Realisierung eines C-Interpreters mit Javascript

Patrick Lukas Starzynski[1]

Abstract:

Im Rahmen diese Arbeit und des Projekt wird ein rudimentärer C-Interpreter zur Ausführung der Programmiersprache C innerhalb der Browserumgebung realisiert. Diese Implementierung basiert konzeptionell auf der fiktiven Programmiersprache Cymbol, die von Terence Parr innerhalb der verfassten Bücher zum Parsergenerator ANTLR [Pa09] [Pa13], erläutert wird. Zur Realisierung wird ein Syntaxbaum, iterativ ver- und bearbeitet, wodurch abschließend ein programmiersprachenunabhängiger Syntaxbaum erzeugt wird, der zur Ausführung der Instruktionen genutzt wird.

Keywords: ANTLR 4; C-Interpreter; Javascript

1 Einführung

Das Interpretieren von Programmiersprachen ist eine Variante zum Ausführen von programmierten Sprachen. Als Beispiele anzuführen sind interpretierte Programmiersprachen wie z.B. Python oder die in dieser Arbeit verwendete Programmiersprache Javascript. Ein Vorteil interpretierter Sprachen ist, dass entsprechende Programme grundsätzlich leichter in unterschiedliche Umgebungen zu portieren sind [Mo11].

Dabei wird, wie bei der Implementierung eines Compilers, ein Lexer zum Lesen und Unterteilen des Programmcodes in entsprechende Token genutzt (siehe Abbildung 1). So können z.B. Schlüsselwörter oder Variablennamen als Token abgebildet werden. Diese Token können im Rahmen der Syntax-Analyse in einer Baumstruktur abgebildet werden. Anschließend wird auf Basis der gewonnenen Informationen ein "Type checking"durch geführt. In dieser Phase kann geprüft werden, ob eine Variable zuvor deklariert oder im jeweiligen Scope bekannt ist. Im Unterschied zum Compiler, der z.B. ausführen Maschinencode erzeugt, wird der Syntaxbaum zur direkten Programmausführung genutzt [Mo11].

[1] FH Bielefeld University of Applied Sciences, Campus Minden, Artilleriestraße 9, 32427 Minden, Deutschland
patrick\protect_lukas.starzynski@fh-bielefeld.de

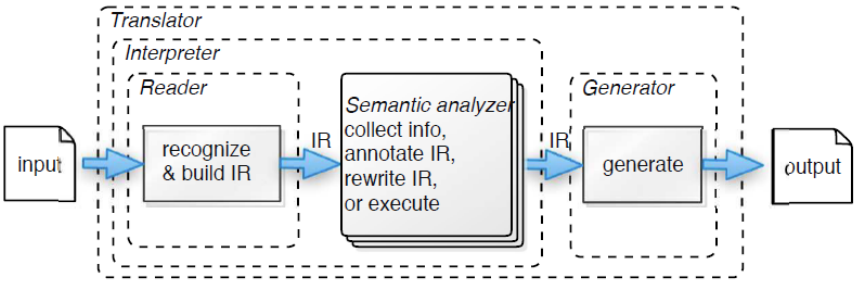

Abb. 1: Phasen des Compilerprogrammierung (Grafik entnommen aus [Pa09])

In dieser Arbeit wird eine Implementierung zur Ausführung von C-Programmcode im Browser vorgestellt, dabei wird erläutert, wie der Prozess vom Parser bis zur Ausführung der Anweisungen, resultierend aus dem Syntaxbaum geschildert. Als Anwendungsfall solch einer Applikation ist die Demonstration entsprechender Funktionalitäten oder als Testareal für sprachspezifische Möglichkeiten. Ein Beispiel für solch ein Werkzeug, ist JSFiddle anzuführen, das (u.a.) diese Funktionen für die Programmiersprache Javascript öffentlich zur Verfügung stellt [JS19].

2 Related Work

In diesem Unterkapitel wird vorgestellt, welche zusätzlichen Werkzeuge und Techniken zur Realisierung des Projektes genutzt wurden.

2.1 Parsergenerator ANTLR v4

ANTLR v4 ist ein Werkzeug zur Generierung von Parsern [Pa13] und dient als Grundlage für dieses Projekt. Ein Vorteil gegenüber anderen Parsergeneratoren ist die Verfügbarkeit innerhalb verschiedenster Programmiersprachen, so z.B. für Javascript. Als Basis zur Generierung des Parser wird eine Grammatik definiert.

```
grammar Hello;
r  : 'hello' ID ;
HELLO: 'hello' ;
ID : [a-z]+ ;
```

In diesem Beispiel wird eine Grammatik *hello* definiert, die einen einfachen Syntaxbaum für z.B. *hello world* erzeugen kann. Dabei werden die Token als reguläre Ausdrücke definiert

und bilden die Blätter des Syntaxbaumes. Die angewandte Regel bildet den (Eltern-) Knoten des Baumes.

Diese definierte Grammatik, ist mit entsprechenden Parametern in konkrete Implementierung für die Zielsprache, z.B. für Javascript, generierbar. Die automatische Generierung umfasst sowohl die Erzeugung eines Lexers, Parsers als auch einem Visitor und Listener die zur Traversierung und ggf. Manipulation des Baumes genutzt werden können [Pa].

Zusätzlich wird eine Sammlung definierter Grammatiken für eine Vielzahl von Programmiersprachen zur Verfügung gestellt. Dies inkludiert die in diesem Projekt genutzte Grammatik für die Programmiersprache C [an19]. Aufgrund der Komplexität des Sachverhalts besteht diese Grammatik, im Gegensatz zum vereinfachten Beispiel, eine tiefe Verschachtelung von Regeln, weshalb die produzierten Teilbäume einen höheren Komplexitätsgrad haben (siehe Abbildung 2).

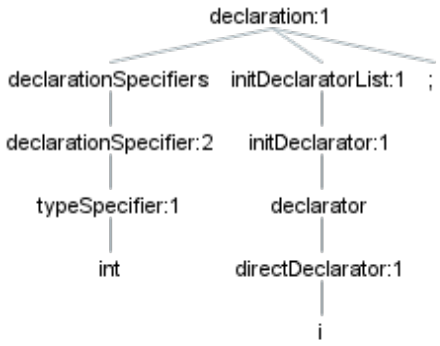

Abb. 2: Erzeugter Teilbaum einer Variablendeklaration

Bei der Traversierung des Syntaxbaumes, bietet ANTLR eine Implementierung des Visitor-Entwurfsmusters als auch einen Listener-Implementierung. Das Visitor-Entwurfsmuster, bietet dabei die Möglichkeit einen entsprechenden Rückgabewert auszuwerten. Dabei wird der Baum mittels Tiefensuche traversiert und folglich jeder Teilbaum besucht. So ist für jeden Knotentyp, definiert durch den Namen der Regel (z.B. *typeSpecifier*), eine überschreibbare Funktion definiert. Während beim Listener für jeden Knotentyp sowohl eine *Enter*, als auch eine *Exit* Funktion implementiert werden kann. Beide Entwurfsmuster lassen sich kombinieren als auch verschachteln, um in Abhängigkeit zur aktuellen Programmphase Aktionen und Manipulationen, anhand der im Syntaxbaum enthalten Informationen, durchführen zu können.

3 C-Interpreter

Die Implementierung im Rahmen dieses Projekt ist in diverse Teilschritte gegliedert und nutzt Funktionalitäten der Programmiersprache Javascript zur Modifikation des generierten Syntaxbaumes. Als Ausgangspunkt für die Implementierung dient ein Programm, zur Ausgabe einer Fibonacci-Folge.

```
int fibonacci(int i) {
        if (i <= 1) { return 1; }
        return fibonacci(i-1) + fibonacci(i-2);
}
int main() {
        int i = 0;
        while (i <= 12) {
                print(fibonacci(i));
                i = i + 1;
        }
        return 0;
}
```

Dieses Programm beinhaltet viele wichtige Bestandteile der zu interpretierenden Sprache. Es wird zwei globale Funktionen definiert, wobei die *main*-Funktion als Startpunkt genutzt wird. Zusätzlich existiert ein rekursiver Funktionsaufruf der *fibonacci*-Funktion, welche einen entsprechenden Funktionsstack erfordert. Darüber hinaus sind Sprunganweisung (*return*-Anweisung) als auch verschiedene Geltungsbereiche für Variablen enthalten.

3.1 Minimierung

Wie in der Abbildung 2 zu sehen ist, existiert bereits bei einer Variablendeklaration eine entsprechend tiefe Baumstruktur. Zur Minimierung von Knoten, mit für die Interpretation, redundanten Informationen. Folglich wird beim ersten Traversieren des Syntaxbaumes, mit einem Visitor, jeder Knoten, der nur einen anderen Knoten, welcher kein Blatt ist entfernt. Resultierend wird der Baum dahingehend modifiziert, dass die zuletzt gültige Regeln inkl. des Tokens erhalten bleibt (siehe Abbildung 4).

Durch diese Minimierung, die Ähnlichkeiten zu einem abstrakten Syntaxbaumes (AST) besitzt, bleiben die spezifischen Eigenschaften der Programmiersprache C, wie z.B. die Definition eines Datentyps, erhalten.

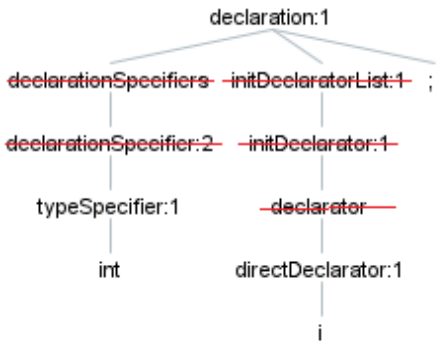

Abb. 3: Erzeugter Teilbaum einer Variablendeklaration

3.2 Symboltabelle

Nach der Vereinfachung des Syntaxbaumes, wird dieser genutzt, um entsprechende Symboltabelle zu erzeugen. Zur Realisierung wurde ein Listener implementiert, wodurch in Abhängigkeit zum Knotentyp ein Wechsel des Geltungsbereichs durchgeführt werden kann. Ebenso, kann eine neue Deklaration von Variablen oder Funktionen dem aktuellen Geltungsbereich hinzugefügt werden.

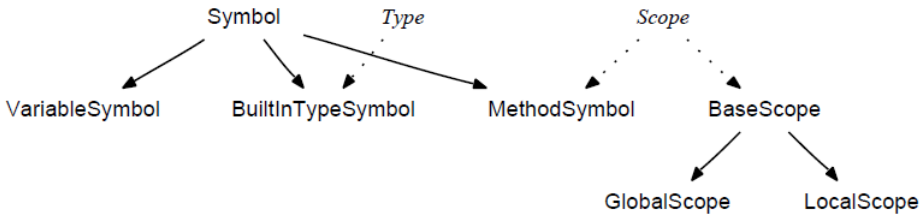

Abb. 4: vereinfachte Darstellung der Symboltabelle (Grafik entnommen aus [Pa09])

Dabei erfolgt eine Unterscheidung zwischen Geltungsbereichen, Symbolen und den Symbolen denen ebenfalls ein Geltungsbereich zugeordnet werden kann.

- Geltungsbereiche:
  - *GlobalScope* - globaler Geltungsbereich
  - *LocalScope* - lokaler Geltungsbereich
- Symbole:
  - *BuiltInTypeSymbol* - Symbole wie z.B. *int* oder *char*
  - *VariableSymbol* - Deklarierte Variablen
- Symbol und Geltungsbereich:
  - *FunctionSymbol* - Paramterliste einer Funktion
  - *StructSymbol* - deklarierter Typ *struct* oder *union*

So wird Starten der Traversierung ein Objekt *GlobalScope* inkl. der Symbole *int*, *char* etc. erstellt. Da die Grammatik für jeden Block die Regel *compoundStatement* nutzt, wird beim *Enter* ein neuer lokaler Geltungsbereich erstellt, welcher den umschließenden Geltungsbereich zugeordnet wird, dies ermöglicht eine spätere Auflösung von definierten Variablem, da zunächst im lokalen Geltungsbereich und anschließend im umschließenden, ggf. globalen Geltungsbereich zu prüfen ob z.B. eine Variable deklariert wurde. Beim Aufruf der *Exit*-Funktion erfolgt ebenso ein Wechsel in den übergeordneten Geltungsbereich.

Zur Erkennung spezieller Symbole wie z.B. Funktions- oder Structdeclarationen werden die Regeln (und die zugehörigen *Ente*r- bzw. *Exit*- Funktionen) genutzt. Jedoch werden diese Geltungsbereiche zusätzlich als Symbole an den übergeordneten Geltungsbereich gebunden.

```
enterStructOrUnionSpecifier(ctx) {
        let type = ctx.structOrUnion().getText();
        let name = ctx.Identifier().getText();
        let struct = new StructSymbol(name, type, this.currentScope);
        this.currentScope.bind(struct);
        this.currentScope = struct;
        ctx.scope = this.currentScope; //Zuordnung Geltungsbereich u. Knoten
}
exitStructOrUnionSpecifier(ctx) {
        this.currentScope = this.currentScope.enclosingScope;
}
```

Diese Verschachtelung erlaubt die Auflösung von Symbolen. Zunächst wird im lokalen Geltungsbereich geprüft ob ein Symbol bekannt ist. Diese Überprüfung erfolgt rekursiv, bis kein weiterer Geltungsbereich vorhanden, oder das Symbol ausgelöst werden kann. Für die Erstellung irrelevant, jedoch zur weiteren Verarbeitung erforderlich, hervorzuheben ist, dass bei der Erstellung eines neuen Geltungsbereichs dieser mit dem korrespondierenden Knoten verknüpft wird.

3.3 Abstrakter Syntaxbaum

Nachdem die Geltungsbereiche und Symbole ermittelt wurde, erfolgt eine erneute Traversierung und abschließende Manipulation des Syntaxbaumes. Diese hat das Ziel die zur Ausführung relevanten Informationen zu extrahieren und einen abstrakten, von der Programmiersprache unabhängigen, Syntaxbaum zu erzeugen.

Dieser abstrakte Syntaxbaum (AST) besteht im wesentlichen aus Operationen und Operanden. Zur Erzeugung wurde ein weiterer Visitor implementiert. Dieser Visitor erstellt für jeden Knoten des Syntaxbaumes, der mit einem Geltungsbereich verknüpft ist, einen AST-Knoten. Anschließend wird der Knoten des Syntaxbaumes mit dem erzeugten AST-Knoten ersetzt. Folglich existieren anschließend die Relationen zwischen dem Geltungsbereich, AST-Knoten und AST-Knoten, Geltungsbereich. Jedem AST-Knoten werden abschließend, sofern existent, Knoten mit entsprechenden Terminalen oder weiteren AST-Knoten zugeordnet. Resultierend bilden Operanden die Blätter eines Teilbaumes, während jeder Elternknoten durch einen Operator abgebildet wird (siehe Abbildung 5).

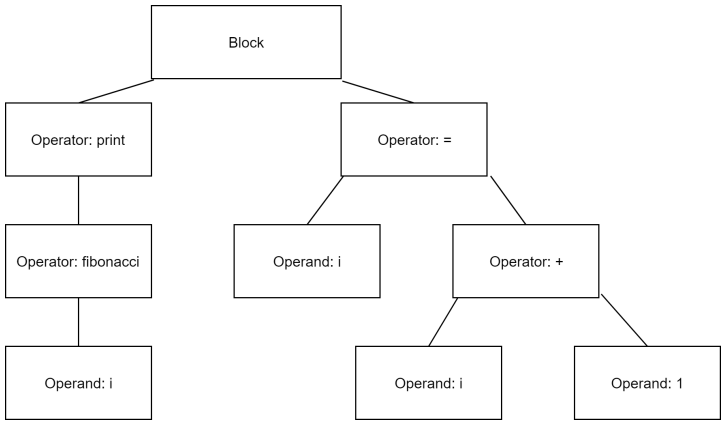

Abb. 5: Abstrakter Syntaxbaum des IF-Blocks aus dem C-Code

Durch die Möglichkeiten von Javascript zur dynamischen Erzeugung von Objekten kann jeder Knoten mit dem entsprechenden Geltungsbereich verknüpft werden. Der erzeugte Syntaxbaum bildet die Grundlage für die Ausführung des C-Codes innerhalb des Dispatchers.

3.4 Dispatcher

Zur Programmausführung wird eine Instanz des Dispatchers erzeugt, dieser erzeugt initial einen virtuellen *MemorySpace*, als auch einen Funktionsstack.

Abschließend wird der AST, welcher mit den zugehörigen Symboltabellen verknüpft ist, innerhalb der *Dispatcher*-Instanz rekursiv verarbeitet. Da im globalen Geltungsbereich, nur die *main*-Funktion ausgeführt werden soll, wird jede Funktionsdeklaration initial übersprungen, folglich werden nur globale Variablen zugewiesen. Anschließend erfolgt der Aufruf der *main*-Funktion. Zur Ausführung wird der AST-Knoten, der dem Funktionssymbol zugewiesen ist, innerhalb einer Switch-Case-Anweisung, geprüft [Pa09].

```
exec(ast) {
        switch(ast.tokentype) {
                case "Block": this.block(ast); break;
                case "Return": this.returnStatement(ast); break;
                case "Assign": this.assign(ast); break;
                ...
                case "Function": return this.call(ast);
        default: throw "Unknown Tokentype " + tokentype;
        }
}
```

Ein Block stellt dabei den Startpunkt zur Ausführung jeder Funktion dar, dazu wird iterativ jeder Kindknoten iterativ erneut an die *exec*-Funktion übergeben und innerhalb der Switch-Case-Anweisung zugeordnet. So erfolgt beim Zuweisen einer Variable ebenfalls ein rekursiver Aufruf der entsprechenden Kindknoten.

```
let left = ast.children[0];
let right = ast.children[1];
let v = this.exec(right);
...
space.put(left.token, v);
```

Sofern kein Fehler, z.B. durch ungültige Zugriffe, bedingt durch Geltungsbereiche auftreten, wird dieser Wert dem aktuellen virtuellen Speicherbereich zugeordnet. Dieser Vorgang ist grundsätzlich identisch für jede Zuweisung und/oder mathematische Operation. Gegensätzlich dazu, muss beim Aufruf einer Funktion ein neuer virtueller Speicherbereich dem Stack hinzugefügt werden und als aktueller Speicherbereich gesetzt werden, in diesem wird der Wert jeder lokal auftretenden Variable gespeichert.

```
call(ast) {
        let fnSymbol = ast.scope.resolve(ast.token);
        if(fnSymbol == null) { throw "Funktion nicht gefunden"; }
        let fspace = new FunctionSpace(fnSymbol);
        let saveSpace = this.currentSpace;
        this.currentSpace = fspace;
                //Parametercheck
        this.stack.push(fspace);
        let result = null;
        try {
                this.exec(fnSymbol.AST.children[0]);
        } catch (e) {
                result = e; this.stack.pop();
                this.currentSpace = saveSpace;
        }
        return result;
}
```

4 Fazit

Schlussendlich ist im Rahmen dieses Projektes ein rudimentärer C-Interpreter realisiert worden, der gegenwärtig nicht alle Funktionen der Programmiersprache verarbeiten kann.

Als ein Beispiel ist der C-Präprozessor zu nennen, dieser ist gegenwärtig, weder durch die Grammatik, als auch durch die Implementierung des Interpreters möglich.

Ebenso ist, für die praktische Nutzung, eine Implementierung für Funktionen der C-Standardbibliotheken sinnvoll, um essentielle Funktionen wie z.B. *malloc* für den Anwender nutzbar zu machen.

Jedoch sind weitere Token, bzw. die resultierenden AST-Knoten, grundsätzlich nur im Dispatcher zu integrieren. Eine Ausnahme stell dabei die Pointerarithmetik der Programmiersprache C dar, diese erfordert einen zusätzlichen Implementierungsaufwand, um die Nutzung von Speicheradresse zu simulieren.

Folglich spiegelt der aktuelle Implementierungsstand nur die grundsätzliche Möglichkeit eines C-Interpreters mittels Javascript wider.

Literaturverzeichnis

[an19] antlr grammars-v4. https://github.com/antlr/grammars-v4/blob/master/c/C.g4.

[JS19] JSFiddle. https://jsfiddle.net/.

[Mo11] Mogensen, Torben: Introduction to Compiler Design. 01 2011.

[Pa] ANTLR v4. https://www.antlr.org/.

[Pa09] Parr, Terence: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf, 1st. Auflage, 2009.

[Pa13] Parr, Terence: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2nd. Auflage, 2013.