OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Ostbayerische Technische Hochschule Regensburg
Fakultät für Informatik und Mathematik

Projekt Compiler Construction

Lukas Wolf
Matrikelnummer: 3216992

bei Prof. Dr. Florian Heinz

Datum: 26.05.2023

Inhaltsverzeichnis

1 Programmiersprache: fblocks

In diesem Kapitel wird die Programmiersprache fblocks vorgestellt. Es wird dabei auf Datentypen, Funktionen und besondere Unterschiede im Vergleich zu anderen Sprachen eingegangen. In Listing 1.1 wird ein Programm zur Demonstration der Syntax dargestellt.

```
1       string name;
2       println("Enter name:");
3       name = readln();
4
5       int i;
6       i = 1;
7       <loop:{
8           <if:i > 10:{break;}:>
9           println("Hello " + name + ", Loop " + i);
10          i = i + 1;
11      }:>
```

Listing 1.1: Das Skript zeigt die Struktur der Programmiersprache fblocks. Es wird dargestellt, wie Konzepte wie Typdefinitionen oder Kontrollstrukturen angewandt werden.

1.1 Datentypen und Operationen

Die Programmiersprache bietet die Datentypen **int**, **bool**, **string**, **list** und **variant**.

int ist ein Ganzzahldatentyp, in dem positive sowie negative Zahlen gespeichert werden. Im fblocks-Quellcode werden Zahlen standardmäßig im Dezimalsystem interpretiert. Negative Zahlen können durch ein vorangestelltes Minus "-" dargestellt werden. Es ist zudem möglich Zahlen im binären, oktalen und hexadezimalen System anzugeben. Folgende Syntax wird dabei verwendet: `<BIN:1001>`, `<OCT:475>` und `<HEX:A08C>`.

bool ist ein Typ zur Speicherung eines Wahrheitswertes. Dieser wird mit **true** oder **false** angegeben. Das Ergebnis von Vergleichsoperationen ist häufig ein solcher Wahrheitswert.

Im Typ **string** werden Zeichenketten gespeichert. Eine Zeichenkette wird durch ein Anführungszeichen begonnen und auch durch ein solches beendet: `"Dies ist ein string"`. Innerhalb einer Zeichenkette kann eine neue Zeile mithilfe von `<NEW>` erzeugt werden. Beliebige Zeichen können mithilfe deren ASCII-Codes eingefügt werden: `<\[Code]>`, beispielsweise `<\34>` für ein Anführungszeichen.

Der Typ **list** wird verwendet, um mehrere Werte in einer indexbasierten Liste zu speichern. Dabei können in einer Liste Werte beliebiger Typen gespeichert werden. Dadurch ist auch

die Speicherung von Listen ineinander möglich. Die Liste kann über die internen Funktionen `list_add`, `list_set`, `list_get` bearbeitet werden. Mit `list_size` wird die Größe der Liste ermittelt. Zusätzlich ist es möglich zwei Listen zu einer neuen zu vereinen durch die "+"-Operation. Durch die Operationen "«" beziehungsweise "»" werden die Elemente der rechten Liste in die Liste auf der linken Seite beziehungsweise umgekehrt eingefügt. Bei der Vergleichsoperation "==" werden die Werte zweier Listen auf Gleichheit geprüft.

Der **`variant`**-Typ wird verwendet, um einen Wert eines beliebigen Typens zu speichern. Der tatsächlich zugrundeliegende Typ bleibt dabei erhalten und wird bei der Verwendung, beispielsweise bei Typprüfungen, berücksichtigt.

1.2 Umgang mit Variablen

Variablen, auch IDs genannt, werden auf Stapeln (Stacks) gespeichert. Dabei wird zwischen einem globalen und einem lokalen Stack unterschieden. Standardmäßig werden Variablen auf dem globalen Stack definiert. Erfolgt die Definition innerhalb einer Funktion, einem **`if`**- oder **`loop`**-Block, so wird die Variable auf dem lokalen Stack angelegt. Diese Blöcke stellen darüber hinaus jeweils einen sogenannten *Scope* dar. Wird ein Scope verlassen, so werden die darin definierten Variablen vom Stack entfernt. In unterschiedlichen Funktionsaufrufen kann der gleiche Variablenname mehrmals vergeben werden, ohne dass die Variablen sich gegenseitig beeinträchtigen. Dies ermöglicht beispielsweise den rekursiven Aufruf von Funktionen.

1.3 Implizite Typkonvertierung

Datentypen sind in einer Hierarchie angeordnet. Bei der impliziten Typkonvertierung wird versucht den niederwertigen Typ in den höherwertigen Typ umzuwandeln. Es gilt folgende Hierarchie: **`bool`** < **`int`** < **`list`** < **`string`**. Der Typ **`variant`** wird auf Basis des tatsächlich zugrundeliegenden Typs eingeordnet. Eine Konvertierung ist nicht für alle Typkombinationen vorgesehen. Tritt ein nicht abgedeckter Fall ein, so wird ein Fehler erzeugt.

Bei einer Operation **`string`** + **`int`** wird beispielsweise geprüft, ob die "+"-Operation für die beiden Datentypen registriert ist. Da diese nicht existiert, wird versucht den **`int`**-Wert in einen **`string`** umzuwandeln. Anschließend wird implizit die Operation **`string`** + **`string`** ausgeführt.

1.4 Funktionen und mehrere Rückgabewerte

Funktionen in fblocks bieten die Möglichkeit mehrere Übergabe- und Rückgabeparameter zu verwenden. In Listing 1.2 wird die Definition von Funktionen und deren Aufruf mit ein oder mehreren Rückgabeparametern dargestellt.

```
1       function f: (int, string) -> (bool, string);
```

```
2       f { int i, string text } = { return { i > 5, text + "def" }; };
3       {bool b, string str} = f(1, "abc");
4
5       function g: () -> (string);
6       g = { return "hello"; };
7       str = g();
```

Listing 1.2: In Zeilen 1 und 5 werden Funktionen definiert. Anschließend wird in Zeilen 2 und 6 der auszuführende Funktionsblock zugewiesen. An dieser Stelle werden zudem die Namen der Übergabeparameter definiert. Funktion `f` besitzt mehrere Rückgabeparameter. Diese werden den Variablen in geschweiften Klammern in Zeile 3 zugewiesen. Diese Variablen werden an dieser Stelle zudem definiert. Das Ergebnis von Funktionen mit einem Rückgabeparameter wie Funktion g kann direkt einer Variablen zugewiesen werden. Diese muss vorher definiert worden sein.

1.5 Kommentare

In der Sprache sind ein- und mehrzeilige Kommentare vorgesehen. Inhalte innerhalb eines Kommentars werden für die Ausführung des Programms vernachlässigt. Mehrzeilige Kommentare werden mit `/*` begonnen. Der umgekehrte Operator beendet den Kommentar: `/* Das ist ein Kommentar */`. Einzeilige Kommentare werden mit `//` eingeleitet und mit dem Ende der Zeile terminiert.

2 Interpreter für fblocks

Dieses Kapitel beschreibt die Verwendung und den Aufbau eines Interpreters für die Programmiersprache fblocks. Dabei wird auf zentrale Konstrukte im Quellcode des Interpreters eingegangen. Zur Demonstration der Programmiersprache und des Interpreters im Ordner *scripts* Beispielprogrammen abgelegt. Skripte mit der Kennzeichnung *demo* wurden zur Erstellung von Listings und Grafiken dieser Dokumentation verwendet.

2.1 Verwendung des Interpreters

Der Interpreter *lang* kann mit und ohne Übergabeparameter ausgeführt werden. Ohne Übergabeparameter kann ein fblocks-Skript manuell über das Terminal eingegeben werden. Die Ausführung wird nach Ende der Eingabe (*Strg+D*) gestartet. Alternativ kann eine Skriptdatei als Parameter übergeben werden. Die Interpretation und Ausführung beginnt automatisch.

2.2 GNU flex und bison

Mithilfe des Lexers GNU flex kann das fblocks-Skript in einzelne Token unterteilt werden. Die Grammatik der Programmiersprache wird auf Basis dieser Token definiert. Durch den Parser GNU bison wird die Syntax des auszuführenden Skripts verifiziert. Während der Evaluation wird ein Abstract Syntax Tree aufgebaut.

2.3 Abstract Syntax Tree

Der Abstract Syntax Tree (AST) repräsentiert das eingelesene fblocks-Skript in einer Baumstruktur. Einzelne AST-Knoten repräsentieren dabei beispielsweise Terme, Operatoren, Funktionsaufrufe, Variablen oder Konstanten. Wird ein Syntaxfehler erkannt, so wird an dieser Stelle ein AST-Knoten vom Typ *ERROR* erzeugt. Sobald das Skript vollständig eingelesen ist, wird der AST rekursiv ausgeführt. In Abbildung 2.1 wird ein solcher AST eines fblocks-Skripts dargestellt.

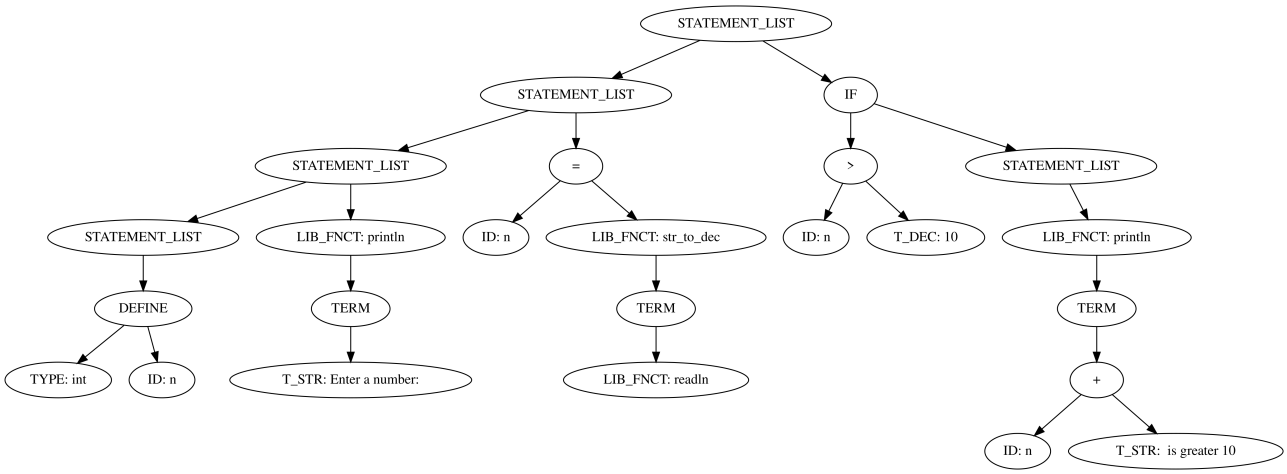

Abbildung 2.1: Der dargestellte Abstract Syntax Tree repräsentiert ein Skript. Im Skript wird ein Anwender aufgefordert eine Zahl einzugeben. Anschließend wird geprüft, ob diese größer als 10 ist, was in diesem Fall ausgegeben wird.

2.3.1 Ergebnisse von AST-Knoten

Zur Kommunikation der Knoten untereinander dient die Struktur `value_t`. Darin wird der Typ eines Ergebnisses und der zugehörige Ergebniswert gespeichert. Das Ergebnis wird an den Elternknoten zurückgegeben, in welchem es weiterverarbeitet wird. Ergebnisse können die Typen der in Abschnitt 1.1 beschriebenen Datentypen annehmen. Funktionsblöcke werden mit dem Typen `type_function` versehen. Darüber hinaus werden die Typen `type_break` und `type_return` zum Kontrollfluss verwendet. Ist für einen Knoten kein Ergebnis vorgesehen, so wird der Typ `type_void` vergeben.

Wird einer Variablen vom Typ **`variant`** ein Wert zugewiesen, so wird dieser in einem Ergebnis vom Typ `type_variant` gekapselt. Wird eine solche Variable in einem Term verwendet und aufgelöst, so wird die Kapselung aufgehoben und der zugrundeliegende Wert zurückgegeben.

Bei der Definition einer Funktion wird ein spezieller AST-Knoten erzeugt. Dieser kapselt Informationen über Typen von Übergabe- und Rückgabeparametern, Namen von Parametern und den auszuführenden Funktionsblock. Ein solcher Knoten ist in Abbildung 2.2 dargestellt. Er ist kein direkter Teil des Abstract Syntax Trees. Stattdessen wird er einer Variablen vom Typ `type_function` als Wert zugewiesen. Die in diesem Knoten gekapselten Informationen werden benötigt, um einen Funktionsaufruf durchzuführen.

2.3.2 AST-Iterator

Für die Navigation im AST wurde ein AST-Iterator implementiert. Dabei wird besonders die Funktion `next_specific_node` angeboten. Über eine Prädikatfunktion kann eine bestimmte Art von Knoten spezifiziert werden, die gesucht werden soll. Es wird so lange über den AST iteriert, bis der nächste Knoten mit den spezifizierten Eigenschaften gefunden wird oder der Baum vollständig durchsucht wurde.

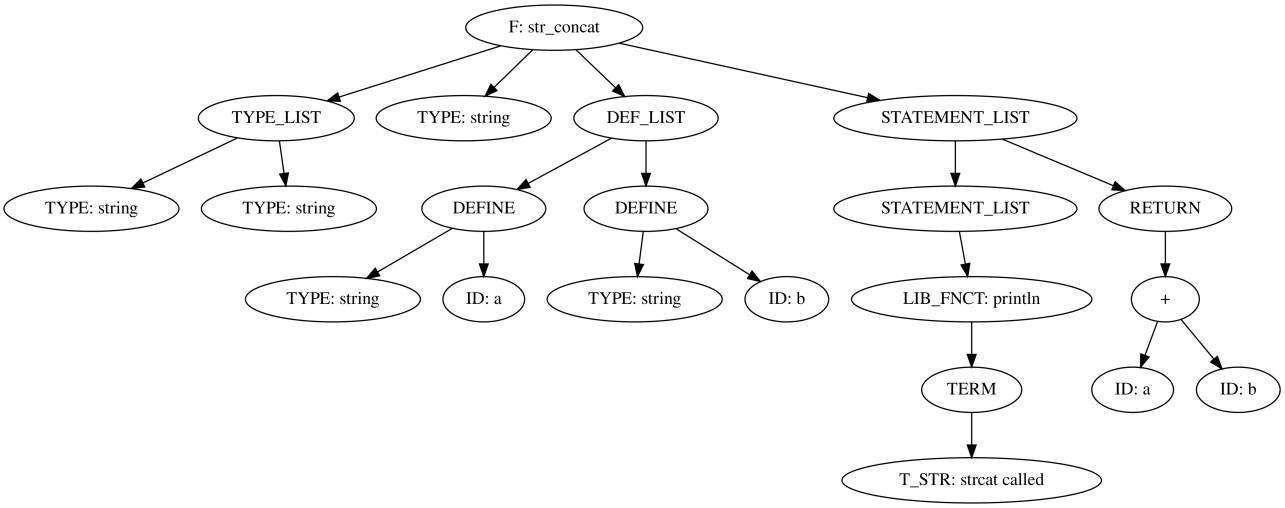

Abbildung 2.2: Der Knoten *F* stellt einen Funktionsknoten für die vom Nutzer definierte Funktion `str_concat` dar. Der erste und zweite Kindknoten beinhalten jeweils eine Typliste für einerseits die zulässigen Typen der Übergabeparameter und andererseits die zulässigen Typen der Rückgabewerte. Im dritten Kindknoten funktionsinternen Namen der Übergabeparameter gespeichert. Im vierten Kindknoten befindet sich der AST zur auszuführenden Funktion.

2.4 Evaluation von Termen

Die Evaluation von Termen wird in einzelne Operationen untergliedert, die durch einen Operator und bis zu zwei Operanden gekennzeichnet sind. Bei einer unären Operation wird dem zweiten Operanden der Typ `type_void` zugewiesen. In einer Tabelle werden alle zulässigen Kombinationen aus Operanden und Operatoren aufgelistet und jeweils einer konkreten Operation zugeordnet. Der Rückgabetyp wird von der Operation bestimmt. Bei der Auswertung eines Terms werden die jeweils benötigten Operationen in der Tabelle gesucht und ausgeführt. Wird eine Operation nicht gefunden, so wird eine implizite Typumwandlung der beiden Operanden durchgeführt. Anschließend wird die Tabelle erneut durchsucht. Kann die Operation weiterhin nicht gefunden werden, ist die Operation in fblocks nicht vorgesehen und es wird ein Fehler erzeugt.

2.5 Native Funktionen der Programmiersprache

Die Sprache fblocks bietet eine Bibliothek von nativen Funktionen. Diese werden vom Interpreter bereitgestellt und können ohne Funktionsdefinition im Skript direkt verwendet werden. Die verfügbaren nativen Funktionen werden in Listing 2.1 aufgelistet.

```
1      function print: (variant) -> ();
2      function println: (variant) -> ();
3      function readln: () -> (string);
4      function list_add: (list, variant) -> ();
5      function list_get: (list, int) -> (variant);
6      function list_set: (list, int, variant) -> ();
7      function list_size: (list) -> (int);
8      function str_to_dec: (string) -> (int);
```

```
9       function str_to_bool: (string) -> (bool);
```

Listing 2.1: Im Listing wird die Signatur der in fblocks verfügbaren nativen Funktionen dargestellt. **variant** als Übergabe- oder Rückgabeparameter bedeutet, dass ein beliebiger Wert übergeben oder zurückgegeben werden kann.

2.6 Fehlerbehandlung

Im Interpreter wird zwischen Syntax- und Laufzeitfehlern unterschieden. Syntaxfehler werden während des Parsens entdeckt. Es wird eine entsprechende Fehlermeldung mit der jeweiligen Zeilennummer ausgegeben. Der Prozess wird fortgesetzt.

Bei der anschließenden Ausführung des ASTs können Laufzeitfehler auftreten. Fehler dieser Art umfassen beispielsweise die Zuweisung eines unstimmigen Wertes zu einer Variable oder den Aufruf einer Funktion mit einer falschen Anzahl an Parametern. Eine Fehlermeldung mit der zugehörigen Zeilennummer und dem ausgeführten AST-Knoten wird ausgegeben. Der Programmablauf wird bei einem solchen Fehler standardmäßig abgebrochen. In der Datei *error.c* kann ein Flag gesetzt werden, um dieses Verhalten zu ändern, wodurch der Programmablauf dennoch fortgesetzt wird. Die Korrektheit des weiteren Ablaufs des Programms ist damit allerdings ungewiss.

2.7 Debugging

Der Interpreter bietet einige Möglichkeiten für das Debuggen eines fblocks-Skripts. Beim Ausführen eines Skripts wird eine visuelle Darstellung des AST erstellt. Dabei wird eine *dot*-Datei erzeugt. Diese kann über das Skript *to_svg.sh* in eine *svg*-Datei umgewandelt werden. In der Datei *debug.c* können Flags gesetzt werden, um die ID und die zugehörige Zeilennummer des Knotens ein- beziehungsweise auszublenden.

Darüber hinaus können in der Datei *debug.c* weitere Einstellungen zur Ausgabe von Debugging-Informationen in der Konsole getroffen werden. Mit der Option `debugging` werden umfangreiche Informationen ausgegeben, die den Verlauf der Ausführung dokumentieren. Durch die Option `debug_tree_node` wird der aktuell ausgeführte AST-Knoten ausgegeben. Damit kann unter anderem die Reihenfolge der Ausführung überprüft werden. Informationen über im Stack angelegte Variablen können mit der Option `debug_stack` angezeigt werden.