C-- Dokumentation

Konzeption der Entwicklung

Zu Beginn der Entwicklung wurden die vorgegebenen Anforderungen gesammelt. Daraufhin wurde sich darauf geeinigt, wie die einzelnen Schlüsselwörter der Sprache heißen sollen, um eine Grammatik definieren zu können. In diesem Zuge wurde auch gleichzeitig ein privates Repository auf Github angelegt, bei dem es 2 Projekte gab. Die Projekte beschreiben die vorgegebenen Anforderungen (Required Features) und die optionalen Anforderungen, die wir selbst noch an unsere Sprache gestellt haben. Diese Projekte sollten bei der späteren Entwicklung genutzt werden, um den Fortschritt an den einzelnen Sprach-Features zu dokumentieren. Dabei gab es für jeden Task noch die Unterpunkte *Implemented* und *Tested*.

Vorgehen während der Entwicklung

Zu Beginn der Entwicklung wurde die gesamte Grammatik definiert und die Features auf diese Grammatik hin zugeschnitten. Dies funktionierte auch bis zu einem gewissen Punkt, an dem wir uns dann allerdings dazu entschieden, Features zu implementieren, für die eine Anpassung der Grammatik notwendig war. Im Zuge dessen entschieden wir uns aus Gründen der Testbarkeit auch von unserem ursprünglichen Plan, der Sprache einen double-ähnlichen Datentyp zu geben, abzuweichen und

stellten die Sprache auf einen einfacheren Datentyp um. Dies war jedoch ein kleinerer Aufwand als angenommen, da die Generierung des Lexers und Parsers durch **Antlr** sehr viel einfacher und reibungsloser von statten ging als angenommen.

So wurden alle vorgegebenen Anforderungen und auch einige der optionalen Anforderungen umgesetzt. Zum Ende des Projektes wurden aus Zeitgründen einige der optionalen Features jedoch nicht mehr realisiert. Nach und nach füllte sich so die Sprache mit Features und mit jedem neu realisierten Feature konnten neue Tests geschrieben werden. Zusätzlich zu diesen Tests schrieben wir auch kleinere Programme in der eigenen Sprache, um den Programmieralltag in der Sprache abschätzen zu können und weitere Notwendigkeiten zu erkennen. Dabei sind besonders die mehreren Compilerfehler die erkannt werden zu nennen aber auch die Standardausgabemethode println() die in der Sprache von Hause aus mitgeliefert wird.

Beschreibung der Sprache

Grundsätzliches

Die C-- Sprache beachtet keinen Leerraum. Für den Compiler ist es unerheblich, ob eine bestimmte Einrückung eingehalten wird, oder ob nach dem Abschluss eines Statements durch `;` ein Linefeed-Zeichen gesetzt wird. Natürlich hilft dies jedoch bei der Strukturierung des Programms und wird von den Entwicklern der Sprache wärmstens empfohlen.
Die Sprache setzt sich aus Statements zusammen, die in der Regel

durch `;` abgeschlossen werden (vgl. C).

Scopes

Ein Programm besteht aus mehreren Scopes. Diese wären `Global`, `Lokal`, `Temporär`. Diese besitzen jeweils unterschiedliche Eigenschaften.

- Global: Funktions-, Variablen- und Konstantendefinitionen erlaubt, Default-scope eines Programms.
- Lokal: Variablen- und Konstantendefinitionen erlaubt. Default Scope innerhalb von Funktionen.
- Temporär: Variablen- und Konstantendefinition erlaubt. Mehrere temporäre Scopes können verschachtelt werden (Wird bspw. durch if-else-Statements und Schleifen aufgespannt). Variablen, die in diesem Scope deklariert werden, haben eine höhere Chance weniger Speicher zu verbrauchen, als andere Variablen (siehe ScopeManager Klasse).

Typen

Die Sprache besitzt nur einen Datentyp (`num`) und zwei Rückgabetypen (`num`, `void`).

- `num` beschreibt einen ganzzahligen Wert, wie er aus der Sprache C bekannt ist. (`int32_t`)
- `void` ist nur gültig als Rückgabetyp und beschreibt, dass die Funktion nichts zurück gibt.

Literale

Literale sind jegliche konstante Zahl die im Code auftaucht. Beispielsweise `13`.

Konstanten

Konstanten sind Werte, die während dem Compileprozess zwischengespeichert werden. Konstanten sind scopeabhängig. Es können nur Konstanten genutzt werden die im gleichen oder in einem höheren Scope definiert wurden. Konstanten werden im späteren Kontext einfach nur substituiert. Konstanten haben keinen Typ, sie speichern nur den String der ihnen zugewiesen wird. Einer Konstanten muss im selben Statement der Wert zugewiesen werden. Da Konstanten aus designtechnischen Gründen erst zur zwischencodegeneration substituiert werden, dürfen nur Literale zugewiesen werden, aber keine Expressions. Konstanten können im globalen scope definiert werden.

- Syntax: `const num <Identifier> = <Wert>;`

Variablen

Variablen stellen veränderbare Speicherbereiche dar, die der Programmierer nutzen kann, um darin Werte zu speichern und diese in weiteren Rechnungen zu benutzen und im Gegensatz zu Konstanten auch überschreiben kann. Die Sichtbarkeit einer Variablen ist von dem Scope abhängig, in dem sie definiert wurde. Wie bei Konstanten können nur Variablen genutzt werden, die im selben Scope oder in einem

höheren deklariert wurden. Variablen müssen erst deklariert werden bevor sie genutzt werden. Variablen können im globalen Scope deklariert werden und sind somit von jeder Funktion veränderbar und zugreifbar. Variablen werden nicht default initialisiert und somit ist das Programm nicht lauffähig, wenn eine uninitialisierte Variable verwendet wird (Dies wird zur laufzeit bestimmt).

- Syntax: `num <Identifier>; <Identifier> = <Wert>`

Standard Arithmetik

Es werden die arithmetischen Operationen `+`, `-`, `*`, `/` für ganzzahlige Datentypen unterstützt. Für die Operation `/` bedeutet dies jedoch, dass keine Rundung vorgenommen wird. Teilt man `5/3` so erhält man nicht, wie vielleicht erwartet, das korrekt gerundete Ergebnis `2`, sondern stets das nach unten abgerundete Ergebnis und damit in diesem Beispiel `1`. Arithmetische Operationen folgen der gewohnten Operatorenpräzedenz, die man aus der Mathematik kennt, sprich von Links nach Rechts und Punkt vor Strich. Natürlich verändert umklammerung der Operationen die Reihenfolge, dies funktioniert auch wie gewohnt.
Operationen müssen Leerzeichen behinhalten d.h. `(1-1)` ist nicht gültig, aber `(1 - 1)`.

Boolsche Algebra

Der Wert `0` entspricht einem Wahrheitswert von Falsch, wohingegen alles andere als Wahr aufgefasst wird. Es werden die meisten

herkömmlichen Boolschen Operationen unterstützt. In folgender Präzedenzreihenfolge.

- `<`, `>`, `<=`, `>=` : Vergleichsoperatoren zweier Werte die in Wahr oder Falsch resultieren.
- `!` : Negiert einen boolschen Ausdruck ((Alles != 0) -> 0, 0 -> 1)
- `==`, `!=` : Prüfung auf Gleichheit und Ungleichheit (funktioniert für Wahrheitswerte und Numerische Werte)
- `&&`, `||` : Boolsche AND- und OR-Verknüpfung.
  Boolsche Operationen müssen keine leerzeichen enthalten d.h. `(0==0)` ist volkommen korrekte syntax, sowie `(0 == 0)`. Dies ist um sie besser von arithmetischen Operationen unterscheiden zu können.

Branch-Statements (If-Else)

Die Sprache C-- unterstützt die Formulierung von Code der nur unter vorherigen Bedingungen ausgeführt wird. Dazu wird ein If-Else-Konstrukt genutzt. Dabei ist es möglich, den Else-Zweig des Statements wegzulassen. Hierbei sind die Wahrheitswerte, die im Abschnitt *Boolsche Algebra* eingeführt wurden, für die `Bedingung` zu beachten.

- Syntax: `if(<Bedingung>) {<Statements>} [else {<Statements>}]`

Loop-Statement (loop)

Das Loop-Statement sorgt dafür, dass ein bestimmter Codeblock so oft ausgeführt wird, bis die formulierte Bedingung den Wahrheitswert 0,

also Falsch, liefert. Die Bedingung wird vor jedem Schleifeneintritt überprüft, sodass es auch möglich ist, dass die Schleife nicht ein einziges Mal durchlaufen wird.

- Syntax: `loop(<Bedingung>) {<Statements>}`

Funktionen

Funktionen bestehen aus Kopf und Körper. Die Rückgabe eines Wertes erfolgt durch ein Return-Statement. Die Parameterliste besteht aus mehreren Deklarationen von Variablen mit Namen und Typ jeweils durch Komma getrennt. Void als Rückgabetyp beschreibt, dass die Funktion keine Daten zurückliefert.

- Kopf: `<Rückgabetyp> <Identifier>(<ParameterListe>)`
- Körper: `{<Statements>}`
- Return-Statement: `return <Ausdruck>;`

Funktionen können aufgerufen werden.

- Syntax: `<Identifier>(<AusdruckListe>)`

Funktionen unterstützen Rekursion.
Der Rückgabetyp einer aufgerufenen Funktion wird aus dem Kontext heraus abgeleitet, um die Programmintegrität zu sichern, kann eine Funktion die einen Wert zurückgibt nur innerhalb einer Expression genutzt werden.

Besondere Funktionen

- Die Sprache besitzt eine eingebaute Ausgabefunktion, die einen Wert auf der Kommandozeile ausgibt. Diese Funktion heißt `println()` und nimmt als Übergabeparameter einen Ausdruck, das zu einem Wert evaluiert werden kann. D.h. Es können z.B. boolsche Berechnungen durchgeführt werden, deren Ergebnisse als `0` oder als `1` auf der Konsole zu sehen sind. Alternativ können auch Literale oder Variablen/Konstanten und allgemein Ausdrücke ausgegeben werden.
- Die Funktion `void main()` stellt eine Besonderheit dar. Wie in C ist sie der Einstiegspunkt des Programms.
- Die Funktion `num get()` scannt stdin nach einem `num` Wert ab und gibt diesen zurück.

Aufgetretene Probleme

Scopes

Wie erkennt man ob eine Variable gültig ist?

Mehrere Variablen mit gleichen Namen sind erlaubt. Unser Ansatz ist es, einen Scopeanager einzuführen, der sobald ein Scope betreten wird oder verlassen wird diese Änderungen erfasst und darauf reagieren kann. Dieser hat mehrere `get` Methoden, denen man nur den Identifier der gefragten Variable übergibt. Dieser Aufruf liefert ein Tupel zurück

das beschreibt, ob es sich um eine Konstante oder Variable handelt, welcher Scope zugrunde liegt und jeweilige Informationen um auf diese Variable/Konstante zuzugreifen.

- Der eigentliche Wert, im Falle einer Konstanten, so dass sie im generierten code nur noch als eingebettete immediate Werte vorkommen.
- Eine Attributreferenz, sofern es sich um eine globale Variable handelt.
- Ein Locals Array Index falls es eine lokale Variable ist.
  Es ist bei mehrfach Deklarationen jedoch nur möglich auf die Variable des innersten zugrunde liegenden Scopes zuzugreifen.

Funktionen

Wie unterscheidet man mehrere Funktionen, bei erlaubtem Overloading?

Ein konkretes Beispiel hierzu wäre: Wie kann mein Compiler erkennen dass `num add(num a, num b)` und `num add(num a, num b, num c)` rund auf verschiedene Funktionen sind?
Unser Ansatz ist es eine Klasse einzuführen, die Metadaten der Funktion speichert. In diesem Fall den Identifier, den Returntyp und die Parameter und deren Typen. Diese Metadaten beschreiben die Signatur der Methode und diese Signatur muss innerhalb eines Programms eindeutig sein.

Der ProgramVisitor hat nun eine Liste von definierten Funktionen, die anfangs leer ist. Sobald eine deklarierte Funktion gefunden wird, wird sie dieser Liste hinzugefügt, sofern nicht bereits eine Funktion mit der gleichen Signatur in dieser Liste vorhanden ist.

Wie kann gewährleistet werden, dass Funktionen die später definiert sind, trotzdem korrekt aufgerufen werden können von Funktionen, die oberhalb definiert sind?

Ein naiver ansatz wäre es, wie von C bekannt Funktionsforwarddeklaration einzuführen. Dies würde jedoch die Sprache relativ verbose gestalten und es sieht nicht so schön aus. Deshalb führen wir einen Validationsschritt ein, der über den generierten Zwischencode läuft (siehe `FunctionCallValidator.java`). Der ProgramVisitor registriert jegliche defnierte Funktion und übersetzt diese. Nun wird diese Liste der registrierten Funktionen direkt an den Validator übergeben und dieser geht Befehl für Befehl durch den Zwischencode und überprüft, ob eine aufgerufene Funktion in oben genannter Liste gefunden wird. Wenn nicht wird ein Error zwischengespeichert. Dieser Validationsschritt erlabut es dem Compiler auch mehrere inkorrekte Funktionsaufrufe zu erkennen und somit mehrere Fehler auszugeben.

Verwendete Technologien

Java 8+

Java Streams erleichtern manche Operationen auf Collection Datenstrukturen erheblich. Diese sind erst seit Java 8 enthalten.
Webseite

ANTLR4

Mächtiges Framework zur Compiler-Frontend Generierung. Nimmt Grammatikdateien als Input und generiert alles bis hin zum AST-visitor. Dieser muss dann implementiert werden um auf bestimmte Kontextknoten im AST zu reagieren und korrekten Code zu generieren.
Webseite

Jasmin

Java Assembler Interface.
Generiert aus Assembly-Code Java `.class` Dateien, welche dann von der JVM ausgeführt werden.
Webseite

Github

Ermöglicht es sehr einfach kollaborativ Code zu entwickeln und gleichzeitig Task-Management zu betreiben.

Webseite

JUnit5

Einfach zu nutzendes Test-Framework, um Code zu testen

Webseite

Maven

Java Build- und Dependency-Tool zum Bauen des Compilers

Webseite

Beispielprogramme

Fakultät Reukrsiv

Code

```
void main(){
    println(fac(4));
}


num fac(num n){
    if(n == 1){
        return 1;
    }
    return n * fac(n - 1);
}
```

Ausgabe

```
24
```

Fibonacci iterativ

Code

```
void main() {

    fib(10);

}


void fib (num n){
    num c;
    num first;
    num second;
    num next;

    first = 0;
    second = 1;
    next = 0;
    c = 0;
    loop(c < n){
```

```
        if (c <= 1) {
            next = c;
        } else {
            next = first + second;
            first = second;
            second = next;
        }
        println(next);
        c = c + 1;
    }
}
```

Ausgabe

```
0
1
1
2
3
5
8
13
21
34
```