Visualisierung der Syntaxanalyse in der erweiterbaren Programmiersprache MOSTflexiPL

Autor:

Lukas Pietzschmann (76010)

Betreuer:

Prof. Dr. Christian Heinlein

Abgabe:

28. Februar 2022

Hochschule Aalen

Studiengang Informatik - Software Engineering

Inhaltsverzeichnis

Seite

1 Einleitung **7**

1.1 Motivation 7

1.2 Ziel 7

2 Grundlagen **8**

2.1 Funktionsweise der Syntaxanalyse des MOSTflexiPL-Compilers 8

2.2 ncurses 10

2.2.1 Initialisierung 11

2.2.2 Farben 11

2.2.3 Tastatur Eingabe 12

2.2.4 Fenster 13

2.2.5 Programm Ende 15

3 Anforderungen **16**

3.1 Nicht-funktionale Anforderungen 16

3.2 Funktionale Anforderungen 16

4 Implementierung **17**

4.1 Eingriffe in den Parser 17

4.2 Entwicklung einer Schnittstelle 19

4.2.1 Anforderungen an die Schnittstelle 19

4.2.2 Entwurf der Schnittstelle 19

4.2.3 Implementierung der Schnittstelle und Events 20

4.3 Implementierung der GUI 23

4.3.1 Aufbau der Oberfläche 24

4.3.2 Spezielle Fenster 25

4.3.3 Scrolling 26

4.3.4 Popups 32

4.3.5 Ausdrucks-Warteschlange 34

4.3.6 Liste aller Operatoren 37

4.3.7 Archiv Rendering 40
4.3.8 Layouting 47
4.3.9 Interaktion zwischen allen Komponenten 52
4.3.10 Hilfsklassen und kleinere Details 58

5 Probleme **61**

Abbildungsverzeichnis

1 Ablauf des Parsers . . . 10
2 Benutzeroberfläche . . . 24
3 Scrolling . . . 26
4 Interner Ablauf des neu Zeichnens eines Archivs . . . 41
5 Berechnung der neuen x-Koordinate im Layouting . . . 50
6 Eingabe-Zustände . . . 55

Tabellenverzeichnis

1 Archiv Beispiel . . . 8
2 Queue Beispiel . . . 9

Listings

1 Initialisierung der Farben . . . 12
2 Callgrind Dateiformat . . . 20
3 events.hpp . . . 20
4 event.hpp (Alle Events) . . . 21
5 event.hpp - Implementierung event und event_with_data . . . 22
6 window_like.hpp . . . 25
7 scrollable::prepare_refresh #1 . . . 28
8 scrollable::prepare_refresh #2 . . . 30
9 scrollable::scroll_y . . . 31
10 scrollable::scroll_y - is_allowed_to_scroll . . . 31
11 popup.hpp . . . 32
12 popup_manager.hpp . . . 33
13 popup_manager::show . . . 34
14 expr_queue.hpp . . . 35
15 expr_queue::push_back . . . 36
16 expr_queue::pop_front . . . 37

17 oper_store.hpp 38
18 oper_store::insert_if_prototyp 39
19 oper_store::get_id_from_oper 40
20 archive::add_comp 42
21 archive::render #1 43
22 archive::render #2 43
23 archive::render #3 45
24 archive::invalidate #1 46
25 archive::invalidate #2 47
26 layouter.hpp 48
27 layouter::notify_dimensions_changed 48
28 layouter::notify_dimensions_changed - layout 49
29 layouter::notify_dimensions_changed - has_intersections 51
30 Auszug aus setup_windows 53
31 step_n_events_forward 54
32 add_expr_to_queue_event::exec 57
33 add_expr_to_queue_event::undo 58
34 expr_repr::flags 60

Eidesstattliche Erklärung

Ich versichere, dass ich diese Ausarbeitung selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Das Gleiche gilt für beigefügte Skizzen und Darstellungen.

Hüttlingen, den 28. Februar 2022

Ort, Datum

Lukas Pietzschmann

1 Einleitung

1.1 Motivation

Die Arbeit von Compilern und/oder Interpretern (im Folgenden werden beide unter dem Begriff Compiler zusammengefasst) kann von außen oft als magisch bezeichnet werden. So versteht der Compiler auf wundersame Art und Weise, was der Entwickler vorher in Form von Programmcode niedergeschrieben hat. Doch was genau passiert während der Laufzeit eines Compilers?
Während die Antwort dieser Frage bei vielen Compilern sicher einfach in die bekannten drei Schritte lexikalische Analyse, syntaktische Analyse und semantische Analyse aufgeteilt werden kann, ist die Beantwortung in Bezug auf den MOSTflexiPL-Compiler nicht ganz so simpel.
Der MOSTflexiPL-Compiler gliedert sein Schaffen nämlich nicht in drei disjunkte Schritte, sondern versucht, umgangssprachlich gesagt, alles in einem zu machen. So wird gleichzeitig der Eingabetext gescannt, als auch versucht, einen korrekten Syntaxbaum aufzuspannen.
Zusätzlich dazu kommt noch die Flexibilität der Sprache selbst. So ist es möglich, nahezu beliebigen Text als Operator zu definieren. Möchte man als Entwickler also beispielsweise zusätzlich zu dem Infix-Operator $*$ noch einen Präfix-Operator $*$, ist dies problemlos möglich. Nun können dadurch aber auch leicht, gerade bei nicht exakt definierten Vorrängen, Mehrdeutigkeiten entstehen. Terminiert der Compiler also andauernd aufgrund einer, vom Entwickler nicht einfach zu erkennenden, Mehrdeutigkeit, wird die eingangs angesprochene Magie schnell zum Frust.

1.2 Ziel

Diese Arbeit möchte nun also eine Möglichkeit schaffen, die Syntaxanalyse des MOSTflexiPL-Compilers zu entmystifizieren.
Dazu soll visuell ausgegeben werden, welche Ausdrücke aktuell aus welchem Teil des Programmcodes entstehen und ob und wie diese Ausdrücke zusammengesetzt werden. Außerdem soll es die Möglichkeit geben, nicht nur den aktuellen Zustand des Compilers zu betrachten, sondern auch, ähnlich zu einem klassischen Debugger, frühere Zustände erneut aufzurufen und in einzelnen Schritten zwischen diesen hin und her zu springen.
Dies soll nicht nur dabei helfen, dem Anwender aufzuzeigen, an welcher Stelle Mehrdeutigkeiten entstehen, sondern auch dem Entwickler des Compilers ein detaillierteres Bild der Funktionsweise des Parsers geben.

Die Visualisierung soll dabei aber nicht wie ein Debugger funktionieren, der das Programm (hier: der Parser) zeitweise anhält, sondern eher wie ein Profiler, der während der Laufzeit Informationen sammelt, um diese dann erst im Nachhinein darzustellen.
Eine detailliertere Beschreibung der tatsächlichen grafischen Ausgabe und deren Funktion wird später in der Arbeit folgen.

2 Grundlagen

Um ein klareres Bild zu bekommen, was genau bei der Visualisierung überhaupt dargestellt wird, beziehungsweise werden kann, erläutert dieses Kapitel zuerst, wie der Parser des MOSTflexiPL-Compilers vorgeht.

2.1 Funktionsweise der Syntaxanalyse des MOSTflexiPL-Compilers

Die zentrale und wichtigste Datenstruktur des Parsvorgangs sind sogenannte Archive. Ein Archiv ist wie eine Tabelle aufgebaut, in der neben bereits vollständig geparsten Ausdrücken, auch Ausdrücke gespeichert werden, die noch nicht abgeschlossen sind. Ein Archiv entsteht dabei immer an einer bestimmten Position im Eingabe-Quelltext. Entsteht ein Archiv beispielsweise an Position 1, bedeutet dies, dass dort alle vollständigen Ausdrücke beginnen und alle unvollständigen enden, beziehungsweise fortgesetzt werden müssen. Diese Position wird in den folgenden Tabellen stets durch einen senkrechten Strich, gefolgt von der tief gestellten Position ($|_1$) dargestellt. Vollständige Ausdrücke haben neben einer Startposition noch eine weitere: die Endposition. Diese wird hier durch einen senkrechten Strich, gefolgt von der hoch gestellten Position ($|^1$), dargestellt.

Vollst. / Unvollst.	$\|_1\ 1\ \|^2$	$\|_1 1+2 \|^4$
$\|_1$ _ - _	$1\|_2$ - _	$1+2\|_4$ - _
$\|_1$ _ + _	$1\|_2$ + _	$1+2\|_4$ + _
$1+\|_1$ _	$1+1\|_2$	X

Tabelle 1: Archiv Beispiel

Dieses Beispiel verdeutlicht den Aufbau eines Archivs. Links finden sich drei unvollständige Ausdrücke. Der Operator + und −, dem noch der linke und rechte Operand fehlt und ein teilweise vervollständigter Plus-Operator. In der oberen Zeile hingegen stehen ein bereits vollständiger Int-Literal Ausdruck und eine vollständige Anwendung des Plus-Operators. Nun wird versucht, diese in alle unvollständigen Ausdrücke einzusetzen. Das kann, wie in Zeile drei, aufgrund einer Präzedenzverletzung, oder ähnlichen Kriterien auch fehlschlagen.
Ist dieser Versuch allerdings erfolgreich, kommt nun eine zweite wichtige Komponente des Parsers ins Spiel:
So werden die neu erzeugten Ausdrücke in einer Queue zwischengespeichert.

1-_	1+_	1+1	1+2-_	1+2+_

Tabelle 2: Queue Beispiel

Nun werden nach und nach Elemente aus der Queue entnommen und fortgesetzt. Der Ausdruck $1|_2$-_ erwartet an Position 1 ein -, ist also fortsetzbar, falls im Quellcode an dieser Stelle auch tatsächlich ein - anliegt. Erwartet ein Ausdruck allerdings einen Operanden, so kann er nicht einfach fortgesetzt werden. Um den benötigten Operanden zu lesen, muss der Ausdruck in ein passendes Archiv geschrieben werden. Passend bezieht sich dabei auf die Position des Archivs, die mit der Position, zu der der Ausdruck fortgesetzt werden konnte, übereinstimmen muss. Konnte der Ausdruck vollständig fortgesetzt werden, wird er ebenfalls in ein passendes Archiv eingetragen. Nun kann es vorkommen, dass kein Archiv zur gewünschten Position gefunden werden konnte. In diesem Fall werden zwei Aktionen angestoßen: Einerseits wird das benötigte Archiv erstellt, zusätzlich werden aber auch Prototyp-Ausdrücke aller definierten Operatoren erstellt und in die Warteschlange eingestellt. Ein Prototyp-Ausdruck ist dabei beispielsweise _-_ aus dem Beispiel oben.
Unabhängig davon, ob das Archiv neu erstellt werden musste und ob der Ausdruck komplett fortgesetzt werden konnte, oder nicht, werden im Archiv nun wieder voll- und unvollständige Ausdrücke kombiniert und die Resultate in die Queue gestellt und somit beginnt die Schleife wieder von vorne.
Nach einer gewissen Anzahl an Schritten muss die Warteschlange a priori leer sein, da alle dort eingestellten Ausdrücke abgearbeitet wurden. Wird dieser Zustand erreicht, ist der Parser am Ende und muss nun entscheiden, welcher der, während der Laufzeit erzeugten und in den Archiven abgespeicherten, Ausdrücke derjenige ist, der das komplette Programm abbildet. Hierzu

werden alle vollständigen Ausdrücke des Archivs an Position 0 betrachtet. Endet nun einer dieser Ausdrücke an Position n, wobei n die Länge des Eingabestrings ist, muss dieser Ausdruck der gewünschte sein. Werden mehrere solche Ausdrücke gefunden, ist das eingegebene Programm nicht eindeutig und der Parser terminiert mit einem Fehler.[Hei21a]

Der folgende Graph veranschaulicht das Arbeiten des Parsers noch einmal anschaulich visuell. Wichtig: Die obige Beschreibung beginnt nicht bei *Einstiegspunkt*, sondern links unten bei *Voll- und unvollständige Ausdrücke kombinieren*!

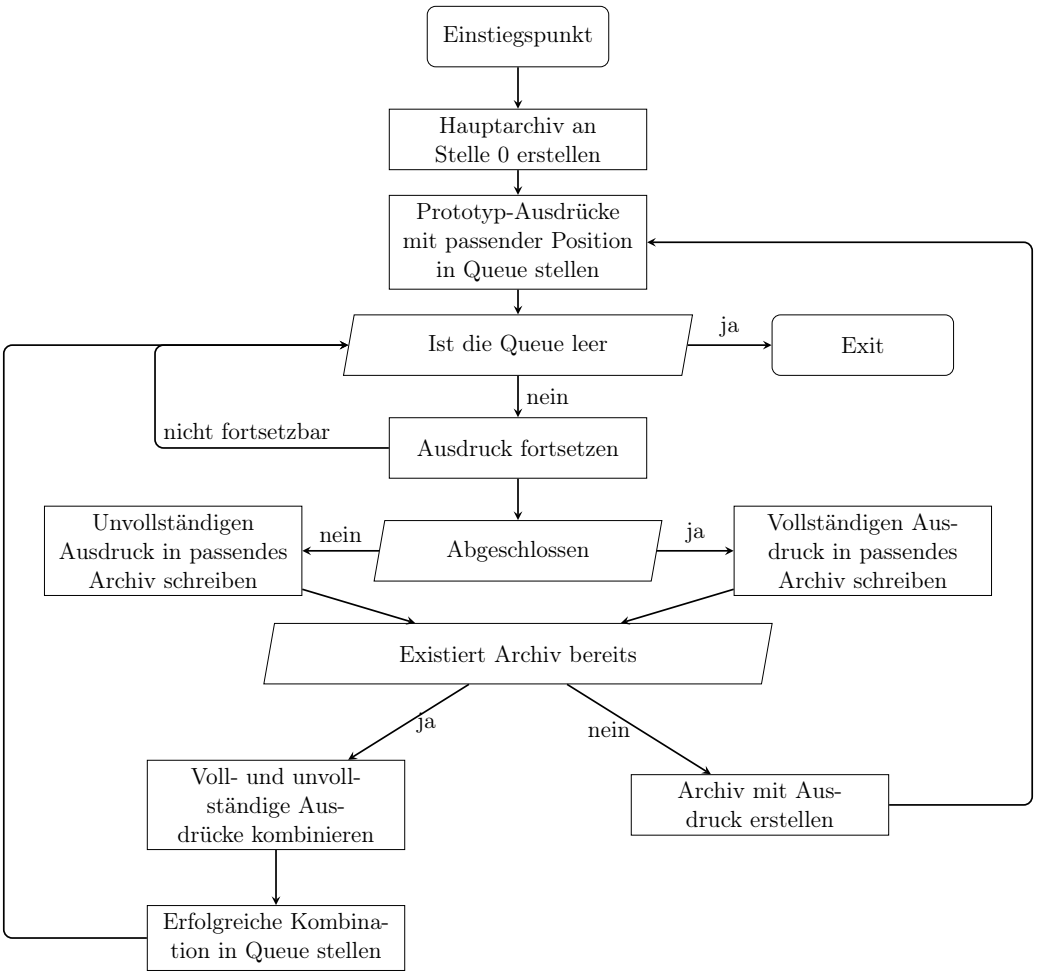


Abbildung 1: Ablauf des Parsers

2.2 ncurses

new **c**urses (kurz: ncurses) ist eine C Bibliothek zum Erstellen textbasierter Benutzeroberflächen. ncurses stellt neben Schnittstellen zum Erzeugen und Verwalten von mehreren Fenstern, auch

Möglichkeiten zur Erkennung von Tastatur und sogar Maus-Eingaben zur Verfügung. All diese Schnittstellen lassen sich über den Header `ncurses.h` einbinden.

Im Folgenden werden einige grundlegende, und für das Projekt essenzielle, Abstraktionen und Prozeduren beschrieben.

2.2.1 Initialisierung

In den meisten Fällen wird `WINDOW* initscr()` die erste ncurses Funktion sein, die in einem Programm aufgerufen wird. Sie initialisiert alle intern verwendeten Datenstrukturen und führt den ersten `refresh()` aus, um den bisherigen Inhalt des Terminals zu löschen. Wurde diese Funktion aufgerufen, können nun alle weiteren ncurses-Funktionen verwendet werden.

2.2.2 Farben

Typischerweise werden als Nächstes Farben initialisiert. Hierfür wird die ncurses-Prozedur `int start_color()` verwendet. Wichtig: Unterstützt das verwendete Terminal keine Farben, liefert `bool has_colors() false` und alle weiteren Verwendungen von farbbezogenen Funktionen werden keine Wirkung haben.
Farben werden immer in Paaren von Vordergrund- und Hintergrundfarbe verwendet. Diese Paare sind über eine eindeutige Zahl identifizierbar und können mit `int init_pair(short id, short f_id, short b_id)` erzeugt werden. Diese ID's werden typischerweise über define-Direktiven gesetzt, so sind beispielsweise `COLOR_WHITE`, `COLOR_BLACK`, oder `COLOR_RED` bereits vordefiniert. Eine "neue" Farbe kann mithilfe der Funktion `int init_color(short id, short r, short g, short b)` registriert werden. Die letzten drei Parameter sind dabei RGB Werte, allerdings nicht im gewohnten Wertebereich von 0 bis 255, sondern hier von 0 bis 1000!

Listing 1: Initialisierung der Farben

```
1  start_color();
2  init_color(COLOR_LIGHT_GREY, 600, 600, 600);
3  init_color(COLOR_GREY, 400, 400, 400);
4  init_pair(STD_COLOR_PAIR, COLOR_WHITE, COLOR_BLACK);
5  init_pair(FOOTER_COLOR_PAIR, COLOR_BLACK, COLOR_RED);
6  init_pair(HEADER_COLOR_PAIR, COLOR_WHITE, COLOR_GREY);
7  init_pair(HIGHLIGHT_EXPR_COLOR_PAR, COLOR_GREEN, COLOR_BLACK);
8  init_pair(MUTED_COLOR_PAIR, COLOR_LIGHT_GREY, COLOR_BLACK);
9  init_pair(AMBIGUOUS_COLOR_PAIR, COLOR_BLACK, COLOR_RED);
10 wbkgd(stdscr, COLOR_PAIR(STD_COLOR_PAIR));
```

Listing 1 - Initialisierung der Farben zeigt, wir in diesem Projekt einzelne Farben und Farbpaare erstellt werden. `wbkgd(stdscr, COLOR_PAIR(STD_COLOR_PAIR))` setzt dabei den Forder- und Hintergrund des Hauptfensters `stdsrc` (standard screen).

2.2.3 Tastatur Eingabe

Als nächstes wird die Eingabe über die Tastatur konfiguriert. Hierfür ist `int getch()` die wichtigste Funktion. Diese verhält sich, je nach vorher konfigurierten Einstellungen anders. Folgende Modi-Gruppen existieren:

Wie lange wird auf Eingaben gewartet?

no-delay

Liegt beim Aufrufen von `getch` kein Eingabetext an, wird ein Fehler zurückgegeben. Andernfalls wird das gelesene Zeichen zurückgegeben.

helf-delay

Liegt in diesem Modus kein Eingabetext an, wird um die, vorher mit `int halfdelay(int tenths)` gesetzte Anzahl an Zehntelsekunden gewartet, bevor ein Fehler zurückgegeben wird.

delay

Hier wartet `getch` solange, bis Text anliegt. Es wird also nie ein Fehler zurückgegeben.

Wann wird eine Eingabe zurückgegeben?

cbreak

In diesem Modus werden eingegebene Zeichen einzeln zurückgegeben.

nocbreak

Hier wird auf einen Zeilenumbruch gewartet, bis Text zurückgegeben wird.

Durch das Aufrufen von ...

```
1 cbreak();
2 timeout(-1);
```

... werden in diesem Projekt der cbreak und delay Modus gesetzt.

2.2.4 Fenster

Die meisten bisher angesprochenen Funktionen arbeiten auf dem `stdsrc`, ncurses erlaubt es dem Entwickler aber auch eigene Fenster zu erstellen und zu manipulieren. Um das zu verwendende Fenster spezifizieren zu können, bieten einige Funktionen eine, durch den Buchstaben "w" geprefixte Variante.

getch()	wgetch(WINDOW* window)
refresh()	wrefresh(WINDOW* window)
...	...

Achtung: Funktionen ohne ein vorstehendes "w" sind meistens as Makro implementiert!

Erstellen von Fenstern

Fenster können mit der Funktion `WINDOW* newwin(int height, int width, int begin_y, int begin_x)` erstellt werden. `begin_y` und `begin_x` sind dabei die absoluten Koordinaten der linken oberen Ecke des neu erstellten Fensters. Der Rückgabewert mit dem Type `WINDOW*` repräsentiert das neue Fenster und sollte gespeichert werden um später in "w-Funktionen" verwendet werden zu können.

Manipulieren von Fenstern

Content

Um Text zu Fenstern hinzuzufügen, existieren einige Funktionen. Die wichtigsten beiden sind dabei `int mvwaddstr(WINDOW* win, int y, int x, const char* str)` und `int mvwaddch(WINDOW* win, int y, int x, const char ch)`. Beide setzen den Cursor des angegebenen Fensters an die spezifizierte `y` und `x`-Position und schreiben ab dieser Stelle den übergebenen Text.

Text wieder zu löschen ist etwas mühsamer. Dazu existieren die Funktionen `int wclrtobot(WINDOW* win)` und `int wclrtoeol(WINDOW* win)`. Während Letztere den Inhalt von der aktuellen Cursor-Position bis zum nächsten Zeilenumbruch löscht, löscht Erstere von der aktuellen Position bis zum Ende des Bildschirms.

Soll der komplette Bildschirm gelöscht werden kann `int werase(WINDOW* win)` verwendet werden.

Position und Größe

Soll ein Fenster an eine andere Position verschoben werden, wird `int mvwin(WINDOW* win, int y, int x)` verwendet. Wie schon `newin` beziehen sich `y` und `x` auch hier auf die absolute Position der linken oberen Ecke.

`int wresize(WINDOW* win, int height, int width)` ermöglicht das Ändern der Fenstergröße. Diese Funktion sollte allerdings mit Bedacht verwendet werden, da sie im Falle einer Vergrößerung neuen Speicher reservieren muss, was gerade in Performance kritischen Situationen nicht erwünscht sein kann.

Dekoration

Sollen Fenster, durch beispielsweise einen Rahmen, klar vom Hintergrund getrennt werden, stellt ncurses die Funktion `int box(WINDOW* win, char vert, char hor)` zur Verfügung. Hier können ASCII-Zeichen für die horizontalen und vertikalen Grenzen angegeben werden. Mit, unter andrem, den Makros `ACS_VLINE` und `ACS_HLINE` bietet ncurses bereits eigene Separatoren.

Nach jeder Änderung an einem Fenster, sollte diese auch dem Benutzer sichtbar gemacht werden. Die Funktion `int wrefresh(WINDOW* win)` muss aufgerufen werden, um die tatsächliche Ausgabe auf dem Terminal zu erhalten, da andere Funktionen lediglich interne Datenstrukturen manipulieren.

Sollen mehrere Fenster auf einmal aktualisiert werden, ist es wesentlich effizienter die Funktion `int wnoutrefresh(WINDOW* win)` für jedes einzelne Fenster, und am Ende `int doupdate()` einmalig, zu verwenden. Intern verwendet ncurses zwei Datenstrukturen um den Terminalbildschirm zu verwalten: Einen physischen Bildschirm, der beschreibt was aktuell tatsächlich angezeigt wird, und einen virtuellen Bildschirm, der den gewünschten Inhalt repräsentiert.
Mehrere Aufrufe von `wrefresh` würden nun intern immer `wnoutrefresh` verwenden, um das angegebene Fenster in den virtuellen Bildschirm zu kopieren, und anschließend, mit `doupdate`, den virtuellen mit dem physischen Bildschirm vergleichen und die tatsächlichen Änderungen auf das Terminal schreiben. So wird also mehrmals auf das Terminal selbst geschrieben.
Mit mehreren händischen Aufrufen von `wnoutrefresh` und einem einzelnen `doupdate` am Ende wird der virtuelle Bildschirm nur einmal mit dem physischen verglichen, ergo wird auch nur einmal auf das Terminal geschrieben!

Entfernen von Fenstern

Wird ein Fenster nicht mehr benötigt, sollte dieses auch explizit gelöscht werden, um den darunterliegenden Speicher wieder freizugeben. Achtung: Auch wenn `newwin` einen Zeiger auf das Fenster zurückgibt, reicht es nicht aus `delete window` beziehungsweise `free(window)` aufzurufen! Ncurses stellt hierzu eine eigene Funktion `int delwin(WINDOW* win)` zur Verfügung. Diese löscht zwar korrekt den dem Fenster zugeordneten Speicher, sie schwärzt aber nicht den Bereich, in dem das Fenster vorher zu sehen war. Dies sollte vorher händisch mit `werase` geschehen.

2.2.5 Programm Ende

Das Gegenstück zu `initscr` ist `int endwin()`. Diese Funktion sollte immer zum Schluss aufgerufen werden, um den ncurses-Modus zu beenden. Um dies sicher zu stellen, können, oder sollten, neben den bereits standardmäßig abgefangenen Signalen `SIGINT`, `SIGTERM` und `SIGTSTP`, auch `SIGABRT` und `SIGKILL` abgefangen und entsprechend behandelt werden.

3 Anforderungen

Da nun die Grundlagen für diese Arbeit besprochen sind, kann sich dieses Kapitel sowohl mit den funktionalen als auch den nicht-funktionalen Anforderungen diskutiert werden.

3.1 Nicht-funktionale Anforderungen

Die einzige, aber sicherlich trotzdem eine sehr wichtige, nicht-funktionale Anforderung bezieht sich auf die Performance. Die Applikation sollte sich auch dann flüssig anfühlen, wenn Events sehr schnell nacheinander ausgeführt werden. Auch sollte das Flickern des Bildschirms vermieden werden. Beides kann durch intelligentes und effizientes Aktualisieren von ncurses Fenstern erreicht werden.

3.2 Funktionale Anforderungen

Die erste Frage die sich bezüglich der Funktionen stellt, ist welche Informationen überhaupt angezeigt werden sollen.
Definitiv sollte der, von Parser bearbeitete, Quelltext dargestellt werden, einerseits, um den Benutzer daran zu erinnern, was die Eingabe war und andererseits, um die Archive daran auszurichten.
Denn diese Archive sollten auch angezeigt werden. Sie sind das wichtigste Element dieser Applikation und sollten in den Vordergrund gestellt werden. Sie sollten alle ihre voll- und unvollständigen Ausdrücke enthalten und Anzeigen, an welcher Position sie im Quelltext entstanden sind. Dies kann sowohl durch eine Nummer über dem Archiv geschehen als auch durch die Positionierung des Archivs genau unter dieser Stelle im Quelltext. Auch sollten Archive kenntlich machen, wann welche Ausdrücke kombiniert werden und ob ein neuer vollständiger Ausdruck eine Mehrdeutigkeit im Programm erzeugt.
Wird eine Mehrdeutigkeit erkannt, ist es oft schwer zu erkennen, welchen Haupt-Operator diese Mehrdeutigkeit hat. `print 1+2` kann sowohl an dem + Operator aufgehangen werden, als auch an dem `print`-Operator. Um diese Unterscheidung machen zu können, soll die Applikation auch eine Liste aller bisher gesehener Operatoren verwalten und diesen Operatoren eine eindeutige Nummer zuweise, die dann eben auch bei allen Ausdrücken angezeigt werden kann.
Auch sollte die Warteschlange der Ausdrücke dargestellt werden. Dies kann dabei helfen, zu

erkennen, zu was zwei Ausdrücke kombiniert werden und wo neue Ausdrücke in den Archiven herkommen.
Eine letzte, sinnvolle Kleinigkeit, könnte das Darstellen von möglichen Benutzereingaben sein. Dies würde neuen Benutzern dabei helfen, sich schnell in die Navigation in der Applikation einzugewöhnen.

4 Implementierung

Die Praktische Umsetzung des Projekts lässt sich grob in drei Phasen aufteilen.
Der erste Schritt war, erst einmal herauszufinden, an welchen Stellen im Parser die Visualisierung überhaupt eingreifen muss. Genau an diesen Stellen wird dann später eine Schnittstelle zwischen Parser und Visualisierung entstehen.
Ist dieser Schritt geschafft, kann die Schnittstelle entwickelt werden. Diese sollte dabei möglichst klein, beziehungsweise simpel sein, um so wenig wie möglich in den Parser-Code eingreifen zu müssen.
Steht nun die Architektur, die Parser und Visualisierung verbindet, kann zum Schluss die Visualisierung selbst entwickelt werden.

Die folgenden Unterkapitel werden nun genauer auf die einzelnen Schritte eingehen.

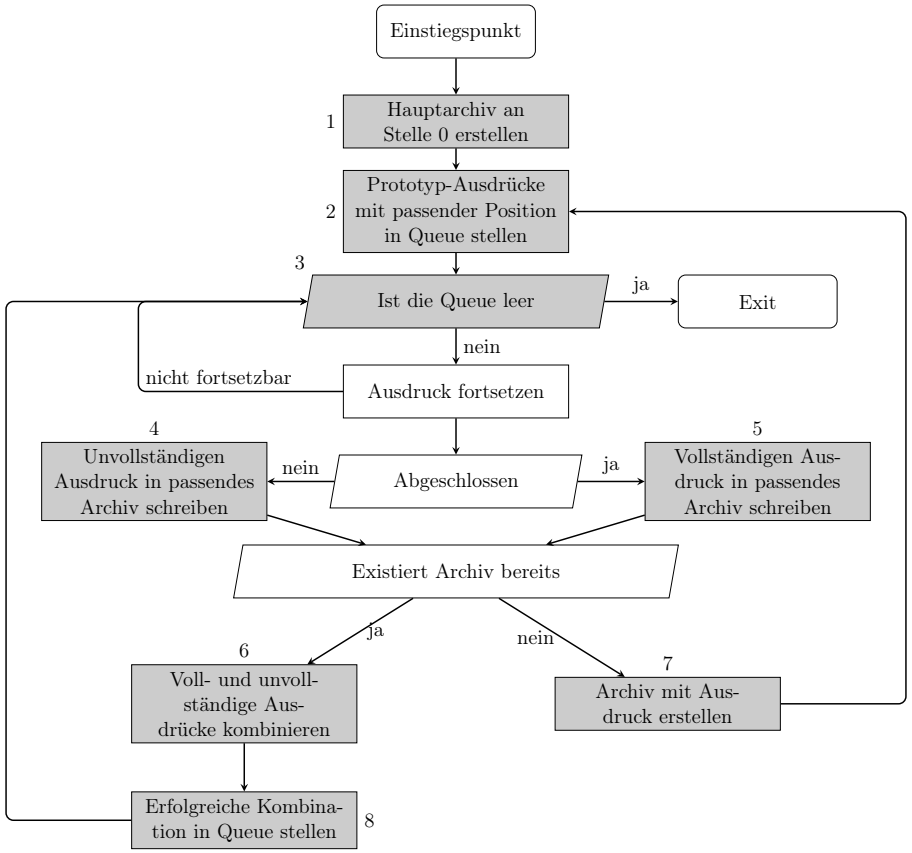
4.1 Eingriffe in den Parser

Um herauszufinden, wo genau in den Parser eingegriffen muss, müssen zuerst die Abläufe des Parsers betrachtet werden. Wie der Parser grundsätzlich arbeitet, wurde bereits in Abbildung 1 - Ablauf des Parsers und Unterabschnitt 2.1 - Funktionsweise der Syntaxanalyse des MOSTflexiPL-Compilers beschrieben. Betrachtet man die Abbildung, lässt sich schnell und leicht einsehen, an welchen Stellen die Visualisierung eingreifen muss. Dies sollte immer passieren, wenn ...

- ein Archiv erstellt wird
- ein Ausdruck in die Queue hineingestellt, oder herausgenommen wird
- ein voll- mit einem unvollständigen Ausdruck kombiniert wird
- ein voll-, oder unvollständiger Ausdruck in ein Archiv geschrieben werden

Wendet man dieses Wissen nun auf Abbildung 1 - Ablauf des Parsers an, stellt man fest, dass die

im Folgenden grau unterlegten Schritte genau diejenigen sind, in die die Visualisierung eingreifen muss.



Wie mapped nun aber diese Grafik auf den tatsächlichen Parser-Code?

Der Kern des Parsers besteht aus neun Funktionen: `arch`, `initial`, `proceed`, `extend` beziehungsweise `finish`, `subscribe` beziehungsweise `publish`, `combine` und `process`.
`arch` ist dabei mit die einfachste Funktion. Sie dient nur als Hilfsfunktion um ein Archiv an der gewünschten Stelle zurückzugeben, beziehungsweise ein neues an dieser Stelle zu erstellen. Sie kommt also an Stelle eins und sieben oben vor.
Die Aufgabe von `process` ist es die Queue nacheinander abzuarbeiten, und deren Elemente an `proceed` weiterzuleiten. Ergo findet der Aufrufe dieser Funktion an Position drei statt.
`proceed` hat nun zwei verschiedene Arbeitsweisen. Die Funktion kann sowohl rekursiv, als auch iterativ arbeiten. Von `process` wird die rekursive Version genutzt. Diese Variante versucht alle möglichen Fortsetzungen des Ausdrucks zu erstellen und diese an entsprechende Funktionen und Archive weiterzuleiten. Die Iterative Variante hingegen, bricht die Rekursion, durch das Einstellen des Ausdrucks in die Warteschlange, auf. Diese Variante wird also an Position zwei (über

`initial`) und acht (über `combine`) verwendet.
`extend` / `subscribe` und `finish` / `publish` machen ähnliche Dinge, zum einen für bereits abgeschlossene Ausdrücke (`finish`, `publish`) und zum anderen für noch nicht abgeschlossene Ausdrücke (`extend`, `subscribe`). Das erste Paar kümmert sich dabei darum, die Verarbeitung eines vollständigen Ausdrucks abzuschließen und ihn in ein passendes Archiv zu stellen. Somit sitzt die Funktion `publish` genau an Stelle fünf im obigen Diagramm. Das zweite Paar hingegen versucht mit `extend` einen Ausdruck fortzuführen und mit `subscribe` wird dieser in ein passendes Archiv gestellt. `subscribe` findet also an Schritt vier im Diagramm statt.
Wie der Name schon sagt, kümmert sich `combine` als nächstes darum, neu in ein Archiv hinzugefügte Ausdrücke, mit den bereits existierenden zu kombinieren. Sie kommt somit an Position sechs vor.

Nun sind also alle Funktionen bekannt, in die die Visualisierung später eingreifen muss. Im nächsten Schritt sollte aber zuerst geklärt werden, wie die Schnittstelle zwischen obigen Funktionen und der Visualisierung aussehen beziehungsweise funktionieren soll.

4.2 Entwicklung einer Schnittstelle

Bevor die Schnittstelle selbst beschrieben werden kann, sollten zuerst Anforderungen and die Schnittstelle spezifiziert werden.

4.2.1 Anforderungen an die Schnittstelle

Um das Single-Responsibility-Prinzip nicht zu verletzten, sollte, wie in der Einleitung zu Abschnitt 4 - Implementierung bereits kurz angerissen, die Schnittstelle recht klein sein. Der Parser soll also ausschließlich für das Parsen zuständig sein und nicht noch etliche Zeilen enthalten, die für die Visualisierung verantwortlich sind!

4.2.2 Entwurf der Schnittstelle

Eine erste Idee könnte sein, sich weiterhin an der in Unterabschnitt 1.2 - Ziel vorgestellten Analogie zu einem Profiler zu orientieren und eine solche Schnittstelle zu betrachten.
Der in Valgrind enthaltene Profiler Callgrind, beispielsweise, erstellt eine Datei, die wie folgt aufgebaut ist:

Listing 2: Callgrind Dateiformat

```
events: Instructions
fl=file1.c
fn=main
16 20
cfn=func1
calls=1 50
16 400
```

Diese Datei lässt sich als Liste an Events interpretieren. Ab Zeile drei kann man so sehen, dass die Funktion `main` Zeile 16 ausführt und dafür 20 Zeiteinheiten benötigt. Zeile 16 ruft dabei `func1` einmal auf, was wiederum 400 Zeiteinheiten benötigt.

Ein ähnliches Event-System wäre auch für diese Arbeit denkbar. Dabei wäre es nicht nur einfach neue Eventtypen hinzuzufügen, sondern die Generierung dieser würde auch nur minimalen Code im Parser benötigen. Mögliche Events wären dann beispielsweise `add_cons_to_archive`, `add_comp_to_archive`, oder `create_archive`. Diese müssten aber nicht extra in eine Datei geschrieben werden, sonder würden ausschließlich im Speicher existieren und von dort aus sequentiell gelesen und abgearbeitet werden. Wobei es wohl auch mit denkbar wenig Aufwand möglich wäre eine Liste an solchen Event zu serialisieren und später wieder zu deserialisieren.

4.2.3 Implementierung der Schnittstelle und Events

Wie oben bereits diskutiert, besteht die Schnittstelle prinzipiell nur aus einer Liste, die nach und nach während der Laufzeit des Parsers mit Events gefüllt wird. Diese Liste lässt sich über die Datei `events.hpp` verwenden.

Listing 3: events.hpp

```
extern std::vector<event*> events;
```

Da die Klasse `event` die Basisklasse aller weiteren Events ist, dadurch also eine, wenn auch niedrige, Vererbungshirarchie entsteht, muss die Liste Zeiger auf `events` enthalten und eben nicht die Objekte selbst.

Die Aufgabe des Parsers ist es jetzt also diese Liste zu populieren. Dies könnte beispielsweise folgendermaßen geschehen:

```
1  events.push_back(new create_archive_event(0));
```

Hiermit würde der Visualisierung mitgeteilt werden, dass ein neues Archiv an Stelle 0 entstanden ist und angezeigt werden muss.

Neben dem Event `create_archive_event` sind in der Datei `event.hpp` noch folgende weitere Events deklariert.

Listing 4: event.hpp (Alle Events)

```
1  class event_group : public event { ... };
2  class create_archive_event : public event { ... };
3  class add_cons_event : public event_with_data { ... };
4  class add_comp_event : public event_with_data { ... };
5  class expr_gets_used_event : public event_with_data { ... };
6  class expr_no_longer_gets_used_event : public event_with_data { ... };
7  class add_expr_to_queue_event : public event_with_data { ... };
8  class remove_expr_from_queue_event : public event_with_data { ... };
```

Die `event_group` Klasse ist dabei nur ein "Hilfsevent", das mehrere Events gruppiert und alle Events zusammen ausführt.
Auffällig ist auch, dass nicht alle Event-Klassen direkt von `event` erben, sondern teilweise auch von `event_with_data`. Die beiden Klassen sind folgendermaßen implementiert:

Listing 5: event.hpp - Implementierung event und event_with_data

```
1  class event {
2  public:
3    explicit event(unsigned int position);
4    virtual event_exec_result exec() = 0;
5    virtual event_exec_result undo() = 0;
6    unsigned int getPosition() const;
7  protected:
8    unsigned int m_position;
9    template <typename Callback>
10   void exec_on_archive_at_pos(unsigned int pos, Callback callback) const;
11 };
12
13 class event_with_data : public event {
14 public:
15   event_with_data(unsigned int position, const Expr& data);
16   Expr getData() const;
17 protected:
18   Expr m_data;
19 };
```

Dabei sieht man, dass ein `event` immer aus einer Position besteht. Die Position ist dabei genau die Position des Archivs, auf dem das Event ausgeführt wird. Die Methode `exec_on_archive_at_pos` ist eine Hilfsmethode, die den übergebenen Callback mit dem Archiv an der Position `pos` als Argument aufruft. Die beiden zentralen Methoden sind allerdings `exec` und `undo`. Sie werden aufgerufen, wenn ein Event ausgeführt werden soll, beziehungsweise wenn ein Event rückgängig gemacht werden soll. Wichtig: Ein Event wird niemals vom Parser rückgängig gemacht, es kann ausschließlich der Benutzer der Visualisierung ein Event rückgängig machen! Beide Methoden geben ein `event_exec_result` zurück. Dieser Typ ist als Enum implementiert und enthält die beiden Möglichkeiten `did_something` und `did_nothing`. Auf die Verwendung dieses Rückgabewerts wird später in Unterunterabschnitt 4.3.9 - Interaktion zwischen allen Komponenten noch genauer eingegangen.

Einigen Events, wie beispielsweise `add_cons_event`, benötigen aber noch mehr Information als lediglich die Position des Archivs, die `event` verfügbar macht. So fügt `add_cons_event` einen Ausdruck zu einem Archiv hinzu, benötigt dazu also auch noch den Ausdruck selbst. Genau diesen zusätzlichen Ausdruck stellt die Klasse `event_with_data` bereit.

Die in Listing 4 - event.hpp (Alle Events) gezeigten Events sind alle ähnlich definiert, da alle die Methoden `exec` und `undo` überschreiben und sonst in der Regel keine eigenen Methoden oder Attribute deklarieren. Events, die auf keinem Archiv arbeiten, wie beispielsweise `remove_expr_from_queue`, definieren zusätzlich einen eigenen Konstruktor, der keine Position erwartet. Wichtig: Definiert ein Event keinen eigenen Konstruktor, muss mit `using event::event`, oder `using event_with_data::event_with_data` explizit der Konstruktor der vererbenden Klasse sichtbar gemacht werden[1].
Auf die tatsächliche Implementierung der beiden entscheidenden Methoden `exec` und `undo` wird später eingegangen, da dazu weitere Schnittstellen nötig sind, die aktuell noch nicht eingeführt wurden.

4.3 Implementierung der GUI

In diesem Kapitel kommen nun alle vorherigen Überlegungen zusammen. Es wird im Folgenden einerseits beschrieben wie die in Abschnitt 3 - Anforderungen eingeführten Anforderungen in ncurses implementiert wurden, als auch wie die Oberfläche mit der in Unterabschnitt 4.2 - Entwicklung einer Schnittstelle entwickelten Schnittstelle zusammenarbeitet, um alle erzeugten Events darzustellen.

[1] `https://en.cppreference.com/w/cpp/language/using_declaration#inheriting_constructors`

4.3.1 Aufbau der Oberfläche

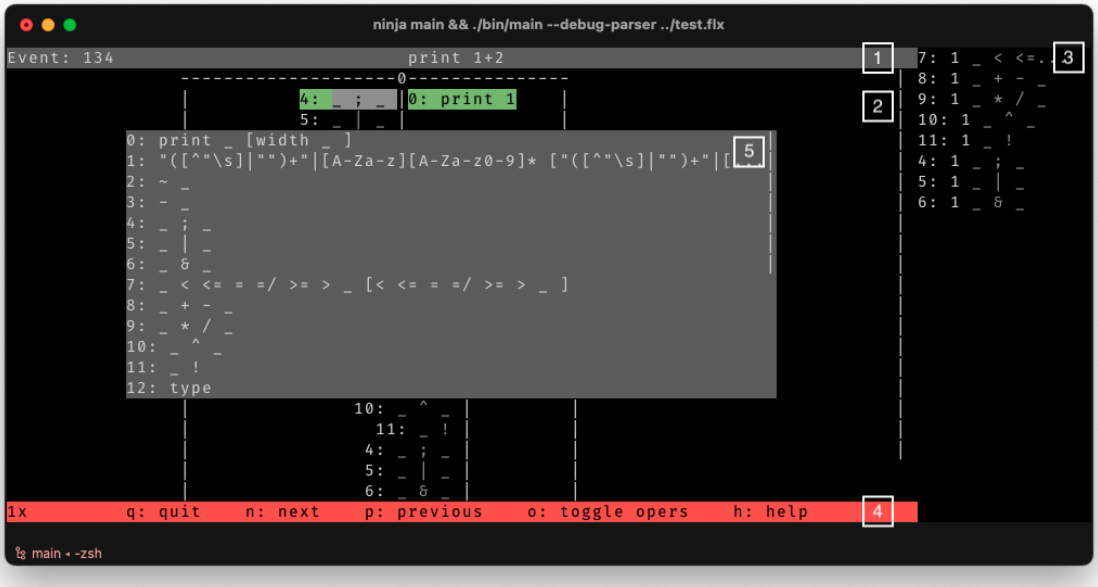


Abbildung 2: Benutzeroberfläche

Abbildung 2 - Benutzeroberfläche zeigt den Aufbau der Oberfläche. Diese lässt sich dabei in vier Bereiche einteilen.

1 Hier wird der Quelltext angezeigt, der vom Parser abgearbeitet wurde.

2 Bereich zwei ist der wahrscheinlich wichtigste. Hier werden alle Archive mit deren voll- und unvollständigen Ausdrücken dargestellt.

3 Ganz rechts im Bild wird die vom Parser verwaltete Queue dargestellt.

4 Der vierte Bereich zeigt Möglichkeiten an, wie der Benutzer mit der Applikation interagieren kann. Außerdem wird der aktuelle Multiplier in der linken unteren Ecke dargestellt. Dieser Bestimmt, wie oft eine Aktion ausgeführt wird.

5 Der letzte Bereich ist nicht durchgehend sichtbar, da er ein Popup ist und vom Benutzer angezeigt oder verborgen werden kann. Hier kann auch anderer Inhalt angezeigt werden, als in obigem Beispiel aktuell sichtbar ist.

Bereich eins und vier sind dabei, mehr oder weniger, statisch und ändern sich während der Laufzeit des Programms nicht drastisch. Bereich vier zeigt zwar je nach Eingabe etwas unterschiedlichen Text an, dieser ist aber nie so lang, dass er nicht in eine Zeile passen würde. Für

diese Beiden Bereiche können und werden also normale ncurses `WINDOW`s verwendet.
Bereich zwei, drei und fünf sind allerdings nicht statisch. Sie können während der Laufzeit beispielsweise ”überlaufen” und Text außerhalb des für den Benutzer sichtbaren Bereichs anzeigen. Hierfür ist der Typ `WINDOW` zu abstrakt und vor allem zu einfach. Hierfür bietet dieses Projekt eine eigene Basisklasse `window_like` an.

4.3.2 Spezielle Fenster

Listing 6: window_like.hpp

```
1  template <typename underlying_window_type>
2  class window_like {
3  public:
4    explicit window_like(underlying_window_type* base, uint32_t width, uint32_t
       ↪ height) : m_underlying_window(base), m_screen_width(width),
       ↪ m_screen_height(height) {}
5    virtual ~window_like() = default;
6
7    virtual void add_n_str(const CH::str&, int x, int y) = 0;
8    virtual void del_line(int x, int y) = 0;
9    virtual void clear() = 0;
10   virtual void prepare_refresh() const = 0;
11   uint32_t get_height() const { return m_screen_height; }
12   uint32_t get_width() const { return m_screen_width; }
13   underlying_window_type* operator*() { return m_underlying_window; }
14 protected:
15   underlying_window_type* m_underlying_window;
16   uint32_t m_screen_width;
17   uint32_t m_screen_height;
18 };
```

Diese Klasse sollte die Basisklasse von allen speziellen Fenstern, wie Popups oder scrollbaren Fenstern, sein. Wobei sie nicht dazu gedacht ist, tatsächlich als Typ einer Variablen verwendet zu werden, sondern nur eine konsistente Schnittstelle bieten soll, um sowohl das Lesen, als auch Schreiben des Codes angenehmer zu gestalten. Ein `window_like` Fenster besitzt immer ein zugrundeliegendes Fenster (`underlying_window_type* underlying_window`). Dieses kann ein ncurses `WINDOW`, oder wiederum ein `window_like` Fenster sein. Außerdem definiert

die Klasse einige grundlegende Schnittstellen, wie Getter für Höhe und Breite und Methoden zum Hinzufügen und Entfernen von Text. `prepare_refresh` sollte dabei nach jeder Änderung aufgerufen werden, um den dargestellten Content zu aktualisieren. Wichtig: Wie der Name `prepare_refresh` bereits andeutet, wird nicht wirklich der physische Speicher, sondern nur der virtuelle (s. Abschnitt 2.2.4 - Manipulieren von Fenstern) aktualisiert. Ergo ist im Anschluss von `prepare_refresh` ein Aufruf zu `doupdate` nötig!

Ein spezielles Fenster in diesem Projekt ist die Klasse `scrollable`. Sie wird für alle Bereiche, exklusive eins und vier verwendet.
Folgender Abschnitt beschreibt, wie ein scrollbares Fenster arbeitet.

4.3.3 Scrolling

Scrolling funktioniert in ncurses über so genannte Pads. Ein Pad kann dabei, im Gegensatz zu einem normalen Fenster, auch deutlich größer als der Terminal Bildschirm selbst sein. Auf einem Pad kann also auch Content außerhalb des sichtbaren Bereichs platziert werden. Nun wird aber nie der komplette Inhalt eines Pads dargestellt, sondern immer nur ein kleiner Ausschnitt. Schreibt man nun also Text auf das Pad und bewegt anschließend den tatsächlich dargestellten Ausschnitt nach unten, erzeugt man somit die Illusion eines scrollenden Fensters.
Abbildung 3 - Scrolling visualisiert dieses Prinzip:

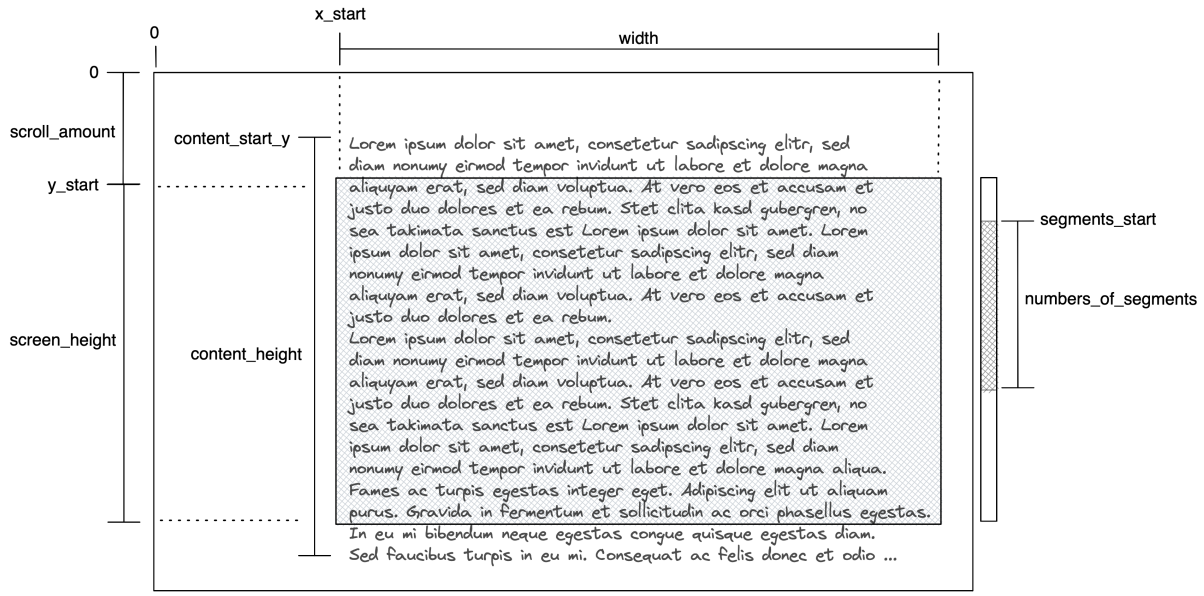


Abbildung 3: Scrolling

Die äußere Box ist dabei das Pad und die innere, leicht grau unterlegte Box, der tatsächlich dargestellte Ausschnitt.
Um nun korrektes Scrollen zu implementieren, verwaltet die Visualisierung mehrere, auch in Abbildung 3 dargestellte, Variablen. Die Bedeutung dieser Variablen wird nach und nach in diesem Kapitel erklärt.

Scrollbar

Die Mathematik hinter dem Scrollen ist relativ simpel. Prinzipiell muss nur das Verhältnis der beiden folgenden Brüche gleich sein:

$$\frac{scroll\_amount}{content\_height - screen\_height} = \frac{segments\_start}{scrollbar\_max\_height - number\_of\_segments} \quad (1)$$

$scrollbar\_max\_height$ ist in diesem Projekt äquivalent zu $screen\_height$! Denkbar wäre grundsätzlich aber auch eine Scrollbar, die nicht die komplette Höhe des Bildschirms ausfüllt, weswegen diese Gleichung hier etwas allgemeiner gehalten ist als eigentlich nötig.
Diese Gleichung kann man nun sinnvoll entweder nach $segments\_start$, oder $scroll\_amount$ auflösen, je nach dem welcher Wert gegeben ist. In diesem Fall, ist $segments\_start$ gegeben. Man kommt also durch Umformen auf folgendes Ergebnis:

$$segments\_start = \frac{scroll\_amount \cdot (scrollbar\_max\_height - number\_of\_segments)}{content\_height - screen\_height} \quad (2)$$

Nun hat man also eine Möglichkeit unter Abhängigkeit verschiedener Parameter die vertikale Position der Scrollbar, beziehungsweise die Position des sich bewegenden Teils der Scrollbar, zu bestimmt. Als nächstes sollte noch die Höhe der Scrollbar berechenbar sein. Diese sollte den Anteil des tatsächlich angezeigten Contents im Verhältnis zu der Höhe des gesamten Contents widerspiegeln.
Hierzu nimmt man das Produkt aus $screen\_height$ und $scrollbar\_max\_height$ (hier also: $screen\_height \cdot screen\_height$) und bildet das Verhältnis zu der Höhe des gesamten Contents ($content\_height$).

$$number\_of\_segments = \frac{screen\_height^2}{content\_height} \quad (3)$$

Grundsätzlich braucht es nicht mehr, um eine funktionierende Scrollbar darzustellen. Allerdings gibt es noch ein paar Fallstricke. Vor allem muss darauf geachtet werden nicht versehentlich durch

0 zu teilen, oder eine minimale, beziehungsweise maximale Größe der Scrollbar festzulegen.
Die folgende Implementierung von `prepare-refresh` ist primär verantwortlich, wie die Scrollbar dargestellt wird und zeigt somit, unter anderem, wie diese Fallstricke überwunden wurden.

Listing 7: scrollable::prepare_refresh #1

```
1  static bool clear_scrollbar = false;
2  uint32_t segments_start;
3  uint32_t number_of_segments;
4
5  if(m_content_height > m_screen_height || m_scroll_amount + m_screen_height >
       ↪ m_screen_height) {
6    const uint32_t internal_content_height = std::max(m_content_height,
         ↪ m_screen_height + m_scroll_amount);
7    const uint32_t internal_scroll_amount = std::clamp(m_scroll_amount, (uint32_t)
         ↪ 0, internal_content_height - m_screen_height);
8
9    number_of_segments = std::max((uint32_t) 1, m_screen_height * m_screen_height /
         ↪  internal_content_height);
10   segments_start = (m_screen_height - number_of_segments) *
         ↪ internal_scroll_amount / (internal_content_height - m_screen_height);
11   clear_scrollbar = true;
12   }else {
13     number_of_segments = 0;
14     if(!clear_scrollbar)
15       clear_scrollbar = true;
16   }
```

Zeile zwei und drei deklarieren die gesuchten Größen. Darauf folgt die Überprüfung, ob überhaupt eine Scrollbar angezeigt werden muss. Diese Überprüfung ist in zwei Bedingungen aufgeteilt:

```
1  m_content_height > m_screen_height
```

Ist diese erste Bedingung nicht erfüllt, muss theoretisch keine Scrollbar angezeigt werden, da die Höhe des Fensters ausreicht, um den vollständigen Content darzustellen. Wichtig dabei ist, dass auf echt größer und nicht größer gleich getestet wird. Sind `m_content_height` und `m_screen_height` nämlich gleich, kann in Zeile zehn im Zweifel durch 0 geteilt werden!
Allerdings gibt es einen Fall, in dem eine Scrollbar angezeigt werden muss, auch wenn der komplette Inhalt in das unterliegende Fenster passt:

```
1  m_scroll_amount + m_screen_height > m_screen_height
```

Ist diese Bedingung erfüllt, wurde über den Content hinaus nach unten gescrollt. Um nun kenntlich zu machen, dass nach oben gescrollt werden kann, um wieder den gesamten Inhalt sichtbar zu machen, sollte auch in diesem Fall die Leiste angezeigt werden.
Ist keine der beiden genannten Bedingungen wahr, ist also keine Scrollbar nötig, wird die Anzahl der zu zeichnenden Segmente auf 0 gesetzt. Andernfalls wird in Zeile neun Gleichung 3 angewandt. Da dabei theoretisch auch ein Wert kleiner 1 heraus kommen kann, wird mit `std::max` eine untere Grenze von 1 gesetzt, da weniger als ein Segment nicht sichtbar ist. Als nächstes wird in Zeile 10 Gleichung 2 angewandt. Anstatt den regulären Wert von *scroll_amount* zu verwenden, muss dieser erst normalisiert werden (Zeile sieben), da der ursprüngliche Wert umgangssprachlich gesagt, erst mal nur speichert, wie oft der Pfeil nach unten gedrückt wurde. Auch *content_height* kann nicht direkt verwendet werden. Dieser Wert muss angepasst werden, je nachdem mit welchem Fall man in diesen Block gekommen ist. Dabei wird in der Regel `m_content_height` verwendet. Ist allerdings die zweite Bedingung erfüllt, so wird `m_screen_height + m_scroll_amount` größer als `m_content_height` sein und somit wird dann diese Summe verwendet. Würde man diese Unterscheidung hier nicht machen, würde die Scrollbar größer werden (also andeuten, dass weniger Content überhängt), wenn der Benutzer bereits über den gesamten Content hinaus gescrollt hat und `m_content_height` kleiner wird. Verwendet man hier also nicht die Höhe des Contents selbst, sondern die feste Höhe des Fensters zusammen mit der Höhe des nach unten gescrollten, teilweise leeren, Raums, bildet die Scrollbar auch genau diese korrekte Höhe ab.
Sind nun also beide gesuchten Größen berechnet, kann im Anschluss die Scrollbar gezeichnet werden.

Listing 8: scrollable::prepare_refresh #2

```
1  for(int i = 0; clear_scrollbar && i < m_screen_height * PAD_HEIGHT_MULTIPLIER; ++
       ↪ i)
2    mvwaddch(m_pad, i, m_width - 1, ' ');
3
4  clear_scrollbar = false;
5
6  for(int i = 0; i < number_of_segments; ++i)
7    mvwaddch(m_pad, segments_start + i + m_scroll_amount + m_content_start_y,
         ↪ m_width - 1, '|');
8
9  pnoutrefresh(m_pad, m_scroll_amount + m_content_start_y, 0, m_y_start, m_x_start,
       ↪  m_y_start + m_screen_height, m_x_start + m_width);
```

Das Flag `clear_scrollbar` dient dabei als simple Optimierung, damit die alte Scrollbar nur komplett entfernt wird, wenn dies auch tatsächlich nötig ist. Ist das also der Fall, ersetzt die erste Schleife, die über die komplette Höhe des Pads iteriert, den kompletten rechten Rand durch Leerschritte. In der zweiten Schleife, die dann nur noch über die anzuzeigenden Segmente iteriert, wird die Scrollbar dann tatsächlich gezeichnet.

Content Offset

In Abbildung 3 - Scrolling fällt auf, dass der Inhalt nicht ganz oben beginnt, sondern erst ab `content_start_y`. Damit es aber immer den Anschein hat, als würde Content ganz oben beginnen, wird alles, was die Klasse `scrollable` zeichnet, um eben diesen `content_start_y`-Offset nach oben geschoben. Daher wird, beispielsweise, in Listing 8 - scrollable::prepare_refresh #2 in Zeile neun zum zweiten Argument `m_scroll_amount` noch zusätzlich dieser Wert dazu addiert, um den Bereich, den das Pad anzeigt, noch weiter nach unten zu verschieben.

Weitere Methoden

Neben der bereits besprochenen `prepare_refresh` Methode, besitzt ein `scrollable` natürlich noch weitere vererbte Methoden, wie `void add_string(const CH::str& str, int x, int y)`, `void del_line(int x, int y)`, oder `void clear()`. Diese machen jeweils genau das, worauf ihr Name bereits schließen lässt, geben dabei aber auch acht, dass

`m_content_height` und `m_content_start_y` immer korrekt gesetzt sind.
Die wichtigste Schnittstelle dieser Klasse wird aber wohl `void scroll_y(int delta)` sein.

Listing 9: scrollable::scroll_y

```
1  const auto& is_allowed_to_scroll = [&]() { ... };
2
3  if(!is_allowed_to_scroll())
4    return;
5
6  m_scroll_amount += delta;
7  prepare_refresh();
```

Die Hauptarbeit dieser Funktion besteht vor allem daraus, sicherzustellen, dass nur gescrollt wird, wenn das auch notwendig beziehungsweise möglich und erlaubt ist. Das Setzten des neues Scroll-Deltas in Zeile zehn und das Aktualisieren des Fensters in Zeile elf findet dann logischerweise auch nur statt, wenn `is_allowed_to_scroll` `true` zurück liefert.

Listing 10: scrollable::scroll_y - is_allowed_to_scroll

```
1  if(m_scroll_amount == 0 && delta < 0)
2    return false;
3  if(m_scroll_amount + m_screen_height > m_content_height) {
4    if(delta < 0)
5      return true;
6    return false;
7  }
8  if(m_content_height < m_screen_height)
9    return false;
10 if(m_scroll_amount + m_screen_height + delta > m_content_height)
11   return false;
12 return true;
```

Dieses Lambda fängt prinzipiell nur Fälle ab, in denen das Scrollen nicht erlaubt ist und gibt am Ende als, so zu sagen, default-Wert `true` zurück.
In folgenden Fällen darf nicht gescrollt werden (Angeordnet nach ihrem Vorkommnis in Listing 10 - scrollable::scroll_y - is_allowed_to_scroll):

- Es wurde bereits nach ganz oben gescrollt und es wurde weiter versucht nach oben zu scrollen

- Der Content würde eigentlich ganz auf den Bildschirm passen, aber durch vorherige Aktionen ist der Inhalt noch etwas runter gescrollt. Es wird versucht weiter runterzuscrollen. (Nur nach oben wäre hier zulässig)
- Der Inhalt passt komplett auf den Bildschirm
- Es wurde bereits maximal nach unten gescrollt

4.3.4 Popups

Neben dem gerade beschriebenen speziellen Fenster `scrollable`, gibt es noch ein Weiteres. Die Klasse `popup` wird dabei ausschließlich von Bereich fünf verwendet.

ncurses besitzt native Unterstützung für Popup artige Fenster an. `PANEL`s können sich, im Gegensatz zu normalen `WINDOW`s, auch überlappen und besitzen zusätzlich zu einer x und y-Koordinate auch noch eine z-Koordinate, die es ermöglicht Panels übereinander anzuordnen. Mit den Funktionen `int show_panel(PANEL* p)` und `int hide_panel(PANEL* p)` lässt sich das Panel dann anzeigen, beziehungsweise verstecken.
Die Klasse `popup` ist nun nur noch ein Wrapper um diese Funktionen. Im Gegensatz zu `scrollable` wird der `underlying_window_type` von dieser Klasse aber nicht auf `WINDOW`, sondern nun auf `scrollable` gesetzt, um auch scrollbaren Inhalt in Popups zu ermöglichen.

Listing 11: popup.hpp

```
1  class popup : public window_like<scrollable> {
2  friend class popup_manager;
3  public:
4    explicit popup(scrollable* window);
5    ~popup() override;
6    void add_n_str(const CH::str& str, int x, int y) override;
7    void del_line(int x, int y) override;
8    void clear() override;
9    void prepare_refresh() const override;
10 private:
11   void show();
12   void hide();
13   PANEL* m_panel;
14   bool m_is_currently_shown {false};
15 };
```

Auffällig ist hierbei auch, dass alle, nicht vererbten, Funktionen `private` deklariert sind. Die Klasse `popup` sieht es gar nicht vor autark verwendet zu werden, weswegen sie ihre Schnittstelle auch gar nicht sichtbar machen muss. So sollten Popups nur indirekt über den, in Zeile zwei als Freund markierten, `popup_manager` verwendet werden.
Wichtig in Zeile zwei ist, dass `popup_manager` mit `friend class` forward-deklariert und nicht mit `#include` eingebunden wird, da sonst eine zyklische Abhängigkeit zwischen `popup` und `popup_manager` entstehen würde!

Die Klasse `popup_manager` bietet nun selber öffentliche Methoden um ein angegebenes Popup anzuzeigen, zu verstecken, oder zu toggeln.

Listing 12: popup_manager.hpp

```
1  class popup_manager {
2  public:
3    using callback = std::function<void()>;
4    static popup_manager& the();
5    /* delete copy assignment oper. and copy constructor  */
6    void insert(popup* popup, const std::optional<callback>& show_callback = {},
         ↪ const std::optional<callback>& hide_callback = {});
7    bool toggle(popup* popup) const;
8    void show(popup* popup) const;
9    void hide(popup* popup) const;
10   void prepare_refresh_for_shown_popups();
11   bool is_one_popup_shown() const;
12 private:
13   popup_manager() = default;
14   std::vector<popup*> m_popups;
15   std::unordered_map<popup*, callback> m_show_callbacks;
16   std::unordered_map<popup*, callback> m_hide_callbacks;
17 };
```

Hierbei wird immer sichergestellt, dass nur ein Popup zu jedem beliebigen Zeitpunkt sichtbar ist. Wird also aktuelle das Hilfe-Popup angezeigt und der Benutzer drückt ”o” um sich alle Operatoren anzeigen zu lassen, so wird automatisch erst das Hilfe-Popup geschlossen, bevor das neue geöffnet wird. Auch verwaltet diese Klasse zu jedem popup zwei Callbacks. Einen so genannten `show_callback` und einen `hide_callback`. Diese werden aufgerufen, wenn das jeweilige Popup angezeigt, beziehungsweise versteckt wird. So registriert beispielsweise das Operator-Popup über

seinen `show_callback` sich selbst als Empfänger aller Scroll-Eingaben. Wichtig hierbei ist, dass diese Callbacks auch nur wirklich dann aufgerufen werden, wenn sich der Zustand des Popups verändert. Wird beispielsweise `show` aufgerufen, obwohl das Popup bereits sichtbar ist, wird der Callback nicht ausgeführt! Die Implementierung von `show` stellt dies wie Folgt sicher.

Listing 13: popup_manager::show

```
1  void popup_manager::show(popup* popup) const {
2    assert(popup != nullptr);
3    assert(!popup->m_is_currently_shown);
4    for(auto* p: m_popups) {
5      if(!p->m_is_currently_shown)
6        continue;
7      hide(p);
8    }
9    m_show_callbacks.at(popup)();
10   popup->show();
11 }
```

In Zeile drei wird mit `assert(!popup->m_is_currently_shown)` sichergestellt, dass `show` nicht auf einem bereits sichtbaren Popup aufgerufen wird. Dabei ist es okay durch die Assertion das Programm zu terminieren, da in diesem Fall die Methode nicht korrekt verwendet wurde und der aufrufende Code somit angepasst werden muss. Mit der Schleife ab Zeile vier wird dann auf allen Popups, die nicht bereits versteckt sind `hide` aufgerufen um sicher zu stellen, dass eben nur ein Popup gleichzeitig angezeigt wird. Kurz bevor das gewünschte Popup dann tatsächlich angezeigt wird, wird noch der entsprechende Callback aufgerufen. `hide` und `toggle` sind ähnlich implementiert.

Da nun alle grundlegenden Fenster-Komponenten bekannt sind, können als nächstes diejenigen Komponenten besprochen werden, die den Fenstern ihren Inhalt geben.

4.3.5 Ausdrucks-Warteschlange

Angefangen mit der Warteschlange des Parsers (s. Tabelle 2 - Queue Beispiel). Die Queue wird von der Klasse `expr_queue` verwaltet und von folgenden Events verwendet:

`add_expr_to_queue_event`

Hier wird ein neuer Ausdruck ans Ende der Queue gestellt

remove_expr_from_queue_event

Hier wird ein Ausdruck am Anfang der Queue entfernt

Schnittstelle

Listing 14: expr_queue.hpp

```
1  class expr_queue {
2  public:
3    static expr_queue& the();
4      /* delete copy assignment oper. and copy constructor */
5    void push_back(const Expr& expr, bool is_comp);
6    bool pop_back();
7    void push_front(const Expr& expr, bool is_comp);
8    bool pop_front();
9  private:
10   expr_queue() = default;
11   int m_y_begin {0};
12   int m_y_end {0};
13 };
```

Die API, die diese Klasse zur Verfügung stellt, ist recht minimalistisch. `push_back` und `pop_front` sind dabei nötig, um das jeweilige Event auszuführen. `pop_back` und `push_front` hingegen, werden verwendet, um das jeweilige Event wieder rückgängig zu machen.
Die Klasse `expr_queue` speichert, beziehungsweise verwaltet, selbst keine echte Queue. Es ist, in diesem Fall, absolut ausreichend immer nur neuen Text zu zeichnen und nicht eine komplette Liste.
Die Attribute `m_y_begin` und `m_y_end` repräsentieren die Grenzen, beziehungsweise die y-Koordinaten der Grenzen der Queue auf dem unterliegenden Fenster. `m_y_begin` ist dabei immer inklusive und `m_y_end` exklusive.

Implementierung

Im Folgenden werden exemplarisch die Implementierung von `expr_queue::push_back` und `expr_queue::pop_front` erleutert.

Listing 15: expr_queue::push_back

```
1 expr_repr er = expr_repr(expr, is_comp ? expr_repr::f_is_comp : 0);
2 CH::str expr_str = truncate_string_to_length(er.as_string(), QUEUE_WIDTH - 2);
3 mvsaddstr(queue_display, m_y_end, 0, expr_str(CH::A | CH::Z  - er.currpart_pos));
4 wattron(**queue_display, A_MUTED);
5 mvsaddstr(queue_display, m_y_end, er.currpart_pos, expr_str(CH::A + er.
    ↪ currpart_pos | CH::Z ));
6 wattroff(**queue_display, A_MUTED);
7 ++m_y_end;
8 queue_display->prepare_refresh();
```

Immer wenn ein Ausdruck dargestellt wird, wird dazu die Klasse `expr_repr` verwendet. Diese wird später in Unterunterabschnitt 4.3.10 - Hilfsklassen und kleinere Details noch näher beschrieben.

Grundsätzlich ist ihre Verwendung aber recht simpel. Um ein Objekt zu erstellen, muss der Ausdruck selbst und ein paar, den Ausdruck beschreibende, Flags übergeben werden. Auf einem `expr_repr` Objekt kann man nun `as_string` aufrufen, um eine Text-Repräsentation des Ausdrucks zu erhalten.

Anschließend wird eben dieser Text auf seine Länge überprüft. Überschreitet diese die Breite des Fensters, wird der überstehende Teil einfach abgeschnitten. Nun werden nacheinander zwei Teile des Ausdrucks separat behandelt. Die Grenze zwischen beiden Teilen ist dabei in `expr_repr::currpart_pos` gespeichert. Zuerst wird der Teil, wenn vorhanden, gezeichnet, der bereits vom Parser gelesen und abgearbeitet wurde. Es folgt der Teil, der eben noch nicht gelesen wurde und als nächstes vom Parser überprüft werden muss. Dieser wird, um ihn vom vorigen Teil unterscheidbar zu machen, anders dargestellt. Hierfür bietet das Projekt das Attribut `A_MUTED`, welches mit `wattron` beziehungsweise `wattroff` aktiviert und deaktiviert wird.

Sind beide Teile gezeichnet, wird zum Schluss die y-Koordinate des Endes erhöht und das Fenster aktualisiert.

Listing 16: expr_queue::pop_front

```
1 bool expr_queue::pop_front() {
2 if(m_y_begin == m_y_end)
3   return false;
4 queue_display->del_line(0, m_y_begin++);
5 queue_display->prepare_refresh();
6 return true;
7 }
```

`pop_front` ist wesentlich simpler aufgebaut. Nach einer Überprüfung, ob die Queue Content enthält, wird dann die lösch-Operation nur noch an das zugrundeliegende `queue_display` (vom Typ `scrollable*`) weitergeleitet. Zum Schluss muss auch hier einer eine Aktualisierung des Fensters folgen.
Die Methoden `push_front` und `pop_back` sind sehr ähnlich zu den beiden hier vorgestellten implementiert.

Soweit zur Implementierung von Bereich drei. Ein weiterer Bereich, der eine eigene Klasse zur Verwaltung verwendet, ist Bereich fünf. Nun steht Bereich fünf allerdings für beliebige Popups. Die im Folgenden beschrieben Klasse verwaltet aber nur das Operator-Popup, das auch in Abbildung 2 - Benutzeroberfläche dargestellt ist.

4.3.6 Liste aller Operatoren

Die Aufgabe dieser Klasse ist schnell erklärt. Sie verwaltet eine Liste aller bisher gesehenen Prototyp-Ausdrücke und weist diesen zusätzlich eine eindeutige ID zu. Diese ID wird aus Gründen der Leserlichkeit einfach ab 0 aufsteigend vergeben. Da ausschließlich Prototypen gespeichert werden, die einen Haupt-Operator besitzen, den noch kein bereits aufgenommener Prototyp besitzt, wird nicht von Prototyp-Liste, sondern eben von Operatoren-Liste gesprochen.

Schnittstelle

Listing 17: oper_store.hpp

```
1  class oper_store {
2  public:
3    static oper_store& the();
4    /* delete copy assignment oper. and copy constructor */
5    bool insert_if_prototyp(const Expr& expr);
6    std::optional<int> get_id_from_oper(const Oper& oper);
7  private:
8    oper_store() = default;
9    std::map<Oper, int> m_opers;
10 };
```

Auch wenn diese Klasse, im Gegensatz zu der vorher vorgestellten Klasse `popup`, eine Datenstruktur mit allen Operatoren und den dazugehörigen IDs verwaltet, wird diese trotzdem nie verwendet, um alle Operatoren neu zu zeichnen. Da die Operatoren und IDs sich nach der initialen Registrierung nie mehr ändern, ist dies hier auch gar nicht notwendig. Das Attribut `m_opers` dient lediglich dazu, bereits registrierte Operatoren zu einem späteren Zeitpunkt mit beziehungsweise in `get_id_from_oper` wieder zu Ihrer vergebenen ID zuordnen zu können.

Implementierung

In Listing 17 - oper_store.hpp ist zu sehen, dass die API dieser Klasse aus lediglich zwei Funktionen besteht. `insert_if_prototyp` fügt einen neuen Ausdruck in die Datenstruktur ein. Allerdings wird dies, wie der Name eben schon sagt, nur getan, falls der Ausdruck tatsächlich ein Prototyp ist und der Haupt-Operator des Ausdrucks noch nicht gesehen wurde.

Listing 18: oper_store::insert_if_prototyp

```
1  static int next_id_and_line = 0;
2  if(expr(beg_) != expr(end_))
3    return false;
4  if(m_opers.find(expr(oper_)) != m_opers.end())
5    return false;
6  m_opers.try_emplace(expr(oper_), next_id_and_line);
7  expr_repr er = expr_repr(expr, expr_repr::f_is_prototype);
8  mvpaddstr(opers_popup, next_id_and_line, 0, er.as_string(POPUP_WIDTH - 1));
9  ++next_id_and_line;
10 return true;
```

Diese beiden Kriterien sind auch das erste das in Zeile zwei und vier überprüft wird. Evaluiert eine der Bedingungen nicht zu `true`, gibt die Funktion `false` zurück. Andernfalls wird ganz unten in Zeile 12 `true` zurückgegeben. Zeile sechs nimmt den neuen Operator dann tatsächlich mit auf. Der nächste Schritt ist, den im Popup angezeigten Text zu erzeugen. `expr_repr::to_string` prefixt den Ausdruck dann automatisch mit seiner ID, weswegen es essentiell ist, dass er vorher schon in `m_opers` eingefügt wurde. Andernfalls kann, die von `expr_repr::to_string` verwendete, Methode `get_id_from_oper` den Ausdruck und seine ID noch nicht finden. Hier ist dann auch das erste Mal, dass `expr_repr::to_string` einen Parameter übergeben bekommt. Dieser bestimmt, zu welcher Länge der Text abgeschnitten wird. Zum Schluss wird dieser Text gezeichnet und die nächste ID noch inkrementiert.
Ausdrücke werden mit `insert_if_prototype` zu dieser Klasse hinzugefügt, sobald sie zur `expr_queue` hinzugefügt werden. Das ist absolut ausreichend, da jeder Ausdruck immer zuerst durch die Queue laufen sollte, bevor er beispielsweise in den Archiven auftritt. Allerdings ist es nicht die Aufgabe der `expr_queue` Klasse, sondern die des `add_expr_to_queue_event` Klasse, beziehungsweise deren `exec` Methode!

Wurde ein Operator mit der gerade beschriebenen Funktion hinzugefügt, kann nun mit `oper_store::get_id_from_oper` die ID dieses Operators abgefragt werden.
Diese Funktion gibt ein `std::optional` zurück. Ist dieses mit einem Wert besetzt, ist dieser die ID selbst, andernfalls wurde der gesuchte Ausdruck noch nicht registriert.

Listing 19: oper_store::get_id_from_oper

```
1 const auto& it = m_opers.find(oper);
2 if(it == m_opers.end())
3   return {};
4 return it->second;
```

In diesem letzten Fall, ist die Bedingung in Zeile vier nicht erfüllt und es wird ein leeres `std::optional` zurückgegeben. Ansonsten wird ein optional in place mit der ID des Ausdrucks gefüllt und zurückgegeben.

Bei den eben beschriebenen Methoden fällt auf, dass es für diese komplett transparent ist, ob das Popup aktuell angezeigt wird, oder eben nicht. Mit diesen beiden Zuständen umzugehen, ist die Aufgabe der in Unterunterabschnitt 4.3.4 - Popups beschriebenen Klasse `popup`. Hier, beziehungsweise bei allen einen Bereich verwaltenden Klassen, wurde also eine Art Model-View-Controller Pattern implementiert. Die View ist dabei immer das Fenster, in diesem Fall also `popup`. Das Model wiederum, wird von den das Fenster verwaltenden Klassen repräsentiert. Der Controller wurde zu diesem Zeitpunkt noch nicht detailliert angesprochen, er besteht aber in der Art und Weise, wie die generierten Events abgearbeitet werden (s. Unterunterabschnitt 4.3.9 - Interaktion zwischen allen Komponenten).

Der letzte, aufwendigste und wichtigste Bereich, der nun noch fehlt, ist Bereich zwei - die Archive.

4.3.7 Archiv Rendering

Da Archive nicht nur einmal gezeichnet und danach nicht mehr verändert werden und zusätzlich drauf geachtet werden muss, dass auch bei einer Änderung der Größe eines Archivs sich alle anderen nach wie vor an der richtigen Stelle befinden und sich nicht überlappen, ist dieser Teil des Codes womöglich mit der Komplexeste.

Bevor es um das Layouting geht, beschreibt dieser Abschnitt aber erst einmal, wie ein einzelnes Archiv auf Änderungen reagiert und neu gezeichnet wird. Die folgende Abbildung zeigt den groben internen Ablauf, nachdem eine Änderung erkannt wurde.

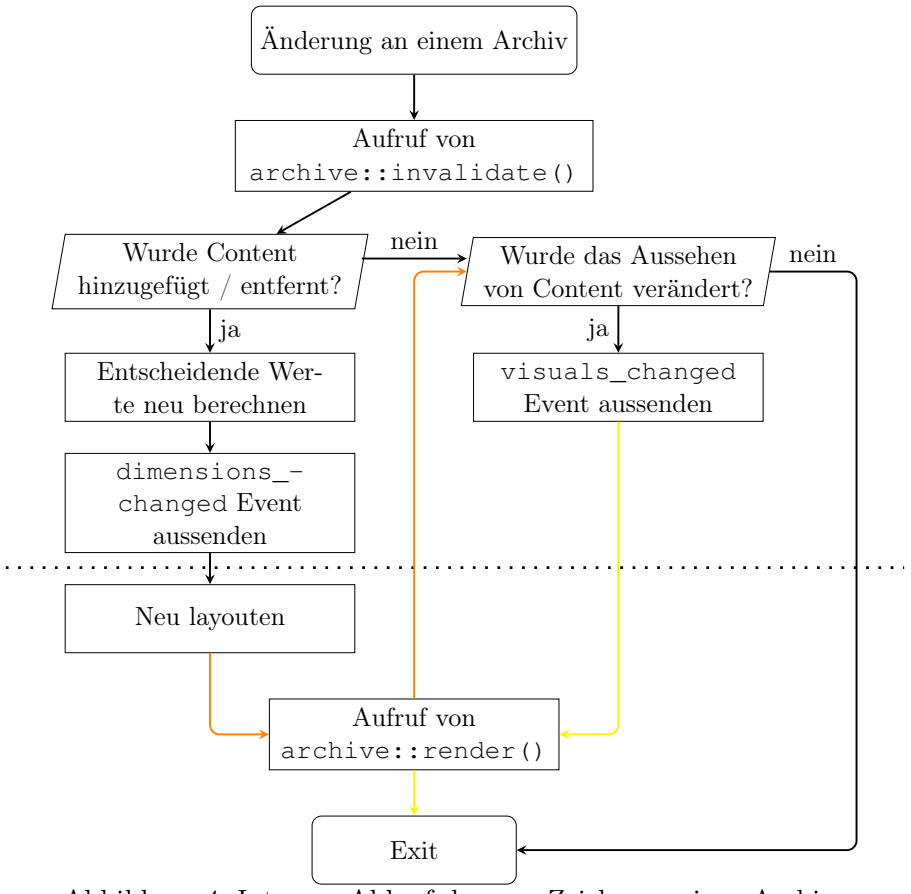


Abbildung 4: Interner Ablauf des neu Zeichnens eines Archivs

Die gestrichelte Linie deutet hierbei den Übergang von der Klasse `archive` zur Klasse `layouter`. Dieser Abschnitt beschäftigt sich ergo primär mit dem oberen Teil der Abbildung.
Um die beiden anfänglichen Tests in der Abbildung beantworten zu können, werden zwei Attribute vom Typ `bool` verwaltet:

`m_dirty_dimensions` Dieses Flag wird gesetzt, sobald ein Ausdruck zum Archiv hinzukommt, oder entfernt wird. Oder sich die Position des Archivs verändert.

`m_dirty_visuals` Dieses Flag wird gesetzt, sobald ein Ausdruck gehighlighted oder als mehrdeutig markiert wird. Wird dieses Flag gesetzt, muss das Archiv nur neu gezeichnet, aber nicht neu gelayouted werden.

`void add_comp(long id, const Expr& expr)` zeigt wie beide Flags verwendet werden:

Listing 20: archive::add_comp

```
1  int flags = 0;
2  for(auto&[_, pair]: m_comp) {
3    auto&[expr, repr] = pair;
4    if(expr(end_) != comp(end_))
5      continue;
6    repr.flags |= expr_repr::f_is_ambiguous;
7    m_dirty_visuals = true;
8    flags = expr_repr::f_is_ambiguous;
9  }
10 flags |= expr_repr::f_is_comp;
11 m_comp.try_emplace(m_comp.size(), comp, (expr_repr) {comp, flags});
12 m_dirty_dimensions = true;
13 invalidate();
```

Die Methode beginnt mit der Überprüfung, ob der eingefügte vollständige Ausdruck eine Mehrdeutigkeit verursacht. Dies geschieht, indem die Endpositionen aller bisher im Archiv gespeicherten vollständigen Ausdrücke mit der Endposition des Neuen verglichen werden. Sind diese Positionen identisch, wird ab Zeile sechs das `f_is_ambiguous` Flag des existierenden und des neuen Ausdrucks gesetzt. Da in diesem Fall die Darstellung eines Ausdrucks geändert wurde, muss in Zeile sieben eben auch das `m_dirty_visuals` Flag auf `true` gesetzt werden. Wichtig zu erkennen ist dabei, dass es ausreichend ist, in einem Archiv nur die Endpositionen zu vergleichen, da alle vollständigen Ausdrücke in einem Archiv zwangsweise dieselbe Startposition haben! Nach dieser initialen Überprüfung, wird, nachdem der Ausdruck in das Archiv mit aufgenommen wurde, auch noch das `m_dirty_dimensions` Flag gesetzt, da sich möglicherweise das Archiv vergrößert hat und deswegen neu gelayouted werden muss.
Zum Schluss wird mit dem Aufruf zu `invalidate` die in Abbildung 4 - Interner Ablauf des neu Zeichnens eines Archivs beschriebene Abfolge in Gang gesetzt. Diese Abfolge resultiert ganz unten dann in einem Aufruf zu `render`. Wie diese Methode das komplette Archiv neu zeichnet, beschreibt folgender Code.

Listing 21: archive::render #1

```
for(int i = 0; i < m_height; ++i) {
  mvsaddstr(main_viewport, i + m_y_start, m_x_start, "|");
  mvsaddstr(main_viewport, i + m_y_start, m_width + m_x_start, "|");
}
for(int i = 0; i <= m_width; ++i) {
  mvsaddstr(main_viewport, m_y_start, i + m_x_start, "-");
  mvsaddstr(main_viewport, m_height + m_y_start - 1, i + m_x_start, "-");
}
```

Zu Beginn werden, mit zwei `for`-Schleifen, die Ränder des Archivs gezeichnet. In den beiden Schleifen wird jeweils zuerst die linke beziehungsweise obere und dann die rechte beziehungsweise untere Seite umrandet.

Als nächstes werden alle voll- und unvollständigen Ausdrücke gezeichnet.

Listing 22: archive::render #2

```
auto comp_it = m_comp.begin();
auto cons_it = m_cons.begin();
for(int i = 1; i < m_height - 1; ++i) {
  mvsaddstr(main_viewport, i + m_y_start, m_divider_x_pos + m_x_start, "|");
  if(comp_it != m_comp.end()) {
    auto&[_, repr] = comp_it->second;
      const auto& string = repr.as_string();
      if(repr.flags & expr_repr::f_is_highlighted)
        wattron(**main_viewport, A_HIGHLIGHT);
      if(repr.flags & expr_repr::f_is_ambiguous)
        wattron(**main_viewport, A_AMBIGUOUS);
      mvsaddstr(main_viewport, i + m_y_start, m_divider_x_pos + 1 + m_x_start,
          ↪ string);
      if(repr.flags & expr_repr::f_is_highlighted)
        wattroff(**main_viewport, A_HIGHLIGHT);
      if(repr.flags & expr_repr::f_is_ambiguous)
        wattroff(**main_viewport, A_AMBIGUOUS);
      ++comp_it;
  }
  if(cons_it != m_cons.end())
    /* ... */
}
```

Dazu werden zwei Iteratoren, für die voll- und unvollständigen Ausdrücke verwaltet. Diese Ausdrücke werden in zwei Variablen mit folgendem Type gespeichert:

```
1  std::map<long, std::pair<Expr, expr_repr>>
```

Der Key dieser `std::map` wird dabei immer auf die aktuelle Größe der Map gesetzt, um eine konsistente Reihenfolge zu gewährleisten. Dadurch, dass eine `std::map` verwendet wird und keine explizite Vergleichsfunktion angegeben ist, werden die Keys automatisch mit `std::less` sortiert[2]. Dadurch werden neue Ausdrücke immer unten eingefügt.
Nun wird also über die Höhe des Archivs iteriert. Dabei wird zuerst in jedem Schritt ein Segment des Trenners zwischen voll- und unvollständigen Ausdrücken gezeichnet. Danach wird überprüft, ob `m_comp` noch Elemente enthält. Dies muss nicht der Fall sein, da die Höhe des Archivs durch die Länge der größeren Map bestimmt wird. Enthält `m_cons` also mehr Elemente (was tatsächlich auch wahrscheinlich ist), ist diese Überprüfung bei den letzten Iterationen eben nicht mehr wahr. Falls aber doch, ist erst mal nur das abgespeicherte `expr_repr` Objekt nötig, das in Zeile sechs durch structured Bindings extrahiert wird. Auf diesem Objekt wird dann wiederum `as_string` ausgeführt, um die Text-Repräsentation des Ausdrucks zu erhalten.
Nun muss auf mehrere Möglichkeiten reagiert werden. Je nachdem, ob das `expr_repr::f_is_highlighted` oder `expr_repr::f_is_ambiguous` gesetzt ist, werden ab Zeile acht ncurses Attribute aktiviert, um den Text in einem alternativen Erscheinungsbild zu zeichnen. Ab Zeile 13, müssen diese dann wieder abgeschaltet werden. In Zeile 12 wird der Ausdruck dann tatsächlich auf dem Bildschirm platziert. Hier müssen nicht, wie in Listing 15 - expr_queue::push_back, zwei einzelne Teile, getrennt durch `expr_repr::currpart_pos`, gezeichnet werden, da bei einem vollständigen Ausdruck diese Position immer am Ende des Ausdrucks sein muss. Als letztes, wird in Zeile 17 der Iterator auf den nächsten vollständigen Ausdruck gesetzt.
Für unvollständige Ausdrücke ist der Ablauf prinzipiell derselbe. Hier muss allerdings nicht extra auf das Flag `expr_repr::f_is_ambiguous` geachtet werden, da ein unvollständiger Ausdruck keine Mehrdeutigkeit verursachen kann. Allerdings muss, wie in Listing 15 - expr_queue::push_back bereits, der Teil vor und hinter `currpart_pos` extra gezeichnet werden.

[2] `https://en.cppreference.com/w/cpp/container/map`

Listing 23: archive::render #3

```
1 const CH::str pos_in_src_str(std::to_string(m_pos_in_src));
2 int pos = main_viewport_horizontal_center + (-src_str_center + m_pos_in_src) -
      ↪ m_x_start;
3 if(pos >= m_width)
4   pos -= (pos - m_width) + *pos_in_src_str;
5 mvsaddstr(main_viewport, m_y_start, pos + m_x_start, pos_in_src_str);
6 m_dirty_visuals = false;
7 m_dirty_dimensions = false;
```

Zum Schluss wird noch die Zahl gezeichnet, die angibt, an welcher Stelle im Quellcode das Archiv entstanden ist. Diese wird immer an der tatsächlichen Stelle im Code gezeigt und nicht zwangsweise in der Mitte des Archivs, oder über dem Trenner.
Nach dem Rendern, werden noch beide `dirty`-Flags zurückgesetzt, da das Archiv nun aktuell sein muss!

Weitere Schnittstellen eines Archivs sind:

`void add_cons(const Expr& cons)` Diese Methode ist denkbar simpel. Sie fügt den Ausdruck zu der Liste an unvollständigen Ausdrücken hinzu. Allerdings muss hier im Gegensatz zu ihrem Gegenstück `add_comp` nicht auf eine Mehrdeutigkeit überprüft werden.

`bool remove_cons(const Expr& cons)` Wie der name schon andeutet, entfernt diese Methode den übergebenen unvollständigen Ausdruck. Es wird `true` zurückgegeben, falls ein Ausdruck gefunden wird, ansonsten `false`.

`bool remove_comp(const Expr& comp)` Diese Methode arbeitet grundsätzlich genauso wie `remove_cons`. Zusätzlich muss sie aber noch überprüfen, ob der entfernte Ausdruck eine Mehrdeutigkeit ausgelöst hatte. Ist das der Fall und wurde nur ein einzelner weiterer Ausdruck mit derselben Länge gefunden, kann das `f_is_ambiguous` entfernt werden, da die Mehrdeutigkeit durch das Entfernen des einen Ausdrucks dann nicht mehr besteht.

`bool set_expr_active(const Expr& expr)` Hiermit wird angezeigt, dass der Ausdruck im Archiv aktuell verwendet, beziehungsweise kombiniert wird. Wird der angegebene Ausdruck im Archiv nicht gefunden, wird `false` zurückgegeben. Ansonsten `true`.

`bool set_expr_inactive(const Expr& expr)` Diese Methode macht die Aktion von `set_expr_active` wieder rückgängig.

`void set_y_start(uint32_t y_start)` Hiermit kann die y-Koordinate der oberen linken Ecke des Archivs gesetzt werden. Achtung: Diese Methode ruft selbst nicht `invalidate` auf, setzt aber trotzdem das `m_dirty_dimensions` Flag. Da der Aufruf zu `invalidate` hier im neuen Layouten des Archivs resultieren würde und das Layouten wiederum `set_y_start` aufruft, würde das in vielen unnötigen Aufrufen von `render` resultieren.

`void set_x_start(uint32_t y_start)` Diese Methode tut das exakt selbe wie `set_y_start` nur eben für die x-Koordinate der oberen linken Ecke des Archivs.

Doch was macht `invalidate` nun eigentlich? Die Methode teilt sich in zwei Teile auf:

Listing 24: archive::invalidate #1

```
1  auto cons_len = longest_str_len(m_cons);
2  auto comp_len = longest_str_len(m_comp);
3  m_divider_x_pos = cons_len + ARCHIVE_X_PADDING;
4  m_width = comp_len + cons_len + 2 * ARCHIVE_X_PADDING;
5  m_height = std::max(m_comp.size(), m_cons.size()) + ARCHIVE_Y_PADDING;
6  for(const archive_change_listener* listener: m_listeners)
7    listener->notify_dimensions_changed(*this);
```

Der erste Teil wird ausgeführt, wenn `m_dirty_dimensions` gesetzt ist und beginnt damit, den längsten voll- und unvollständigen Ausdruck zu finden. Als nächstes wird die x-Koordinate der Trennlinie berechnet. Diese liegt genau neben den unvollständigen Ausdrücken, ergo wird deren maximale Länge zusammen mit dem Padding für die linke Seite verrechnet, um auf den neuen Wert zu kommen. Die neue Breite berechnet sich dann aus der maximalen Länge beider Ausdrucks-Kategorien zusammen mit dem Padding für beide Seiten. Die Höhe wiederum setzt sich aus der größten Anzahl an voll- oder unvollständigen Ausdruck zusammen, mit dem Padding.
Sind dann also alle neuen Größen berechnet, werden alle Listener für diese Änderung benachrichtigt.

Der zweite Teil der Methode wird ausgeführt, wenn das Flag `m_dirty_visuals` gesetzt ist.

Listing 25: archive::invalidate #2

```
1  for(const archive_change_listener* listener: m_listeners)
2    listener->notify_visuals_changed(*this);
```

Hier müssen keine extra Berechnungen durchgeführt werden. Dies ist nebenbei auch der Hauptgrund dafür, dass zwei Flags verwaltet werden und eben nicht nur ein einzelnes, bei dem immer alle Werte neu berechnet werden.
Hier werden also ausschließlich die registrierten Listener benachrichtigt. Ein, beziehungsweise sogar der einzige, Listener ist hier der `layouter`. Das heißt, am Ende von `invalidate` wird, in beiden besprochenen Fällen, die in Abbildung 4 - Interner Ablauf des neu Zeichnens eines Archivs gestrichelt angedeutete Grenze zwischen dem Archiv und dem Layouting überschritten.

Sollte es in Zukunft mehr Interessenten an den beiden Events `dimensions_changed` und `visuals_changed` geben, können sich diese mit `void register_as_listener(archive_change_listener* listener)` und `void unregister_as_listener(archive_change_listener* listener)` registrieren und wieder entfernen. Die von diesen beiden Methoden erwartete Klasse `archive_change_listener`, muss öffentlich von einem Interessenten geerbt werden und bietet die beiden rein virtuellen Callback Methoden

```
1  virtual void notify_dimensions_changed(archive& a) const = 0;
2  virtual void notify_visuals_changed(archive& a) const = 0;
```

Wurde nun also der `layouter` aufgerufen, beschreibt der folgende Abschnitt wie dieser die Archive auf dem Fenster platziert.

4.3.8 Layouting

Wie bereits in Abschnitt 3 - Anforderungen angesprochen, sollen Archive so platziert werden, dass die Trennlinie zwischen voll- und unvollständigen Ausdrücken stets an der Position im Quellcode ist, an der auch das Archiv entstanden ist. So sieht man in Abbildung 2 - Benutzeroberfläche gut, wie sowohl die Trennlinie als auch die Position über der Trennlinie genau am Anfang des Quellcodes stehen. Kann aus platztechnischen Gründen das Archiv nicht an die gewünschte Stelle bewegt werden, ist zumindest die Positions-Nummer immer an der entsprechenden Stelle im Quellcode.
Ein weiteres, a priori klares, Kriterium ist zusätzlich, dass Archive sich in keinem Fall überlappen

dürfen.

Im Folgenden wird gezeigt, wie nun beide Kriterien eingehalten werden.

Der Header des Layouters ist recht simpel:

Listing 26: layouter.hpp

```
1  class layouter : public archive_change_listener {
2  public:
3    static layouter& the();
4      /* delete copy assignment oper. and copy constructor */
5    void notify_dimensions_changed(archive& archive_to_layout) const override;
6    void notify_visuals_changed(archive& archive_to_rerender) const override;
7  private:
8    layouter() = default;
9  };
```

Wie oben bereits angemerkt, muss die Klasse `layouter` öffentlich von `archive_change_listener` erben, um die in Zeile fünf und sechs zu sehenden Methoden zur Verfügung zu haben. Mehr Schnittstellen benötigt die Klasse nicht, da ein Aufruf einer dieser beiden Methoden genügend Information liefert, um die gewünschte Aktion auszuführen.

Während `notify_visuals_changed` sehr einfach, nur durch einen Aufruf zu `archive::render()` und einem folgenden `prepare_refresh()` des `main_viewports`, implementiert ist, ist `notify_dimensions_changed` etwas komplexer.

Listing 27: layouter::notify_dimensions_changed

```
1  const auto& layout = [](archive& archive_to_layout) { ... };
2  for(archive& a: arch_windows)
3    a.m_is_layouted = false;
4  for(archive& a: arch_windows) {
5    layout(a);
6    a.m_is_layouted = true;
7  }
8  main_viewport->clear();
9  for(auto& window: arch_windows)
10   window.render();
11 main_viewport->prepare_refresh();
```

Die Hauptarbeit dieser Methode findet in dem Lambda `layout` statt, weswegen die Implementierung in Listing 27 - layouter::notify_dimensions_changed recht einfach wirkt.
Jedes Archiv besitzt ein privates Flag `m_is_layouted`, auf das durch eine friend-Deklaration in der Klasse `archive`, hier zugegriffen werden kann. Diese Variable wird auch nicht von dem jeweiligen Archiv selbst verwaltet, sondern ausschließlich von der hier beschriebenen Methode. Zu Beginn wird dieses Flag für alle Archive zurückgesetzt. Ist dieses Zurücksetzen abgeschlossen, wird für jedes Archiv `layout` aufgerufen und anschließend mit `m_is_layouted = true` angezeigt, dass dieser Prozess für dieses bestimmte Archiv abgeschlossen ist. Wurden dann also alle Archive neu gelayouted, wird in Zeile acht der komplette Bildschirm gelöscht und daraufhin alle Archive neu gezeichnet. Diese Stelle des Codes könnte deutlich optimiert werden, indem nur tatsächlich veränderte Archive gelöscht und neu gezeichnet werden und nicht grundsätzlich alle. Abschnitt 5 - Probleme geht dabei näher auf diese Optimierungsmöglichkeit ein!

Wie bereits gesagt, findet der eigentliche Prozess des Layoutens in dem Lambda `layout` statt.

Listing 28: layouter::notify_dimensions_changed - layout

```
1 auto has_intersections = [&](const archive::rect& rect_to_test) { ... };
2 uint32_t x_coordinate = std::max(0, (main_viewport_horizontal_center -
    ↪ src_str_center + archive_to_layout.get_pos_in_src()) - archive_to_layout.
    ↪ get_divider_x_pos());
3 uint32_t y_coordinate = 0;
4 while(has_intersections({x_coordinate, y_coordinate, archive_to_layout.get_width
    ↪ (), archive_to_layout.get_height()}))
5   ++y_coordinate;
6 archive_to_layout.set_y_start(y_coordinate);
7 archive_to_layout.set_x_start(x_coordinate);
```

Zu Beginn, wird die neue x-Koordinate des Archivs bestimmt. Diese sollte, wie bereits erwähnt, so gewählt werden, dass der Trenner genau an der Position im Quelltext steht, an der das Archiv entstanden ist.

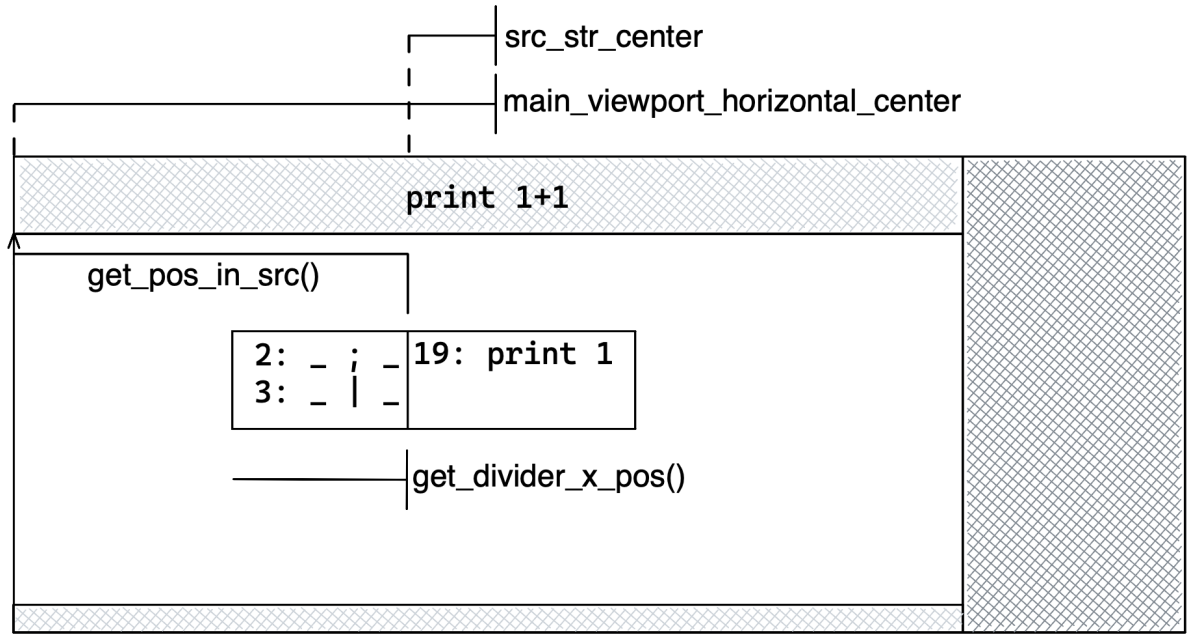


Abbildung 5: Berechnung der neuen x-Koordinate im Layouting

Abbildung 5 - Berechnung der neuen x-Koordinate im Layouting zeigt eine vereinfachte Abbildung der Applikation, in der alle nötigen Werte zur Berechnung der neuen Koordinate eingetragen sind.

Zuerst wird mit `main_viewport_horizontal_center - src_str_center` die absolute Position des Anfangs des Quelltextes berechnet. Nun muss man diese Position um `get_pos_in_src` nach rechts verschieben, um die absolute Position des Archivs im Quellcode zu bekommen. Würde man das Archiv nun an dieser Stelle zeichnen, wäre allerdings nicht der Trenner an dieser Position, sondern die obere linke Ecke des Archivs, weswegen zum Schluss noch um die Position des Trenners nach links geschoben werden muss, um alles korrekt auszurichten.

Um als nächstes die neue y-Koordinate zu bestimmen, reicht es nun nicht mehr das Archiv als Einzelnes zu betrachten, sondern es müssen alle Archive zusammen betrachtet werden. In der Reihenfolge der Berechnung der beiden Koordinaten fällt schon auf, dass das Layouting in jedem Fall die Position auf der x-Achse, vor der vertikalen Position priorisiert. Zeile vier deutet schon an, wie genau die vertikale Position bestimmt wird. So wird ein Archiv so lange nach unten geschoben, bis es mit keinem anderen mehr überlappt. `has_intersections` überprüft dabei das übergebene `archive::rect` auf eine Überschneidung mit einem bereits gelayouteten Archiv. Ein `archive::rect` ist dabei eine stark vereinfachte Version eines Archivs, die ausschließlich die Position und Größe abspeichert. Diese Klasse hier zu verwenden ist eine simple Optimierung,

die es erlaubt nicht immer neue Archive zu erstellen, um auf Überlappungen zu prüfen, sondern eben nur ein solches vereinfachtes Objekt.
Wurde also nach einer beliebigen Anzahl an Iterationen eine y-Koordinate gefunden, an der das Archiv keine Überschneidungen mehr erzeugt, wird diese vertikale zusammen mit der horizontalen Koordinate in Zeile sechs und sieben angewandt.
`has_intersections` ist wie Folgt implementiert.

Listing 29: layouter::notify_dimensions_changed - has_intersections

```
1 return std::any_of(arch_windows.begin(), arch_windows.end(), [&](const archive& a
    ↪ ) {
2     if(!a.m_is_layouted || a == archive_to_layout)
3         return false;
4   return a.intersects_with(rect_to_test);
5 });
```

`std::any_of` gibt dabei genau dann `true` zurück, wenn mindestens ein Element zwischen den übergebenen Iteratoren (hier die Liste `arch_windows`, die alle derzeit sichtbaren Archive enthält) die übergebene Funktion beziehungsweise Bedingung erfüllt. In der hier verwendeten Bedingung wird zuerst getestet, ob das Archiv hinter dem aktuellen Iterator genau das Archiv ist, mit dem `layout` aufgerufen wurde. Ist das der Fall, muss nicht auf eine Überschneidung getestet werden, da ein Element sich zwangsweise immer mit sich selbst überschneiden würde. Auch muss nicht auf Überschneidungen getestet werden, wenn das aktuell betrachtete Archiv noch nicht final gelayouted wurde, seine (y-) Position also noch nicht fix ist. In diesen beiden Fällen wird `false` zurückgegeben, da dieser Wert keinen Einfluss auf den Rückgabewert von `std::any_of` hat. Tritt keiner der beiden Fälle ein, wird mit `bool archive::intersects_with(const archive::rect&)` überprüft, ob das in der while-Schleife in Listing 28 - layouter::notify_dimensions_changed - layout erzeugte `archive::rect` sich mit dem aktuellen Archiv überschneidet. Gibt also zumindest ein Aufruf dieser Funktion `true` zurück, gibt auch `std::any_of` `true` zurück und die Schleife beginnt eine nächste Iteration mit der um eins inkrementierten y-Koordinate.

Nun sind also alle Einzelteile der Visualisierung beschrieben. Es wurde sowohl die fensterartigen Klassen wie `popup` und `scrollable` besprochen, als auch Klassen wie `expr_queue`, oder `oper_store`, die diese Fenster verwenden, um Ihren Inhalt zu präsentieren. Was an dieser Stelle allerdings noch fehlt, ist das größere Zusammenspiel all dieser Komponenten. Was sind

also die Schritte, die von einem im Parser erzeugten Event, zu dem tatsächlichen Archiv auf dem Bildschirm des Anwenders führen?

4.3.9 Interaktion zwischen allen Komponenten

Nachdem der Parser durchgelaufen ist und, wie in Unterunterabschnitt 4.2.2 - Entwurf der Schnittstelle besprochen, eine Liste mit Events gefüllt hat, kann die Visualisierung gestartet werden. Die einzige Methode die dafür verwendet werden muss, ist `int start_visualizer(const CH::str& source_string, int event_to_skip_to = 0)` und kann über den Header `visualizer.hpp` eingebunden werden. Diese Funktion bekommt als Parameter einerseits den Quelltext, den der Parser bearbeitet hat, und andererseits optional einen Event-Index, zu dem gesprungen werden soll. Wird dieser nicht angegeben, wird einfach bei dem ersten Event (Index 0) gestartet.

Start der Visualisierung

Hier werden dann diverse, im Laufe des Programms verwendete, konstante Werte berechnet. So beispielsweise

```
1  src_str_center = *src_str / 2;
```

ganz zu beginn, oder

```
1  main_viewport_horizontal_center = main_viewport->get_width() / 2;
2  main_viewport_vertical_center = main_viewport->get_height() / 2;
```

in `void setup_windows()`. Diese Funktion initialisiert, wie der Name bereits sagt, außerdem auch alle verwendeten Fenster.

Listing 30: Auszug aus setup_windows

```
1  main_viewport = new scrollable(width - QUEUE_WIDTH - 1, height - HEADER_HEIGHT -
       ↪ FOOTER_HEIGHT - 1, 0, HEADER_HEIGHT);
2  wbkgd(**main_viewport, COLOR_PAIR(STD_COLOR_PAIR));
3  main_viewport->prepare_refresh();
4
5  src_display = newwin(HEADER_HEIGHT, width - QUEUE_WIDTH, 0, 0);
6  wbkgd(src_display, COLOR_PAIR(HEADER_COLOR_PAIR));
7  mvwaddnstr(src_display, 0, getmaxx(src_display) / 2 - src_str_center, &src_str.
       ↪ elems[0], *src_str);
8  wnoutrefresh(src_display);
9
10 queue_display = new scrollable(QUEUE_WIDTH - 1, height - 1, width - QUEUE_WIDTH,
       ↪ 0);
11 wbkgd(**queue_display, COLOR_PAIR(STD_COLOR_PAIR));
12 queue_display->prepare_refresh();
```

Beim Erstellen von Fenstern muss grundsätzlich auf ”off-by-one” Fehler geachtet werden. Aus diesem Grund werden beispielsweise in Zeile neun und eins jeweils eins von Breite und Höhe abgezogen, damit die äußere Kante des Fensters nicht mehr zum Fenster selbst gehört.
Weiterhin zeigt Listing 30 - Auszug aus setup_windows wie alle drei Typen von Fenstern in diesem Projekt erstellt werden.

Eine weitere Initialisierung, die nun stattfinden muss, ist das Setzten von `next_event_it`. Diese Variable speichert einen Iterator aus der Liste aller Events, der stets auf das als nächstes auszuführende Event zeigt. Dieser Iterator wird von den beiden Funktionen `void step_n_events_forward(int n)` und `void step_n_events_backward(int n)` verwendet, um, wie die Namen schon sagen, die nächsten `n` Events auszuführen, beziehungsweise rückgängig zu machen. Folgender Code-Ausschnitt zeigt exemplarisch, wie `step_n_events_forward` implementiert ist.

Listing 31: step_n_events_forward

```
1 assert(n >= 0);
2 for(int i = 0; i < n; ++i) {
3   if(next_event_it == events.end())
4     return false;
5   ++current_event_index;
6   if((*(next_event_it++))->exec() == event::did_nothing)
7     --i;
8 }
9 return true;
```

Diese Funktion testet in Zeile drei zuerst, ob denn überhaupt noch ein nachfolgendes Event existiert. Ist dies nicht der Fall, wird `false` zurückgegeben. Andernfalls wird der aktuelle Index inkrementiert, der ausschließlich dazu verwendet wird, dem Benutzer mitzuteilen, das wievielte Event ausgeführt wurde. Das `if` Statement wirkt auf den ersten Blick möglicherweise etwas verwirrend. Zuerst wird `next_event_it` dereferenziert, um das tatsächliche `event` hinter dem Iterator zu erhalten. Da Events, durch die in Unterunterabschnitt 4.2.3 - Implementierung der Schnittstelle und Events beschriebene Vererbungshierarchie, als Zeiger abgespeichert werden, muss als Nächstes der -> Operator verwendet werden, um auf dem `event` Objekt `exec` aufzurufen, und das Event somit auszuführen. Ist all das getan, wird durch den postfix ++ Operator der Iterator auf das nächste Event gesetzt. In Unterunterabschnitt 4.2.3 - Implementierung der Schnittstelle und Events wurde ebenfalls angesprochen, dass `event::exec` und `event::undo` entweder `event::did_something`, oder `event::did_nothing` zurückgeben kann. Letzteres deutet dabei an, dass das Ausführen des Events keinen Einfluss auf die Visualisierung hatte. Ist das also der Fall, wird Zeile sieben ausgeführt und der Schleifenzähler `i` dekrementiert, um ein Event zusätzlich auszuführen. Das hat zur Folge, dass keinen Einfluss habende Events praktisch übersprungen werden. Der Parameter `n` bezieht sich also nicht auf die Anzahl der tatsächlich ausgeführten Events, sondern auf die Anzahl der Events mit dem Rückgabewert `event::did_something`.

`step_n_events_backward` funktioniert äquivalent zu der eben besprochenen Funktion!

Bevor in `start_visualizer` nun also auf Benutzer-Eingaben gewartet wird, wird noch ...

```
1 step_n_events_forward(event_to_scip_to);
```

... aufgerufen, um an das übergebene Event zu springen. Danach wird die aktuelle Event-Nummer

und der Multiplier noch angezeigt und mit `doupdate` der komplette Bildschirm das erste Mal dem Benutzer angezeigt.

Mögliche Benutzereingaben

Nun muss also auf Benutzer-Eingaben reagiert werden. Hierzu werden verschiedene Zustände verwendet:

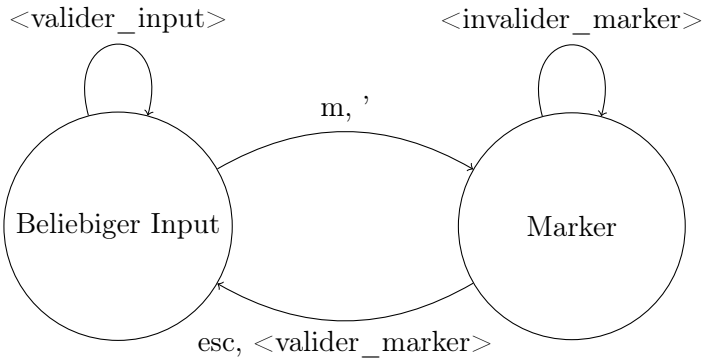


Abbildung 6: Eingabe-Zustände

Gestartet wird in dem linken Zustand. Hier hat der Benutzer nun diverse Möglichkeiten:

q beendet die Visualisierung.

n führt das nächste Event aus.

p macht das letzte Event rückgängig

o zeigt alle Operatoren in einem Popup

h zeigt alle Tastenbelegungen in einem Popup

↑ scrollt die Archive, oder das Operator-Popup einen Schritt nach unten

↓ scrollt die Archive, oder das Operator-Popup einen Schritt nach oben

w scrollt in der Queue einen Schritt nach unten.

s scrollt in der Queue einen Schritt nach oben.

m registriert einen Marker auf den aktuellen Event-Index.

' springt zu einem bereits registrierten Marker.

<Ziffer> modifiziert den Multiplier.

Wählt der Benutzer m oder ', wechselt das Programm in den rechten Zustand in Abbildung 6 - Eingabe-Zustände. Hier wird ein Marker erwartet. Als Marker sind alle ASCII Zeichen zwischen 97 und 122, sprich alle klein geschriebenen Buchstaben. Natürlich können auch Buchstaben, die sonst zu einer Funktion gebunden wären, verwendet werden. Wird also ein korrekter Marker, oder die Taste Escape eingegeben, wechselt der Automat wieder in seinen Startzustand.
Die letzte Option des Startzustandes (<Ziffer>) verhält sich etwas anders als alle anderen. Gibt der Benutzer nacheinander eine 1 und 2 ein, wird dies nicht als zwei disjunkte Aktionen, sondern als die Zahl 12 interpretiert. Diese fungiert dann als Multiplikator für die meisten anderen Aktionen. Wird also nacheinander 1, 2, n eingegeben, wird 12 mal die Aktion n ausgeführt. Wird eine Aktion gewählt, die den Multiplier nicht nutzen kann, beispielsweise m, verfällt dieser.
Werden in einem der beiden Zustände Eingaben getätigt, die der Zustand nicht verarbeiten kann, löst dies ein Blinken und Piepen des Terminals aus.

Die Implementierung dieses Zustandssystems ist recht simpel und und ohne Überraschungen. Es wird lediglich eine kleinere Optimierung vorgenommen. Nach jeder ausgeführten Aktion, besteht die Möglichkeit zu einem von zwei Labels zu springen. `skip_with_refresh` ist dabei der Default. Hier werden alle relevanten Informationen neu gezeichnet und mit `doupdate` auch veränderte Fenster aktualisiert. Ist allerdings ein Fehler aufgetreten, wurde keine, den Bildschirm ändernde, Aktion ausgeführt! Ist dies also der Fall, kann das Label `skip_without_refresh`, das alle Aktualisierungen überspringt und zuletzt nur das oben angesprochene Blinken und Piepen erzeugt.

Architektur

Da nun langsam das Ende der Arbeit erreicht ist, und die größten und wichtigsten Komponenten alle beschrieben sind, lässt sich vielleicht schon erahnen, welcher Architektur diese Applikation gefolgt ist.

Und zwar wurde das Command Processor Pattern verwendet.s Dieses Design-Pattern kapselt Aktionen in eigene Objekte, deren Ausführung dann von einer zentralen Komponente gesteuert wird [Bus+96, S. 173]. Dabei sollte es eine abstrakte Klasse geben, von der alle Aktionen abgeleitet werden und die um zusätzliche Methoden ergänzt werden kann, um den Aktionen neue Funktionalität hinzufügen zu können. Der Command-Processor hat in diesem Pattern die größte Kontrolle, er kann entscheiden, ob, wann und beispielsweise wie oft einzelne Aktionen ausgeführt

werden. Die letzte Komponente ist der so genannte Supplier. Er stellt weiter Komponenten beziehungsweise Funktionen zur Verfügung, die zum Ausführen der Aktionen relevant sind [Bus+96, 176ff.].
Die Aktionen, oder hier: Events, wurden bereits in Unterunterabschnitt 4.2.3 - Implementierung der Schnittstelle und Events ausführlich besprochen. Der Command-Processor ist hier in den Funktionen `step_n_events_forward` und `step_n_events_backward`, beziehungsweise auchd er Liste `events` versteckt. Welche Komponente die Rollte des Suppliers hier übernimmt, ist allerdings nicht ganz offensichtlich. Diese Aufgabe wird sich nämlich von mehreren Komponenten geteilt. So gehören beispielsweise der `oper_store`, oder die `expr_queue` dazu. Also vereinfacht gesagt, alle, einen Bereich verwaltende, Fenster gehören zum Supplier.
Alle einzelnen Komponenten aus diesem Pattern sind also bereits erklärt. Was als letztes nun noch fehlt, ist die tatsächliche Implementierung von `event::exec` und `event::undo` diverser Events.

Ausführen und rückgängig machen von Events

Exemplarisch wird hier das Event `add_expr_to_queue_event` betrachtet.

Listing 32: add_expr_to_queue_event::exec

```
1 if(!m_is_comp)
2   oper_store::the().insert_if_prototyp(m_data);
3 expr_queue::the().push_back(m_data, m_is_comp);
4 return event::did_something;
```

Wie in Unterunterabschnitt 4.3.6 - Liste aller Operatoren bereits beschrieben, ist die erste Aufgabe von `add_expr_to_queue_event::exec`, den Ausdruck im `oper_store` zu registrieren. Da dort nur Prototypen registriert werden, ist es etwas effizienter `oper_store::insert_if_prototyp` erst gar nicht aufzurufen, falls der Ausdruck des Events ein vollständiger ist. Zeile drei fügt den Ausdruck dann der `expr_queue` hinzu und gibt anschließend `did_something` zurück, da das Hinzufügen in die Warteschlange nicht fehlschlagen kann.

Das Entfernen aus der Queue kann, wie der Folgende Code zeigt, allerdings schon fehlschlagen.

Listing 33: add_expr_to_queue_event::undo

```
1  return expr_queue::the().pop_back() ? event::did_something : event::did_nothing;
```

Je nach Rückgabewert von `expr_queue::pop_back` kann hier also auch `did_nothing` zurückgegeben werden. Dies sollte in der Realität aber eigentlich nicht vorkommen, da vor `undo` zwangsweise `exec` aufgerufen wurde und dadurch immer ein Ausdruck zum Löschen vorhanden sein muss, wenn man in `undo` ankommt.
Tatsächlich wäre es hier sogar sinnvoller, mit einer Assertion abzufangen, dass ein zu löschender Ausdruck vorhanden ist. Um hier aber auch die Verwendung von `did_nothing` zeigen zu können, wurde dies hier aber nicht getan.

Alle anderen Events sind, von dem Gedanken her, ähnlich implementiert. Die meisten werden eine der, einen Bereich verwaltenden Klasse verwenden, um einen Ausdruck hinzuzufügen, zu entfernen, oder zu modifizieren.

4.3.10 Hilfsklassen und kleinere Details

In diversen Code-Ausschnitten wurden immer wieder kleinere Komponenten, oder Konzepte verwendet, auf die nicht direkt eingegangen wurde. Diese kleineren Bestandteile bekommen nun hier ihre Erwähnung.

Oft wurden in diesem Projekt Makros verwendet, um konstante Werte leserlicher zu gestalten. Diese Makros, wie beispielsweise ...

```
1  #define ARCHIVE_X_PADDING 5
2  #define ARCHIVE_Y_PADDING 2
3  #define REPLACE_WITH_ID_THRESHOLD 15
4  /* ... */
```

... und sind alle in der Datei `constants.hpp` definiert.
Weitere Dateien mit konstanten Werten, sind `coordinated.hpp` und `windows.hpp`. Erstere enthält Positions bezogene Werte, wie:

```
1  extern int width;
2  extern int height;
3  extern int main_viewport_horizontal_center;
4  /* ... */
```

In `windows.hpp` hingegen, sind alle, in `setup_windows` erstellten, Fenster deklariert.
Alle drei Header binden selbst keine weiteren Dateien aus diesem Projekt ein, können also immer verwendet werden, ohne auf zirkuläre Abhängigkeiten achten zu müssen.

Neben Dateien mit Konstanten, gibt es auch noch einen Header, der Hilfsfunktionen zur Verfügung stellt. `utils.hpp` enthält die Funktionen `void center_text_hor(WINDOW* window, const CH::str& str, uint32_t y)` und `CH::str truncate_string_to_length(const CH::str& string, uint32_t length)`. Erstere zentriert einen Text horizontal in einem Fenster und wird beispielsweise in `setup_windows` verwendet. `truncate_string_to_length` wird immer dann verwendet, wenn ein Text ab einer bestimmten Länge angeschnitten werden soll. Dabei wird der Text aber nicht nur abgeschnitten, sondern am Ende durch `...` ergänzt, um die Verkürzung kenntlich zu machen.

Wie `truncate_string_to_length` schon angedeutet hat, wird in diesem Projekt, anstelle der Klasse `std::string` aus der Standard-Bibliothek, die Klasse `CH::str` aus der Bibliothek libCH verwendet. Da ncurses diese Klasse aber nicht kennt und selbst `char*` verwendet, wird oft ein kleiner Trick angewandt um `CH::str` zu einem Character-Zeiger zu konvertieren.
`CH::str` ist nichts anderes als eine Liste an `chars` [Hei21b], auf die mit `str.elems` zugegriffen werden kann. Da ein klassischer C-String nichts anderes, als ein Zeiger auf das erste Element ist, kann die Konvertierung ganz einfach folgendermaßen erfolgen:

```
1 &str.elems[0];
```

Wichtig dabei ist allerdings, dass der so erhaltene C-String nicht durch ein Null-Byte terminiert ist. Diese Konvertierung sollte also nur mit Funktionen verwendet werden, die auch die Länge des Textes als Argument übergeben bekommen (beispielsweise `mvwaddnstr`)!

Eine weitere Methode, die auch mit `CH::str` Objekten arbeitet, ist `expr_repr::as_string`. Wie bereits erwähnt, wird diese Klasse, beziehungsweise diese Methode immer verwendet, wenn ein Ausdruck als Text ausgegeben wird. Beim Erstellen eines `expr_reprs`, wird neben der `Expr` selbst, auch eine Hand voll Flags angegeben:

Listing 34: expr_repr::flags

```
enum flags {
  f_is_comp = 1 << 0,
  f_is_highlighted = 1 << 1,
  f_is_ambiguous = 1 << 2,
  f_is_prototype = 1 << 3,
};
```

Diese Flags bestimmen, wie der von `as_string` erzeugte Text zusammengesetzt wird. Das `f_is_prototype`-Flag wird dabei vom Konstruktor der Klasse selbst gesetzt.

`as_string`. Formatiert den Text dann folgendermaßen:

```
<Text> ::= (ID | ?): <Ausdruck>
<Ausdruck> ::= (Signatur | Gelesener_Text | Gelesener_Text Signatur)
```

Zu Beginn steht entweder die ID des Haupt-Operators des Ausdrucks, die die Klasse `oper_store` bereitstellt, oder ein Fragezeichen, falls der Operator noch nicht gesehen wurde, was in der Regel aber nicht passieren sollte. Der, auf die ID folgende, Ausdrucks-Text, kann auf drei verschiedene Möglichkeiten aufgebraut sein:

Signatur Es wird ausschließlich die Signatur des Ausdrucks dargestellt, falls der Ausdruck ein Prototyp ist, also das `f_is_prototype` Flag gesetzt ist.

Gelesener Text Ist der Ausdruck vollständig (`f_is_comp`), wird der exakte Quelltext angezeigt, den der Ausdruck repräsentiert.

Gelesener Text Signatur Ist ein Ausdruck teilweise vollständig, wird der gelesene Text so weit angezeigt, wie der Ausdruck bereits abgearbeitet wurde. Darauf folgt der Teil der Signatur, der als nächstes erwartet wird. Diese Grenze wird durch das Attribut `currpart_` in dem offenen Typ `Expr` [Hei21a, S. 7], beziehungsweise `currpart_pos` in `expr_repr` bestimmt.

Da aus Platz-Gründen Archive und die Queue nicht zu breit werden sollten, wird, im letzten Schritt dieser Methode, der Ausdrucks-Text auf eine maximale Länge gekürzt. Dies wird allerdings nur mit Prototypen, die bereits eine ID haben, gemacht, da das die einzigen Ausdrücke sind, die im Operatoren-Popup denselben Text als auch in den Archiven haben (da sowohl in den Archiven als auch im Popup für Prototypen nur die Signatur dargestellt wird).

In einigen hier beschriebenen Klassen, kann einem der Kommentar ”delete copy assignment oper. and copy constructor” zusammen mit der statischen Methode `the` aufgefallen sein. Diese Klassen sind als Singleton implementiert. Genauer gesagt, sind das hier alle Klasse, die einen bestimmten Bereich verwalten. Also: `expr_queue`, `layouter`, `oper_store` und `popup_manager`.
Alle diese Klassen lassen sich, durch das Löschen des Kopier-Konstruktors und Kopier-Zuweisungs-Operators, nicht kopieren und die einzige Instanz dieser Klassen, kann über die `the` Methode erhalten werden.

5 Probleme

Das einzige, größere Problem, das hier leider nicht mehr behoben wurde, ist ein nicht-funktionales. Wie in Unterunterabschnitt 4.3.8 - Layouting bereits angerissen, löscht das Layouting grundsätzlich immer den gesamten Bildschirm und zeichnet daraufhin alle Archive neu. Nun ist `archive::render` keine extrem aufwändige Methode und dieses Problem, deswegen auch kein extrem schlimmes, eine gute Optimierungsmöglichkeit bietet es aber wohl trotzdem.

Schöner wäre es, wenn `layouter::notify_dimensions_changed` nur noch das übergebene Archiv vom Bildschirm löschen würde und eben nicht alle. In `archive::render` könnte zu beginnt dann überprüft werden, ob eines der beiden `dirty`-Flags gesetzt ist und falls nicht, würde diese Methode direkt terminieren. Die Klasse `layouter` könnte dann trotzdem alle Archive neu zeichnen, da bei den den meisten Archiven, dieser Aufruf nicht in einem tatsächlichen neu zeichnen resultieren würde.
Diese Optimierung würde bei `print 1 + 2` 28 komplette Ausführungen von `archive::render` verhindern.
Das größte Problem bei der Umsetzung, sind die eingeschränkten Möglichkeiten Content auf dem Bildschirm zu löschen. Da man nicht einfach einen Bereich angeben kann, sondern nur von einer Startposition, bis zu einem Zeilenumbruch, oder dem Fenster-Ende löschen kann (s. Abschnitt 2.2.4 - Manipulieren von Fenstern sind diese Möglichkeiten leider etwas zu eingeschränkt, um ein Archiv zuverlässig und effizient zu löschen.

Literatur

[Bus+96] Frank Buschmann u. a. *Pattern-Oriented Software Architecture*. Bd. 1. Wiley, 1996. ISBN: 9780471958697.

[Hei21a] Prof. Dr. Christian Heinlein. *Kernteile eines MOSTflexiPL-Compilers*. 2021.

[Hei21b] Prof. Dr. Christian Heinlein. *LibCH C++-Bibliothek*. 2021.