebenen Argumente und die Anzahl der Argumente:

SyntaxTree.h

```
typedef struct{
  char* name;
  int compiledNr;
  Expr** args;
  int argsNr;
} FunCallExpr;
```

2.3.8 Vereinigung der Ausdrücke

Alle die Strukturen für Baumknoten vereinigen wir zu einen Summentypen:

SyntaxTree.h

```
union ExprUnion{
  BinExpr binExpr;
  VariableExpr variableExpr;
  LiteralExpr literalExpr;
  AssignExpr assignExpr;
  IfElseExpr ifElseExpr;
  WhileExpr whileExpr;
  FunCallExpr funCallExpr;
};
```

Ein Ausdruck hat nun zwei Attribute: den Knotentyp und dem eigentlichen Knoten:

SyntaxTree.h

```
struct Ex{
  ExprType type;
  union ExprUnion expr;
};
```

2.3.9 Funktionsdefinitionen

Ein zusätzliches Struct stellt eine Funktionsdefinition dar. EIne Funktionsdefinition ist aber kein Ausdruck und taucht deshelb nicht in der Union `ExprUnion` auf.

Eine Funktionsdefinition hat einen Funktionsnamen, einen Array von Variablennummern, für die Argumente der Funktion, die Anzahl der Argumente (also die Stelligkeit der Funktion) und einen Ausdruck als Funktionsrumpf:

SyntaxTree.h

```
typedef struct{
  char* name;
  int* args;
  int argsNr;
  Expr* body;
} FunDef;
```

Ein Programm unserer kleinen Programmiersprache ist ein Array von Funktionsdefinitionen:

SyntaxTree.h

```
typedef struct{
  FunDef* funs;
  int funsNr;
}  Program;
```

Das Ergebnis eines Parsers, der für einen Quelltext die hier definierte Struktur erzeugt, kann ein Programm oder ein einzelner Ausdrucj sein.

SyntaxTree.h

```
typedef union {Program prog;Expr* expr;} ParseResult;
```

2.4 Konstruktoren

Für alle Baumknoten schreiben wir nun Konstruktorfunktionen.

SyntaxTree.h

```
Expr* newBinExpr(BinOp op,Expr* left, Expr* right);
Expr* newVariableExpr(unsigned int varNr);
Expr* newLiteralExpr(long int value);
Expr* newIfElseExpr(Expr* cond, Expr* ifCase, Expr* elseCase);
Expr* newWhileExpr(Expr* cond, Expr* body);
Expr* newAssignExpr(unsigned int varNr,Expr* rhs);
Expr* newFunCallExpr(char* name, Expr** args, int argsNr);
```

Spezielle konstruktorfunktionen werden für die verschiedenen Operatorausdrücke vorgesehen:

SyntaxTree.h

```
Expr* newAddExpr(Expr* left, Expr* right);
Expr* newSubExpr(Expr* left, Expr* right);
Expr* newMultExpr(Expr* left, Expr* right);
Expr* newDivExpr(Expr* left, Expr* right);
Expr* newModExpr(Expr* left, Expr* right);
Expr* newLtExpr(Expr* left, Expr* right);
Expr* newGtExpr(Expr* left, Expr* right);
Expr* newLeExpr(Expr* left, Expr* right);
Expr* newGeExpr(Expr* left, Expr* right);
Expr* newEqExpr(Expr* left, Expr* right);
Expr* newNeqExpr(Expr* left, Expr* right);
Expr* newAndExpr(Expr* left, Expr* right);
Expr* newOrExpr(Expr* left, Expr* right);
Expr* newSeqExpr(Expr* left, Expr* right);
```

Wir beginnen nun mit der Implementierung der ersten Konstrukte.Hierzu brauchen wir folgende Includes:

SyntaxTree.c

```
#include "SyntaxTree.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

2.4.1 Konstruktor für Operatorknoten

Für die Konstruktorfunktionen ist zunächst ein Heapobjekt für einen allgemeinen Ausdruck zu allokieren. Dann ist der spezielle Ausdruck als Struct-Objekt aus den Argumenten zu erzeugen. Dieser wird im Union-Attribut entsprechend gesetzt und der entsprechenden Enum-Wert in dem Aufzählungsattribut gesetzt. Der Zeiger auf das entsprechende Heapobjekt wird zurückgegeben.

SyntaxTree.c

```
Expr* newBinExpr(BinOp op,Expr* left, Expr* right){
  Expr* this = malloc(sizeof(Expr));
  BinExpr expr = {op,left,right};
  this->type = BIN_OP;
  this->expr.binExpr = expr;
  return this;
}
```

Die Konstruktoren für spezielle Operatoren verwenden diese Funktion:

SyntaxTree.c

```
Expr* newAddExpr(Expr* left, Expr* right){
  return newBinExpr(ADD,left,right);
}
Expr* newSubExpr(Expr* left, Expr* right){
  return newBinExpr(SUB,left,right);
}
Expr* newMultExpr(Expr* left, Expr* right){
  return newBinExpr(MULT,left,right);
}
Expr* newDivExpr(Expr* left, Expr* right){
  return newBinExpr(DIV,left,right);
}
Expr* newModExpr(Expr* left, Expr* right){
  return newBinExpr(MOD,left,right);
}
Expr* newLtExpr(Expr* left, Expr* right){
  return newBinExpr(LT,left,right);
}
Expr* newGtExpr(Expr* left, Expr* right){
  return newBinExpr(GT,left,right);
}
Expr* newLeExpr(Expr* left, Expr* right){
  return newBinExpr(LE,left,right);
}
Expr* newGeExpr(Expr* left, Expr* right){
  return newBinExpr(GE,left,right);
}
Expr* newEqExpr(Expr* left, Expr* right){
  return newBinExpr(EQ,left,right);
}
Expr* newNeqExpr(Expr* left, Expr* right){
  return newBinExpr(NEQ,left,right);
}
Expr* newAndExpr(Expr* left, Expr* right){
  return newBinExpr(AND,left,right);
}
Expr* newOrExpr(Expr* left, Expr* right){
  return newBinExpr(OR,left,right);
}
Expr* newSeqExpr(Expr* left, Expr* right){
  return newBinExpr(SEQ,left,right);
}
```

Aufgabe 1 Implementieren Sie jetzt die Konstruktoren der übrigen Baumknoten. Gehen Sie dabei schematisch genauso vor, wie in `newBinExpr` vorgemacht.

- Schreiben Sie den Konstruktor für Variablen:

SyntaxTree.c

```
Expr* newVariableExpr(unsigned int varNr){
  return NULL; /*ToDo*/
}
```

- Schreiben Sie den Konstruktor für Zahlenliterale:

SyntaxTree.c

```
Expr* newLiteralExpr(long int value){
  return NULL; /*ToDo*/
}
```

- Schreiben Sie den Konstruktor für Zuweisungsausdrücke:

SyntaxTree.c

```
Expr* newAssignExpr(unsigned int varNr,Expr* rhs){
  return NULL; /*ToDo*/
}
```

- Schreiben Sie den Konstruktor für Verzweigungen:

SyntaxTree.c

```
Expr* newIfElseExpr(Expr* cond, Expr* ifCase, Expr* elseCase){
  return NULL; /*ToDo*/
}
```

- Schreiben Sie den Konstruktor für Schleifen:

SyntaxTree.c

```
Expr* newWhileExpr(Expr* cond, Expr* body){
  return NULL; /*ToDo*/
}
```

- Schreiben Sie den Konstruktor für Funktionsaufrufe:

SyntaxTree.c

```
Expr* newFunCallExpr(char* name, Expr** args, int argsNr){
  return NULL; /*ToDo*/
}
```

2.5 Destruktoren

Eine datenstruktur wie unser Santaxbaum besteht aus vielen Referenzobjekten im Heap. Jeder Konstruktor, der so einen Heapknoten erzeugt hat, hat hierzu die Funktion `malloc` verwendet. Wir müssen all diese Referenzen, wenn der Baum nicht mehr benötigt wird, mit einem Aufruf von `free` wieder löschen. Sinnvoller Weise schreibt man passend zu jeder Konstuktorfunktion eine Destrukterfunktion:

SyntaxTree.h

```
void deleteExpr(Expr* this);
void deleteBinExpr(BinExpr binExpr);
void deleteIfElseExpr(IfElseExpr expr);
void deleteWhileExpr(WhileExpr expr);
void deleteAssignExpr(AssignExpr expr);
void deleteFunCallExpr(FunCallExpr binExpr);
```

Allgemein lässt sich so eine Destruktorfunktion für den allgemeinen Baumknoten `Expr` definieren.

Hierzu wird ein `switch` über den Knotentyp gemacht und dann die entsprechende Destruktorfunktion aufgerufen.

SyntaxTree.c

```
void deleteExpr(Expr* this){
  switch (this->type){
  case BIN_OP: deleteBinExpr(this->expr.binExpr);break;
  case IF_ELSE: deleteIfElseExpr(this->expr.ifElseExpr);break;
  case WHILE: deleteWhileExpr(this->expr.whileExpr);break;
  case ASSIGN: deleteAssignExpr(this->expr.assignExpr);break;
  case FUN_CALL: deleteFunCallExpr(this->expr.funCallExpr);break;
  default:break;
  }
  free(this);
}
```

Aufgabe 2 Implementieren Sie jetzt die einzelnen Destruktorfunktionen:

SyntaxTree.c

```
void deleteBinExpr(BinExpr binExpr){
  //TODO
}
void deleteIfElseExpr(IfElseExpr expr){
  //TODO
}
```

```
void deleteWhileExpr(WhileExpr expr){
  //TODO
}
void deleteAssignExpr(AssignExpr expr){
  //TODO
}
void deleteFunCallExpr(FunCallExpr funCallExpr){
  //TODO
  }
```

3 Programme auswerten

Schließlich geht es darum, dass wir ein Programm auswerten. Es soll das Ergebnis, das aus einer Zahl bestehtg, berechnet werden.

Hierzu benötigen wir die Funktion `eval`.

SyntaxTree.h

```
long int eval(Expr* this, long int* env, Program p);
```

Sie erhält ein Array der Länge 100, in dem für die Variablen gespeichert ist, welchen Wert sie momentan enthält.

Die Funktion `eval` Funktion erhält folgende Parameter:

- `Expr* this`: einen Ausdruck, dessen Ergebnis auszuwerten ist.
- `long int* env`: einen Array der Länge 100, in dem für die 100 durchnummerierten Variablen Werte gespeichert sind. Wir nennen diesen Array `env` für *enviroment*, der Umgebung, in der die Auswertung stattfindet.
- `Program p`: das Programm, bestehend aus einem Array der Funktionsdefinition. Um eine Funktion für einen Funktionsaufruf aufzulösen, muss die Funktion mit entsprechenden Namen hier gesucht werden. Ist allerdings einmal der Index der Funktion so gefunden worden, dann kann dieser in dem `FunCallExpr`-Knoten als `compiledNr` gespeichert werden. `compiledNr` ist auf `-1` gesetzt, wenn der Index in diesem Array noch nicht bekannt ist.

Die Funktion `eval` macht, so wie auch schion bei der allgemeinen Destruktorfunktion, ein Switch über die verschiedenen Baumknotenarten des Syntaxbaums und ruft eine für diesen Knoten spezialisierte Auswertungsfunktion auf:

SyntaxTree.c

```
long int eval(Expr* this, long int* env,Program p){
  switch (this->type){
  case LIT: return evalLiteralExpr(this->expr.literalExpr,env, p);
  case BIN_OP: return evalBinOp(this->expr.binExpr,env, p);
```

```
  case ASSIGN: return evalAssignExpr(this->expr.assignExpr,env, p);
  case VAR: return evalVariableExpr(this->expr.variableExpr,env, p);
  case WHILE: return evalWhileExpr(this->expr.whileExpr,env, p);
  case IF_ELSE: return evalIfElseExpr(this->expr.ifElseExpr,env, p);
  case FUN_CALL: return evalFunCallExpr(&(this->expr.funCallExpr),env, p);
  default: return -1;
  }
}
```

Wir benötigen also die folgenden Auswertungsfunktionen.

SyntaxTree.h

```
long int evalBinOp(BinExpr expr, long int* env,Program p);
long int evalVariableExpr(VariableExpr expr, long int* env,Program p);
long int evalLiteralExpr(LiteralExpr expr, long int* env,Program p);
long int evalIfElseExpr(IfElseExpr expr, long int* env,Program p);
long int evalWhileExpr(WhileExpr expr, long int* env,Program p);
long int evalAssignExpr(AssignExpr expr, long int* env,Program p);
long int evalFunCallExpr(FunCallExpr* expr, long int* env,Program p);

#endif
```

3.1 Auswertung von Zahlenliteralen

Am einfachsten sind Ausdrücke auszuwerten, die ein Zahlenliteral darstellen. Dann repräsentiert dieses Literal genau eine Zahl, die das entsprechende Ergebnis ist:

SyntaxTree.c

```
long int evalLiteralExpr(LiteralExpr expr, long int* env,Program p){
  return expr.value;
}
```

3.2 Auswertung von Variablen

Beim Auswerten einer Variablen, ist deren Wert in der Umgebung nachzuschlagen. Da die Variablen durchnummeriert sind, muss der Wert am entsprechenden Index in der Umgebung nachgeschlagen werden:

SyntaxTree.c

```
long int evalVariableExpr(VariableExpr expr, long int* env,Program p){
  return env[expr.varNr];
}
```

3.3 Auswertung von Verzweigungen

Um eine If-Verzweigung auszuwerten ist zunächst die Bedingung auszuwerten. Abhängig von deren Ergebnis, ist der *ifCase* oder der *elseCase* auszuwerten:

SyntaxTree.c

```
long int evalIfElseExpr(IfElseExpr expr, long int* env,Program p){
  return eval(expr.cond,env, p)
    ?eval(expr.ifCase,env, p)
    :eval(expr.elseCase,env, p);
}
```

Auch in C steht die Zahl 0 für *false* und alle anderen Zahlen für *true*.

Aufgabe 3 Implementieren Sie jetzt die Auswertungsfunktionen für die übrigen Konstrukte:

- Schreiben Sie die Auswertungsfunktion für Operatorausdrücke. Hierzu sind rekursiv die beiden Operanden auszuwerten. Dann ist ein Switch über den Operator zu machen, in dem entschieden wird, wie die beiden Operanden zum Ergbnis des Ausdrucks zu verrechnen sind.

  SyntaxTree.c

  ```
  long int evalBinOp(BinExpr expr, long int* env, Program p){
    //TODO
  }
  ```

- Schreiben Sie nun die Funktion, die die Zuweisung auswertet. Hierzu ist die rechte Seite `rhs` des Zuweisungsknotens auszuwerten. Dessen Ergebnis ist dann am passenden Index in der Umgebung `env` zu speichern. Das Ergebnis der Zuweisung soll wie in C oder Java auch, der zugewiesenen Wert sein.

  SyntaxTree.c

  ```
  long int evalAssignExpr(AssignExpr expr, long int* env, Program p){
    //TODO
  }
  ```

- Schreiben Sie nun die Funktion, die eine Schleife auswertet. Das Ergebnis ist hierbei nicht entscheidend. Sie müssen nur dafür sorgen, dass der Rumpf auch wirklich solange immer wieder ausgewertet wird, bis die Bedingung zu 0 auswertet.

  SyntaxTree.c

  ```
  long int evalWhileExpr(WhileExpr expr, long int* env, Program p){
    //TODO
  }
  ```

- Schließlich ist der wohl komplexeste Ausdruck auszuwerten. Der Aufruf einer Funktion. Gegeben ist der Code, der die korrekte Funktion aus dem Programm sucht und checkt, ob diese mit der passenden Parameterzahl aufgerufen wurde.

  Ergänzen Sie die Funktion um den offenen Part, in dem der Funktionsrumpf in einer neuen Umgebung mit den Werten der Argumente gesetzt ausgewertet wird.

SyntaxTree.c

```
long int evalFunCallExpr(FunCallExpr* expr, long int* env, Program p){
  //Schleifenvariable
  int i;

  //weiß ich noch nicht die Nummer der Funktion?
  if (expr->compiledNr<0){
    //durchlaufe die Funktionsdefinitionen
    for (i=0;i<p.funsNr;i++){
      //habe ich die aufgerufene Funktion gefunden?
      if (strcmp(expr->name, p.funs[i].name)==0){
        //dann merke ich mir die Nummer
        expr->compiledNr = i;
        break;
      }
    }
    // ich habe keine Funktion gefunden.
    if (i>=p.funsNr){
      fprintf(stderr,"unknown function: %s\n",expr->name);
      return -1;
    }
  }
  // ich hole die Definition der aufgerufenen Funktion
  FunDef* fun = p.funs+(expr->compiledNr);

  //falsche Argumentanzahl!
  if (fun->argsNr != expr->argsNr){
    fprintf( stderr
           ,"wrong nr of arguments for function %s: expected %d but found %d\n"
           , expr->name
           , fun->argsNr
           , expr->argsNr);
    return -1;
  }else{
    /* TODO
    Erzeugen Sie eine neue Umgebung für 100 long int.

    Iterieren Sie über die args von fun.
    Setzen Sie darin für jedes Argument, das Ergenis seiner Auswertung am
    passenden Index in der neuen Umgebung

    Werten Sie den Funktionsrumpf in der neuen Umgebung aus.

    Löschen Sie die neue Umgebung.
```

```
      Geben Sie das Ergebnis der Auswertung des Rumpfs als Ergebnis zurück.
       */

    }

  }
```

Ein kleiner Beispielaufruf der Funktionen:

Ex1.c

```c
#include "SyntaxTree.h"
#include <stdio.h>

int main(){
  long int env[] = {0,0,0,0,0,0,0,0,0,0,0,0,0};
  Program p = {NULL,0};

  Expr* e1  = newLiteralExpr(42);
  printf("%ld\n",eval(e1,env,p));
  deleteExpr(e1);

  Expr* e5 = newIfElseExpr(newGtExpr(newLiteralExpr(4),newLiteralExpr(0))
                           ,newLiteralExpr(42)
                           ,newDivExpr(newLiteralExpr(1),newLiteralExpr(0)));
  printf("%ld\n",eval(e5,env,p));
  deleteExpr(e5);
}
```

Anbei finden Sie auch einen Lexer und einem Parser, mit dem ein Quelltext gelesen und der entsprechende anstrakte Syntaxbaum erzeugt wird. Diese sind mit den Tools `flex` und `bison` (oder `lex` und `yacc` zu bauen.) Beigelegte sind aberbereits die von diesen Programmen generierten Quelltexte.

Zusammen mit dem Programm `LLInterpreter.h` erhalten Sie so einen interaktiven Interpreter für die Programmiersprache LL.

Mit dem beigelegten Kommandozeilenprogramm `LLInterpreter` können Sie eine LL-Datei laden und dann dann Ausdrücke auswerten lassen. Hier eine kleine Beispielsession zur Arbeit mit dem LLInterpreter.

Shell

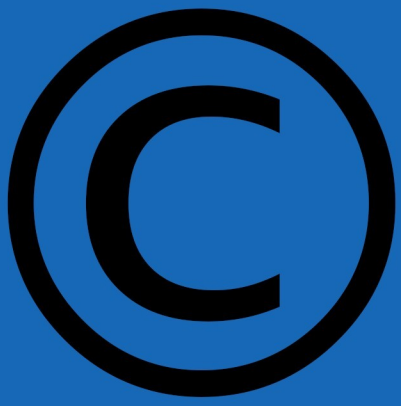
```
panitz@panitz-ThinkPad-T430:~/ll$ ./LLInterpreter test3.ll
parsing file test3.ll

Welcome. Type :? for help.
>: :?
Commands:
  :q   quit the programm
  :l   list defined functions
```

```
  :?   this help
  1+1  evaluate expression
>: :l
doppelt
fac
>: 1+1

2
>: fac(doppelt(3))

720
>: :q
panitz@panitz-ThinkPad-T430:~/ll$
```



Baumimplementierung
mit Union, Struct und Enum
Interpreter für Ausdrücke
Parser und Lexer