DIPLOMARBEIT

Entwicklung und Konzeption einer interpretierten Programmiersprache

Lukas Singer
lukas42singer@gmail.com

Studiengang: Technische Informatik (DI (FH))
Fakultät: Angewandte Computer- und Biowissenschaften
Matrikel Nr: 35051
Seminargruppe: KT11wIA-F

Prof. Dr.-Ing. Uwe Schneider
Prof. Dr. Thomas Haenselmann

19. Jänner 2016

DIPLOMA THESIS

Development and conception of an intepreted programming language

Lukas Singer
lukas42singer@gmail.com

Course of studies: Computer Engineering (DI (FH))
Faculty: Applied Computer Sciences & Biosciences
Student no: 35051
Seminar group: KT11wIA-F

Prof. Dr.-Ing. Uwe Schneider
Prof. Dr. Thomas Haenselmann

January 19th, 2016

Bibliografische Beschreibung:

Name: Singer, Lukas

Thema: Entwicklung und Konzeption einer interpretierten Programmiersprache.

Betreuer: Prof. Dr.-Ing. Uwe Schneider, Prof. Dr. Thomas Haenselmann

Hochschule Mittweida, Fakultät: Angewandte Computer- und Biowissenschaften. Diplomarbeit. Inzing, 2016. 115 Seiten: davon 1 Eidesstattliche Erklärung, 1 Glossar, 1 Inhaltsverzeichnis, 3 Abbildungen, 3 Tabellen, 1 Literaturverzeichnis und 66 Seiten Anhang.

Referat:

Diese Diplomarbeit behandelt sowohl die theoretischen Grundlagen zur Entwicklung und Konzeption einer Programmiersprache als auch die praktische Umsetzung eines Interpreters. Es wird auf die eingesetzten Werkzeuge, sowie auf die zugrundeliegenden Mechanismen eingegangen. Anhand von Beispielen werden zuerst die Bausteine des Interpreters erläutert und später einige Konzepte wie Built-In-Funktionen und Speicherverwaltung genauer vorgestellt.

Ergebnis der Diplomarbeit ist ein in C implementierter Interpreter für die entworfene Programmiersprache sowie Konzepte und Grundlagen zur Erweiterung und Weiterentwicklung der Programmiersprache.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Inzing, 19. Jänner 2016

Abstrakt

Die vorliegende Diplomarbeit baut auf die Ergebnisse des Praxisprojekts[1] auf und geht weiter auf Theorien und Techniken der Compiler- und Interpreterentwicklung sowie auf die konkrete Implementierung einer Programmiersprache ein.

Nach einem kurzen Rückblick, welcher die Ergebnisse des Praxisprojekts und somit die Ausgangsituation für die Diplomarbeit betrachtet, wird auf den Aufbau des Interpreters und dessen Bestandteile Lexer und Parser eingegangen. Anhand eines Beispiels wird der komplexe Parsevorgang erläutert um die zugrundeliegenden Mechanismen eines LR-Parsers verständlich darzustellen.

Im Kapitel *2. Built-In-Funktionen* werden fest im Interpreter implementierte Funktionen kategorisiert und die Motivation und Gründe für die jeweiligen Funktionstypen aufgezeigt. In diesem Kapitel wird anhand von vielen Beispielen bereits ein Einblick in die internen Mechanismen des Interpreters gegeben.

Das Kapitel *3. Interpreter* behandelt nun die internen Strukturen und Abläufe. Es wird auf die Kernelemente des Interpreters eingegangen und jeweils anhand von realen Implementierungen gezeigt, wie das Programm verschiedene Aufgaben umsetzt und wie diese zusammenspielen. Es wird auf die Umsetzungen der virtuellen Maschine, der Ausführungsumgebung, der Operatoren, der Fehlerbehandlung und der Datentypen näher eingegangen, sodass ein vollständiges Bild von den internen Abläufe des Interpreters entsteht.

Bevor auf die automatische Speicherverwaltung mittels Referenzzählung genauer eingegangen wird, zeigt ein Kapitel noch auf, welche Optimierungspotentiale in einer Interpreterimplementierung möglich sind und wie es einige einfache Konzepte erlauben möglichst sichere und dennoch performante Multithreadinganwendungen in einer interpretierten Programmiersprache umzusetzen.

Ergebnis der Diplomarbeit ist eine vollständig in C implementierte und dadurch plattformunabhängige Programmiersprache. Sie bietet neben einer einfachen Schnittstelle für Built-In-Funktionen vor allem eine einfache und leicht verständliche Syntax. Dank des schlanken Typsystems und der automatischen Speicherverwaltung werden dem Programmierer viele Aufgaben wie Typdeklarationen oder Speicherallokation abgenommen, sodass beim Programmieren der Fokus auf Lösungen und Algorithmen liegt.

1 *Basisarbeit und Konzept zum Entwurf und zur Implementierung einer Programmiersprache* [1]

Inhaltsverzeichnis

Glossar

AST	Abstract syntax tree (abstrakter Syntaxbaum)
BNF	Backus-Naur-Form
C	Programmiersprache C
CPU	Central processing unit
DBL	Double
DLL	Dynamic link library
ELIF	Else if
ENV	Environment
EOF	End of file
GNU	Gnu is not Unix
ID	Identifikator
INT	Integer
LALR	Lookahead left to right
LR	Left to right
OS	Operating system (Betriebssystem)
PFOR	Parallel-for
POSIX	Portable operating system interface
QPL	Quick programming language
RAM	Random access memory
SO	Shared object
STMNT	Statement
STR	String
VM	Virtuelle Maschine

0. Rückblick und Ausgangssituation

Im Februar 2015 wurde das Praxisprojekt unter dem Titel *Basisarbeit und Konzept zum Entwurf und zur Implementierung einer Programmiersprache* fertiggestellt. Nach einer kurzen allgemeinen Einführung über Programmiersprachen wurden darin Herangehensweisen und Ideen diskutiert, die bei der Entwicklung einer Programmiersprache von Bedeutung sind.

Verschiedene Programmiersprachen wurden anhand ihrer Implementierungen gegenübergestellt und die Unterschiede zwischen interpretiertem und nativem Code wurden betrachtet. Es wurde auch ein Einblick in verschiedene Programmierparadigmen gegeben um für den späteren Entwurf der eigenen Programmiersprache auf ein breites Repertoire an Ideen und Einflüssen zurückgreifen zu können.

Anhand eines Praxisbeispiels wurde gezeigt, wie praxisrelevant kleine Programmier- oder Skriptsprachen sind und wie die Mechanismen in einem einfachen selbst geschriebenen Kommandointerpreter intern funktionieren.

Um eine solide Grundlage für diese Diplomarbeit zu schaffen, wurde im Praxisprojekt auch bereits ein erster Blick auf formale Sprachen und die Anwendung der Tools GNU Bison und Flex geworfen. Anhand von einfachen Beispielen wurde gezeigt, wie mit Hilfe dieser Tools ein abstrakter Syntaxbaum erzeugt werden kann.

Den Hauptteil des Praxisprojekts und auch den wertvollsten Beitrag für die vorliegende Diplomarbeit lieferte das Kapitel *Sprachdefinition*, in welchem bereits viele Ideen für die Programmiersprache diskutiert wurden. Neben Beschreibungen der angestrebten Datentypen und der geplanten Kontrollkonstrukte wurden die verfügbaren Operatoren und deren Verhalten beschrieben.

Zu den Ergebnissen des Praxisprojekts zählt neben den formulierten Anforderungen an die Programmiersprache eine Syntaxbeschreibung in BNF, die dem Anhang beigefügt wurde und als Vorlage für Lexer und Parser der QPL diente.

1. Techniken des Compiler- und Interpreterbaus

In diesem Kapitel werden die Bestandteile eines Compilers/Interpreters betrachtet und deren Zusammenspiel erläutert. Es wird auf die theoretischen Grundlagen sowie auf die konkrete praktische Umsetzung in QPL eingegangen.

Rückblickend auf das Praxisprojekt [1] wird weiter auf Parser und Lexer eingegangen und deren praktische Umsetzung dargestellt.

1.1 Logischer Aufbau

[2] Compiler und Interpreter sind sich im Aufbau ähnlich. Der Unterschied besteht hauptsächlich darin, dass ein Compiler generierten Code zur späteren Ausführung in eine Datei speichert, wohingegen ein Interpreter den generierten Code sofort ausführt. Im Detail gibt es natürlich noch mehr Unterschiede, wobei die Grenze zwischen Compiler und Interpreter oft fließend ist.

Ein Compiler bzw. Interpreter besteht aus folgenden Bestandteilen:

1. Frontend: nimmt Befehle entgegen, steuert die weiteren Bestandteile und lädt Quelldateien.
2. Lexer: liest die Quelldateien und erzeugt einen Tokenstream.
3. Parser: liest den Tokenstream und erzeugt einen abstrakten Syntaxbaum.

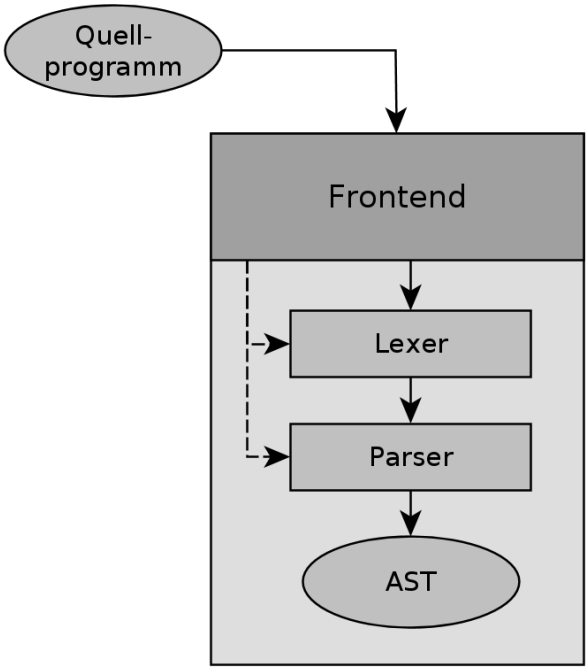
Je nachdem ob es sich um Compiler oder Interpreter handelt werden diese drei grundlegenden Bestandteile noch um folgende ergänzt:

1. Codegenerator (Compiler): erzeugt einen Code aus dem abstrakten Syntaxbaum.
2. Ausführungseinheit (Interpreter): führt einzelne Sequenzen oder den gesamten abstrakten Syntaxbaum aus.

Die konkrete Implementierung der QPL setzt die drei Grundbausteine (Frontend, Lexer, Parser) und eine Ausführungseinheit um. Die Ausführungseinheit übernimmt einen komplett geparsten AST und führt diesen aus. Im Gegensatz zu vielen anderen Interpreterimplementierungen ermöglicht diese Architektur eine vollständige syntaktische Prüfung des Quelltextes bevor dieser ausgeführt wird.

Folgende Abbildung stellt den architektonischen Zusammenhang der beschriebenen

Elemente dar.



Übersicht Frontend, Lexer und Parser

1.2 Lexer

[3] Der Lexer stellt das erste Gate in einem Compiler oder Interpreter dar. Seine Aufgabe besteht darin die Eingabe (das Quellprogramm) einzulesen und einen Tokenstream zu erzeugen. Alle weiteren Bestandteile arbeiten mit internen Codedarstellungen, nur der Lexer hat Kontakt mit dem Quellprogramm. Diese Tatsache erlaubt es im Lexer neben der Erzeugung des Tokenstreams noch weitere Aufgaben zu erledigen, die an anderen Stellen nicht mehr möglich sind[2]:

- Zuordnung von Zeilennummern zu Code
- Zuordnung von Zeilennummern zu Fehlermeldungen
- Entfernung von Kommentaren und Leerraum
- Steuern des Präprozessors und Auflösen von Makros

Die Aufgaben eines Lexers können weiter in Scannen und lexikalisches Analysieren unterteilt werden. Zum Scannen gehören die deutlich einfacheren Aufgaben wie Entfernung von überflüssigem Leerraum und Entfernung von Kommentaren. Die eigentliche

2 Nicht alle hier genannten Aufgaben sind für QPL relevant.

lexikalische Analyse löst weitaus komplexere Aufgaben, wie die Erkennung von Literalen, Bezeichnern und Schlüsselwörtern.

Der Lexer der QPL wurde mit Hilfe des Tools Flex erstellt. Das Eingabeformat von Flex bietet eine Abstraktion der Interna des Lexers, sodass die erwarteten und gültigen Tokens mittels regulärer Ausdrücke beschrieben werden können [4].

Beim Entwurf eines Lexers ist es wichtig Token, Muster und Lexeme unterscheiden zu können und zu wissen wofür diese verwendet werden.

Tokens sind Paare aus eindeutigen Bezeichnern und zugehörigem optionalen Attributwert. Beispiele für Tokens sind Schlüsselwörter (z.B.: T_IF) und Literale (z.B.: T_NUMBER)

Im Folgenden sind ein paar Fakten und Regeln gelistet, die beim Entwurf eines Lexers berücksichtigt werden müssen:

1. Ein Token je Schlüsselwort
2. Ein oder mehr Tokens für Operatoren
3. Ein Token für Bezeichner
4. Ein oder mehr Tokens für Konstanten
5. Ein Token für jedes Satzzeichensymbol (Klammern, Punkt, Semikolon, …)

Unter *Muster* versteht man die möglichen Erscheinungsformen von Tokens. Das Schlüsselwort `if` (T_IF) hat zum Beispiel das eindeutige Muster `if`, wohingegen ein Zahlenliteral zum Beispiel durch folgenden regulären Ausdruck beschrieben werden kann: `(0|[1-9][0-9]*)`. Das Muster beschreibt eine beliebige Zahlenfolge.

Ein *Lexem* ist die tatsächliche Darstellung eines Tokens im Quelltext. Also ein Ausschnitt des Quelltextes der von einem Muster für ein Token abgedeckt ist.

Folgendes Beispiel stellt die Beziehung zwischen *Token*, *Muster* und *Lexem* dar:

```
printf("Total = %d\n", score);
```

Hierbei entsprechen die Ausdrücke `printf` und `score` als Lexeme dem Muster des Tokens für Identifikatoren. `"Total = %d\n"` entspricht als Lexem dem Muster des Tokens für Stringliterale.

Wichtige Attribute für Tokens die vom Parser für den Aufbau des abstrakten Syntaxbaumes sowie zur Generierung von Fehlermeldungen oder Hinweisen und auch zur Steuerung des

Laufzeitverhaltens benötigt werden, sind unter anderem die zugehörige Zeilennummern und das zugehörige Literal.

1.2.1 Lexikalische Fehler

Im Lexer können bereits erste Fehleranalysen durchgeführt werden um Rückmeldungen in Form von Fehlermeldungen mit anschließendem Abbruch oder Warnhinweisen zu geben. Die meisten Fehler können jedoch nicht vom Lexer erkannt werden. Als Beispiel möchte ich hier einen Tippfehler bei Schlüsselwörtern angeben.

```
fi( a == f(x) ) { ...
```

Hier ist offensichtlich, dass es sich um einen Schreibfehler handelt und statt `fi` das Schlüsselwort `if` eingesetzt werden sollte. Der Lexer erkennt `fi` jedoch als Identifikator und gibt somit auch den Identifikator als Token dem Parser weiter, wo dann auch der Fehler erkannt wird.

Eine Art von Fehlern, die jedoch vom Lexer sicher erkannt werden können, sind Fehler in Literalen. Verwendet eine Programmiersprache beispielsweise vorzeichenbehaftete 32 Bit Ganzzahlen für die Darstellung von ganzzahligen Literalen dürfen diese nur im Bereich -2147483648 bis 2147483647[3] liegen. Werte die außerhalb dieses Bereichs liegen können bereits vom Lexer erkannt werden und zu einer Fehlermeldung führen. Weiters ist abhängig von der Darstellung von Stringliteralen oft nur eine bestimmte Anzahl an Zeichen in einem Stringliteral möglich, eine Überschreitung dieser Obergrenze kann auch bereits vom Lexer erkannt werden und zu einer Fehlermeldung führen.

Neben dieser aktiven Fehleranalyse können in einem Lexer auch passive Fehleranalysen zum Einsatz kommen. Wird ein Quelltextausschnitt gelesen, der von keinem *Muster* erkannt wird, führt dies auch zum Abbruch mit Fehlermeldung. Als Beispiel hierfür möchte ich Umlaute oder nicht verwendete Sonderzeichen nennen.

Diese passive Fehleranalyse ist implizit und resultiert daraus, dass es nicht für jede mögliche Eingabe ein passendes *Muster* gibt.

1.3 Parser

[5] Der Parser ist das zweite wichtige Gate, das vom Quellprogramm passiert werden muss,

3 Annahme: Signed 32 Bit Integer, unsigned wäre 0 bis 4294967295 möglich.

bevor es zur Ausführung kommt. Die Aufgabe des Parsers besteht darin, aus dem vom Lexer erzeugten Tokenstream einen abstrakten Syntaxbaum zu erzeugen. Eingabe und Ausgabe des Parsers sind interne Codedarstellungen, der Parser kennt das zugrundeliegende Quellprogramm nicht und führt dieses auch nicht aus.

Für die Erstellung des Parsers für QPL wurde Bison[4] eingesetzt. Die Beschreibung des Parsers erfolgt durch eine BNF-ähnliche kontextfreie Grammatik[5], welche in eine C-Implementierung eines Lookahead-LR-Parsers (LALR Parser) umgesetzt wird [6].

1.3.1 LR-Parser

LR-Parser gehören zur Familie der Bottom-Up-Parser. Die Bezeichnung LR setzt sich aus folgenden Teilen zusammen:

L: Die Eingabetokens werden von *links* nach rechts gelesen und zwischengespeichert.

R: Die Ausgabe wird in umgekehrter Reihenfolge als *rechts*bündiger Baum erzeugt.

Die wesentlichen Vorteile eines LR-Parsers und die Gründe, warum die meisten heute eingesetzten Parser diese Technik umsetzen, sind [2]:

- frühes Erkennen von Syntaxfehlern.
- jede beliebige kontextfreie Grammatik kann umgesetzt werden.
- können durch Verzicht auf Zurückverfolgung (Backtracking) sehr effizient implementiert werden

Die Funktionsweise eines LR-Parsers lässt sich am einfachsten anhand eines Beispiels erläutern. Es wird eine äußerst einfache Grammatik angenommen, um den Fokus auf den Parsevorgang zu richten. Die Grammatik lässt sich in BNF folgendermaßen darstellen:

```
E ::= E "*" B
    | E "+" B
    | B

B ::= "0"
    | "1"
```

4 GNU Bison ist eine freie Implementierung des UNIX Tools Yacc

5 Eine Beschreibung der Eingabesyntax mit einfachen Beispielen kann im Praxisprojekt „Basisarbeit und Konzept zum Entwurf und zur Implementierung einer Programmiersprache“, Lukas Singer, 2015 nachgelesen werden.

Die abgebildete Grammatik erlaubt Additionen und Subtraktionen von Einsen und Nullen. Aus der Grammatik lassen sich folgende Reduktionsregeln ableiten:

```
R1:  E * B ← E

R2:  E + B ← E

R3:  B ← E

R4:  0 ← B

R5:  1 ← B
```

Um nun eine Eingabe zu parsen bietet es sich an, eine Tabelle aufzustellen, in der zu jedem Zustand und zu jeder Eingabe bzw. zu jedem Folgezeichen eine Shift- oder Reduce-Aktion hinterlegt ist.

Zustand	**Aktion**					**GOTO**	
	*****	**+**	**0**	**1**	**EOF**	**E**	**B**
0			S1	S2		3	4
1	R4	R4	R4	R4	R4		
2	R5	R5	R5	R5	R5		
3	S5	S6			OK		
4	R3	R3	R3	R3	R3		
5			S1	S2			7
6			S1	S2			8
7	R1	R1	R1	R1	R1		
8	R2	R2	R2	R2	R2		

Parsetabelle

Wie die Tabelle zu lesen ist, wird anhand des Beispiels erklärt. Es wird von folgender Eingabe mit abschließendem EOF[6] ausgegangen:

```
1+1
```

1. Der Parser startet im Zustand 0. Es wird das erste Zeichen eingelesen. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 0 bei Eingabe 1 die Aktion S2 ausgeführt werden muss.
   Es wird in den Zustand 2 gewechselt und es wird 2 auf den Stack gelegt (Shift).
2. Der Parser befindet sich im Zustand 2. Es wird das nächste Zeichen gelesen. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 2 bei Eingabe + die

6 EOF: End Of File; ASCII Wert: 0x04; abschließendes Zeichen einer Datei oder eines Streams.

Aktion R5 (1 ← B) angewandt werden muss.
2 wird vom Stack genommen und verworfen (reduziert). Nun wird aus Zustand 0 das GOTO-Ziel für B entnommen.
Es wird in den Zustand 4 gewechselt und es wird 4 auf den Stack gelegt.

3. Der Parser befindet sich im Zustand 4. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 4 bei Eingabe + die Aktion R3 (B ← E) angewandt werden muss.
   4 wird vom Stack genommen und verworfen. Es wird in den Zustand 3 gewechselt und es wird 3 auf den Stack gelegt.
4. Der Parser befindet sich im Zustand 3. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 3 bei Eingabe + die Aktion S6 angewandt werden muss.
   Es wird in den Zustand 6 gewechselt und es wird 6 auf den Stack gelegt.
5. Der Parser befindet sich im Zustand 6. Es wird das nächste Zeichen gelesen. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 6 bei Eingabe 1 die Aktion S2 angewandt werden muss.
   Es wird in den Zustand 2 gewechselt und es wird 2 auf den Stack gelegt.
6. Der Parser befindet sich im Zustand 2. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 2 bei Eingabe 1 die Aktion R5 (B ← 1) angewandt werden muss.
   2 wird vom Stack genommen und verworfen. Aus dem Eintrag der zugehörigen GOTO-Tabelle geht hervor, dass in den Zustand 8 gewechselt werden muss. Es wird in den Zustand 8 gewechselt und es wird 8 auf den Stack gelegt.
7. Der Parser befindet sich im Zustand 8. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 8 bei Eingabe 1 die Aktion R2 (E + B ← E) angewandt werden muss.
   8, 6 und 3 werden vom Stack genommen und verworfen. Nun wird aus Zustand 0 das GOTO-Ziel für E entnommen.
   Es wird in den Zustand 3 gewechselt und es wird 3 auf den Stack gelegt
8. Der Parser befindet sich im Zustand 3. Es wird EOF festgestellt. Aus der Aktionstabelle kann entnommen werden, dass im Zustand 3 bei EOF die Eingabe gültig war und anhand der Reduktionsregeln auf ein Startsymbol reduziert werden konnte.

1.3.2 Semantische Fehler

Den Großteil der Fehleranalyse, die vor der Ausführung des Programms durchgeführt werden kann, wird vom Parser erledigt. Der Parser kann syntaktische Fehler erkennen und hierzu Fehlermeldungen erzeugen.

Kann eine Eingabe nicht anhand der sich ergebenden Reduktionsregeln auf ein Startsymbol reduziert werden, führt dies zu Parserfehlern, die zum Abbruch des Parsevorgangs führen.

Typische semantische Fehler, die vom Parser erkannt werden können, sind unabgeschlossene Statements, fehlende schließende Klammern oder falsche Verwendung von Operatoren. Folgende Beispiele stellen Eingaben dar, die vom Lexer nicht als Fehler erkannt werden, beim Parsevorgang jedoch zum Abbruch führen:

```
x = if( 1 == 1 );

a = b + * - / c;

y = func( y = 42 );

a + b = c - d;
```

Die Fehleranalyse vom Parser ist wie die des Lexers passiv. Das bedeutet, dass in der Grammatikbeschreibung keine Beschreibungen von Fehlerzuständen enthalten sind, sondern sich Fehler aus dem Nichtvorhandensein von passenden Regeln für einen Ausdruck ergeben.

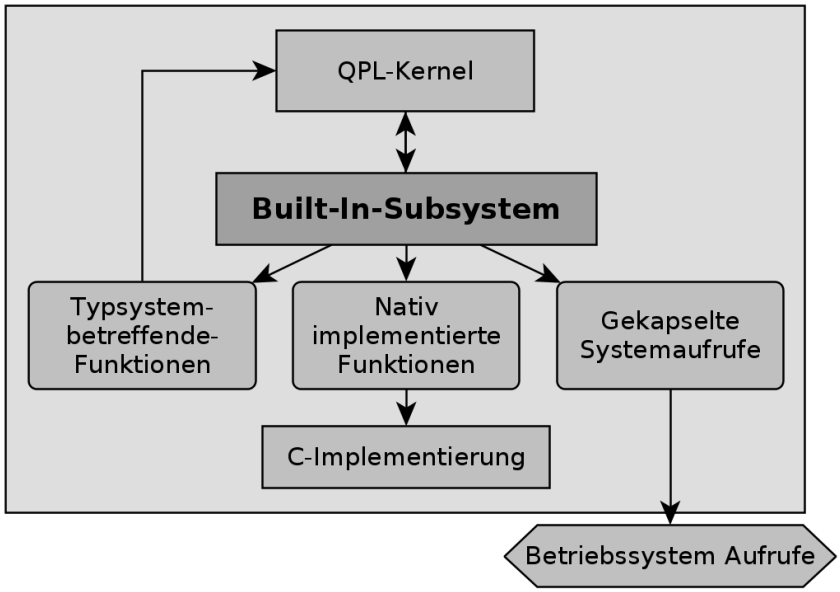
2. Built-In-Funktionen

Da es sich bei QPL um eine interpretierte Programmiersprache handelt, die keine Möglichkeit zum Linken[7] gegen Systembibliotheken hat, muss jegliche Funktionalität die über die Grenzen von Berechnungen hinaus geht, als Built-In-Funktion zur Verfügung gestellt werden. Diese Herangehensweise bringt den Vorteil mit sich, dass die Sprache als relativ sicher angesehen werden kann. Mit ihr ist es nicht ohne weiteres möglich das Betriebssystem anzugreifen, da alle Operationen über den Layer der Built-In-Funktionen geregelt werden.

In QPL gibt es folgende Arten von Built-In-Funktionen:

1. gekapselter Zugriff auf Systemfunktionen, wie zum Beispiel: Terminal Ein-/Ausgabe, Dateioperationen und Systemaufrufe.
2. performanceoptimierte Implementierungen von Standardfunktionen, wie zum Beispiel rechenintensive String- oder mathematische Operationen.
3. typsystembetreffende Funktionen wie Umwandlung von einem in einen anderen Datentyp oder Auslesen des Datentyps einer Variable.

Folgende schematische Darstellung zeigt die Kategorien und Kommunikationskanäle des Built-In-Subsystems:



Built-In-Subsystem

7 Statisches Linken: Verbinden von bereits compiliertem Code zu einer ausführbaren Datei.
Dynamisches Linken: Laden von Funktions- und Speicheradressen aus separaten compilierten Dateien (*.dll, *.so)

Im Folgenden wird ein detaillierterer Einblick in einen Auszug aus den implementierten Built-In-Funktionen gegeben. Es wird auf je eine der oben genannten Funktionstypen eingegangen.

2.1 Gekapselte Systemfunktionen

Als Beispiel für gekapselte Systemfunktionen dient die Funktionsfamilie der Standard Ein-/Ausgabe. Zu den Funktionen zählen: `print()`, `println()`, `read()` und `readln()`. Die Implementierung der beiden Ausgabefunktionen geht auf die C-Funktion `printf()` mit Ziel `stdout` aus der C-Standardbibliothek `stdio.h` zurück.

Die genannten Funktionen sind in der Datei `builtin.c` implementiert. Folgender Quellcodeausschnitt zeigt die Implementierung der `print()` Funktion, `println()` ist analog dazu mit abschließendem Zeilenumbruch implementiert.

```
ast_t* builtin_print(ast_t* ast) {
  ast_t* str;
  str = builtin_to_string(ast);
  printf("%s", str->data.s);
  free_ast(str);
  return NULL;
}
```

Wie man an der Deklaration sieht, nimmt die Funktion `builtin_print()` einen Pointer auf ein Element des abstrakten Syntaxbaumes und gibt aus Kompatibilitätsgründen ebenso einen Pointer auf ein solches zurück[8]. Mittels `builtin_to_string()`, der Built-In-Funktion für Typumwandlungen nach String, wird die Stringrepräsentation des übergebenen Elements ermittelt, das bedeutet, dass die QPL-Funktion `print()` aus Sicht des Programmierers eine implizite Typumwandlung vornimmt. Nach der Ausgabe wird die erzeugte Stringrepräsentation nicht mehr benötigt und via `free_ast()` freigegeben.

Jede Built-In-Funktion muss mit einem Aufruf von `set_ast_to_id()` der Ausführungseinheit unter einem Namen bekannt gemacht werden. Um die gezeigte

8 Der Rückgabewert ist immer `NULL`, was zu einem Laufzeitfehler führt

Funktion `builtin_print()` an den Namen `print()` zu binden ist folgender Aufruf nötig:

```
set_ast_to_id(env, "print",
    create_builtin_1(&builtin_print));
```

Die Variable `env` stellt die aktuelle Ausführungsumgebung dar an die der neue Bezeichner gebunden wird. Der String `"print"` ist die ID bzw. der Name über den auf die Funktion zugegriffen werden kann, `create_builtin_1()` ist eine Funktion die aus dem übergebenen Funktionspointer ein gültiges Element des abstrakten Syntaxbaumes erstellt.

Als nächstes Beispiel für Built-In-Funktionen zur Standard-Ein-/Ausgabe soll die Funktion `readln()` untersucht werden. Um Eingaben über die Kommandozeile lesen zu können, wird von `stdin` zeilenweise, also vom ersten anstehenden Zeichen bis zum Erreichen eines Zeilenumbruchs oder End of File gelesen und das Ergebnis als String zurückgegeben. Die Funktion ist ebenfalls in `builtin.c` implementiert.

```
ast_t* builtin_readln(void) {
#define CHUNK_SIZE (80)
  ast_t* result = NULL;
  size_t size = CHUNK_SIZE;
  size_t idx = 0;
  char* s = (char*)check_malloc(size * sizeof(char));
  int c = fgetc(stdin);
  while(c >= 0 && c != '\n') {
    s[idx] = (char)((c > CHAR_MAX) ?
        (c - (UCHAR_MAX + 1)) : c);
    idx++;
    if(idx >= size) {
      size += CHUNK_SIZE;
      s = (char*)check_realloc(s, size * sizeof(char));
    }
    c = fgetc(stdin);
```

```
  }
  s = (char*)check_realloc(s, (idx + 1) * sizeof(char));
  s[idx] = '\0';
  result = create_string(s);
  result->ref_count = 0;
  return result;
#undef CHUNK_SIZE
}
```

Die Funktion übernimmt keine Parameter und gibt einen Zeiger auf ein Element des abstrakten Syntaxbaumes zurück. Der Rückgabewert entspricht immer einem String-Element, welches wie jede andere String-Variable weiterverarbeitet werden kann. Es wird in 80 Byte Blöcken immer mehr zusätzlicher Speicher reserviert, der zum Einlesen von `stdin` verwendet wird bis ein Zeilenende oder End of File erreicht wurde. Aus den eingelesenen Bytes mit abschließendem `'\0'` wird mit der Funktion `create_string()` ein neues Element des abstrakten Syntaxbaumes erzeugt, das von der Funktion zurückgegeben wird.

Um die Funktion in QPL verfügbar zu machen, muss analog zur `print()` Funktion ein `set_ast_to_id()` Aufruf erfolgen, der in diesem Fall folgendermaßen aussieht:

```
set_ast_to_id(env, "readln",
    create_builtin_0(&builtin_readln));
```

2.2 Nativ implementierte Standardfunktionen

Als nächstes Beispiel für Built-In-Funktionen sollen optimierte Standardfunktionen genannt werden. Das Ziel dieser ist es nicht, wie bei den bisher gezeigten Funktionen Betriebssystemaufrufe zu kapseln, sondern rechenintensive Algorithmen in C zu implementieren und diese in QPL verfügbar zu machen. Durch den natürlichen Overhead einer interpretierten Sprache sind Implementierungen des selben Algorithmus meist deutlich weniger performant als die selben Implementierungen in nativem Code.

Ein Beispiel für eine solche oft benötigte Funktion ist die `replace()` Funktion:

```
ast_t* builtin_replace(ast_t* str, ast_t* old, ast_t* new) {
  ast_t* s = builtin_to_string(str);
  ast_t* o = builtin_to_string(old);
  ast_t* n = builtin_to_string(new);
  char* str = replace_str(s->data.s, o->data.s, n->data.s);
  ast_t* result = create_string(str);
  result->ref_count = 0;
  dec_ref(s);
  dec_ref(o);
  dec_ref(n);
  return result;
}
```

In der gezeigten Ausführung kann die Funktion dazu verwendet werden Teile eines Strings durch andere zu ersetzen. Die Verwendung in QPL und damit auch die Einsatzmöglichkeit der Funktion wird in folgendem QPL-Codebeispiel dargestellt:

```
old = "hello boring world";
new = replace(old, "boring", "awesome");
println("old: " $ old);
println("new: " $ new);
```

Das Programm erzeugt erwartungsgemäß folgende Ausgabe:

```
old: hello boring world
new: hello awesome world
```

Die Implementierung der Funktion `replace_str()` wurde mit vielen Kommentaren im Anhang *B.2 replace_str()* angefügt, sodass die genauen Mechanismen des zugrundeliegenden Algorithmus nachvollzogen werden können.

2.3 Typensystembetreffende Funktionen

Die dritte und letzte Kategorie von Built-In-Funktionen bilden jene das Typsystem betreffende Funktionalitäten. Die Architektur der Programmiersprache lässt es nicht zu auf andere Weise als über diese Built-In-Funktionen explizite Typumwandlungen durchzuführen oder Informationen wie den momentanen Typ einer Variable auszulesen. Manche Built-In-Funktionen nutzen intern die zur Verfügung gestellten Typumwandlungen um sicher zu stellen, dass korrekte Parameter übergeben wurden. Diese implizite Typumwandlung fällt aus Sicht des Programmierers nicht in diese Kategorie, sondern wird als Sprachfeature gesehen.

Für das bessere Verständnis wird als Beispiel die Funktion `typeof()` gezeigt, welche die String-Repräsentation des momentanen Datentyps einer Variable zurückgibt.

```
ast_t* builtin_type_of(ast_t* var) {
  char* typ = strdup(get_ast_type_name(var->type));
  ast_t* result = create_string(typ);
  result->ref_count = 0;
  return result;
}
```

Bekannt gemacht wird die Funktion der Interpreterumgebung wieder mit einem Aufruf der Funktion `set_ast_to_id()`. Die Funktion `typeof()` kann nun beispielsweise dazu verwendet werden abhängig vom aktuellen Datentyp das Verhalten von Funktionen zu ändern oder andere Aktionen im Programm auszulösen. Sie ermöglicht auch eine Art überladene Funktionen zu implementieren und dadurch einfachere Schnittstellen und Bibliotheksfunktionen bereit zu stellen.

Folgendes kurzes Anwendungsbeispiel zeigt die Verwendung der `typeof()` Funktion:

```
s = "hello world";
i = 42;
d = 3.1415;
b = true;
f = @(x) { @ = x * x; };
```

```
print_var_info = @ (var) {
  println("var: '" $ str(var) $ "' - " $ typeof(var));
};

print_var_info(s);
print_var_info(i);
print_var_info(d);
print_var_info(f);
print_var_info(b);
```

Die produzierte Ausgabe sieht erwartungsgemäß folgendermaßen aus:

```
var: 'hello world' - string
var: '42' - integer
var: '3.1415' - double
var: 'function <0x241d1d0>' - function
var: 'true' - bool
```

2.4 Erweiterungsmöglichkeit durch Built-In-Funktionen

Die Built-In-Funktionen bilden ein zentrales und wichtiges Element der Programmiersprache. Wie in den Beispielen gezeigt, decken sie einen vielseitigen Einsatzbereich ab. Sie bilden einerseits die Schnittstelle zu Aufrufen an das Betriebssystem und somit die Möglichkeit mit anderen Anwendungen, Rechnern oder Netzwerken zu kommunizieren[9], andererseits werden mit ihnen Funktionen zur Verfügung gestellt, die implementiert in QPL deutlich weniger performant oder gar nicht umsetzbar wären.

Durch einen einfachen Funktionsaufruf kann eine Built-In-Funktion der Ausführungsumgebung bekannt gemacht werden. Dies ermöglicht die Sprache auf einfache Art zu erweitern und stellt so sicher, dass sie laufend an neue Anforderungen angepasst und verbessert werden kann.

Diese einfache Art der Integration von Built-In-Funktionen ermöglicht es weiter eine Shared

9 Sofern für die gewünschte Schnittstelle (z.B; Sockets, Pipes, …) bzw. die dafür benötigten Built-In-Funktionen implementiert sind und diese vom Betriebssystem unterstützt werden.

Object[10] Schnittstelle zu implementieren, wodurch vom Programmierer eigene Quasi-Built-In-Funktionen in einer nativen Programmiersprache implementiert werden können. Diese Schnittstelle wird es dem Programmierer ermöglichen die Sprache einerseits als Script- oder Pluginsprache für beliebige Anwendungen einzusetzen oder in QPL geschriebene Anwendungen um spezielle Schnittstellen oder Funktionen zu erweitern.

10 Windows: DLL (Dynamic Link Library)
Linux: SO (Shared Object)

3. Interpreter

Der QPL-Interpreter ist das Herzstück der Programmiersprache. Er übernimmt den von Lexer und Parser erzeugten abstrakten Syntaxbaum und arbeitet diesen Schritt für Schritt ab. Der Interpreter besteht aus mehreren Teilbereichen, die alle für unterschiedliche Aufgaben verantwortlich sind und den abstrakten Syntaxbaum in engem Zusammenspiel abarbeiten.

Die wichtigsten Elemente des Interpreters sind:

- **Virtuelle Maschine:**
  zuständig für das Laden des jeweils nächsten Befehls und Aufrufen der zum Befehl gehörenden Funktionen.
- **Environment:**
  implementiert als Hashmap[11] in der alle Variablen (Primitive Datentypen, Funktionen, Listen, …) mit Namen referenziert sind.
- **Operatoren:**
  Sammlung von Funktionen, in denen die Funktionalitäten der verfügbaren Operatoren für die verschiedenen Datentypen implementiert sind.
- **Fehlerbehandlung:**
  Erzeugung von Fehlermeldungen zum jeweiligen Fehlerfall mit möglichst informativem Inhalt über die Fehlerursache.
- **Datentypen:**
  Datencontainer für Variablen und Konstanten sowie Informationen über Anwendung von Operatoren.

3.1 Implementierung der virtuellen Maschinen

[8] Die virtuelle Maschine oder auch Ausführungseinheit bietet eine zentrale Funktion an, die vom Hauptprogramm nach dem Parsevorgang aufgerufen wird. Ab hier übernimmt die virtuelle Maschine den kompletten Programmablauf, die den abstrakten Syntaxbaum abarbeitet.

11 Es kommt eine selbst implementierte Hashmap zum Einsatz. Details und nähere Informationen dazu sind im Anhang *B.1 Hashmap* angefügt. Als Hashing Algorithmus wurde ein nicht kryptografischer FNV Hasher implementiert [7].

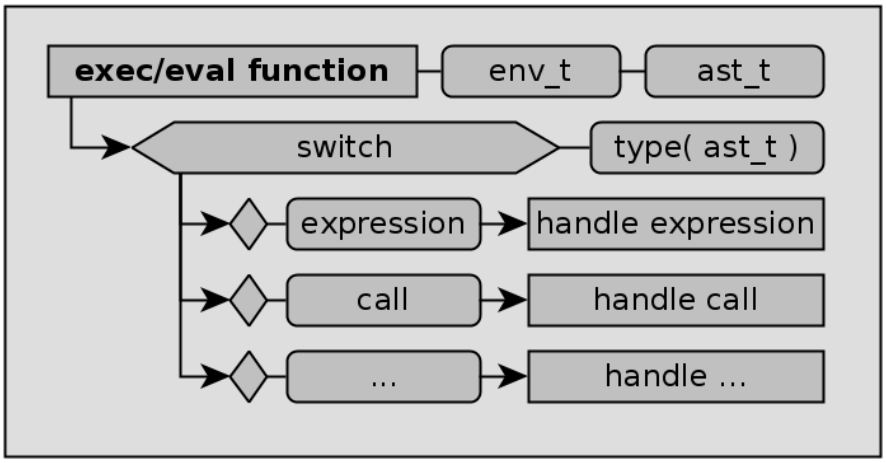
Vorbereitend wird ein initiales Environment erstellt, welches neben dem abstrakten Syntaxbaum der Funktion `vm_exec()` übergeben wird.

Innerhalb der VM gibt es verschiedene Kategorien von Funktionen die entweder eine Auswertung von Elementen des abstrakten Syntaxbaumes durchführen (`eval_*()` Funktionen), oder ein oder mehrere Befehle ausführen (`exec_*()` Funktionen).

Die Familie der Evaluierungsfunktionen gibt jeweils eine `ast_t` Struktur zurück, die dann weiterverwendet werden kann. Sie übernehmen ihrerseits ein Environment und ein Element des abstrakten Syntaxbaumes, welches ausgewertet wird. Benötigt werden die Evaluierungsfunktionen für alle Ausdrücke, die ein Ergebnis haben, wie Berechnungen oder Funktionsaufrufe.

Die Familie der Ausführungsfunktionen hat keinen Rückgabewert. Sie behandeln jene Elemente des abstrakten Syntaxbaumes, die kein Ergebnis haben, wie zum Beispiel Gruppen von Statements, Kontrollkonstrukte wie `if` oder `while` Anweisungen oder abgeschlossene Ausdrücke wie Zuweisungen.

Beide Funktionsfamilien arbeiten auf ähnliche im Folgenden vereinfacht dargestellte Weise:



exec/eval Funktionsschema

Es wird zuerst ausgewertet um welchen genauen Typ es sich handelt (eine Vorauswahl wurde zu diesem Zeitpunkt bereits getroffen[12]) und im Anschluss wird die korrespondierende Detailimplementierung aufgerufen. Die Detailimplementierung kann das weitere Vereinfachen oder Zusammenfassen eines Ausdrucks, also wieder eine Ausführungs- oder Evaluierungsfunktion oder ein anderes Element der virtuellen Maschine zum Beispiel ein Operator sein.

12 Auf Basis der Vorentscheidung wurde ausgewählt, welche Funktion aufgerufen werden muss.

Im Folgenden wird als Beispiel auf die Implementierung des `if` Statements eingegangen und der zugehörige Quellcode erläutert.

Zur Erinnerung: bedingte Anweisungen bestehen in QPL aus einem `if`-Statement gefolgt von keinem, einem oder mehreren `elif`-Statements, gefolgt von keinem oder einem `else`-Statement. Die zugehörige Grammatikbeschreibung in BNF sieht folgendermaßen aus:

```
<conditional> ::= <if_stmnt> [<elif_stmnt>]* [<else_stmnt>]
<if_stmnt> ::= "if" "(" <expression> ")" <block>
<elif_stmnt> ::= "elif" "(" <expression> ")" <block>
<else_stmnt> ::= "else" <block>
<block> ::= "{" <stmnts> "}"
```

Als ersten Teilbereich des Beispiels betrachten wir die Funktion `exec_if()`. Die Funktion überprüft die Bedingung des übergebenen `ast_t` Elements und führt abhängig von dessen Wahrheitswert die zugehörenden Statements aus. War das Ergebnis positiv und wurden die Statements ausgeführt, gibt `exec_if()` einen Wert ungleich Null zurück um das Ergebnis der Bedingung nach außen zu signalisieren. `exec_if()` wird für `if` und `elif` Ausdrücke verwendet.

```
int exec_if(env_t* env, ast_t* ast) {
  int result = 0; /* not null when condition was true,
                     null when condition was false */
  ast_t* cond = eval_expression(env,
      ast->data.if_statement.condition);
  if(cond->type != at_bool) {
    error_expected(NULL, "bool",
        get_ast_type_name(cond->type));
  } else {
    if(cond->data.b) {
      exec_statements(env, ast->data.
          if_statement.statements);
      result = 1;
    }
```

```
    dec_ref(cond);
  }
  return result;
}
```

Die Funktion `exec_if()` entspricht nicht ganz dem Muster der `exec/eval` Funktionen. Sie ist ein Sonderfall und wird auch nur an einer Stelle in der virtuellen Maschine verwendet, nämlich von der Funktion `exec_conditional()`, welche ihrerseits dem Muster entspricht. `exec_conditional()` kann einen kompletten bedingten Codeteil behandeln. Es wird das `if`-Statement, die optionalen `elif`-Statements und das optionale `else`-Statement behandelt und die von den Bedingungen abhängigen Statements ausgeführt.

Dank der zuvor gezeigten Funktion `exec_if()` gestaltet sich die Implementierung von `exec_conditional()` nicht sehr komplex. Der Programmablauf folgt der Funktionsweise der `if-elif-else` Beziehung. Zuerst wird der `if`-Teil ausgewertet. Wurde er ausgeführt kehrt die Funktion sofort zurück, darauf folgen die `elif`-Zweige, sobald einer ausgeführt wurde kehrt die Funktion zurück. Wenn die Funktion bis hierher ausgeführt wird und ein `else`-Zweig vorhanden ist, wird dieser ausgeführt.

Im Folgenden ist die Funktion `exec_conditional()` angeführt:

```
void exec_conditional(env_t* env, ast_t* ast) {
  /* the IF */
  if(exec_if(env, ast->data.conditional.if_statement)) {
    return;
  } else {
    if(ast->data.conditional.elif_statements != NULL) {
      size_t i;
      /* the ELIF(s) */
      for(i = 0;
          i < ast->data.conditional.elif_statements->data.
              elif_statements.count;
          i++) {
```

```
            if(exec_if(
                env, ast->data.conditional.elif_statements->
                    data.elif_statements.elif_statements[i])) {
              return;
            }
          }
        }
        if(ast->data.conditional.else_statement != NULL) {
          /* the ELSE */
          exec_statements(env, ast->data.conditional.
              else_statement);
        }
      }
    }
```

3.2 Implementierung des Environments

Das Environment ist ein wichtiger Programmteil, der alle Variablen und Container die zur Laufzeit eines QPL-Programms erzeugt werden, beinhaltet. Jeder Funktionsaufruf in QPL erzeugt eine neue Umgebung in der Variablen abgelegt werden können. Am Ende jeder Funktion wird die hierfür erzeugte Umgebung wieder freigegeben.

Ein QPL-Script selbst erhält auch ein Environment, welches bei Programmstart erzeugt und bei Programmende freigegeben wird. Diese übergeordnete Umgebung wird bei Programmstart mit den Built-In-Funktionen vorinitialisiert. Wie im Kapitel über Built-In-Funktionen beschrieben geschieht dies über `set_ast_to_id()`-Aufrufe was die Funktionen an Namen im Environment bindet.

Wird in einer QPL-Funktion auf eine Variable lesend zugegriffen, wird diese zuerst in der aktuellen Umgebung gesucht. Sollte sie nicht gefunden werden, wird in der übergeordneten Umgebung danach gesucht. Wird die verwendete Variable in keiner Umgebung gefunden, erzeugt dies einen Laufzeitfehler.

Im Gegensatz zum lesenden Zugriff auf Variablen wird beim schreibenden Zugriff immer nur auf die aktuelle Umgebung geachtet. Wird einer Variable ein Wert zugewiesen wird diese in der aktuellen Umgebung überschrieben, falls vorhanden, oder neu erzeugt, falls noch nicht vorhanden.

Um diesen Mechanismus zu verdeutlichen, werden die Funktionen `get_ast_by_id()` und `set_ast_to_id()` näher untersucht.

Die Funktion `get_ast_by_id()` ruft als erstes `get_value()`, eine Hashmap-Funktion, auf und prüft dann, ob diese einen gültigen Wert zurückgegeben hat. Falls kein gültiger Wert gefunden wurde und die Umgebung eine übergeordnete Umgebung hat, wird `get_ast_by_id()` mit der übergeordneten Umgebung rekursiv aufgerufen. Der Wert von `parent` der obersten Umgebung ist `null`, daher bricht die Rekursion spätestens an dieser Stelle ab und die Funktion gibt `null` zurück.

```
ast_t* get_ast_by_id(env_t* env, const char* id) {
  ast_t* ast = (ast_t*)get_value(env->map, id);
  if(ast == NULL && env->parent != NULL) {
    ast = get_ast_by_id(env->parent, id);
  }
  return ast;
}
```

Die Funktion `set_ast_to_id()` weist der übergebenen Umgebung die Variable zu. Wurde bei der Zuweisung eine möglicherweise Vorhandene ersetzt, wird von dieser die Referenzzählung dekrementiert[13]. Im Folgenden ist die einfache Funktion dargestellt.

```
void set_ast_to_id(env_t* env, const char* id, ast_t* ast) {
  ast_t* old = (ast_t*)add_value(env->map, id, ast);
  inc_ref(ast);
  if(old != NULL) {
    dec_ref(old);
  }
}
```

13 Beschreibung von Inkrement und Dekrement des Referenzzählers im Kapitel *5. Speicherverwaltung*

3.3 Implementierung der Operatoren

In QPL werden Operatoren für verschiedene Anwendungen zur Verfügung gestellt. Mittels Operatoren können mathematische und logische Ausdrücke gebildet oder Strings verkettet werden.

Folgende Tabelle gibt einen Überblick über alle verfügbaren Operatoren und deren Verwendung:

Operator	Bedeutung	Verwendung
`+`	Addition	`i = 1 + 2;`
`-`	Subtraktion	`i = 2 – 1;`
`*`	Multiplikation	`d = 3.0 * 2.0;`
`/`	Division	`d = 10.0 / 3;`
`%`	Modulo	`i = 10 % 3;`
`$`	Verkettung	`s = "Hallo " $ "Welt.";`
`&`	Logisches Und	`b = true & true;`
`|`	Logisches Oder	`b = true | false;`
`<`	Kleiner Vergleich	`b = 1 < 2;`
`>`	Größer Vergleich	`b = 2 > 1;`
`<=`	Kleiner-Gleich Vergleich	`b = 1 <= 2;`
`>=`	Größer-Gleich Vergleich	`b = 2 >= 1;`
`==`	Gleichheits Vergleich	`b = 1 == 1;`
`!=`	Ungleichheits Vergleich	`b = 1 != 2;`

QPL-Operatoren [1]

Die zugrundeliegenden Mechanismen der Operatoren aus QPL-Sicht wurden bereits im Praxisprojekt diskutiert und können dort nachgelesen werden [1]. Folgend wird detaillierter auf die Implementierung eingegangen.

Jeder der verfügbaren Operatoren hat zwei Argumente und ein Ergebnis. Die Implementierung ähnelt daher einer Built-In-Funktion mit zwei Parametern. Abhängig von den Typen der Argumente müssen Operatoren jedoch entscheiden, welche Implementierung verwendet werden soll bzw. in welchen Typ das Ergebnis umgewandelt werden muss. Werden beispielsweise zwei ganzzahlige Variablen mit einem mathematischen Operator kombiniert, muss das Ergebnis wiederum ganzzahlig sein.

Alle mathematischen Berechnungen werden im C-Datentyp `long double` durchgeführt.

Dies ermöglicht, dass Berechnungen im größtmöglichen Wertebereich und gleichzeitig mit größtmöglicher Genauigkeit durchgeführt werden. Weiters weisen dadurch alle mathematischen Operatoren dieselbe Funktionssignatur auf und es kann bequem mit Funktionszeigern gearbeitet werden.

Im Folgenden wird anhand eines Beispiels der Additionsoperator + näher erläutert. Für Additionen steht wie für alle anderen Operatoren eine Funktion der Form

```
ast_t* eval_add(env_t* env, ast_t* ast1, ast_t* ast2);
```

zur Verfügung, welche `eval_math()` aufruft, worin die Datentypen der Argumente und des Ergebnisses ermittelt und anschließend die übergebene Funktion aufgerufen wird. Die Funktion `eval_math()` hat folgende Signatur, die es erlaubt alle mathematischen Funktionen mit ihr abzubilden:

```
ast_t* eval_math(
    env_t* env, operator_t op, ast_t* ast1, ast_t* ast2,
    long double (math_func)(long double, long double)
);
```

Die eigentlichen Berechnungsfunktionen, die als `math_func` übergeben werden sind trivial. Hier ist die Funktion, die die eigentliche Addition durchführt:

```
long double __add_func(long double d1, long double d2) {
  return d1 + d2;
}
```

Die Vergleichsoperatoren (<, <=, >, >=, ==, !=) nutzen denselben Mechanismus wie die mathematischen. Die hierfür implementierte Funktion lautet:

```
ast_t* eval_gtlt(
    env_t* env, operator_t op, ast_t* ast1, ast_t* ast2,
    char (comp_func)(long double,long double)
);
```

3.4 Implementierung der Fehlerbehandlung

Abseits jener Fehler die während der lexikalischen Analyse und des Parsevorgangs erkannt werden können[14], gibt es eine Vielzahl von Fehlerzuständen, die erst während der Programmausführung erkannt werden können. Generell führen alle Fehlerzustände zum Abbruch des Programms und zur Beendigung des Interpreters.

Wird der Interpreter durch einen Fehlerzustand beendet, gibt die QPL-Anwendung einen Fehlercode an das Betriebssystem zurück um zu signalisieren, dass es sich um kein geordnetes Programmende gehandelt hat.

Bei Fehlerzuständen muss zwischen zwei Arten von Fehlern unterschieden werden.

1. Das QPL-Script weist Fehler auf (z.B: Zugriff auf nicht zugewiesene Variable, falsche Verwendung von Operatoren, Typkonflikte, …)
2. Innerhalb des Interpreters ist ein Fehler aufgetreten (Inkonsistenzen im abstrakten Syntaxbaum, Überläufe, Fehler in der Referenzzählung, …)

Die erste Art von Fehlern ist die unproblematischere. In vielen Fällen werden zu diesen Fehlern Fehlermeldungen erzeugt, welche dem Programmierer genaue Hinweise auf die Ursache liefern. Wird zum Beispiel der Verkettungsoperator auf einen Integer und einen String angewandt wie im Folgenden gezeigt, führt dies zu einer eindeutigen Fehlermeldung auf Grund welcher der Programmierer seinen Quellcode anpassen kann:

```
x = 1 $ "a";
```

```
error: cannot apply "$" to "integer" and "string"
```

Im Gegensatz dazu, führen interne Fehler des Interpreters nicht zu solch sprechenden Fehlermeldungen, sondern zu einem Stacktrace des QPL-Interpreters. Ohne den Quellcode des QPL-Interpreters können diese Fehler nicht weiter analysiert oder gelöst werden. Im Folgenden ist ein Stacktrace abgebildet, der auf Grund eines Fehlers im Referenzzählungssystem aufgetreten ist:

```
Error: signal 11:
./qpl(sigsegv_handler+0x34)[0x40695c]
/lib/x86_64/libc.so.6(+0x352f0)[0x7fca8ee8]
./qpl(inc_ref+0xc)[0x4093ad]
```

14 Siehe Kapitel *1.2 Lexer* und *1.3 Parser*

```
./qpl(eval_expression+0x6a)[0x408291]
./qpl(eval_call+0x268)[0x40807f]
./qpl(exec_statements+0x9c)[0x408929]
./qpl(vm_exec+0x2e)[0x408b32]
./qpl(main+0xe9)[0x406a96]
/lib/x86_64/libc.so.6(__libc_start_main+0xf0)[0x7fca8ee7]
./qpl(_start+0x29)[0x402eb9]
```

Man sieht, dass alle bis zum Auftreten des Fehlers aufgerufenen Funktionen von unten nach oben ausgegeben werden. Mit diesen Informationen kann nun der Interpreter weiter debugged werden.

Fehler dieser Art sollten selten auftreten, werden sich aber nicht zuverlässig vermeiden lassen. Wie in jeder Software können sich auch in einem Interpreter Programmfehler befinden. Eine große Schwierigkeit dabei ist es, dass vom Anwender der Programmiersprache nicht immer erkannt werden kann, ob es sich bei einem auftretenden Problem um eines aus seinem Quellcode oder um einen internen Fehler des Interpreters handelt.

Fehlermeldungen werden standardmäßig auf `stderr`[15] ausgegeben, wozu folgende Funktion verwendet wird:

```
void error(position_t* pos, const char* emsg) {
  if(pos == NULL) {
    fprintf(stderr, "error: %s\n", emsg);
  } else {
    if(pos->file == NULL) {
      fprintf(stderr,"error (line: %d): %s\n",
          pos->line, emsg);
    } else {
      fprintf(stderr, "error (%s:%d): %s\n",
          pos->file, pos->line, emsg);
    }
```

15 Standard Error Stream: Bevorzugtes Ausgabeziel für Fehlermeldungen [9]

```
  }
  exit(1);
}
```

Die gezeigte Funktion wird allerdings nie direkt aufgerufen. Es gibt für jeden Fehlertyp eine Wrapper-Funktion, die auf den Fehler angepasste Parameter übernimmt und den eigentlichen Fehlertext zusammenbaut. Als Beispiel für solch eine Wrapper-Funktion wird eine Funktion gezeigt, die dem Programmierer den Hinweis geben soll, dass eine Funktion mit einer falschen Anzahl an Parametern aufgerufen wurde. Man sieht, dass `error_paramcount()` genau jene Informationen erfordert, die zur Generierung der Fehlermeldung benötigt werden.

```
void error_paramcount(position_t* pos,
    const char* func, int paramcount, int callargcount) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,
      "function \"%s\" expects %d arguments, \
       called with %d",
      func, paramcount, callargcount);
  error(pos, buf);
}
```

Durch die Wrapper-Funktionen wird sichergestellt, dass Fehlermeldungen immer im selben Stil und mit relevantem Inhalt generiert werden.

Für die Erzeugung des Stacktrace kommen die C-Bibliotheksfunktionen `backtrace()` und `backtrace_symbols_fd()` in Kombination mit einem Signal-Handler zum Einsatz. Hierzu wird zuerst mittels `signal()` ein Handler für `SIGSEGV` (Zugriffsverletzungen) installiert, welcher im Fehlerfall den Stacktrace erzeugt und ausgibt. Im Folgenden ist der hierfür relevante Codeausschnitt abgebildet [10]:

```
void sigsegv_handler(int sig) {
  void* array[25];
  size_t size;
  size = backtrace(array, 25);
```

```
    fprintf(stderr, "Error: signal %d:\n", sig);
    backtrace_symbols_fd(array, size, stderr);
    exit(1);
}

...
    signal(SIGSEGV, sigsegv_handler);
...
```

Mit den hier vorgestellten Methoden werden alle Laufzeitfehler behandelt, sodass der Anwender im Fehlerfall mit möglichst aufschlussreichen Fehlermeldungen bei der weiteren Fehlersuche unterstützt wird.

3.5 Implementierung der Datentypen

In QPL stehen vier Basisdatentypen zur Verfügung, die vom Anwender genutzt und kombiniert werden können. Die verfügbaren Datentypen sind aus vielen anderen Programmiersprachen bekannt, jedoch so vereinfacht, dass sich der Programmierer nicht um hardwarenahe Details kümmern muss.

Daten werden immer in Elementen des abstrakten Syntaxbaumes gehalten. Sie sind in einer `union`[16] Struktur umgesetzt, sodass für alle Datentypen die gleiche Struktur eingesetzt werden kann.

Innerhalb der Struktur kommen für die verschiedenen Datentypen verschiedene Grunddatentypen zum Einsatz. Folgende Liste stellt die QPL-Datentypen und ihre Repräsentationen in C dar:

16 `union` Strukturen sind Strukturen, die für mehrere Felder den selben Speicherbereich nutzen [11]

QPL-Datentyp	C-Datentyp
`integer`	`intmax_t`
`double`	`long double`
`bool`	`char`
`string`	`char*`
`list`	`ast_t**`

Datentypen

Insbesondere bei Strings und Listen muss in C genau auf das Reservieren und Freigeben von Speicherplatz geachtet werden. Der QPL-Interpreter übernimmt hier die gesamte Speicherverwaltung, vom Anfordern bzw. Vergrößern und Verkleinern des benötigten Speicherplatzes bis zum Freigeben von nicht mehr benötigten Variablen[17].

Variablen werden in QPL nicht deklariert, sie erhalten Ihren Typ aus der jeweils letzten Zuweisung. Das bedeutet, dass ein Bezeichner auch innerhalb eines Codeblocks nicht zwingend immer den selben Datentyp haben muss. Das Typsystem bietet dem Programmierer die Möglichkeit zur Laufzeit den Typ einer Variable zu ermitteln und Konvertierungen zwischen Datentypen durchzuführen.

Folgende Funktionen stehen zum Konvertieren von Variablen zur Verfügung:

`int()` Gibt die Integer-Repräsentation des Inhalts der übergebenen Variable zurück. Kann keine gültige Umwandlung durchgeführt werden, führt dies zu einem Laufzeitfehler.
Beispiele:

```
i = int(3.1415);     /* i ist nun 3 */
i = int("42");       /* i ist nun 42 */
i = int(true);       /* i ist nun 1 */
i = int("test");     /* laufzeitfehler der form:
    "error: cannot convert 'string' to 'integer'" */
```

`dbl()` Gibt die Double-Repräsentation des Inhalts der übergebenen Variable zurück. Auch hier führen ungültige Umwandlungsversuche zu einem Laufzeitfehler.
Beispiele:

```
d = dbl(23);         /* d ist nun 23.0 */
d = dbl("2.718");    /* d ist nun 2.718 */
d = dbl(false);      /* d ist nun 0.0 */
d = dbl("test");     /* laufzeitfehler der form:
    "error: cannot convert 'string' to 'double'" */
```

17 Anhand von Referenzzählung; siehe *5. Speicherverwaltung*.

`str()` Gibt die String-Repräsentation des Inhalts der übergebenen Variable zurück. `str()` gibt für jeden beliebigen Variableninhalt einen String zurück und führt nie zu einem Laufzeitfehler. Konvertierungen nach String sind immer sicher.
Beispiele:

```
s = str(13);        /* s ist nun "13" */
s = str(1.4142);    /* s ist nun "1.4142" */
s = str(true);      /* s ist nun "true" */
```

`bool()` Gibt die Boolean-Repräsentation des Inhalts der übergebenen Variable zurück. `bool()` gibt jeden beliebigen Variableninhalt einen boolschen Wert zurück und führt nie zu einem Laufzeitfehler. Konvertierungen nach Boolean sind immer sicher.
Beispiele:

```
b = bool(27);     /* b ist nun true */
b = bool(0.0);    /* b ist nun false */
b = bool("");     /* b ist nun false */
```

4. Optimierung

In diesem Kapitel werden Methoden zur Optimierung verschiedener Teilbereiche aus dem Gebiet Compiler- und Interpreterentwicklung betrachtet und vorgestellt. Nicht alle der hier vorgestellten Methoden, insbesondere das Unterkapitel *4.2 Theorien & Techniken zu Parallelität* wurden praktisch umgesetzt, weshalb dies als Ausblick auf zukünftige Optimierungspotentiale zu sehen ist.

4.1 Theorien & Techniken zu Performanceoptimierung

Unter Performanceoptimierung können im Compiler- und Interpreterbau sehr viele Techniken zusammengefasst werden. Hier soll vor allem auf Laufzeitoptimierungen des Interpreters eingegangen werden.

Bereits beim Erzeugen des abstrakten Syntaxbaumes kann dieser nach verschiedenen Mustern durchsucht, und Teile davon zusammengefasst oder entfernt werden. Werden zum Beispiel zwei oder mehr Literale kombiniert, können diese unter Umständen bereits vorab berechnet und durch ein einziges Element des abstrakten Syntaxbaumes ersetzt werden. Weiters können bedingte Anweisungen oft von vornherein ein- oder ausgeschlossen werden. Folgende Beispiele zeigen Bedingungen, welche vor der eigentlichen Ausführung ausgewertet werden können [2].

Diese Bedingungen führen immer zur Ausführung des darauf folgenden Codeblocks. Sie müssten zur Laufzeit nicht ausgewertet werden.

```
if (true) { /* ... */ }
if (b == true | b == false) { /* ... */ }
if (i >= 3 | i <= 3) { /* ... */ }
```

Diese Bedingungen führen immer dazu, dass der darauf folgende Codeblock nicht ausgeführt wird, er kann aus dem abstrakten Syntaxbaum entfernt werden.

```
if (false) { /* ... */ }
if (b==true & b == false) { /* ... */ }
if (i < 3 & i > 3) { /* ... */ }
```

Kommt diese Art von Optimierung zum Einsatz muss sich der Interpreter sicher sein, dass

durch das Entfernen der Auswertungen keine Seiteneffekte entfernt werden. Da QPL im Gegensatz zu manchen C-ähnlichen Sprachen keine Zuweisungen in Bedingungen erlaubt, sind an dieser Stelle Funktionsaufrufe die einzige Möglichkeit für Seiteneffekte. Folgende Bedingung, die auf den ersten Blick den zuvor gezeigten entspricht, darf aufgrund von möglichen Seiteneffekten nicht wegoptimiert werden:

```
if (func(x) | true) { /* ... */ }
```

Weiters kann mittels Compileroptionen eine geeignete Optimierungsstufe für den Interpreter gewählt werden. Der GCC Compiler bietet hierzu folgende Optionen an[12]:

`-O0` Keine Optimierung (Standardauswahl)

`-O1` Viele „ungefährliche“ Optimierungen, die relativ wenig Einfluss auf Debuggingmöglichkeiten haben.
Beispiele: Branch prediction, Vorberechnung der CPU Registerbenutzung, Inkrement und Dekrement Optimierung, Vorberechnung von konstanten Ausdrücken.

`-O2` Viele Optimierungen (inklusive aller aus `-O1`) die auch die Debugging möglichkeiten einschränken.
Beispiele: Auflösung von `assert` Statements, Ausrollen von Schleifen, Inlining von kurzen Funktionen, Anpassung von Funktionsadressen, Ändern der Auswertereihenfolge von boolschen und arithmetischen Ausdrücken.

`-O3` Beinahe alle verfügbaren Optimierungen (inklusive `-O1` und `-O2`).
Beispiele: Funktionsinlining, Redundanz von Codesegmenten, aufwändige Quellcodeanalyse und Anpassung, Schleifenvektorisierung.

`-Os` Optimierung für kleine Dateigröße. Nutzt alle `-O2` Optimierungen die die Codegröße nicht erhöhen (zum Beispiel Inlining) und fügt noch weitere spezielle Anpassungen hinzu.

`-Ofast` Alle verfügbaren Optimierungen (inklusive `-O1`, `-O2` und `-O3`) und zusätzlich noch weitere, die allerdings nicht zwingend dem Standard entsprechen und daher äußerst vorsichtig einzusetzen sind.
Beispiele: schnelle mathematische Funktionen, Anpassung von Arrays.

`-Og` Optimierungsoptionen aus den verschiedenen Stufen, jedoch nur solche, die kaum Einfluss auf die Debuggingmöglichkeiten haben und den Compiliervorgang nicht wesentlich aufwändiger machen.

Für den produktiven Einsatz wird QPL daher mit `-O3` kompiliert, was abhängig von der Zielarchitektur und den ausgeführten Algorithmen zu einem Performancezuwachs von bis

zu 300% führt [13].

4.1.1 Optimierung des Switch-Statements

Einige Optimierungsstrategien, welche `switch`-Statements optimieren wurden detaillierter untersucht, da diese erheblichen Einfluss auf das Laufzeitverhalten des Interpreters haben. Die Optimierung kann durch den Kommandozeilenparameter `-ftree-switch-conversion` aktiviert werden.

[14] Ein `switch`-Statement kann man sich wie eine Aneinanderreihung von `if-else-if`-Blöcken vorstellen. Ohne Optimierung wird dies vom Compiler auch so umgesetzt, was zur Folge hat, dass sehr viele Vergleichsoperationen ausgeführt werden müssen. Je nach Größe und Eigenschaften des `switch`-Statements können jedoch Optimierungen durchgeführt werden, die eine deutliche Performancesteigerung mit sich bringen.

Ein einfacher Ansatz ersetzt die Vergleiche durch eine Sprungtabelle. Durch diesen Mechanismus muss anstatt der vielen Vergleiche nur noch die Berechnung eines Tabellenindexes durchgeführt werden. Folgendes Pseudocode Beispiel zeigt eine Gegenüberstellung des unoptimierten und optimierten Codes.

```
if var = case_0 then func_0()
elif var = case_1 then func_1()
...
```

An diesem Code sieht man deutlich, dass unter Umständen sehr viele Vergleiche durchgeführt werden müssen, der folgende Codeausschnitt zeigt eine effizientere Methode:

```
// initialisierung zur compilezeit:
func[0] := func_0
func[1] := func_1
...
// switch statement zur laufzeit:
func[var]()
```

Somit konnten die zuvor gesehenen Vergleiche durch eine einfache Dereferenzierung ersetzt werden. Die Zuordnung von auszuführendem Code zum jeweiligen Offset kann bereits zur Compilezeit durchgeführt werden.

[15] In verschiedenen Konstellationen ist es jedoch nicht möglich oder sinnvoll diese Methode einzusetzen. Es gibt noch weitere Herangehensweisen, die performanter als die `if-else-if`-Kette sind und gleichzeitig flexibler eingesetzt werden können als die gezeigte Sprungtabelle. Unter Verwendung einer Hashtabelle können zum Beispiel auch weit auseinander liegende `case`-Ziele auf ähnliche Weise realisiert werden, zusätzlich zur Dereferenzierung muss jedoch ein Hashwert gebildet und einige wenige Vergleichsoperationen durchgeführt werden. Im Anhang *B.1 Hashmap* sind die grundlegenden Mechanismen einer Hashtabelle gezeigt wie sie hier eingesetzt werden kann.

4.2 Theorien & Techniken zu Parallelität

In modernen Anwendungen wird das parallele Verarbeiten immer wichtiger. Nahezu alle gängigen Prozessoren bieten mehrere Rechenkerne, weshalb Anwendungen die rechenintensive Aufgaben auf mehrere Threads aufteilen, besonders von ihnen profitieren können.

Bei der Verteilung der Aufgaben können vielseitige Probleme auftreten, die Anwendungen trotz eingesetzter Parallelität ausbremsen oder sogar zum Stillstand[18] bringen können. Neben diesen Problemen sind Dateninkonsistenzen durch ungesicherte Schreib- und Lesezugriffe auf Variablen ein Problem, welches häufig in parallelen Anwendungen auftritt [16].

Parallelität wird vom Interpreter durch POSIX-kompatible Threads umgesetzt. Diese stehen abstrahiert durch Built-In-Funktionen zur Verfügung [17]. Mit der Abstraktionsebene werden bereits einige Elemente eingeführt, die die Arbeit mit Threads stark vereinfachen und das Fehlerpotential minimieren.

Durch das konsequente Schützen jedes Variablenzugriffs wird dem Programmierer ermöglicht, jeden Lese- und jeden Schreibvorgang als atomar[19] zu betrachten. Aufgrund des dadurch immer geschützten Zugriffs auf Variablen, wird es für den Programmierer wesentlich einfacher und übersichtlicher parallel laufenden Code zu schreiben.

Weiters kann zum Beispiel die `for-each` Schleife in einer weiteren Variante zur Verfügung gestellt werden, die die Bearbeitung in Threads auslagert. Hierbei ist zu beachten, dass es nicht sinnvoll ist, für jede Bearbeitung einen eigenen Thread zu erzeugen der nur für einen

18 Deadlock, Verklemmung durch Synchronisierungs- oder Sperrmechanismen.
19 Atomar lat. Unteilbar. Lese- Schreibzugriffe werden (auch bei mehreren Zyklen lange Operationen) nicht unterbrochen.

Durchlauf verwendet wird. Um optimale Ergebnisse zu erzielen ist es besser einen Threadpool zu verwenden der `"Anzahl der CPU Kerne" + 1` Threads erzeugt und die Aufgaben auf diese aufteilt [18].

Eine mögliche Schreibweise für die parallele `for-each` Schleife wäre folgende:

```
elements = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
pfor(element, elements) {
  some_complex_calculation(element);
}
```

Um den Unterschied zur nicht parallelen for-each Schleife darzustellen muss das neue Schlüsselwort `pfor` eingeführt werden.

Dieser Code würde auf einem Rechner mit vier CPU Kernen dazu führen, dass fünf Threads erzeugt werden. Den Threads werden nun die Iterationselemente und die Anweisungen der `for-each` Schleife zugewiesen. Sobald ein Thread seine Aufgaben erledigt hat wird ihm ein neues Iterationselement zugewiesen, so lange bis alle Elemente bearbeitet und alle Threads ihre Ausführung beendet haben.

Abhängig von der CPU-Kernanzahl und des gewählten Algorithmus können durch dieses Verfahren erhebliche Performancegewinne erzielt werden.

Um den Overhead von Threaderzeugung und Threadzerstörung noch weiter zu verringern, könnte während des Parsvorgangs bereits darauf geachtet werden, ob im Programm parallele `for-each` Schleifen verwendet werden oder nicht. Kommen welche zum Einsatz macht es Sinn, den Threadpool bereits beim Start des Interpreters zu erzeugen und für alle `for-each` Schleifen den gleichen Threadpool zu verwenden [19].

Um dem Programmierer bei der Fehlersuche in parallelen Anwendungen zu helfen, wird es die Möglichkeit geben, die Verwendung von Threads über einen Kommandozeilenparameter zu deaktivieren. Wird eine Anwendung mit `--no-threads` aufgerufen führt der Interpreter alle `pfor` Anweisungen so aus, als wären es `for` Anweisungen.

5. Speicherverwaltung

[20] Ein zentraler Punkt bei der Entwicklung einer Programmiersprache ist die Speicherverwaltung. Viele Programmiersprachen haben keine automatische Speicherverwaltung, sodass der benötigte Speicherplatz im RAM vom Programmierer angefordert und der nicht mehr benötigte wieder freigegeben werden muss. Beispiele hierfür sind die C-Funktionsfamilie `malloc()` und `free()`[20] sowie die C++ Schlüsselwörter `new` und `delete` [21].

Einige Programmiersprachen verzichten auf die Speicherverwaltung durch den Programmierer und geben selbst den nicht mehr benötigten Speicher frei. Hierfür gibt es zwei verschiedene Ansätze die weit verbreitet sind [22].

Garbage Collection
: Bezeichnet eine Gruppe von Verfahren die sich um das automatisierte Freigeben von nicht mehr benötigtem Speicherplatz kümmern.
Es gibt verschiedene Mechanismen wie Programme Speicherstellen zum Freigeben markieren, welche dann vom Garbage Collector freigegeben werden können.
Der Garbage Collector selbst läuft meistens parallel zur eigentlichen Anwendung und prüft zyklisch nach ob Speicherplatz freigegeben werden kann.

Referenzzählung
: Ist ein Verfahren bei dem sich zu jeder Variable gemerkt wird von wie vielen Stellen aus darauf zugegriffen werden kann, also wie viele Referenzen auf eine Variable vorhanden sind.
Sobald die Anzahl der Referenzen auf Null fällt wird der zugehörige Speicher freigegeben.

Bei QPL kommt wie in vielen anderen interpretierten Programmiersprachen Referenzzählung zum Einsatz. Jedes Element im abstrakten Syntaxbaum hat ein Feld, das für die Referenzzählung verwendet wird. Der initiale abstrakte Syntaxbaum weist für jedes Element einen Referenzzähler von -1 auf, was bedeutet, dass diese Elemente nicht von der Referenzzählung berücksichtigt werden sollen.

Generell gilt, dass jede Zuweisung den Wert des Referenzzählers des zugewiesenen Elements um Eins erhöht. Wird eine Variable überschrieben, oder verliert sie durch Verlassen einer Umgebung[21] ihre Gültigkeit, so wird der Referenzzähler des betroffenen

20 Dazu gehören: `malloc()`, `calloc()`, `realloc()` und `free()`
21 siehe *3.2 Implementierung des Environments*

Elements um Eins reduziert. Diese beiden Operationen werden nur dann ausgeführt, wenn der Referenzzähler eine positive Zahl ist. Auf diese Weise können statische Elemente wie Literale einfach von der Referenzzählung ausgeschlossen werden.

Weist ein Element nach einer Reduktion einen Wert von Null auf bedeutet das, dass die Variable nicht mehr zugänglich ist und der Speicher freigegeben werden kann.

Im Folgenden sind die beiden Referenzzählungsfunktionen abgebildet, die das beschriebene Verhalten noch einmal verdeutlichen sollen:

```
void inc_ref(ast_t* ast) {
  if(ast->ref_count >= 0) {
    ast->ref_count++;
  }
}

void dec_ref(ast_t* ast) {
  if(ast->ref_count > 0) {
    ast->ref_count--;
  }
  if(ast->ref_count == 0) {
    free_ast(ast);
  }
}
```

Diese Funktionen werden an vielen Stellen innerhalb des Interpreters aufgerufen. Folgendes Beispiel soll einen Eindruck davon vermitteln, wie die Referenzzählung in QPL genau arbeitet. Um die Übersicht zu wahren wird nur die Instanz betrachtet, die in der ersten Zeile durch Zuweisung eines Literals erzeugt wird:

```
var = 42; /* var wird eine integer variable zugewiesen. die
           * referenzzählung beginnt. ref_count wird 1. */
sqr = @(x) {
  /* eine einfache funktion zur berechnung des quadrats */
```

```
  @ = x * x;
};
erg = sqr(var); /* funktionsaufruf mit var als parameter.
                 * der parameter x der funktion sqr zeigt
                 * auf die selbe instanz wie var.
                 * ref_count wird 2.
                 * beim verlassen der funktion verliert x
                 * seine gültigkeit.
                 * ref_count wird 1. */
var = 0; /* zuweisung eines neuen werts auf var.
          * ref_count wird 0.
          * die instanz wird freigegeben. */
```

Das Beispiel zeigt, wie sich die Referenzzählung einer Instanz während des Programmablaufs ändert, insbesondere die Zeile `erg = sqr(var);` ist sehr interessant, weil durch den Funktionsaufruf eine neue Umgebung erzeugt wird, in der die betrachtete Instanz ebenfalls enthalten ist. Das bedeutet, dass die Referenzzählung beim Betreten der Funktion erhöht und beim Verlassen reduziert werden muss.

Durch diese Methode gewinnt man mit relativ kleinem Overhead eine robuste Speicherverwaltung, die dem Programmierer alle Details der Speicherverwaltung abstrahiert [23].

6. Schluss

Abschließend werden hier einerseits die Ergebnisse und der Inhalt der Diplomarbeit zusammengefasst und andererseits ein Ausblick auf zukünftige Entwicklungen, Änderungen und Anpassungen gegeben.

6.1 Zusammenfassung

Bereits nach vergleichsweise kurzer Zeit konnte eine Programmiersprache entwickelt werden, die sich im praktischen Einsatz durch den schnellen Workflow und die einfache Syntax abhebt. Dank des flexiblen Built-In-Subsystems ist es einfach möglich die Programmiersprache um grundlegende Funktionen zu erweitern und so für ein breites Einsatzspektrum interessant zu machen.

Zu Beginn der Diplomarbeit wurden die Grundlagen und Konzepte zu Lexer und Parser aufgegriffen und detailliert beleuchtet. Die gezeigten Mechanismen stellen die Grundlage des von *Flex* generierten Lexers und des von *Bison* generierten Parsers dar.

Ein besonderes Merkmal der Programmiersprache sind die Built-In-Funktionen, die in einem ausführlichen Kapitel anhand vieler Beispiele gezeigt wurden. Neben einer Kategorisierung der Funktionen wurde gezeigt, wie man Built-In-Funktionen der Ausführungseinheit bekanntmachen kann.

Einen zentralen Teil der Diplomarbeit stellt das Kapitel *3. Interpreter* dar. In ihm wurde auf die Architektur, die Bestandteile und das Zusammenspiel der Elemente des Interpreters eingegangen. Es wurde gezeigt, wie die virtuelle Maschine den abstrakten Syntaxbaum abarbeitet und auf Basis verschiedener Typen des abstrakten Syntaxbaumes entscheidet, welche Programmteile zur Ausführung benötigt werden. Neben Erläuterungen zur Ausführungsumgebung, Operatoren und Datentypen wurde gezeigt, wie Fehlermeldungen in unterschiedlichen Situationen generiert werden.

Im Kapitel *4. Optimierung* wurden Strategien zur Verbesserung des Laufzeitverhaltens vorgestellt, welche einerseits auf die Optimierung des abstrakten Syntaxbaumes und andererseits auf die Optimierung des Interpreters abzielen. Weiters wurden Überlegungen zu Parallelität angestellt, die Mechanismen wie Threads, Locking und atomaren Variablenzugriff umfassen.

Zum Schluss der Arbeit wurde noch ein Blick auf die Speicherverwaltung geworfen. Es wurden zwei Strategien zur Verwaltung von dynamischem Speicher gegenübergestellt und auf die implementierte Referenzzählung näher eingegangen.

Im Anhang *C. Project QPL* ist der vollständige Quelltext der Programmiersprache zum aktuellen Zeitpunkt angefügt. Weiters sind im Anhang Informationen über die genutzten Tools und eine Anleitung zum Erstellen des Interpreters zu finden.

6.2 Ausblick

Nach Beendigung des Studiums werde ich die Programmiersprache QPL weiter betreuen und weiterentwickeln. Der vollständige Quellcode ist auf github[22] verfügbar und kann von jedem verwendet und eingesehen werden.

QPL wird von mir neben `bash` und `python` bereits jetzt für einige alltägliche Aufgaben genutzt und soll in Zukunft noch mehr verwendet werden. Zu diesem Zweck werden über die einfache Built-In-Schnittstelle noch weitere Funktionen implementiert, die die Zusammenarbeit mit dem Betriebssystem und die Integration in Shell-Scripte vereinfachen. Als Beispiele sollen hier ein eingebauter Parameterparser für Kommandozeilenparameter und abstrahierte Built-In-Funktionen zur einfachen Arbeit mit Pipes erwähnt werden.

Bisher unterstützt die Programmiersprache noch keine Parallelität. Um die möglichen Einsatzbereiche von QPL zu erweitern, können noch entsprechende Funktionen implementiert werden. Dazu sollen die Überlegungen aus dem Kapitel *4.2 Theorien & Techniken zu Parallelität* herangezogen und verfeinert werden.

Weiters bieten sich interpretierte Programmiersprachen als Pluginsprachen für größere Anwendungen an. Um dies zu ermöglichen muss eine Schnittstelle definiert werden, die es erlaubt, QPL in andere Anwendungen einzubetten und Daten mit der Hostanwendung auszutauschen.

Zusammengefasst kann man sagen, dass mit QPL eine neue Programmiersprache entwickelt wurde, die flexibel erweiterbar und einfach zu programmieren ist. Die bisher implementieren Funktionen und die geplanten Erweiterungen stellen in Aussicht, dass QPL zukünftig in immer mehr Bereichen eingesetzt wird.

22 Verfügbar unter github.com/linluk/project-qpl.

Literaturverzeichnis

[1]: Lukas Singer; Praxisprojekt - Basisarbeit und Konzept zum Entwurf und zur Implementierung einer Programmiersprache; 2015; Hochschule Mittweida

[2]: Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman; Compiler - Prinzipien, Techniken und Werkzeuge; Pearson Studium; 2008; ISBN 978-3-8273-7097-6

[3]: Christian Wagenknecht, Michael Hielscher; Formale Sprachen, abstrakte Automaten und Compiler; Vieweg +Teubner; 2009; ISBN 978-3-8348-0624-6

[4]: The Flex Project; sourceforge.net; Flex; 2008; http://flex.sourceforge.net/

[5]: Reinhard Wilhelm, Helmut Seidl, Sebastian Hack; Übersetzerbau - Band 2: Syntaktische und semantische Analyse; Springer-Verlag Berlin Heidelberg; 2012; ISBN 978-3-642-01135-1 (e-Book)

[6]: Free Software Foundation; gnu.org; GNU Bison; 2015; https://www.gnu.org/software/bison/

[7]: Fowler, Noll, Vo; ietf.org; The FNV Non-Cryptographic Hash Algorithm; 2016; https://tools.ietf.org/html/draft-eastlake-fnv-10

[8]: Ian D. Craig; Virtual Machines; Springer-Verlag London Limited; 2005; ISBN: 978-1-85233-969-2

[9]: cplusplus.com; cplusplus.com; Standard error stream; 2015; http://www.cplusplus.com/reference/cstdio/stderr/

[10]: Free Software Foundation; gnu.org; The GNU C Library - Backtraces; 2015; http://www.gnu.org/software/libc/manual/html_node/Backtraces.html

[11]: Helmut O.B. Schellong; Moderne C-Programmierung; Springer-Verlag Berlin Heidelberg; 2014; ISBN 978-3-642-54436-1

[12]: Free Software Foundation; gcc.gnu.org; A GNU Manual - Options That Control Optimization; 2016; https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[13]: Michael Larabel; Phoronix Media; The Performance Between GCC Optimization Levels; 2012; http://www.phoronix.com/scan.php?page=article&item=gcc_47_optimizations

[14]: Tobias Fechner; Universität Hamburg, Fachbereich Informatik; Laufzeitkosten in C; 2014; https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2013_2014/epc-1314-fechner-laufzeitkosten-ausarbeitung.pdf

[15]: Jasper Neumann, Jens Henrik Göbbert; Jasper Neumann; Improving Switch Statement Performance with Hashing Optimized at Compile Time; 2013; http://programming.sirrida.de/hashsuper.pdf

[16]: Uwe Schneider; Taschenbuch der Informatik; Fachbuchverlag Leipzig im Carl Hanser Verlag; 2012; ISBN 978-3-446-42638-2

[17]: Dick Buttlar, Jacqueline Farrell, Bradford Nichols; PThreads Programming; O'Reilly Media; 1996; ISBN 978-1-56592-115-3

[18]: Thomas Rauber; Parallele Programmierung; Springer-Verlag Berlin Heidelberg; 2012; ISBN 978-3-642-13604-7 (e-Book)

[19]: Brian Goetz; IBM; Thread pools and work queues; 2002; http://www.ibm.com/developerworks/java/library/j-jtp0730/index.html

[20]: Reinhard Wilhelm, Helmut Seidl; Compiler Design - Virtual Machines; Springer-Verlag Berlin Heidelberg; 2010; ISBN 978-3-642-14908-5

[21]: Jürgen Wolf; C von A bis Z; Rheinwerk Computing; 2009; ISBN 978-3-8362-1411-7

[22]: David F. Bacon, Perry Cheng, V. T. Rajan; IBM; A Unified Theory of Garbage Collection; 2004; https://www.cs.virginia.edu/~cs415/reading/bacon-garbage.pdf

[23]: Ben Klemens; C im 21. Jahrhundert; O'Reilly Media; 2013; ISBN 978-3-95561-385-3

[24]: Chris DiBona; google.com; Bidding farewell to Google Code; 2015; http://google-opensource.blogspot.de/2015/03/farewell-to-google-code.html

[25]: Drew Neil; Practical Vim; The Pragmatic Programmers; 2012; ISBN 978-1-934356-98-2

Anhang

A. Entwicklungsumgebung

Entwickelt wurde die Programmiersprache zuerst unter Debian 8 und später unter Ubuntu 15.10.

Zur Quellcodeverwaltung wurde zuerst Svn und später Git eingesetzt. Der Wechsel von Svn nach Git wurde auf Grund des angekündigten Endes des Dienstes Google Code vollzogen [24].

Als Parsergenerator wurden GNU Bison in der Version 3.0.2 eingesetzt.

Als Lexergenerator wurde Flex in der Version 2.5.39 eingesetzt.

Als C Compiler wurde der C Compiler der GNU Compiler Collection zuerst in der Version 4.9.2 und später in der Version 5.2.1 eingesetzt.

Als Editor wurde Vim in der Version 7.4 mit einigen Plugins (clang_complete, CRefVim, syntastic, gundo.vim, …) eingesetzt [25].

Als Buildsystem wurde GNU Make in der Version 4.0 eingesetzt.

B. Detailimplementierungen

B.1 Hashmap

Eine Hashmap ist ein assoziatives Array. Sie besteht aus Schlüssel- und Wertepaaren (Key, Value). Über die Schlüssel kann auf die Werte zugegriffen werden.

Aus dem Schlüssel wird mit einem einfachen Algorithmus ein Hashwert gebildet. Der Rest der ganzzahligen Division mit der Länge des Arrays ergibt den Index des Elements. Ist dieser Platz bereits belegt, wird der errechnete Index so lange inkrementiert bis ein freier Platz gefunden wurde. Durch diese Herangehensweise können Elemente auch bei großen Arrays in relativ konstanter Zeit gefunden werden, da neben der Hashfunktion meist nur noch wenige Vergleichsoperationen durchgeführt werden müssen.

Ein Element der Hashmap wird durch folgende Struktur beschrieben:

```
typedef struct key_value_s {
```

```
  size_t hash;
  char* key;
  void* value;
} key_value_t;
```

Die eigentliche Hashmap wird durch folgende Struktur beschrieben:

```
typedef struct map_s {
  size_t size;  /* maximum count */
  size_t count; /* element count */
  size_t index; /* iter index (see iter_first() & iter_next()) */
  key_value_t* items;
} map_t;
```

Das Feld `items` ist das eigentliche Array und beinhaltet die Elemente. Dieses Array muss immer größer sein als die Anzahl der Elemente und leere Elemente müssen zwingend gekennzeichnet sein (durch ein Flag oder durch den initialisierten Wert `null`).

Ein Kernelement der Implementierung ist die Umsetzung des FNV-1 Hashing Algorithmus welche hier dargestellt ist:

```
#define FNV_PRIME_32 (16777619)
#define FNV_OFFSET_BASIS_32 (2166136261)
size_t get_hash(const char* key) {
  size_t hash = FNV_OFFSET_BASIS_32;
  size_t len = strlen(key);
  size_t idx = 0;
  while (idx < len) {
    hash *= FNV_PRIME_32;
    hash ^= key[idx];
    idx++;
  }
  return hash;
}
```

Der Hashing Algorithmus liegt als Internet-Draft dem IETF vor [7].

B.2 replace_str()

```
char* replace_str(
    const char* str, const char* old, const char* new) {
  size_t olen; /* length of old sub string */
  size_t nlen; /* length of new sub string */
  size_t slen; /* length of source string */
  size_t rlen; /* length of result string */
  char* tmp;
  char* result;
  ptrdiff_t doffset;
  ptrdiff_t soffset;
  ptrdiff_t clen;
  int cnt;
  olen = strlen(old);
  nlen = strlen(new);
  slen = strlen(str);
  /* to get the count of replace patterns */
  cnt = 0; /* we initialize with count = 0 */
  tmp = strstr(str, old); /* find the first occourence */
  while(tmp != NULL) { /* check if old string was found */
    cnt++; /* increment the count */
    tmp = strstr((tmp + olen), old); /* find the next occurence */
  }
  if(cnt == 0) {
    /* if there is nothing to replace
     * return a copy of the source string */
    result = strdup(str); /* copy the original string to result */
  } else {
    /* otherwise allocate memory for the result string */
    rlen = slen + cnt * (nlen - olen);
```

```
    result = (char*)check_malloc((1 + rlen)  * sizeof(char));
    doffset = 0;
    soffset = 0;
    clen = 0;
    /* and start replacing every pattern of the old string with
     * the new string ... */
    tmp = strstr(str, old);
    while(tmp != NULL) {
      /* by considering the offset ... */
      clen = tmp - (str + soffset);
      /* and copying a part of the source string to the end of the
       * result string ... */
      strncpy(result + doffset, str + soffset, clen);
      doffset += clen;
      soffset += clen + olen;
      /* and then appending the new pattern to it ... */
      strncpy(result + doffset, new, nlen);
      doffset += nlen;
      /* until all old patterns are found */
      tmp = strstr(str + soffset, old);
    }
    /* if the string hasn't ended wit an old pattern there is a
     * last part of the source string to append ... */
    strncpy(result + doffset, str + soffset, rlen – doffset);
    /* before we can append the terminating zero */
    result[rlen] = '\0';
  }
  return result;
}
```

C. Project QPL

Als Quellcodeverwaltung kommt Git zum Einsatz. Der hier abgedruckte Quellcode entspricht dem Commit mit der ID: 963fff2a4829d1366f11a121142f590194f49dfe.

Zum erzeugen des Interpreters wird ein C-Compiler sowie die Tools GNU Bison und Flex benötigt. Es ist ein Makefile enthalten, welches den C-Compiler der GNU Compiler Collection nützt.

Durch folgende Befehle lässt sich der Interpreter via Git herunterladen und compilieren:

```
 ~ $ git clone https://github.com/linluk/project-qpl
Klone nach 'project-qpl' ...
remote: Counting objects: 401, done.
remote: Total 401 (delta 0), reused 0 (delta 0), pack-reused 400
Empfange Objekte: 100% (401/401), 92.18 KiB | 0 bytes/s, Fertig.
Löse Unterschiede auf: 100% (311/311), Fertig.
Prüfe Konnektivität ... Fertig.
 ~ $ cd project-qpl/
 ~/project-qpl $ make
bison parser.y
flex lexer.l
gcc -c -g -rdynamic -Wall parser.c
gcc -c -g -rdynamic -Wall lexer.c
gcc -c -g -rdynamic -Wall main.c
gcc -c -g -rdynamic -Wall ops.c
gcc -c -g -rdynamic -Wall env.c
gcc -c -g -rdynamic -Wall error.c
gcc -c -g -rdynamic -Wall vm.c
gcc -c -g -rdynamic -Wall ast.c
gcc -c -g -rdynamic -Wall utils.c
gcc -c -g -rdynamic -Wall map.c
gcc -c -g -rdynamic -Wall builtin.c
```

```
gcc -g -rdynamic parser.o lexer.o main.o ops.o env.o error.o vm.o
ast.o utils.o map.o builtin.o -o qpl
 ~/project-qpl $
```

C.1 QPL-Sourcecode

```
================================================================================
===  Dateiname: cat_files.sh
================================================================================
#!/bin/bash

# this script prints the content of all the source files in the
# current directory.
# you can use xclip to pipe the output to the clipboard so that it
# can easily be pastet to whereever you want.
#
#   $ ./cat_files.sh | xclip -selection clipboard

echo_80() {
  for ((i = 0; i < 79; i++)) {
    echo -ne "$1"
  }
  echo ""
}

remove_double_newlines() {
  # the sed command makes a single blank line from multiple
  sed -e '/./b' -e :n -e 'N;s/\n$//;tn'
}

for file in cat_files.sh Makefile *.h *.c *.l *.y
do
  echo_80 "="
  echo "===  Dateiname: $file"
  echo_80 "="
  cat $file | remove_double_newlines
  echo ""
done

================================================================================
===  Dateiname: Makefile
================================================================================
#
# Copyright (C) 2014 - 2016 Lukas Singer
#
# This file is part of 'qpl'.
#
# 'qpl' is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
```

```
# 'qpl' is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
#
# The name 'qpl' stands for Quick Programming Language and is a working
# title.  It may changes in the future.
#
# Source code can be found under: <https://github.com/linluk/project-qpl/>.
#

LEX=flex
YACC=bison
CC=gcc
#-rdynamic is needed for stack trace on crash
CFLAGS=-g -rdynamic -Wall
LFLAGS=-g -rdynamic
BIN=qpl
OBJ=parser.o lexer.o main.o ops.o env.o error.o vm.o ast.o utils.o map.o \
    builtin.o

BIN: $(OBJ)
	$(CC) $(LFLAGS) $(OBJ) -o $(BIN)

%.o: %.c parser.c lexer.c
	$(CC) -c $(CFLAGS) $<

parser.c: parser.y
	$(YACC) $<

lexer.c: lexer.l
	$(LEX) $<

clean:
	rm -f $(BIN) $(OBJ) lexer.c lexer.h parser.c parser.h parser.output


================================================================================
===  Dateiname: ast.h
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 ******************************************************************************/

#ifndef __AST_H__
#define __AST_H__

/* lib */
#include <stdint.h>

/* own */
#include "map.h"

typedef enum ast_type_e {
  at_identifier,
  at_integer, at_double, at_string, at_bool,
  at_list,
  at_expression,
  at_assignment,
  at_call, at_callargs,
  at_statements,
  at_conditional,
  at_if, at_elif,
  at_while, at_dowhile,
  at_function, at_params,
  at_builtin,
    /* add values before this */
} ast_type_t;

typedef enum call_type_e {
  ct_named,    /* "normal" function call (f.e: str(), doSomething(), ...) */
  ct_anonymous /* anonymous call (f.e:  fun(x){@=x*x;}(3), ...) */
} call_type_t;

typedef enum operator_e {
  op_add, op_sub,
  op_mul, op_div, op_mod,
  op_lt, op_gt, op_le, op_ge,
  op_eq, op_neq,
  op_and, op_or,
  op_cat,
  op_deref
} operator_t;

typedef struct position_s {
  char* file;
  int line;
```

```
} position_t;

/* THE abstract syntax tree structure */
typedef struct ast_s {
  ast_type_t type;
  int ref_count; /* positive value: ref_count -> call free when it becomes zero
                    negative value: created by parser -> never call free */
  union { /* data */
    intmax_t i;
    long double d;
    char b;
    char* s;
    char* id;
    struct { /* list */
      size_t count;
      struct ast_s** elements;
    } list;
    struct { /* expression */
      operator_t op;
      struct ast_s* left;
      struct ast_s* right;
    } expression;
    struct { /* assignment */
      char* id;
      struct ast_s* right;
    } assignment;
    struct { /* statements */
      size_t count;
      struct ast_s** statements;
    } statements;
    struct { /* call */
      call_type_t call_type;
      union { /* call.function */
        char* id;                /* use this when call_type is ct_named */
        struct ast_s* function;  /* use this when call_type is ct_anonymous */
      } function;
      struct ast_s* callargs;
    } call;
    struct { /* callargs */
      size_t count;
      struct ast_s** callargs;
    } callargs;
    struct { /* conditional */
      struct ast_s* if_statement;
      struct ast_s* elif_statements;
      struct ast_s* else_statement;
    } conditional;
    struct { /* if_statement */
      struct ast_s* condition;
      struct ast_s* statements;
    } if_statement;
    struct { /* elif_statements */
      size_t count;
      struct ast_s** elif_statements;
    } elif_statements;
    struct { /* while_statement */
```

```
      struct ast_s* condition;
      struct ast_s* statements;
    } while_statement;
    struct { /* dowhile_statement */
      struct ast_s* condition;
      struct ast_s* statements;
    } dowhile_statement;
    struct { /* function */
      struct ast_s* params;
      struct ast_s* statements;
    } function;
    struct { /* params */
      size_t count;
      char** params;
    } params;
    struct { /* builtin */
      size_t paramcount;
      union { /* builtin.function */
        struct ast_s* (*builtin_0)();
        struct ast_s* (*builtin_1)(struct ast_s*);
        struct ast_s* (*builtin_2)(struct ast_s*, struct ast_s*);
        struct ast_s* (*builtin_3)(struct ast_s*, struct ast_s*, struct ast_s*);
      } function;
    } builtin;
  } data;
} ast_t;

ast_t* create_integer(intmax_t value);
ast_t* create_double(long double value);
ast_t* create_bool(char value);
ast_t* create_string(char* value);
ast_t* create_list(ast_t* list, ast_t* element);
ast_t* create_identifier(char* id);
ast_t* create_expression(operator_t op, ast_t* left, ast_t* right);
ast_t* create_assignment(char* id, ast_t* right);
ast_t* create_statement(ast_t* statements, ast_t* statement);
ast_t* create_call(char* id, ast_t* function, ast_t* callargs);
ast_t* create_callarg(ast_t* callargs, ast_t* callarg);
ast_t* create_conditional(ast_t* if_statement, ast_t* elif_statements, ast_t*
       else_statement);
ast_t* create_if(ast_t* condition, ast_t* statements);
ast_t* create_elif(ast_t* elif_statements, ast_t* elif_statement);
ast_t* create_while(ast_t* condition, ast_t* statements);
ast_t* create_dowhile(ast_t* condition, ast_t* statements);
ast_t* create_function(ast_t* params, ast_t* statements);
ast_t* create_param(ast_t* params, char* id);
ast_t* create_builtin_0(ast_t*(*builtin_0)());
ast_t* create_builtin_1(ast_t*(*builtin_1)(ast_t*));
ast_t* create_builtin_2(ast_t*(*builtin_2)(ast_t*,ast_t*));
ast_t* create_builtin_3(ast_t*(*builtin_3)(ast_t*,ast_t*,ast_t*));
const char* get_ast_type_name(ast_type_t ast);
const char* get_op_str(operator_t op);

void inc_ref(ast_t* ast);
void dec_ref(ast_t* ast);
```

```
char is_numeric_ast_type(ast_type_t ast);

void print_ast(ast_t* ast, int indent);

void free_ast(ast_t* ast);

#endif /* __AST_H__ */


================================================================================
===  Dateiname: builtin.h
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

#ifndef __BUILTIN_H__
#define __BUILTIN_H__

#include "ast.h"
#include "env.h"

void populate_env(env_t* env);

/* stdin stdout */
ast_t* builtin_print(ast_t* ast);
ast_t* builtin_println(ast_t* ast);
ast_t* builtin_read(void);
ast_t* builtin_readln(void);

ast_t* builtin_to_string(ast_t* ast);
ast_t* builtin_to_integer(ast_t* ast);
ast_t* builtin_to_double(ast_t* ast);
ast_t* builtin_to_bool(ast_t* ast);
```

```
ast_t* builtin_replace(ast_t* str, ast_t* old, ast_t* new);

ast_t* builtin_type_of(ast_t* var);

ast_t* builtin_run(ast_t* command);

/* file i/o */
ast_t* builtin_fread(ast_t* filename);
ast_t* builtin_fwrite(ast_t* filename, ast_t* content);

#endif /* __BUILTIN_H__ */


================================================================================
===  Dateiname: env.h
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

#ifndef __ENV_H__
#define __ENV_H__

#include "map.h"
#include "ast.h"

typedef struct env_s {
  struct env_s* parent;
  map_t* map;
} env_t;

env_t* create_env(void);
```

```
void free_env(env_t* env);

ast_t* get_ast_by_id(env_t* env, const char* id);

void set_ast_to_id(env_t* env, const char* id, ast_t* ast);

#endif /* __ENV_H__ */


================================================================================
===  Dateiname: error.h
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

#ifndef __ERROR_H__
#define __ERROR_H__

#include "ast.h"

void error_expected(position_t* pos, const char* expected, const char* received);
void error_unexpected(position_t* pos, const char* unexpected);
void error_assign(position_t* pos, const char* from, const char* to);
void error_apply(position_t* pos, const char* op, const char* arg1, const char*
      arg2);
void error_id(position_t* pos, const char* id);
void error_paramcount(position_t* pos, const char* func, int paramcount, int
      callargcount);
void error_unsupported(position_t* pos, const char* unsupported);
void error_convert(position_t* pos, const char* from, const char* to);
void error_astgen(position_t* pos, const char* err);
void error_failed(position_t* pos, const char* what);
```

```
#endif /* __ERROR_H__ */


================================================================================
===  Dateiname: map.h
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

#ifndef __MAP_H__
#define __MAP_H__

typedef struct key_value_s {
  size_t hash;
  char* key;
  void* value;
} key_value_t;

typedef struct map_s {
  size_t size;  /* maximum count */
  size_t count; /* element count */
  size_t index; /* iter index ( see iter_first() and iter_next() ) */
  key_value_t* items;
} map_t;

map_t* create_map(size_t size);

/* return NULL  -> added new value.
 * return !NULL -> updated value old value returned
 */
void* add_value(map_t* map, const char* key, void* value);

void* delete_value(map_t* map, const char* key);
```

```
void* get_value(map_t* map, const char* key);

/* returns the first element (not the first one added!)
 * returns NULL if map is empty.
 */
void* iter_first(map_t* map);

/* returns the next element
 * returns NULL if called after last element
 */
void* iter_next(map_t* map);

/* if param != 0 -> call free() on every value
*/
void free_map(map_t* map, int free_values);

#endif /* __MAP_H__ */


================================================================================
===  Dateiname: ops.h
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

#ifndef __OPS_H__
#define __OPS_H__

#include "env.h"
#include "ast.h"

/* math ops */
```

```
ast_t* eval_add(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_sub(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_mul(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_div(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_mod(env_t* env, ast_t* ast1, ast_t* ast2);

/* logical ops */
ast_t* eval_and(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_or(env_t* env, ast_t* ast1, ast_t* ast2);

/* compare ops */
ast_t* eval_gt(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_lt(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_ge(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_le(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_eq(env_t* env, ast_t* ast1, ast_t* ast2);
ast_t* eval_neq(env_t* env, ast_t* ast1, ast_t* ast2);

/* string ops */
ast_t* eval_cat(env_t* env, ast_t* ast1, ast_t* ast2);

#endif /* __OPS_H__ */


================================================================================
===  Dateiname: utils.h
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 ******************************************************************************/

#ifndef __UTILS_H__
#define __UTILS_H__
```

```
#include <stdlib.h>

/* if allocation fails program terminates! */
void* check_malloc(size_t size);
void* check_realloc(void* ptr, size_t size);

/* sets the pointer to NULL and frees the allocated value */
void free_null(void** p);

char* replace_str(const char* str, const char* old, const char* new);

char is_str_int(const char* str);
char is_str_dbl(const char* str);

#endif /* __UTILS_H__ */


================================================================================
===  Dateiname: vm.h
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

#ifndef __VM_H__
#define __VM_H__

#include "map.h"
#include "ast.h"
#include "env.h"

void vm_exec(env_t* env, ast_t* ast);

void exec_statements(env_t* env, ast_t* ast);
```

```
void exec_assignment(env_t* env, ast_t* ast);
void exec_conditional(env_t* env, ast_t* ast);
void exec_while(env_t* env, ast_t* ast);
void exec_dowhile(env_t* env, ast_t* ast);

int exec_if(env_t* env, ast_t* ast);

ast_t* eval_call(env_t* env, ast_t* ast);
ast_t* eval_expression(env_t* env, ast_t* ast);

#endif /* __VM_H__ */


================================================================================
===  Dateiname: ast.c
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl>.
 *
 *******************************************************************************/

/* lib */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* own */
#include "utils.h"
#include "error.h"
#include "env.h"
#include "vm.h"

/* self */
#include "ast.h"
```

```
/* ast type names, these are returned by get_ast_type_name() */
static const char* at_identifier_str = "identifier";
static const char* at_integer_str = "integer";
static const char* at_double_str = "double";
static const char* at_string_str = "string";
static const char* at_bool_str = "bool";
static const char* at_list_str = "list";
static const char* at_expression_str = "expression";
static const char* at_assignment_str = "assignment";
static const char* at_call_str = "call";
static const char* at_callargs_str = "callargs";
static const char* at_statements_str = "statements";
static const char* at_conditional_str = "conditional";
static const char* at_if_str = "if";
static const char* at_elif_str = "elif";
static const char* at_while_str = "while";
static const char* at_dowhile_str = "do-while";
static const char* at_function_str = "function";
static const char* at_params_str = "params";
static const char* at_builtin_str = "builtin";
static const char* at_unknown_str = "unknown";

/* operator strings, these are returned by get_op_str() */
static const char* op_add_str = "+";
static const char* op_sub_str = "-";
static const char* op_mul_str = "*";
static const char* op_div_str = "/";
static const char* op_mod_str = "%";
static const char* op_lt_str = "<";
static const char* op_gt_str = ">";
static const char* op_le_str = "<=";
static const char* op_ge_str = ">=";
static const char* op_eq_str = "==";
static const char* op_neq_str = "!=";
static const char* op_and_str = "&";
static const char* op_or_str = "|";
static const char* op_cat_str = "$";
static const char* op_unknown_str = "unknown";

/* prototypes */
ast_t* create_ast(ast_type_t type);

/* prototypes in "ast.c" */
ast_t* create_ast(ast_type_t type) {
  ast_t* ast;
  ast = check_malloc(sizeof(ast_t));
  ast->type = type;
  ast->ref_count = -1; /* created by parser --> never call free! */
  return ast;
}

/* prototypes in "ast.h" */
ast_t* create_integer(intmax_t value) {
  ast_t* ast;
  ast = create_ast(at_integer);
  ast->data.i = value;
```

```
  return ast;
}

ast_t* create_double(long double value) {
  ast_t* ast;
  ast = create_ast(at_double);
  ast->data.d = value;
  return ast;
}

ast_t* create_bool(char value) {
  ast_t* ast;
  ast = create_ast(at_bool);
  ast->data.b = value;
  return ast;
}

ast_t* create_string(char* value) {
  ast_t* ast;
  ast = create_ast(at_string);
  ast->data.s = value;
  return ast;
}

ast_t* create_list(ast_t* list, ast_t* element) {
  if(list == NULL) {
    list = create_ast(at_list);
    list->data.list.count = 0;
    list->data.list.elements = NULL;
  }
  if(list->type != at_list) {
    fprintf(stderr, "ast generation error\n");
    exit(1);
  }
  if(element != NULL) {
    list->data.list.count++;
    list->data.list.elements = (ast_t**)check_realloc(
        list->data.list.elements,
        list->data.list.count * sizeof(ast_t*)
    );
    list->data.list.elements[list->data.list.count - 1] = element;
  }
  return list;
}

ast_t* create_identifier(char* id) {
  ast_t* ast;
  ast = create_ast(at_identifier);
  ast->data.id = id;
  return ast;
}

ast_t* create_expression(operator_t op, ast_t* left, ast_t* right) {
  ast_t* ast;
  ast = create_ast(at_expression);
  ast->data.expression.op = op;
```

```
  ast->data.expression.left = left;
  ast->data.expression.right = right;
  return ast;
}

ast_t* create_assignment(char* id, ast_t* right) {
  ast_t* ast;
  ast = create_ast(at_assignment);
  ast->data.assignment.id = id;
  ast->data.assignment.right = right;
  return ast;
}
ast_t* create_statement(ast_t* statements, ast_t* statement) {
  if(statements == NULL) {
    statements = create_ast(at_statements);
    statements->data.statements.count = 0;
    statements->data.statements.statements = NULL;
  }
  if(statements->type != at_statements) {
    error_astgen(NULL,"statement");
  }
  statements->data.statements.count++;
  statements->data.statements.statements = (ast_t**)check_realloc(
      statements->data.statements.statements,
      statements->data.statements.count * sizeof(ast_t*)
  );
  statements->data.statements.statements[statements->data.statements.count - 1] =
       statement;
  return statements;
}

ast_t* create_call(char* id, ast_t* function, ast_t* callargs) {
  ast_t* ast;
  ast = create_ast(at_call);
  if(id != NULL) {
    ast->data.call.call_type = ct_named;
    ast->data.call.function.id = id;
  } else {
    if(function != NULL) {
      ast->data.call.call_type = ct_anonymous;
      ast->data.call.function.function = function;
    } else {
      error_astgen(NULL,"call");
    }
  }
  if(callargs == NULL) {
    callargs = create_ast(at_callargs);
    callargs->data.callargs.count = 0;
    callargs->data.callargs.callargs = NULL;
  };
  ast->data.call.callargs = callargs;
  return ast;
}

ast_t* create_callarg(ast_t* callargs, ast_t* callarg) {
  if(callargs == NULL) {
```

```
    callargs = create_ast(at_callargs);
    callargs->data.callargs.count = 0;
    callargs->data.callargs.callargs = NULL;
  }
  if(callargs->type != at_callargs) {
    error_astgen(NULL,"callargs");
  }
  callargs->data.callargs.count++;
  callargs->data.callargs.callargs = (ast_t**)check_realloc(
      callargs->data.callargs.callargs,
      callargs->data.callargs.count * sizeof(ast_t*)
  );
  callargs->data.callargs.callargs[callargs->data.callargs.count - 1] = callarg;
  return callargs;
}

ast_t* create_conditional(ast_t* if_statement, ast_t* elif_statements, ast_t*
       else_statement) {
  ast_t* ast;
  ast = create_ast(at_conditional);
  ast->data.conditional.if_statement = if_statement;
  ast->data.conditional.elif_statements = elif_statements;
  ast->data.conditional.else_statement = else_statement;
  return ast;
}

ast_t* create_if(ast_t* condition, ast_t* statements) {
  ast_t* ast;
  ast = create_ast(at_if);
  ast->data.if_statement.condition = condition;
  ast->data.if_statement.statements = statements;
  return ast;
}

ast_t* create_elif(ast_t* elif_statements, ast_t* elif_statement) {
  if(elif_statements == NULL) {
    elif_statements = create_ast(at_elif);
    elif_statements->data.elif_statements.count = 0;
    elif_statements->data.elif_statements.elif_statements = NULL;
  }
  if(elif_statements->type != at_elif) {
    error_astgen(NULL,"elif");
  }
  elif_statements->data.elif_statements.count++;
  elif_statements->data.elif_statements.elif_statements = (ast_t**)check_realloc(
      elif_statements->data.elif_statements.elif_statements,
      elif_statements->data.elif_statements.count * sizeof(ast_t*)
  );
  elif_statements->data.elif_statements.elif_statements[elif_statements-
       >data.elif_statements.count - 1] = elif_statement;
  return elif_statements;
}

ast_t* create_while(ast_t* condition, ast_t* statements) {
  ast_t* ast;
  ast = create_ast(at_while);
```

```
  ast->data.while_statement.condition = condition;
  ast->data.while_statement.statements = statements;
  return ast;
}

ast_t* create_dowhile(ast_t* condition, ast_t* statements) {
  ast_t* ast;
  ast = create_ast(at_dowhile);
  ast->data.dowhile_statement.condition = condition;
  ast->data.dowhile_statement.statements = statements;
  return ast;
}

ast_t* create_function(ast_t* params, ast_t* statements) {
  ast_t* ast;
  ast = create_ast(at_function);
  if(params == NULL) {
    params = create_ast(at_params);
    params->data.params.count = 0;
    params->data.params.params = NULL;
  }
  ast->data.function.params = params;
  ast->data.function.statements = statements;
  return ast;
}

ast_t* create_param(ast_t* params, char* param) {
  if(params == NULL) {
    params = create_ast(at_params);
    params->data.params.count = 0;
    params->data.params.params = NULL;
  }
  if(params->type != at_params) {
    error_astgen(NULL,"param");
  }
  params->data.params.count++;
  params->data.params.params = (char**)check_realloc(
      params->data.params.params,
      params->data.params.count * sizeof(char*)
  );
  params->data.params.params[params->data.params.count - 1] = param;
  return params;
}

ast_t* create_builtin_0(ast_t*(*builtin_0)()) {
  ast_t* ast;
  ast = create_ast(at_builtin);
  ast->data.builtin.paramcount = 0;
  ast->data.builtin.function.builtin_0 = builtin_0;
  return ast;
}

ast_t* create_builtin_1(ast_t*(*builtin_1)(ast_t*)) {
  ast_t* ast;
  ast = create_ast(at_builtin);
  ast->data.builtin.paramcount = 1;
```

```
  ast->data.builtin.function.builtin_1 = builtin_1;
  return ast;
}

ast_t* create_builtin_2(ast_t*(*builtin_2)(ast_t*,ast_t*)) {
  ast_t* ast;
  ast = create_ast(at_builtin);
  ast->data.builtin.paramcount = 2;
  ast->data.builtin.function.builtin_2 = builtin_2;
  return ast;
}

ast_t* create_builtin_3(ast_t*(*builtin_3)(ast_t*,ast_t*,ast_t*)) {
  ast_t* ast;
  ast = create_ast(at_builtin);
  ast->data.builtin.paramcount = 3;
  ast->data.builtin.function.builtin_3 = builtin_3;
  return ast;
}

void inc_ref(ast_t* ast) {
  if(ast->ref_count >= 0) {
    ast->ref_count++;
  }
}

void dec_ref(ast_t* ast) {
  if(ast->ref_count > 0) {
    ast->ref_count--;
  }
  if(ast->ref_count == 0) {
    free_ast(ast);
  }
}

char is_numeric_ast_type(ast_type_t ast) {
  return (ast == at_double) || (ast == at_integer);
}

const char* get_ast_type_name(ast_type_t ast) {
  switch(ast) {
    case at_assignment: return at_assignment_str;
    case at_bool: return at_bool_str;
    case at_call: return at_call_str;
    case at_identifier: return at_identifier_str;
    case at_callargs: return at_callargs_str;
    case at_conditional: return at_conditional_str;
    case at_double: return at_double_str;
    case at_dowhile: return at_dowhile_str;
    case at_elif: return at_elif_str;
    case at_expression: return at_expression_str;
    case at_function: return at_function_str;
    case at_if: return at_if_str;
    case at_integer: return at_integer_str;
    case at_params: return at_params_str;
    case at_statements: return at_statements_str;
```

```
    case at_string: return at_string_str;
    case at_while: return at_while_str;
    case at_builtin: return at_builtin_str;
    case at_list: return at_list_str;
  }
  return at_unknown_str;
}

const char* get_op_str(operator_t op) {
  switch(op) {
    case op_add: return op_add_str;
    case op_sub: return op_sub_str;
    case op_mul: return op_mul_str;
    case op_div: return op_div_str;
    case op_mod: return op_mod_str;
    case op_lt: return op_lt_str;
    case op_gt: return op_gt_str;
    case op_le: return op_le_str;
    case op_ge: return op_ge_str;
    case op_eq: return op_eq_str;
    case op_neq: return op_neq_str;
    case op_and: return op_and_str;
    case op_or: return op_or_str;
    case op_cat: return op_cat_str;
  }
  return op_unknown_str;
}

void print_ast(ast_t* ast, int indent){
  size_t i;
  for(i = 0; i < indent; i++) {
    printf(" ");
  }
  if(ast != NULL) {
    const char* tn = get_ast_type_name(ast->type);
    switch(ast->type) {
      case at_identifier:
        printf("%s: %s\n",tn,ast->data.id);
        break;
      case at_integer:
        printf("%s: %jd\n",tn,ast->data.i);
        break;
      case at_double:
        printf("%s: %Lf\n",tn,ast->data.d);
        break;
      case at_string:
        printf("%s: \"%s\"\n",tn,ast->data.s);
        break;
      case at_bool:
        printf("%s: %s\n",tn,ast->data.b > 0 ? "true" : "false");
        break;
      case at_list:
        printf("%s: %zu\n",tn,ast->data.list.count);
        for(i = 0; i < ast->data.list.count; i++) {
          print_ast(ast->data.list.elements[i], indent + 2);
        }
```

```
    break;
  case at_expression:
    printf("%s: %s\n",tn,get_op_str(ast->data.expression.op));
    print_ast(ast->data.expression.left,indent+2);
    print_ast(ast->data.expression.right,indent+2);
    break;
  case at_assignment:
    printf("%s: %s\n",tn,ast->data.assignment.id);
    print_ast(ast->data.assignment.right,indent+2);
    break;
  case at_call:
    switch(ast->data.call.call_type) {
      case ct_anonymous:
        printf("%s: <anonymous>\n", tn);
        print_ast(ast->data.call.function.function, indent+2);
        break;
      case ct_named:
        printf("%s: %s\n",tn, ast->data.call.function.id);
        break;
    }
    print_ast(ast->data.call.callargs,indent+2);
    break;
  case at_callargs:
    printf("%s: %zu\n",tn, ast->data.callargs.count);
    for(i = 0; i < ast->data.callargs.count; i++) {
      print_ast(ast->data.callargs.callargs[i],indent+2);
    }
    break;
  case at_statements:
    printf("%s:\n",tn);
    for(i = 0; i < ast->data.statements.count; i++) {
      print_ast(ast->data.statements.statements[i],indent+2);
    }
    break;
  case at_conditional:
    printf("%s:\n",tn);
    print_ast(ast->data.conditional.if_statement,indent+2);
    print_ast(ast->data.conditional.elif_statements,indent+2);
    print_ast(ast->data.conditional.else_statement,indent+2);
    break;
  case at_if:
    printf("%s:\n",tn);
    print_ast(ast->data.if_statement.condition,indent+2);
    print_ast(ast->data.if_statement.statements,indent+2);
    break;
  case at_elif:
    printf("%s: %zu\n",tn,ast->data.elif_statements.count);
    for(i = 0; i < ast->data.elif_statements.count; i++) {
      print_ast(ast->data.elif_statements.elif_statements[i],indent+2);
    }
    break;
  case at_while:
    printf("%s:\n",tn);
    print_ast(ast->data.while_statement.condition,indent+2);
    print_ast(ast->data.while_statement.statements,indent+2);
    break;
```

```
      case at_dowhile:
        printf("%s:\n",tn);
        print_ast(ast->data.dowhile_statement.condition,indent+2);
        print_ast(ast->data.dowhile_statement.statements,indent+2);
        break;
      case at_function:
        printf("%s:\n",tn);
        print_ast(ast->data.function.params,indent+2);
        print_ast(ast->data.function.statements,indent+2);
        break;
      case at_builtin:
        printf("%s:\n",tn);
        break;
      case at_params:
        printf("%s: %zu (",tn,ast->data.params.count);
        for(i = 0; i < ast->data.params.count; i++) {
          if(i != 0) {
            printf(", ");
          }
          printf(ast->data.params.params[i]);
        }
        printf(")\n");
        break;
      /* removed default -> i want to have a compilerwarnings! */
    }
  } else {
    printf("/* empty */\n");
  }
}

void free_ast(ast_t* ast) {
  size_t i;
  if(ast != NULL) {
    switch(ast->type) {
      case at_identifier:
        free(ast->data.id);
        break;
      case at_integer:
        break;
      case at_double:
        break;
      case at_string:
        free(ast->data.s);
        break;
      case at_bool:
        break;
      case at_expression:
        free_ast(ast->data.expression.left);
        free_ast(ast->data.expression.right);
        break;
      case at_assignment:
        free(ast->data.assignment.id);
        free_ast(ast->data.assignment.right);
        break;
        break;
      case at_call:
```

```
    switch(ast->data.call.call_type) {
      case ct_anonymous:
        free_ast(ast->data.call.function.function);
        break;
      case ct_named:
        free(ast->data.call.function.id);
        break;
    }
    free_ast(ast->data.call.callargs);
    break;
  case at_list:
    for(i = 0; i < ast->data.list.count; i++) {
      free_ast(ast->data.list.elements[i]);
    }
    free(ast->data.list.elements);
    break;
  case at_callargs:
    for(i = 0; i < ast->data.callargs.count; i++) {
      free_ast(ast->data.callargs.callargs[i]);
    }
    break;
  case at_statements:
    for(i = 0; i < ast->data.statements.count; i++) {
      free_ast(ast->data.statements.statements[i]);
    }
    break;
  case at_conditional:
    free_ast(ast->data.conditional.if_statement);
    free_ast(ast->data.conditional.elif_statements);
    free_ast(ast->data.conditional.else_statement);
    break;
  case at_if:
    free_ast(ast->data.if_statement.condition);
    free_ast(ast->data.if_statement.statements);
    break;
  case at_elif:
    for(i = 0; i < ast->data.elif_statements.count; i++) {
      free_ast(ast->data.elif_statements.elif_statements[i]);
    }
    break;
  case at_while:
    free_ast(ast->data.while_statement.condition);
    free_ast(ast->data.while_statement.statements);
    break;
  case at_dowhile:
    free_ast(ast->data.dowhile_statement.condition);
    free_ast(ast->data.dowhile_statement.statements);
    break;
  case at_function:
    free_ast(ast->data.function.params);
    free_ast(ast->data.function.statements);
    break;
  case at_builtin:
    break;
  case at_params:
    for(i = 0; i < ast->data.params.count; i++) {
```

```
          free(ast->data.params.params[i]);
        }
        break;
        /* removed default -> i want to have compilerwarnings! */
    }
    free(ast);
  }
}
```

```
================================================================================
===  Dateiname: builtin.c
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

/* lib */
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>
#include <limits.h>

/* own */
#include "ast.h"
#include "env.h"
#include "utils.h"
#include "error.h"

/* self */
#include "builtin.h"

void populate_env(env_t* env) {
```

```
  /* stdin stdout */
  set_ast_to_id(env, "print", create_builtin_1(&builtin_print));
  set_ast_to_id(env, "println", create_builtin_1(&builtin_println));
  set_ast_to_id(env, "read", create_builtin_0(&builtin_read));
  set_ast_to_id(env, "readln", create_builtin_0(&builtin_readln));

  /* conversion */
  set_ast_to_id(env, "str", create_builtin_1(&builtin_to_string));
  set_ast_to_id(env, "int", create_builtin_1(&builtin_to_integer));
  set_ast_to_id(env, "dbl", create_builtin_1(&builtin_to_double));

  set_ast_to_id(env, "replace", create_builtin_3(&builtin_replace));

  set_ast_to_id(env, "typeof", create_builtin_1(&builtin_type_of));
  set_ast_to_id(env, "run", create_builtin_1(&builtin_run));

  set_ast_to_id(env, "fread", create_builtin_1(&builtin_fread));
  set_ast_to_id(env, "fwrite", create_builtin_2(&builtin_fwrite));
}

ast_t* builtin_print(ast_t* ast) {
  ast_t* str;
  str = builtin_to_string(ast);
  printf("%s",str->data.s);
  free_ast(str);
  return NULL;
}

ast_t* builtin_println(ast_t* ast) {
  ast_t* result;
  result = builtin_print(ast);
  printf("\n");
  return result;
}

ast_t* builtin_read(void) {
  ast_t* result;
  int c;
  char* s;
  c = fgetc(stdin);
  if(c >= 0) {
    s = (char*)check_malloc(2 * sizeof(char));
    s[1] = '\0';
    s[0] = (char)((c > CHAR_MAX) ? (c - (UCHAR_MAX + 1)) : c);
  } else {
    s = (char*)check_malloc(1 * sizeof(char));
    s[0] = '\0';
  };
  result = create_string(s);
  result->ref_count = 0;
  return result;
}

ast_t* builtin_readln(void) {
#define CHUNK_SIZE (80)
  ast_t* result = NULL;
```

```
  size_t size = CHUNK_SIZE;
  size_t idx = 0;
  char* s = (char*)check_malloc(size * sizeof(char));
  int c = fgetc(stdin);
  while(c >= 0 && c != '\n') {
    s[idx] = (char)((c > CHAR_MAX) ? (c - (UCHAR_MAX + 1)) : c);
    idx++;
    if(idx >= size) {
      size += CHUNK_SIZE;
      s = (char*)check_realloc(s,size * sizeof(char));
    }
    c = fgetc(stdin);
  }
  s = (char*)check_realloc(s,(idx + 1) * sizeof(char));
  s[idx] = '\0';
  result = create_string(s);
  result->ref_count = 0;
  return result;
#undef CHUNK_SIZE
}

ast_t* builtin_to_string(ast_t* ast) {
  ast_t* str = NULL;
  char buf[50];
  switch(ast->type) {
    case at_bool: sprintf(buf,"%s",ast->data.b == 0 ? "false" : "true"); break;
    case at_double: sprintf(buf,"%Lf",ast->data.d); break;
    case at_integer: sprintf(buf,"%jd",ast->data.i); break;
    case at_string: str = create_string(strdup(ast->data.s)); break;
    case at_function: sprintf(buf,"function <%p>",ast); break;
    default: error_convert(NULL,get_ast_type_name(ast-
       >type),get_ast_type_name(at_string));
  }
  if(str == NULL) {
    str = create_string(strdup(buf));
  }
  str->ref_count = 0;
  return str;
}

ast_t* builtin_to_integer(ast_t* ast) {
  ast_t* num = NULL;
  switch(ast->type) {
    case at_bool: num = create_integer(ast->data.b == 0 ? 0 : 1); break;
    case at_double: num = create_integer((intmax_t)ast->data.d); break;
    case at_integer: num = create_integer(ast->data.i); break;
    case at_string:
      if(is_str_int(ast->data.s)) {
        num = create_integer(strtoimax(ast->data.s,NULL,10));
      } else {
        error_convert(NULL,get_ast_type_name(ast-
       >type),get_ast_type_name(at_integer));
      }
      break;
    default: error_convert(NULL,get_ast_type_name(ast-
       >type),get_ast_type_name(at_integer));
```

```
  }
  num->ref_count = 0;
  return num;
}

ast_t* builtin_to_double(ast_t* ast) {
  ast_t* dbl = NULL;
  switch(ast->type) {
    case at_bool: dbl = create_double(ast->data.b == 0 ? 0.0 : 1.0); break;
    case at_double: dbl = create_double(ast->data.d); break;
    case at_integer: dbl = create_double((long double)ast->data.i); break;
    case at_string:
      if(is_str_dbl(ast->data.s)) {
        dbl = create_double(atof(ast->data.s));
      } else {
        error_convert(NULL,get_ast_type_name(ast-
       >type),get_ast_type_name(at_integer));
      }
      break;
    default: error_convert(NULL,get_ast_type_name(ast-
       >type),get_ast_type_name(at_integer));
  }
  dbl->ref_count = 0;
  return dbl;
}

ast_t* builtin_to_bool(ast_t* ast) {
  ast_t* bol = NULL;
  switch(ast->type) {
    case at_bool: bol = create_bool(ast->data.b); break;
    case at_double: bol = create_bool(ast->data.d != 0.0); break;
    case at_integer: bol = create_bool(ast->data.i != 0); break;
    case at_string: bol = create_bool(strlen(ast->data.s) != 0); break;
    default: error_convert(NULL, get_ast_type_name(ast-
       >type),get_ast_type_name(at_bool));
  }
  bol->ref_count = 0;
  return bol;
}

ast_t* builtin_replace(ast_t* str, ast_t* old, ast_t* new) {
  /* TODO : implement for lists too */
  ast_t* s = builtin_to_string(str);
  ast_t* o = builtin_to_string(old);
  ast_t* n = builtin_to_string(new);
  ast_t* result = create_string(replace_str(s->data.s, o->data.s, n->data.s));
  result->ref_count = 0;
  dec_ref(s);
  dec_ref(o);
  dec_ref(n);
  return result;
}

ast_t* builtin_type_of(ast_t* var) {
  char* typ = strdup(get_ast_type_name(var->type));
  ast_t* result = create_string(typ);
```

```
  result->ref_count = 0;
  return result;
}

ast_t* builtin_run(ast_t* command) {
#define CHUNK_SIZE (80)
  if(command->type != at_string) {
    error_expected(NULL, get_ast_type_name(at_string), get_ast_type_name(command-
       >type));
  }
  /* whats with stderr? do i need stderr too?? this only gives me stdout */
  FILE* pipe = popen(command->data.s, "r");
  if(pipe == NULL) {
    error_failed(NULL,"popen()");
  }
  size_t size = CHUNK_SIZE;
  size_t idx = 0;
  char* s = (char*)check_malloc(size * sizeof(char));
  int c = fgetc(pipe);
  while(c != EOF) {
    s[idx] = (char)((c > CHAR_MAX) ? (c - (UCHAR_MAX + 1)) : c);
    idx++;
    if(idx >= size) {
      size += CHUNK_SIZE;
      s = (char*)check_realloc(s,size * sizeof(char));
    }
    c = fgetc(pipe);
  }
  pclose(pipe);
  s = (char*)check_realloc(s,(idx + 1) * sizeof(char));
  s[idx] = '\0';
  ast_t* result = create_string(s);
  result->ref_count = 0;
  return result;
#undef CHUNK_SIZE
}

ast_t* builtin_fread(ast_t* filename) {
#define CHUNK_SIZE (256)
  if(filename->type != at_string) {
    error_expected(NULL, get_ast_type_name(at_string), get_ast_type_name(filename-
       >type));
  }
  FILE* file = fopen(filename->data.s, "r");
  if(file == NULL) {
    error_failed(NULL, "fopen()");
  }
  size_t size = CHUNK_SIZE;
  size_t idx = 0;
  char* s = (char*)check_malloc(size * sizeof(char));
  int c;
  while( (c = fgetc(file)) != EOF ) {
    s[idx] = (char)((c > CHAR_MAX) ? (c - (UCHAR_MAX + 1)) : c);
    idx++;
    if(idx >= size) {
      size += CHUNK_SIZE;
```

```
      s = (char*)check_realloc(s,size * sizeof(char));
    }
  }
  fclose(file);
  s = (char*)check_realloc(s, (idx + 1) * sizeof(char));
  s[idx] = '\0';
  ast_t* result = create_string(s);
  result->ref_count = 0;
  return result;
#undef CHUNK_SIZE
}

ast_t* builtin_fwrite(ast_t* filename, ast_t* content) {
  if(filename->type != at_string) {
    error_expected(NULL, get_ast_type_name(at_string), get_ast_type_name(filename-
       >type));
  }
  if(content->type != at_string) {
    error_expected(NULL, get_ast_type_name(at_string), get_ast_type_name(content-
       >type));
  }
  FILE* file = fopen(filename->data.s, "w");
  if(file == NULL) {
    error_failed(NULL, "fopen()");
  }
  fprintf(file, "%s", content->data.s);
  //fprintf(stdout, "%s", content->data.s);
//  char c;
//  size_t idx = 0;
//  while( (c = content->data.s[idx]) != '\0') {
//    fputc(c, file);
//    idx++;
//  }
  fclose(file);
  return NULL;
}


================================================================================
===  Dateiname: env.c
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 ******************************************************************************/

/* lib */
#include <stdlib.h>

/* own */
#include "utils.h"
#include "map.h"
#include "ast.h"

/* self */
#include "env.h"

/* prototypes in "env.h" */
env_t* create_env(void) {
  env_t* env = (env_t*)check_malloc(sizeof(env_t));
  env->map = create_map(0);
  env->parent = NULL;
  return env;
}

void free_env(env_t* env) {
  ast_t* tmp;
  if((tmp = iter_first(env->map)) != NULL) {
    do {
      //      dec_ref(tmp);
    } while((tmp = iter_next(env->map)) != NULL);
  }
  free_map(env->map, 0);
  free(env);
}

ast_t* get_ast_by_id(env_t* env, const char* id) {
  ast_t* ast = (ast_t*)get_value(env->map, id);
  if(ast == NULL && env->parent != NULL) {
    ast = get_ast_by_id(env->parent, id);
  }
  return ast;
}

void set_ast_to_id(env_t* env, const char* id, ast_t* ast) {
  ast_t* old = (ast_t*)add_value(env->map, id, ast);
  inc_ref(ast);
  if(old != NULL) {
    dec_ref(old);
  }
}
```

```
================================================================================
===  Dateiname: error.c
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

/* lib */
#include <stdlib.h>
#include <stdio.h>

/* own */
#include "ast.h"

#define ERROR_BUFFER_LENGTH (255)

/* prototypes */
void error(position_t* pos, const char* emsg);

/* prototypes in "error.c" */
void error(position_t* pos, const char* emsg) {
  if(pos == NULL) {
    fprintf(stderr, "error: %s\n",emsg);
  } else {
    if(pos->file == NULL) {
      fprintf(stderr,"error (line: %d): %s\n", pos->line, emsg);
    } else {
      fprintf(stderr,"error (%s:%d): %s\n", pos->file, pos->line, emsg);
    }
  }
  exit(1);
}
```

```
/* prototypes in "error.h" */
void error_expected(position_t* pos, const char* expected, const char* received) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"expected: \"%s\", received: \"%s\"", expected, received);
  error(pos,buf);
}

void error_unexpected(position_t* pos, const char* unexpected) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"unexpected \"%s\"", unexpected);
  error(pos,buf);
}

void error_assign(position_t* pos, const char* from, const char* to) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"cannot assign \"%s\" to \"%s\"", from, to);
  error(pos,buf);
}

void error_apply(position_t* pos, const char* op, const char* arg1, const char*
       arg2) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"cannot apply \"%s\" to \"%s\" and \"%s\"", op, arg1, arg2);
  error(pos,buf);
}

void error_id(position_t* pos, const char* id) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"identifier \"%s\" not found", id);
  error(pos,buf);
}

void error_paramcount(position_t* pos, const char* func, int paramcount, int
       callargcount) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"function \"%s\" expects %d arguments, called with %d", func,
       paramcount, callargcount);
  error(pos,buf);
}

void error_unsupported(position_t* pos, const char* unsupported) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"unsupported feature: \"%s\"",unsupported);
  error(pos,buf);
}

void error_convert(position_t* pos, const char* from, const char* to) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf,"cannot convert \"%s\" into \"%s\"",from,to);
  error(pos,buf);
}

void error_astgen(position_t* pos, const char* err) {
  char buf[ERROR_BUFFER_LENGTH];
  if (err != NULL) {
```

```
    sprintf(buf,"AST create error: \"%s\"", err);
  } else {
    sprintf(buf,"AST create error");
  }
  error(pos, buf);
}

void error_failed(position_t* pos, const char* what) {
  char buf[ERROR_BUFFER_LENGTH];
  sprintf(buf, "%s failed", what);
  error(pos, buf);
}


================================================================================
===  Dateiname: main.c
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

#include <stdio.h>

/* needed to create a stack trace on crash */
#include <execinfo.h>
#include <signal.h>

#include "ast.h"
#include "vm.h"
#include "env.h"
#include "builtin.h"
#include "utils.h"

int yyparse(ast_t** ast_dest); /* get rid of implicit declaration warning */
```

```
extern int yydebug;
#include "lexer.h"
#include "parser.h"

void sigsegv_handler(int sig) {
  void *array[25];
  size_t size;
  // get void*'s for all entries on the stack
  size = backtrace(array, 25);
  // print out all the frames to stderr
  fprintf(stderr, "Error: signal %d:\n", sig);
  backtrace_symbols_fd(array, size, STDERR_FILENO);
  exit(1);
}

int main(int argc, char** argv) {
  signal(SIGSEGV, sigsegv_handler);

  ast_t* ast = NULL;

  if(argc > 1) {
    yyin = fopen(argv[1], "r");
  } else {
    yyin = stdin;
    printf("type your code and press <CTRL> + <D> \n");
  }

  //  yydebug = 1;

  if(yyin == NULL) {
    printf("yyin == NULL\n");
  }

  yyparse(&ast);

  //  print_ast(ast,0);

  if (yyin != stdin) {
    fclose(yyin);
  }

  if(ast != NULL) {

    env_t* env = create_env();
    populate_env(env);
    vm_exec(env, ast);

    //    free_ast(ast);
    free_env(env);

  } else {
    printf("ast == NULL\n");
    return 1;
  }
  return 0;
}
```

```
================================================================================
===  Dateiname: map.c
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h> /* TODO: refactor hash from string functions to pointer
       arithmetic */

#include "utils.h"

#include "map.h"

/* defines */
#define FNV_PRIME_32 (16777619)
#define FNV_OFFSET_BASIS_32 (2166136261)

//#define MAP_PARANOID_INFINITY_LOOP_CHECK
//#define DO_MAP_DEBUG

#define MAP_THRESHOLD (5)
#define MIN_MAP_SIZE (15)

/* prototypes */
size_t get_hash(const char* key);

void error_exit(char* emsg);

int is_empty_item(key_value_t item);
```

```
void rehash(map_t* map, size_t new_size);  /* TODO: rename to repopulate */

#ifdef DO_MAP_DEBUG
void debug_print_map(map_t* map, const char* msg);
#endif /* DO_MAP_DEBUG */

/* prototypes in "map.c" */
size_t get_hash(const char* key) {
  /* implements fnv-1 hashing function */
  size_t hash;
  size_t len;
  size_t idx;
  hash = FNV_OFFSET_BASIS_32;
  len = strlen(key);
  idx = 0;
  while (idx < len) {
    hash *= FNV_PRIME_32;
    hash ^= key[idx];
    idx++;
  }
  return hash;
}

#ifdef DO_MAP_DEBUG
void debug_print_map(map_t* map, const char* msg) {
  size_t i;
  printf("*** MAP DEBUG PRINT ***\n(%s)\n",msg);
  for(i = 0; i < map->size; i++) {
    if(is_empty_item(map->items[i])) {
      printf("element on position %zu is empty\n",i);
    }else{
      printf("element (%s) %zu has hash %zu, want to be element %zu\n",map-
       >items[i].key,i,map->items[i].hash,map->items[i].hash % map->size);
    }
  }
  printf("map size: %zu\nmap count: %zu\n",map->size,map->count);
}
#endif /* DO_MAP_DEBUG */

void error_exit(char* emsg) {
  fprintf(stderr, "error: \"%s\"", emsg);
  exit(1);
}

int is_empty_item(key_value_t item) {
  return (item.key == NULL && item.hash == 0 && item.value == NULL);
//  return (item.key == NULL);
}

void rehash(map_t* map, size_t new_size) {
 #ifdef DO_MAP_DEBUG
  debug_print_map(map,"before rehash()");
#endif /* DO_MAP_DEBUG */
  if(map->count > 0) {
    key_value_t tmp[map->count];
```

```
    size_t i1, i2;
    // extract all items
    i1 = 0;
    i2 = 0;
    while(i1 < map->size) {
      if(!is_empty_item(map->items[i1])) {
        tmp[i2].key = map->items[i1].key;
        tmp[i2].value = map->items[i1].value;
        tmp[i2].hash = map->items[i1].hash;
        i2++;
      }
      i1++;
    }
    free(map->items);
    // resize map
    i1 = map->count + MAP_THRESHOLD;
    i2 = i1 > MIN_MAP_SIZE ? i1 : MIN_MAP_SIZE;
    map->size = new_size > i2 ? new_size : i2;
    map->items = (key_value_t*)check_malloc(sizeof(key_value_t) * map->size);
    for(i1 = 0; i1 < map->size; i1++) {
      map->items[i1].key = NULL;
      map->items[i1].value = NULL;
      map->items[i1].hash = 0;
    }
    // repopulate map
#ifdef DO_MAP_DEBUG
      printf("*** MAP DEBUG REMAP ***\n");
#endif /* DO_MAP_DEBUG */
      for(i1 = 0; i1 < map->count; i1++) {
      i2 = tmp[i1].hash % map->size;
      while(!is_empty_item(map->items[i2])) {
        i2++;
        if(i2 >= map->size) {
          i2 = 0;
        }
      }
#ifdef DO_MAP_DEBUG
      printf("remap %s to %zu wants to be at %zu\n",tmp[i1].key,i2,tmp[i1].hash %
       map->size);
#endif /* DO_MAP_DEBUG */
      map->items[i2].key = tmp[i1].key;
      map->items[i2].value = tmp[i1].value;
      map->items[i2].hash = tmp[i1].hash;
    }
  }
#ifdef DO_MAP_DEBUG
  debug_print_map(map,"after rehash()");
#endif /* DO_MAP_DEBUG */
}

/* prototypes in "map.h" */
map_t* create_map(size_t size) {
  // should work
  map_t* map;
  size_t i;
  map = (map_t*)check_malloc(sizeof(map_t));
```

```
  map->size = size > MIN_MAP_SIZE ? size : MIN_MAP_SIZE;
  map->count = 0;
  map->index = 0;
  map->items = (key_value_t*)check_malloc(sizeof(key_value_t) * map->size);
  for(i = 0; i < map->size; i++) {
    map->items[i].key = NULL;
    map->items[i].value = NULL;
    map->items[i].hash = 0;
  }
#ifdef DO_MAP_DEBUG
  printf("*** MAP DEBUG CREATE ***\ncreated map at %p\nmap size: %zu\n",map,map-
      >size);
#endif /* DO_MAP_DEBUG */
  return map;
}

void* add_value(map_t* map, const char* key, void* value) {
  // should work
  size_t idx;
  size_t hash;
  size_t start;
  hash = get_hash(key);
  start = hash % map->size;
  idx = start;
#ifdef DO_MAP_DEBUG
  printf("*** MAP DEBUG ADD ***\nwill add value: %p with key: %s\n",value,key);
#endif /* DO_MAP_DEBUG */
  while(1) {
    if(is_empty_item(map->items[idx])) {
      /* insert item */
      map->items[idx].key = strdup(key);
      map->items[idx].value = value;
      map->items[idx].hash = hash;
      map->count++;
      if((map->count + MAP_THRESHOLD) >= map->size) {
        rehash(map, map->size + 2 * MAP_THRESHOLD);
      }
      return NULL;
    } else {
      if(strcmp(map->items[idx].key, key) == 0) {
        /* update item */
        void* old_value;
        old_value = map->items[idx].value;
        map->items[idx].value = value;
        return old_value;
      }
    }
    idx++;
    if(idx >= map->size) {
      idx = 0;
    }
#ifdef MAP_PARANOID_INFINITY_LOOP_CHECK
    if(start == idx) {
      error_exit("add_or_set_value() failed! (infinity loop)\n");
    }
#endif /* MAP_PARANOID_INFINITY_LOOP_CHECK */
```

```
  }
#ifdef DO_MAP_DEBUG
  debug_print_map(map,"after add_value()");
#endif /* DO_MAP_DEBUG */
}

void* delete_value(map_t* map, const char* key) {
  // should work
  size_t idx;
  size_t start;
#ifdef DO_MAP_DEBUG
  printf("*** MAP DEBUG DELETE ***\nwill delete key %s\n",key);
#endif /* DO_MAP_DEBUG */
  start = get_hash(key) % map->size;
  idx = start;
  while(1) {
    if(is_empty_item(map->items[idx])) {
      return NULL; // nothing to delete
    } else {
      if(strcmp(map->items[idx].key, key) == 0) {
        void* tmp;
        tmp = map->items[idx].value;
        free(map->items[idx].key);
        map->items[idx].key = NULL;
        map->items[idx].value = NULL;
        map->items[idx].hash = 0;
        map->count--;
        // TODO: shrink map (or maybe not??)
        rehash(map, map->size);
        return tmp;
      }
    }
    idx++;
    if(idx >= map->size) {
      idx = 0;
    }
#ifdef MAP_PARANOID_INFINITY_LOOP_CHECK
    if(start == idx) {
      error_exit("delete_value() failed! (infinity loop)\n");
    }
#endif /* MAP_PARANOID_INFINITY_LOOP_CHECK */
  }
#ifdef DO_MAP_DEBUG
  debug_print_map(map,"after delete_value()");
#endif /* DO_MAP_DEBUG */
}

void* get_value(map_t* map, const char* key) {
  // should work
  size_t idx;
  size_t start;
  start = get_hash(key) % map->size;
  idx = start;
  while(1) {
    if(is_empty_item(map->items[idx])) {
      return NULL;
```

```
    } else {
      if(strcmp(map->items[idx].key, key) == 0) {
        return map->items[idx].value;
      }
    }
    idx++;
    if(idx >= map->size) {
      idx = 0;
    }
#ifdef MAP_PARANOID_INFINITY_LOOP_CHECK
    if(start == idx) {
      error_exit("get_value() failed! (infinity loop)\n");
    }
#endif /* MAP_PARANOID_INFINITY_LOOP_CHECK */
  }
}

void* iter_first(map_t* map) {
  size_t idx = 0;
  map->index = idx;
  while(idx < map->size) {
    if(!is_empty_item(map->items[idx])) {
      map->index = idx;
      return map->items[idx].value;
    }
    idx++;
  }
  return NULL;
}

void* iter_next(map_t* map) {
  size_t idx;
  idx = map->index + 1;
  while(idx < map->size) {
    if(!is_empty_item(map->items[idx])) {
      map->index = idx;
      return map->items[idx].value;
    }
    idx++;
  }
  return NULL;
}

void free_map(map_t* map, int free_values) {
  if(free_values != 0) {
    void* tmp;
    tmp = iter_first(map);
    if(tmp != NULL) {
      do {
        free(tmp);
        tmp = iter_next(map);
      } while(tmp != NULL);
    }
  }
  free(map->items);
  free(map);
```

```
}

#ifdef FNV_PRIME_32
#  undef FNV_PRIME_32
#endif /* FNV_PRIME_32 */

#ifdef FNV_OFFSET_BASIS_32
#  undef FNV_OFFSET_BASIS_32
#endif /* FNV_OFFSET_BASIS_32 */

#ifdef MAP_PARANOID_INFINITY_LOOP_CHECK
#  undef MAP_PARANOID_INFINITY_LOOP_CHECK
#endif /* MAP_PARANOID_INFINITY_LOOP_CHECK */

#ifdef DO_MAP_DEBUG
#  undef DO_MAP_DEBUG
#endif /* DO_MAP_DEBUG */

#ifdef MAP_THRESHOLD
#  undef MAP_THRESHOLD
#endif /* MAP_THRESHOLD */

#ifdef MIN_MAP_SIZE
#  undef MIN_MAP_SIZE
#endif /* MIN_MAP_SIZE */


================================================================================
===  Dateiname: ops.c
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/
```

```
/* lib */
#include <stdio.h>
#include <string.h>
#include <stdint.h>

/* own */
#include "ast.h"
#include "vm.h"
#include "env.h"
#include "error.h"
#include "utils.h"

/* self */
#include "ops.h"

/* prototypes */
ast_t* eval_math(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2,long double
       (math_func)(long double,long double));
long double __add_func(long double d1, long double d2);
long double __sub_func(long double d1, long double d2);
long double __mul_func(long double d1, long double d2);
long double __div_func(long double d1, long double d2);

ast_t* eval_gtlt(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2, char
       (comp_func)(long double,long double));
char __gt_func(long double d1, long double d2);
char __lt_func(long double d1, long double d2);
char __ge_func(long double d1, long double d2);
char __le_func(long double d1, long double d2);
ast_t* eval_eqneq(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2);

/* prototypes in "ops.c" */
ast_t* eval_math(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2,long double
       (math_func)(long double,long double)) {
  ast_t* result = NULL;
  switch(ast1->type) {
    case at_integer:
      switch(ast2->type) {
        case at_integer:
          result = create_integer((intmax_t)math_func((long double)ast1->data.i,
       (long double)ast2->data.i));
          break;
        case at_double:
          result = create_double(math_func((long double)ast1->data.i, ast2-
       >data.d));
          break;
        default:
          error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
          break;
      }
      break;
    case at_double:
      switch(ast2->type) {
        case at_integer:
```

```
          result = create_double(math_func(ast1->data.d, (long double)ast2-
       >data.i));
          break;
        case at_double:
          result = create_double(math_func(ast1->data.d, ast2->data.d));
          break;
        default:
          error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
          break;
      }
      break;
    default:
      error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
      break;
  }
  return result;
}

long double __add_func(long double d1, long double d2) {
  return d1 + d2;
}

long double __sub_func(long double d1, long double d2) {
  return d1 - d2;
}

long double __mul_func(long double d1, long double d2) {
  return d1 * d2;
}

long double __div_func(long double d1, long double d2) {
  // TODO : div by zero
  return d1 / d2;
}

ast_t* eval_gtlt(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2, char
       (comp_func)(long double,long double)) {
  ast_t* result = NULL;
  switch(ast1->type) {
    case at_integer:
      switch(ast2->type) {
        case at_integer:
          result = create_bool(comp_func((long double)ast1->data.i, (long
       double)ast2->data.i));
          break;
        case at_double:
          result = create_bool(comp_func((long double)ast1->data.i, ast2-
       >data.d));
          break;
        default:
          error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
          break;
      }
```

```
      break;
    case at_double:
      switch(ast2->type) {
        case at_integer:
          result = create_bool(comp_func(ast1->data.d, (long double)ast2-
       >data.i));
          break;
        case at_double:
          result = create_bool(comp_func(ast1->data.d, ast2->data.d));
          break;
        default:
          error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
          break;
      }
      break;
    default:
      error_apply(NULL,get_op_str(op), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
      break;
  }
  return result;
}

char __gt_func(long double d1, long double d2) {
  return d1 > d2 ? 1 : 0;
}

char __lt_func(long double d1, long double d2) {
  return d1 < d2 ? 1 : 0;
}

char __ge_func(long double d1, long double d2) {
  return d1 >= d2 ? 1 : 0;
}

char __le_func(long double d1, long double d2) {
  return d1 <= d2 ? 1 : 0;
}

ast_t* eval_eqneq(env_t* env, operator_t op, ast_t* ast1, ast_t* ast2) {
  ast_t* result;
  result = NULL;
  if(ast1->type == ast2->type) {
    switch(ast1->type) {
      case at_bool:
        result = create_bool(ast1->data.b == ast2->data.b ? 1 : 0);
        break;
      case at_double:
        result = create_bool(ast1->data.d == ast2->data.d ? 1 : 0);
        break;
      case at_integer:
        result = create_bool(ast1->data.i == ast2->data.i ? 1 : 0);
        break;
      case at_string:
        result = create_bool(strcmp(ast1->data.s, ast2->data.s) == 0 ? 1 : 0);
```

```
        break;
      default:
        error_apply(NULL,get_op_str(op_eq), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
        break;
    }
    if(op == op_neq) {
      result->data.b = result->data.b == 0 ? 1 : 0;
    }
  } else {
    error_apply(NULL,get_op_str(op_eq), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
  }
  return result;
}

/* prototypes in "ops.h" */
/* math ops */
ast_t* eval_add(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_math(env,op_add,ast1,ast2,&__add_func);
}

ast_t* eval_sub(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_math(env,op_sub,ast1,ast2,&__sub_func);
}

ast_t* eval_mul(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_math(env,op_mul,ast1,ast2,&__mul_func);
}

ast_t* eval_div(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_math(env,op_div,ast1,ast2,&__div_func);
}

ast_t* eval_mod(env_t* env, ast_t* ast1, ast_t* ast2) {
  ast_t* result = NULL;
  if(ast1->type == at_integer && ast2->type == at_integer) {
    result = create_integer(ast1->data.i % ast2->data.i);
  } else {
    error_apply(NULL,get_op_str(op_mod), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
  }
  return result;
}

/* logical ops */
ast_t* eval_and(env_t* env, ast_t* ast1, ast_t* ast2) {
  ast_t* result = NULL;
  if(ast1->type == at_bool && ast2->type == at_bool) {
    result = create_bool(ast1->data.b && ast2->data.b);
  } else {
    error_apply(NULL,get_op_str(op_and), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
  }
  return result;
}
```

```
ast_t* eval_or(env_t* env, ast_t* ast1, ast_t* ast2) {
  ast_t* result = NULL;
  if(ast1->type == at_bool && ast2->type == at_bool) {
    result = create_bool(ast1->data.b || ast2->data.b);
  } else {
    error_apply(NULL,get_op_str(op_and), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
  }
  return result;
}

/* compare ops */
ast_t* eval_gt(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_gtlt(env, op_gt, ast1, ast2, &__gt_func);
}

ast_t* eval_lt(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_gtlt(env, op_lt, ast1, ast2, &__lt_func);
}

ast_t* eval_ge(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_gtlt(env, op_ge, ast1, ast2, &__ge_func);
}

ast_t* eval_le(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_gtlt(env, op_ge, ast1, ast2, &__le_func);
}

ast_t* eval_eq(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_eqneq(env, op_eq, ast1, ast2);
}

ast_t* eval_neq(env_t* env, ast_t* ast1, ast_t* ast2) {
  return eval_eqneq(env, op_neq, ast1, ast2);
}

/* string ops */
ast_t* eval_cat(env_t* env, ast_t* ast1, ast_t* ast2) {
  ast_t* result = NULL;
  if(ast1->type == at_string && ast2->type == at_string) {
    result = create_string(NULL);
    result->data.s = (char*)check_malloc((strlen(ast1->data.s) + strlen(ast2-
       >data.s) + 1) * sizeof(char));
    strcpy(result->data.s, ast1->data.s);
    strcat(result->data.s, ast2->data.s);
  } else {
    error_apply(NULL,get_op_str(op_cat), get_ast_type_name(ast1->type),
       get_ast_type_name(ast2->type));
  }
  return result;
}


================================================================================
===  Dateiname: utils.c
```

```
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>

#include "utils.h"

void __error_exit(char* emsg) {
  fprintf(stderr, "%s", emsg);
  exit(1);
}

void* check_malloc(size_t size) {
  void* p;
  p = malloc(size);
  if(p == NULL) {
    __error_exit("memory allocation failed!\n");
  }
  return p;
}

void* check_realloc(void* ptr, size_t size) {
  void* p;
  p = realloc(ptr,size);
  if(p == NULL) {
    __error_exit("memory allocation failed!\n");
  }
  return p;
}
```

```
void free_null(void** p) {
  if((*p) != NULL) {
    free(*p);
    *p = NULL;
  }
}

char is_str_int(const char* str) {
  if((str[0] == '-' || str[0] == '+') || (str[0] >= '0' && str[0] <= '9')) {
    size_t i;
    i = 1;
    while(str[i]) {
      if(str[i] < '0' || str[i] > '9') {
        return 0;
      }
      i++;
    }
  } else {
    return 0;
  }
  return 1;
}

char is_str_dbl(const char* str) {
  if((str[0] == '-' || str[0] == '+') || (str[0] >= '0' && str[0] <= '9')) {
    size_t i;
    char p;
    i = 1;
    p = 0;
    while(str[i]) {
      if(str[i] < '0' || str[i] > '9') {
        if(p == 0 && str[i] == '.') {
          p = 1;
        } else {
          return 0;
        }
      }
      i++;
    }
  } else {
    return 0;
  }
  return 1;
}

char* replace_str(const char* str, const char* old, const char* new) {
  size_t olen;
  size_t nlen;
  size_t slen;
  size_t rlen;
  char* tmp;
  char* result;
  ptrdiff_t doffset;
  ptrdiff_t soffset;
  ptrdiff_t clen;
```

```
  int cnt;
  olen = strlen(old);
  nlen = strlen(new);
  slen = strlen(str);
  /* get count of replace patterns */
  cnt = 0;
  tmp = strstr(str,old);
  while(tmp != NULL) {
    cnt++;
    tmp = strstr((tmp+olen),old);
  }
  /* check if there is nothing to replace */
  if(cnt == 0) {
    result = strdup(str); /* copy the original string to result */
  } else {
    /* create a buffer for the "new" string */
    rlen = slen + cnt * (nlen - olen);
    result = (char*)check_malloc((1 + rlen)  * sizeof(char));
    doffset = 0;
    soffset = 0;
    clen = 0;
    tmp = strstr(str, old);
    while(tmp != NULL) {
      clen = tmp - (str + soffset);
      strncpy(result + doffset, str + soffset, clen);
      doffset += clen;
      soffset += clen + olen;
      strncpy(result + doffset, new, nlen);
      doffset += nlen;
      tmp = strstr(str + soffset, old);
    }
    strncpy(result + doffset, str + soffset, rlen - doffset);
    result[rlen] = '\0';
  }
  return result;
}


================================================================================
===  Dateiname: vm.c
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
```

```
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 ****************************************************************************/

/* lib */
#include <stdlib.h>
#include <stdio.h>

/* own */
#include "ast.h"
#include "ops.h"
#include "env.h"
#include "utils.h"
#include "error.h"

/* self */
#include "vm.h"

ast_t* eval_call(env_t* env, ast_t* ast) {
  ast_t* func = NULL;
  ast_t* result = NULL;
  char* fn = NULL;  /* function name */
  switch(ast->data.call.call_type) {
    case ct_anonymous:
      fn = "<anonymous>";
      func = ast->data.call.function.function;
      break;
    case ct_named:
      fn = ast->data.call.function.id;
      func = get_ast_by_id(env, fn);
      if(func == NULL) {
        error_id(NULL, fn);
      };
      break;
  }
  switch(func->type) {
    case at_function:{
      size_t i;
      if(ast->data.call.callargs->data.callargs.count != func-
       >data.function.params->data.params.count) {
        error_paramcount(NULL, fn, func->data.function.params->data.params.count,
       ast->data.call.callargs->data.callargs.count);
      }
      env_t* inner = create_env();
      inner->parent =  env;
      for(i = 0; i < func->data.function.params->data.params.count; i++) {
        set_ast_to_id(
            inner,
            func->data.function.params->data.params.params[i],
```

```
          eval_expression(env,ast->data.call.callargs-
    >data.callargs.callargs[i])
      );
    }
    /* execute the function */
    exec_statements(inner, func->data.function.statements);
    /* get the result */
    inner->parent = NULL; /* must be NULL, get_ast_by_id() also searches the
     parent environment */
    result = get_ast_by_id(inner, "@");
    free_env(inner);
    break;
  }
  case at_builtin:
    if(ast->data.call.callargs->data.callargs.count != func-
     >data.builtin.paramcount) {
      error_paramcount(NULL, fn, func->data.function.params->data.params.count,
     ast->data.call.callargs->data.callargs.count);
    }
    switch(func->data.builtin.paramcount) {
      case 0:
        result = func->data.builtin.function.builtin_0();
        break;
      case 1: {
        ast_t* p = eval_expression(env,ast->data.call.callargs-
    >data.callargs.callargs[0]);
        result = func->data.builtin.function.builtin_1(p);
        dec_ref(p);
        break;
      }
      case 2: {
        ast_t* p1 = eval_expression(env,ast->data.call.callargs-
    >data.callargs.callargs[0]);
        ast_t* p2 = eval_expression(env,ast->data.call.callargs-
    >data.callargs.callargs[1]);
        result = func->data.builtin.function.builtin_2(p1,p2);
        dec_ref(p1);
        dec_ref(p2);
        break;
      }
      case 3: {
        ast_t* p1 = eval_expression(env, ast->data.call.callargs-
    >data.callargs.callargs[0]);
        ast_t* p2 = eval_expression(env, ast->data.call.callargs-
    >data.callargs.callargs[1]);
        ast_t* p3 = eval_expression(env, ast->data.call.callargs-
    >data.callargs.callargs[2]);
        result = func->data.builtin.function.builtin_3(p1,p2,p3);
        dec_ref(p1);
        dec_ref(p2);
        dec_ref(p3);
        break;
      }
      default:
        printf("\n\n*** HINT TO DEVELOPER ***\nimplement builtincall in
    vm.c\n\n");
        exit(1);
```

```
          /* if you create a builtin function with more parameters then you have
       to add a case here */
          break;
      }
      break;
    default:
      error_expected(NULL, get_ast_type_name(at_function), get_ast_type_name(func-
       >type));
      break;
  }
  return result;
}

ast_t* eval_expression(env_t* env, ast_t* ast) {
  switch(ast->type) {
    /* valid */
    case at_call: return eval_call(env,ast);
    case at_identifier: return get_ast_by_id(env, ast->data.id);
    case at_expression: {
      ast_t* result = NULL;
      ast_t* left = eval_expression(env, ast->data.expression.left);
      ast_t* right = eval_expression(env, ast->data.expression.right);
      inc_ref(left);
      inc_ref(right);
      switch(ast->data.expression.op) {
        case op_add: result = eval_add(env, left, right); break;
        case op_mul: result = eval_mul(env, left, right); break;
        case op_div: result = eval_div(env, left, right); break;
        case op_sub: result = eval_sub(env, left, right); break;
        case op_mod: result = eval_mod(env, left, right); break;
        case op_and: result = eval_and(env, left, right); break;
        case op_or: result = eval_or(env, left, right); break;
        case op_gt: result = eval_gt(env, left, right); break;
        case op_ge: result = eval_ge(env, left, right); break;
        case op_lt: result = eval_lt(env, left, right); break;
        case op_le: result = eval_le(env, left, right); break;
        case op_eq: result = eval_eq(env, left, right); break;
        case op_neq: result = eval_neq(env, left, right); break;
        case op_cat: result = eval_cat(env, left, right); break;
        case op_deref: {
          ast_t* index = eval_expression(env, right);
          if (index->type != at_integer) {
            // TODO: error -> index must be an integer!
          } else {
            switch(left->type) {
              case at_list: result = left->data.list.elements[index->data.i];
            }
          }
        }
      }
      result->ref_count = 0;
      dec_ref(left);
      dec_ref(right);
      return result;
    }
    /* no need to evaluate */
```

```
    case at_integer:
    case at_bool:
    case at_double:
    case at_string:
    case at_function:
    case at_statements:
    case at_list:
      return ast;

    /* invalid */
    case at_assignment:
    case at_callargs:
    case at_conditional:
    case at_dowhile:
    case at_elif:
    case at_if:
    case at_params:
    case at_while:
    case at_builtin:
      error_expected(NULL,"expression",get_ast_type_name(ast->type));
  }
  return NULL; /* this should never happen */
}

int exec_if(env_t* env, ast_t* ast) {
  int result = 0; /* not null when condition was true,
                     null when condition was false */
  ast_t* cond = eval_expression(env, ast->data.if_statement.condition);
  if(cond->type != at_bool) {
    error_expected(NULL,get_ast_type_name(at_bool),get_ast_type_name(cond->type));
  } else {
    if(cond->data.b) {
      exec_statements(env, ast->data.if_statement.statements);
      result = 1;
    }
    dec_ref(cond);
  }
  return result;
}

void exec_conditional(env_t* env, ast_t* ast) {
  /* if */
  if(exec_if(env, ast->data.conditional.if_statement)) {
    return;
  } else {
    if(ast->data.conditional.elif_statements != NULL) {
      size_t i;
      /* elif */
      for(i = 0; i < ast->data.conditional.elif_statements-
       >data.elif_statements.count; i++) {
        if(exec_if(env, ast->data.conditional.elif_statements-
       >data.elif_statements.elif_statements[i])) {
          return;
        }
      }
    }
```

```
    if(ast->data.conditional.else_statement != NULL) {
      /* else */
      exec_statements(env, ast->data.conditional.else_statement);
    }
  }
}

void exec_while(env_t* env, ast_t* ast) {
  ast_t* cond = eval_expression(env, ast->data.while_statement.condition);
  if(cond->type != at_bool) {
    error_expected(NULL,get_ast_type_name(at_bool),get_ast_type_name(cond->type));
  } else {
    while(cond->data.b) {
      exec_statements(env, ast->data.while_statement.statements);
      dec_ref(cond);
      cond = eval_expression(env, ast->data.while_statement.condition);
    }
    dec_ref(cond);
  }
}

void exec_dowhile(env_t* env, ast_t* ast) {
  /* same as exec_while() but with a call of exec_statements() before. */
  exec_statements(env, ast->data.while_statement.statements);
  ast_t* cond = eval_expression(env, ast->data.while_statement.condition);
  if(cond->type != at_bool) {
    error_expected(NULL,get_ast_type_name(at_bool),get_ast_type_name(cond->type));
  } else {
    while(cond->data.b) {
      exec_statements(env, ast->data.while_statement.statements);
      dec_ref(cond);
      cond = eval_expression(env, ast->data.while_statement.condition);
    }
    dec_ref(cond);
  }
}

void exec_statements(env_t* env, ast_t* ast) {
  size_t i;
  for(i = 0; i < ast->data.statements.count; i++) {
    switch(ast->data.statements.statements[i]->type) {
      case at_assignment:
        exec_assignment(env, ast->data.statements.statements[i]);
        break;
      case at_call: {
        ast_t* tmp = eval_call(env, ast->data.statements.statements[i]);
        if(tmp != NULL) {
          if(tmp->ref_count == 0) {
            free_ast(tmp);
          }
        }
        break;
      }
      case at_conditional:
        exec_conditional(env, ast->data.statements.statements[i]);
        break;
```

```
      case at_while:
        exec_while(env, ast->data.statements.statements[i]);
        break;
      case at_dowhile:
        exec_dowhile(env, ast->data.statements.statements[i]);
        break;
      default:
        error_unexpected(NULL,get_ast_type_name(ast-
       >data.statements.statements[i]->type));
        break;
    }
  }
}

void exec_assignment(env_t* env, ast_t* ast) {
  ast_t* right = NULL;
  switch(ast->data.assignment.right->type) {
    case at_function:
      right = ast->data.assignment.right;
      break;
    case at_bool:
    case at_double:
    case at_integer:
    case at_string:
    case at_statements:
      right = ast->data.assignment.right;
      break;
    case at_identifier:
      right = get_ast_by_id(env, ast->data.assignment.right->data.id);
      break;
    case at_expression:
      right = eval_expression(env, ast->data.assignment.right);
      break;
    case at_call:
      right = eval_call(env, ast->data.assignment.right);
      break;
    default:
      break;
  }
  if(right == NULL) {
    error_assign(NULL,"NULL",ast->data.assignment.id);
  }
  set_ast_to_id(env, ast->data.assignment.id, right);
}

void vm_exec(env_t* env, ast_t* ast) {
  if(ast->type == at_statements) {
    exec_statements(env,ast);
  } else {
    // TODO : -----
  }
}


================================================================================
===  Dateiname: lexer.l
```

```
================================================================================
/*******************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *******************************************************************************/

%option noyywrap
%option nounput
%option noinput
%option outfile="lexer.c" header-file="lexer.h"

%{
/* lib */
#include <stdlib.h>
#include <inttypes.h>

/* own */
#include "ast.h"
#include "utils.h"
#include "error.h"

/* created by 'bison' */
#include "parser.h"

void lex_integer(char* value);
void lex_double(char* value);
void lex_string(char* value);
void lex_bool(char* value);
void lex_operator(char* value);
void lex_id(char* value);
void lex_error(void);
void lex_inc_line(void);
%}

%x X_COMMENT
```

```
%%
"//".*                          { /* skip */ }
"/*"                            { BEGIN(X_COMMENT); }
<X_COMMENT>{
"*/"                            { BEGIN(INITIAL); }
[^*\n]+                         { /* skip */ }
"*"                             { /* skip */ }
\n                              { lex_inc_line(); }
}
[ \t]                           { /* skip white */ }
\(                              { return T_LPAREN; }
\)                              { return T_RPAREN; }
\{                              { return T_LBRACKET; }
\}                              { return T_RBRACKET; }
"["                             { return T_LSQBRACKET; }
"]"                             { return T_RSQBRACKET; }
\=                              { return T_ASSIGN; }
"\n"                            { lex_inc_line(); }
";"                             { return T_DELIMITER; }
","                             { return T_COMMA; }
":"                             { return T_COLON; }
"if"                            { return T_IF; }
"elif"                          { return T_ELIF; }
"else"                          { return T_ELSE; }
"while"                         { return T_WHILE; }
"do"                            { return T_DO; }
"true"|"false"                  { lex_bool(yytext); return T_BOOL; }
[a-zA-Z_][a-zA-Z0-9_]*          { lex_id(yytext); return T_ID; }
@                               { lex_id(yytext); return T_AT; }
(0|[1-9][0-9]*)                 { lex_integer(yytext); return T_INTEGER; }
(0|[1-9][0-9]*)\.[0-9]+         { lex_double(yytext); return T_DOUBLE; }
\"[^\"\n]*\"                    { lex_string(yytext); return T_STRING; }
"+"|"-"                         { lex_operator(yytext); return T_ADDOP; }
"*"|"/"|"%"                     { lex_operator(yytext); return T_MULOP; }
">"|"<"|">="|"<="|"=="|"!="     { lex_operator(yytext); return T_CMPOP; }
"&"                             { lex_operator(yytext); return T_ANDOP; }
"|"                             { lex_operator(yytext); return T_OROP; }
"$"                             { lex_operator(yytext); return T_STROP; }
.                               { lex_error(); }
%%

void lex_integer(char* value) {
  yylval.i = strtoimax(value,NULL,10);
}

void lex_double(char* value) {
  yylval.d = atof(value);
}

void lex_string(char* value) {
  yylval.s = replace_str(yytext,"\"",""); // trim the delimiting quotes
}

void lex_bool(char* value) {
  if(strcmp(value,"true") == 0) {
```

```
    yylval.b = 1;
  } else if(strcmp(value,"false") == 0) {
    yylval.b = 0;
  } else {
    position_t pos;
    pos.file = NULL;
    pos.line = yylineno;
    error_unexpected(&pos,value);
  }
}

void lex_operator(char* value) {
#define OP(x)  {yylval.o = (x);}
#define OPB(x) {OP(x); break;}
  if(strlen(value) == 1) {
    switch(value[0]) {
      case '+': OPB(op_add);
      case '-': OPB(op_sub);
      case '*': OPB(op_mul);
      case '/': OPB(op_div);
      case '%': OPB(op_mod);
      case '>': OPB(op_gt);
      case '<': OPB(op_lt);
      case '&': OPB(op_and);
      case '|': OPB(op_or);
      case '$': OPB(op_cat);
      default:{
        position_t p;
        p.file = NULL;
        p.line = yylineno;
        error_unexpected(&p,value);
      }
    }
  } else {
    /* */ if(strcmp(value,"!=") == 0) {
      OP(op_neq);
    }else if(strcmp(value,"==") == 0) {
      OP(op_eq);
    }else if(strcmp(value,">=") == 0) {
      OP(op_ge);
    }else if(strcmp(value,"<=") == 0) {
      OP(op_le);
    } else {
      position_t p;
      p.file = NULL;
      p.line = yylineno;
      error_unexpected(&p,value);
    }
  }
#undef OP
#undef OPB
}

void lex_id(char* value) {
  yylval.id = strdup(value);
}
```

```
void lex_error(void) {
  position_t p;
  p.file = NULL;
  p.line = yylineno;
  error_unexpected(&p,"unknown token");
}

void lex_inc_line(void) {
  yylineno++;
}


================================================================================
===  Dateiname: parser.y
================================================================================
/*****************************************************************************
 *
 * Copyright (C) 2014 - 2016 Lukas Singer
 *
 * This file is part of 'qpl'.
 *
 * 'qpl' is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * 'qpl' is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with 'qpl'.  If not, see <http://www.gnu.org/licenses/>.
 *
 * The name 'qpl' stands for Quick Programming Language and is a working
 * title.  It may changes in the future.
 *
 * Source code can be found under: <https://github.com/linluk/project-qpl/>.
 *
 *****************************************************************************/

%error-verbose
%output "parser.c"
%defines "parser.h"
%{
/* lib */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
/* own */
#include "ast.h"
/* created by 'flex' */
#include "lexer.h"

#define YYDEBUG (1)
```

```
%}

%parse-param { ast_t** ast_dest }

/* this union defines the yyval structure -> the $$, $1..$N values */
%union {
  intmax_t i;        /* integer */
  long double d;     /* double */
  char b;            /* bool */
  char* s;           /* string */
  enum operator_e o; /* operator */
  char* id;          /* identifier */
  struct ast_s* ast; /* abstract syntax tree */
  /* ast_t doesnt work!
  * i have to use "struct ast_s" instead!
  * why? astgen.h is included above!
  */
}

%token T_LBRACKET T_RBRACKET T_LPAREN T_RPAREN T_LSQBRACKET T_RSQBRACKET T_COLON
       T_ASSIGN T_WHILE T_COMMA T_DELIMITER T_DO
%token<id> T_ID T_AT
%token<o> T_ADDOP T_MULOP T_CMPOP T_ANDOP T_OROP T_STROP T_IF T_ELIF T_ELSE
%token<i> T_INTEGER
%token<d> T_DOUBLE
%token<s> T_STRING
%token<b> T_BOOL
%type<ast> statement statements assignment expression term factor conditional
       if_statement elif_statement elif_statements else_statement loop while_loop
       do_while_loop block call callargs non_empty_callargs function params
       non_empty_params list list_values
%type<id> assignleft

%start program

%{
void yyerror(ast_t** ast, const char* const emsg);
%}

%%

program : statements { (*(ast_t**)ast_dest) = $1; }
        ;

statements :  { $$ = NULL; }
           | statements statement { $$ = create_statement($1, $2); }
           ;

statement : assignment T_DELIMITER { $$ = $1; }
          | conditional { $$ = $1; }
          | loop { $$ = $1; }
          | call T_DELIMITER { $$ = $1; }
          ;

assignment : assignleft T_ASSIGN expression { $$ = create_assignment($1, $3); }
           | assignleft T_ASSIGN function { $$ = create_assignment($1, $3); }
```

```
          ;

assignleft : T_ID { $$ = $1; }
           | T_AT { $$ = $1; }
           ;

expression : term T_CMPOP expression { $$ = create_expression($2, $1, $3); }
           | term T_STROP expression { $$ = create_expression($2, $1, $3); }
           | term T_OROP expression { $$ = create_expression($2, $1, $3); }
           | term T_ADDOP expression { $$ = create_expression($2, $1, $3); }
           | term { $$ = $1; }
           /*| list { $$ = $1; }*/
           ;

term : term T_ANDOP factor { $$ = create_expression($2, $1, $3); }
     | term T_MULOP factor { $$ = create_expression($2, $1, $3); }
     | factor { $$ = $1; }
     ;

factor : T_INTEGER { $$ = create_integer($1); }
       | T_DOUBLE { $$ = create_double($1); }
       | T_BOOL { $$ = create_bool($1); }
       | T_STRING { $$ = create_string($1); }
       | T_ID { $$ = create_identifier($1); }
       | call { $$ = $1; }
       | T_LPAREN expression T_RPAREN { $$ = $2; }
       | list { $$ = $1; }
       | factor T_LSQBRACKET expression T_RSQBRACKET { $$ =
      create_expression(op_deref, $1, $3); }
       ;

conditional : if_statement elif_statements else_statement { $$ =
       create_conditional($1, $2, $3); }
            ;

if_statement : T_IF T_LPAREN expression T_RPAREN block { $$ = create_if($3, $5); }
             ;

elif_statement : T_ELIF T_LPAREN expression T_RPAREN block { $$ = create_if($3,
       $5);/* create_if() here and combine them to elif_statement_S_ */ }
               ;

elif_statements :  { $$ = NULL; }
                | elif_statements elif_statement { $$ = create_elif($1, $2); }
                ;

else_statement :  { $$ = NULL; }
               | T_ELSE block { $$ = $2; }
               ;

loop : while_loop { $$ = $1; }
     | do_while_loop { $$ = $1; }
     ;

while_loop : T_WHILE T_LPAREN expression T_RPAREN block { $$ = create_while($3,
       $5); }
```

```
          ;

do_while_loop : T_DO block T_LPAREN expression T_RPAREN { $$ = create_dowhile($4,
       $2); }
              ;

list : T_LSQBRACKET list_values T_RSQBRACKET { $$ = $2; }
     | T_LSQBRACKET T_RSQBRACKET { $$ = NULL; }
     ;

list_values : expression { $$ = create_list(NULL, $1); }
            | list_values T_COMMA expression { $$ = create_list($1, $3); }
            ;

block : T_LBRACKET statements T_RBRACKET { $$ = $2; }
      ;

call : T_ID T_LPAREN callargs T_RPAREN { $$ = create_call($1, NULL, $3); }
     | function T_LPAREN callargs T_RPAREN { $$ = create_call(NULL, $1, $3); }
     ;

callargs : non_empty_callargs { $$ = $1; }
         |  { $$ = NULL; }
         ;

non_empty_callargs : expression { $$ = create_callarg(NULL, $1); }
                   | non_empty_callargs T_COMMA  expression { $$ =
       create_callarg($1, $3); }
                   ;

function : T_AT T_LPAREN params T_RPAREN block { $$ = create_function($3, $5); }
         ;

params : non_empty_params { $$ = $1; }
       |  { $$ = NULL; }
       ;

non_empty_params : T_ID { $$ = create_param(NULL, $1); }
                 | non_empty_params T_COMMA T_ID { $$ = create_param($1, $3); }
                 ;

%%

void yyerror(ast_t** ast, const char* const emsg) {
  fprintf(stderr, "parse error (line %d): \"%s\"\n", yylineno, emsg);
  exit(1);
}
```