Lernziele:
- Dynamische Datenstruktur für Ausdrücke
- Umgang mit Algebraischen Datentypen (ADTs)
- Objektorientierte Programmierung (OOP) in einer Sprache, die nativ nicht OOP unterstützt
- Vertiefung der Programmierkenntnisse

Vorbereitung: Compiler aktualisieren

1. Wechselt in das Verzeichnis **abc-llvm** mit dem Source-Code für den Compiler.
2. Führt das Kommando **git pull** aus.
3. Führt das Kommando **make** aus.
4. Führt das Kommando **sudo make install** aus.

Vorbemerkung

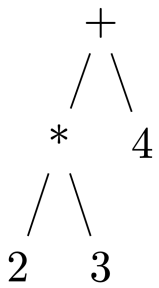
Wir werden hier tatsächlich weiter am Compiler arbeiten, auch wenn es zunächst nicht danach aussieht. In den ersten Schritten implementieren wir eine Datenstruktur für Ausdrücke (Expression-Trees). Diese Datenstruktur wird später in den Compiler integriert. Der Parser wird dann zum Beispiel nicht mehr direkt Assembler-Code erzeugen, sondern einen Expression-Tree erzeugen und zurückgeben. Diesen Expression-Tree kann man dann (wenn man will) optimieren und anschließend Assembler-Code erzeugen.

Schritt 1 (Algebraische Datenstruktur für Ausdrücke)

Analog zu verketteten Listen werden wir eine dynamische Datenstruktur implementieren. Im Gegensatz zu Listen können jedoch Knoten eine unterschiedliche Anzahl von Nachfolgern haben. Im ersten Entwurf werden wir zwei Arten von Knoten unterstützen:

- Binary-Nodes haben zwei Nachfolger.
- Primary-Nodes haben keinen Nachfolger.

Ziel wird es sein, einfache Expression-Trees zu erzeugen und auch mit solchen Bildern zu visualisieren:



Bevor man sich den Kopf zerbricht, wie diese Datenstruktur aussehen sollte, kann man sich erst einmal überlegen, wie man am Ende die Datenstruktur benutzen kann, um einen Baum wie oben anzulegen und zu visualisieren.

Dazu folgende Gedanken:

- Da es eine Datenstruktur ähnlich einer Liste ist, wird man am Ende irgendwie ein **struct Expr** deklarieren und wird irgendwie analog mit Zeigern auf Objekte vom Typ **Expr** arbeiten.
- Irgendwie sollte es möglich sein, einen Knoten für einen Integer-Wert anzulegen. Zum Beispiel soll mit

  ```
  createIntegerExpr(2)
  ```

  ein Knoten ohne Nachfolger für den Zahlenwert 2 angelegt werden. Die Funktion soll dann einen Zeiger darauf zurückgeben.

- Irgendwie sollte es möglich sein, einen Knoten für eine Summe oder ein Produkt anzulegen, also einen Knoten mit zwei Nachfolgern. Dazu sollte es eine Funktion geben, der man die Art der Operation (Addition oder Multiplikation) übergibt und Zeiger auf die zwei Operanden. Zum Beispiel sollte es dann mit

  ```
  createBinaryExpr(EXPR_MUL, createIntegerExpr(2), createIntegerExpr(3))
  ```

  möglich sein, einen Knoten anzulegen für eine Multiplikation, der zwei Nachfolger (einen linken und einen rechten Nachfolger) hat.
- Dann sollte es eine Funktion **printExprTree** geben, der man einen Zeiger auf den Expression-Tree übergibt und die dann das Bild dazu erzeugt.
- Und da man mit dynamischem Speicher arbeitet, muss man diesen bestimmt wieder freigeben. Also braucht man wahrscheinlich eine Funktion **releaseExpr**, der man einen Zeiger auf einen Expression-Tree übergibt und die sich darum irgendwie kümmert.

Ein erstes Ziel wäre also, dass man dann dieses Programm **xtest_expr.abc** benutzen könnte, um obigen Expression-Tree zu erzeugen und zu visualisieren:

```
1 @ <stdio.hdr>
2 @ "expr.hdr"
3
4 fn main()
5 {
6     local expr: -> Expr;
7
8     expr = createBinaryExpr(EXPR_ADD,
9                             createBinaryExpr(EXPR_MUL,
10                                              createIntegerExpr(2),
11                                              createIntegerExpr(3)),
12                             createIntegerExpr(4));
13     printExprTree(expr);
14     printf("\n");
15
16     releaseExpr(expr);
17 }
```

Listing 1: xtest_expr.abc

Nun kann man sich überlegen, wie man ein **struct Expr** und die notwendigen Funktionen deklarieren kann. Eine simple Idee ist, dass man zunächst mit einer enum-Deklaration symbolische Konstanten für die möglichen Knotenarten deklariert:

```
enum ExprKind
{
    EXPR_ADD,
    EXPR_MUL,

    EXPR_INTEGER,
};
```

und dann im **struct Expr** alle Informationen vereint, die man für die verschiedenen Knotenarten benötigt:

```
struct Expr
{
    kind:           ExprKind;
    left, right:    ->Expr;
    decimal:        u64;
};
```

Mit dem Member **kind** kann dann festgelegt werden, ob es sich um einen Binary-Node handelt (Werte **EXPR_ADD** oder **EXPR_MUL**) oder einen Primary-Node (Wert **EXPR_INTEGER**). Auf die Member **left** und **right** sollte man dann in einem Binary-Node zugreifen, und auf **decimal** nur in einem Primary-Node.

Damit kann man bereits wie folgt einen Header expr.hdr erstellen für den ersten Entwurf dieser Datenstruktur:

```
enum ExprKind
{
    EXPR_ADD,
    EXPR_MUL,

    EXPR_INTEGER,
};

struct Expr
{
    kind:           ExprKind;
    left, right:    ->Expr;
    decimal:        u64;
};

// constructors (caller becomes owner)
extern fn createBinaryExpr(kind: ExprKind, left: ->Expr, right: ->Expr): ->Expr;
extern fn createIntegerExpr(val: u64): ->Expr;

// destructor
extern fn releaseExpr(expr: ->Expr);

// meothods
extern fn printExprTree(expr: ->Expr);
```

Listing 2: expr.hdr

Aufgaben:

1. Schreibt das obige Testprogramm **xtest_expr.abc** und die Headerdatei **expr.hdr** ab. Bevor ihr in der nächsten Teilaufgabe mit der Implementierung der Datenstruktur beginnt, überprüft, dass der Compiler das Testprogramm ohne Fehler oder Warnungen mit der Option "-c" (nur ein Object-File erzeugen nicht linken)

   ```
   abc xtest_expr.abc -c
   ```

   übersetzt und ein Objekt-File erzeugt, es also insbesondere keine Tippfehler gibt.
2. Nachdem das Interface steht und sich mit dem Testprogramm verträgt, könnt ihr mit der Implementierung der Datenstruktur beginnen. Was in der Regel selten funktioniert (oder zumindest viel Zeit kostet), ist eine Vorgehensweise, bei der man erst alle Funktionen implementiert und dann testet, ob das Programm funktioniert.

   Meine Empfehlung ist, hier mit sogenannten Stubs anzufangen. Das sind Implementierungen, die im Prinzip nichts tun außer den Zweck zu erfüllen, dass man ein ausführbares Programm hat. Verwendet zum Beispiel die in Listing 3 gezeigten Stubs. Um Tippfehler auszuschließen, übersetzt das Programm mit

   ```
   abc xtest_expr.abc expr.abc
   ```

   und führt es aus.

3. Als erste Funktion könnt ihr nun zum Beispiel **createInteger** implementieren und dann gleich testen, ob sich das Programm übersetzen und ausführen lässt. Zu sehen ist dann noch nichts. Aber so kann man zumindest frühzeitig Tippfehler ausschließen. Und man be-

```
fn createIntegerExpr(val: u64): ->Expr
{
    local n: -> Expr = malloc(sizeof(*n));
    assert(n);
    n->kind = EXPR_INTEGER;
    n->decimal = val;
    return n;
}
```

```
@ <stdio.hdr>
@ "expr.hdr"

fn createBinaryExpr(kind: ExprKind, left: ->Expr, right: ->Expr): ->Expr
{
    printf("not implemented\n");
    return left;
}

fn createIntegerExpr(val: u64): ->Expr
{
    printf("not implemented\n");
    return nullptr;
}

fn releaseExpr(expr: ->Expr)
{
    printf("not implemented\n");
}

fn printExprTree(expr: ->Expr)
{
    printf("not implemented\n");
}
```

Listing 3: expr.abc (mit stubs)

merkt frühzeitig, dass für die Funktion malloc der Header stdlib.hdr eingebunden werden muss ;-).

4. Da jetzt zumindest eine Knotenart angelegt werden kann, bietet es sich an, als nächstes die Funktion **printExprTree** zu implementieren. Damit man sieht, dass das Programm tatsächlich etwas tut. Folgende Implementierung könnt ihr übernehmen (damit kann später der Expression-Tree leicht visualisiert werden):

```
fn printExprTree(expr: ->Expr)
{
    if (expr->kind == EXPR_INTEGER) {
        printf(" [ %llu ]", expr->decimal);
    } else if (expr->kind >= EXPR_ADD && expr->kind <= EXPR_MUL) {
        printf(" [ ");
        switch (expr->kind) {
            case EXPR_ADD: printf("+"); break;
            case EXPR_MUL: printf("*"); break;
            default: assert(0);
        }
        printExprTree(expr->left);
        printExprTree(expr->right);
        printf("] ");
    }
}
```

Die Idee der Implementierung ist folgende:

- Hat ein Primary-Node zum Beispiel den Wert 42, dann wird **[42]** ausgegeben.
- Bei einem Binary-Node wird (nach einem [) erst der Operator (+ oder *), dann der linke Nachfolgeknoten und dann der rechte Nachfolgeknoten ausgegeben (und dann ein]).

Es werden also Ausgaben der Art **[* [2] [3]]** oder **[+ [* [2] [3]] [4]]** erzeugt, die textuell den Expression-Tree darstellen.

5. Überlegt euch nun, wie man **createBinaryExpr** implementieren kann! Im Erfolgsfall sollte das Programm folgende Ausgabe erzeugen:
   [+ [* [2] [3]] [4]]
   not implemented

6. Es wäre jetzt natürlich richtig und wichtig, die Funktion **releaseExpr** zu implementieren. Aber noch wichtiger ist es, zu testen, dass unsere Datenstruktur tatsächlich eine Baumstruktur wie anfangs gezeigt darstellen kann, also überhaupt von Nutzen ist. Deshalb visualisieren wir die Ausgabe des Programms zunächst mit Hilfe von LaTeX.

   Schreibt in eine Datei **tree_header.tex** folgenden LaTeX-Code:

   ```
   \documentclass[preview, margin=0.2cm]{standalone}
   \usepackage{forest}
   \begin{document}
   \begin{forest}
   ```

   Und in eine Datei **tree_footer.tex** diese Zeile:

   ```
   \end{forest}
   \end{document}
   ```

   Danach könnt ihr mit folgendem Kommando (den Namen **a.out** ggf. mit dem Namen eures ausführbaren Programms ersetzen) die Datei **tree.tex** erzeugen:

   ```
   (cat tree_header.tex; ./a.out ; cat tree_footer.tex) > tree.tex
   ```

   Dabei wird der Inhalt von **tree_header.tex**, die Ausgabe von **a.out** und der Inhalt von **tree_-footer.tex** in dieser Reihenfolge in die Datei **tree.tex** geschrieben. Mit folgendem Kommando kann dann die PDF-Datei **tree.pdf** mit der Visualisierung erzeugt werden (falls LaTeX installiert ist):

   ```
   lualatex tree.tex
   ```

7. Nachdem die Datenstruktur nützlich zu sein scheint, lohnt es sich, die Funktion **releaseExpr** zu implementieren. Beachtet dabei, dass der Speicher eines Knotens erst dann freigegeben werden sollte, wenn für jeden Nachfolgeknoten der Speicher schon freigegeben ist. Hier kommen also, wie bei **printExprTree**, rekursive Aufrufe ins Spiel. Und tatsächlich kann man hier die Implementierung von **printExprTree** zunächst übernehmen und dann anpassen (statt einen Knoten auszugeben, wird dann der Speicher freigegeben).

Schritt 2: gitHub!

Wir haben jetzt im Compiler-Projekt mehrere Bausteine (oder Baustellen):
- Einen Lexer (bereits in Header-File, Source-File und Testprogramm zerlegt)
- Einen Parser mit Code-Generator (noch alles in einem Sourcefile)
- Eine Datenstruktur für Expression-Trees (bereits in Header-File, Source-File und Testprogramm zerlegt)

Alles muss nur noch etwas verbessert werden und mit einander integriert werden. Damit bei dem Projekt dabei nicht den Überblick verlieren sollten wir dafür eine Versionsverwaltung wie git benutzten. Wer sich mit git bereits auskennt soll einfach ein Repository anlegen mit allem was zum Projekt gehört (also simple.isa, lexer.hdr, lexer.abc, xtest_parser.abc, expr.hdr, expr.abc und xtest_expr.abc).

Für alle anderen gibt es hier eine Anleitung für die ersten Schritte mit git. Dabei wird der Web-Service gitHub verwendet (man kann natürlich auch Alternativen wie zum Beispiel gitLab verwenden). Legt euch beim Web-Service eurer Wahl einen Account an.

Anleitung für gitHub: Repository anlegen

1. GitHub öffnen und anmelden
   - Gehe zu GitHub und melde dich mit deinen Zugangsdaten an.
2. Neues Repository erstellen
   - Klicke oben rechts auf das "+"-Symbol und wähle "New repository" aus.
3. Repository-Details eingeben
   - Gib im Feld "Repository name" abc-project ein.
   - Wähle die Option "Public" oder "Private", je nach Wunsch.
   - Setze ein Häkchen bei "Add a README file".
   - Klicke auf "Create repository".

4. Dateien zum Repository hinzufügen
   - Zum neuen Repository navigieren
     - Nach der Erstellung wirst du automatisch zur Hauptseite des neuen Repositories weitergeleitet.
   - Dateien hochladen
     - Klicke auf die Schaltfläche "Add file" und wähle "Upload files" aus.
   - Dateien auswählen
     - Ziehe die Dateien expr.hdr, expr.abc und test_expr.abc in den Upload-Bereich oder klicke auf "choose your files", um sie manuell auszuwählen.
   - Änderungen committen
     - Scrolle nach unten und gib eine Commit-Nachricht ein, z.B. "Add initial project files".
     - Klicke auf "Commit changes".
5. Dateien überprüfen
   - Nach dem Commit wirst du zur Hauptseite des Repositories weitergeleitet.
   - Hier solltest du nun die Dateien **expr.hdr**, **expr.abc** und **test_expr.abc** sehen.
6. Terminal öffnen und SSH-Schlüssel generieren
   - Gib den folgenden Befehl ein und drücke Enter. Ersetze your_email@example.com durch deine GitHub-E-Mail-Adresse:
     `ssh-keygen -t ed25519 -C "your_email@example.com"`
7. Aufforderungen folgen:
   - Du wirst aufgefordert, einen Speicherort für den Schlüssel anzugeben. Drücke Enter, um den Standardpfad zu akzeptieren.
   - Optional: Gib ein sicheres Passwort ein und bestätige es
8. SSH-Schlüssel zu GitHub hinzufügen
   - Öffne den öffentlichen Schlüssel in deinem Texteditor oder gib ihn auf dem Terminal mit **cat** aus. Standardmäßig befindet sich dieser im Verzeichnis **~/.ssh** und hat die Endung **.pub**:
     `cat ~/.ssh/id_ed25519.pub`
   - Kopiere den gesamten Inhalt des Schlüssels.
9. SSH-Schlüssel zu GitHub hinzufügen
   - Gehe zu GitHub und melde dich an.
   - Klicke oben rechts auf dein Profilbild und wähle "Settings" aus.
   - Im linken Menü wähle "SSH and GPG keys" und dann die Schaltfläche "New SSH key".
   - Gib einen Titel ein (z.B. "My Laptop") und füge den kopierten öffentlichen Schlüssel in das Feld "Key" ein.
   - Klicke auf "Add SSH key".
10. SSH-Verbindung zu GitHub testen
    - Führe den folgenden Befehl im Terminal aus, um die SSH-Verbindung zu GitHub zu testen:
      `ssh -T git@github.com`
    - Wenn alles korrekt eingerichtet ist, solltest du eine Nachricht wie diese erhalten:
      `Hi yourusername! You've successfully authenticated, but GitHub does not provide shell access.`
11. Repository mit SSH-URL klonen
    - Repository-URL ändern
      - Öffne die Seite deines neuen Repositories auf GitHub.
      - Klicke auf den grünen Button "Code" und wähle "SSH" aus.
      - Kopiere die SSH-URL des Repositories.
    - Repository klonen
      - Klone das Repository mit der SSH-URL:
        `git clone git@github.com:yourusername/abc-project.git`
    - Damit sollte eine Unterverzeichnis abc-project angelegt werden mit allen Dateien des Repositories. Kopiert in das Verzeichnis die weiteren Files **lexer.hdr, lexer.abc, xtest_parser.abc, simple.isa**
    - Weitere (restlichen) Dateien hinzufügen und pushen
      - Wechsel in das geklonte Verzeichnis: `cd abc-project`
      - Füge die Dateien hinzu, committe und pushe sie:
        ```
        git add lexer.hdr, lexer.abc, xtest_parser.abc simple.isa
        git commit -a
        git push
        ```
      - Bei "git commit -a" wird der Text-Editor vim gestartet: Drückt "i" um in den Einfügemodus zu gelangen, dann gebt "Initial commit" ein, dann drückt Esc und gebt ":wq" ein.