

WILHELM BÜCHNER HOCHSCHULE

MASTERTHESIS

**Realisierung eines Source-to-Source Compilers
zwischen Xamarin.Forms und Flutter zur
automatisierten Transformation bestehender mobiler
Anwendungen**

Author:

Julian Pasqué

Betreuer:

Dr. Thomas Kalbe

Verteilte und mobile Anwendungen

Fachbereich Informatik

Matrikelnummer: 902953

11. März 2021

Zusammenfassung

Abstract

Inhaltsverzeichnis

Inhaltsverzeichnis	III
Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Quellcodeverzeichnis	VI
1 Einleitung	1
1.1 Motivation	2
1.2 Ziel der Arbeit	2
1.3 Gliederung	3
2 Compiler	4
2.1 Grundbegriffe	5
2.2 Compiler Struktur	6
2.3 Lexikalische Analyse	7
2.4 Syntaxanalyse	8
2.5 Semantische Analyse	9
2.6 Zwischencodeerzeugung	10
2.7 Codeoptimierung	10
2.8 Codeerzeugung	11
2.9 Der .NET Compiler Roslyn	11
Literaturverzeichnis	12

Abbildungsverzeichnis

2.1	Programmiersprachen als Schnittstelle	4
2.2	Phasen eines Compilers	6
2.3	Lexer Beispiel	7
2.4	Interaktion zwischen Lexer und Parser	8
2.5	Syntaxbaum	9
2.6	Typüberprüfung	9
2.7	Zwischendarstellungen	10

Tabellenverzeichnis

2.1	Token-Beispiele	7
-----	---------------------------	---

Quellcodeverzeichnis

1 Einleitung

Die Entwicklung von verschiedenen mobilen Geräten mit unterschiedlichsten Hardwarekomponenten und Betriebssystemen hat einen stark fragmentierten Markt ergeben.¹ Diese Situation hat einen direkten Einfluss auf die Softwareentwicklung, da die dedizierte Programmierung für die einzelnen Plattformen ressourcenintensiv ist. Durch Realisierung von Web- und hybriden Apps können Softwareprojekte von der darunterliegenden Plattform abstrahieren und plattformübergreifend verwendet werden. Diese Anwendungen haben jedoch, wie schon ausführlich im wissenschaftlichen Diskurs ausgeführt, eine schlechtere Performance und nur begrenzten Zugriff auf die plattformspezifischen Funktionalitäten.²

Durch die Kombination der Vorteile von Web- und hybriden Anwendungen mit denen von nativen konnten Frameworks wie Xamarin.Forms und Flutter Programmierern die Möglichkeit bieten, ihre Anwendungen auf mehreren Plattformen bereit zu stellen. Diese Apps haben neben einer guten Performance auch Zugriff auf sämtliche plattformspezifischen Funktionalitäten. Durch die Abstraktion von Hardware und Betriebssystem können Apps mit einer gemeinsamen Quelltextbasis und somit mit geringerem Ressourcenaufwand entwickelt werden.³

Der Möglichkeit, Ressourcen zu sparen, steht das Risiko der Abhängigkeit gegenüber, da sich die oben genannten Frameworks zur Cross-Plattform-Entwicklung in den verwendeten Programmiersprachen sowie ihrer Arbeitsweise grundlegend unterscheiden. Ein Wechsel zwischen den einzelnen Alternativen ist daher mit enormen Arbeitsaufwänden verbunden.⁴

¹Vgl. Joorabchi 2016, S. 3.

²Vgl. Keist, Benisch und Müller 2016, S. 110ff.

³Vgl. Vollmer 2017, S. 295.

⁴Vgl. Wissel, Liebel und Hans 2017, S. 64.

1.1 Motivation

Im Mai 2020 hat Microsoft mit dem Multi-platform App User Interface (.NET MAUI) einen Nachfolger für das Xamarin.Forms Framework angekündigt, der im Herbst 2021 zusammen mit der sechsten Hauptversion des .NET Frameworks veröffentlicht werden soll. Zum aktuellen Zeitpunkt ist bereits bekannt, dass der Umstieg grundlegende Änderungen mit sich bringt und Anwendungen, die mit Hilfe von Xamarin.Forms entwickelt wurden, angepasst werden müssen.⁵

Für Xamarin.Forms Entwickler wird es also unausweichlich sein, tiefgreifende Modifizierungen an bereits realisierten Anwendungen vorzunehmen, um in der Zukunft von Aktualisierungen zu profitieren. Unternehmen und einzelne Programmierer stehen vor der Entscheidung, ob ein Umstieg auf das leistungsfähige Flutter sinnvoller ist, als die Anpassungen für das neue noch nicht erprobte .NET MAUI, das federführend von einer Firma entwickelt wird, welche leichtfertig mit der Abhängigkeit von Entwicklern umgeht.

Ein automatisierter Umstieg auf das von Google entwickelte Framework Flutter würde also nicht nur die Anpassungen an .NET MAUI vermeiden, sondern die mobile Anwendung auf eine vermeintlich zukunftsichere Basis stellen. Denn obwohl Google in der Vergangenheit schon manche Projekte eingestellt hat, wie zum Beispiel Google Nexus oder Google Hangouts, ist damit bei Flutter aufgrund des Erfolges nicht zu rechnen. Nach offizieller Aussage von Tim Sneath, dem Produkt Manager des Frameworks, haben im Jahr 2020 mehr als zwei Millionen Entwickler Flutter verwendet und über 50.000 mobile Anwendungen programmiert.⁶ Neben der hohen Verbreitung des Frameworks, konnte das Portal Stackoverflow in seinen jährlichen Umfragen auch eine hohe Beliebtheit unter Softwareentwicklern in den Jahren 2019⁷ und 2020⁸ ermitteln. Im März 2021 hat Flutter darüber hinaus die zweite Hauptversion von Flutter veröffentlicht, welche zusätzlich Support für die Entwicklung von Webseiten zur Verfügung stellt.⁹

1.2 Ziel der Arbeit

Im Rahmen dieser Arbeit soll ein Source-To-Source Compiler zwischen den Frameworks Xamarin.Forms und Flutter realisiert werden, mit dessen Hilfe die folgende

⁵Vgl. Hunter 2020, Abgerufen am 28.10.2020.

⁶Vgl. Sneath 2020, Abgerufen am 28.10.2020.

⁷Vgl. Stack Exchange Inc. 2019, Abgerufen am 28.10.2020.

⁸Vgl. Stack Exchange Inc. 2020, Abgerufen am 28.10.2020.

⁹Vgl. Sells 2021, Abgerufen am 28.10.2020.

zentrale Forschungsfrage beantwortet werden soll: "Können Apps komplett automatisiert von Xamarin.Forms zu Flutter übersetzt werden, oder sind manuelle Arbeitsschritte erforderlich?"

1.3 Gliederung

Um diese Forschungsfrage beantworten zu können, wird in Kapitel 2 auf die theoretischen Grundlagen von Software-Übersetzern eingegangen. Anschließend wird in Kapitel ?? auf den Entwurf des in dieser Arbeit zu implementierenden Compiler eingegangen, bevor in Kapitel ?? die Unterschiede zwischen den Frameworks Xamarin.Forms und Flutter behandelt werden. Darauf aufbauend wird in Kapitel ?? der Source-To-Source Compiler realisiert und in dem darauf folgen Kapitel ?? getestet bevor in Kapitel ?? die Forschungsfrage beantwortet wird und ein Fazit gezogen wird.

2 Compiler

Programmiersprachen dienen als Verständigungsmittel zwischen Programmierern und Rechenanlagen wie z.B Smartphones. Diese Sprachen haben sich in der Vergangenheit dabei immer mehr an die Terminologie eines bestimmten Anwendungsgebietes angenähert. Durch diese Entwicklung eigneten sich Programmiersprachen direkt für die Dokumentation von entwickelten Algorithmen und Anwendungen, entfernten sich jedoch weiter von den Gegebenheiten des realen Rechners.¹⁰ Die Beziehung zwischen Softwareentwicklern und Rechenanlagen mit Hilfe von Programmiersprachen werden in Abbildung 2.1 dargestellt.

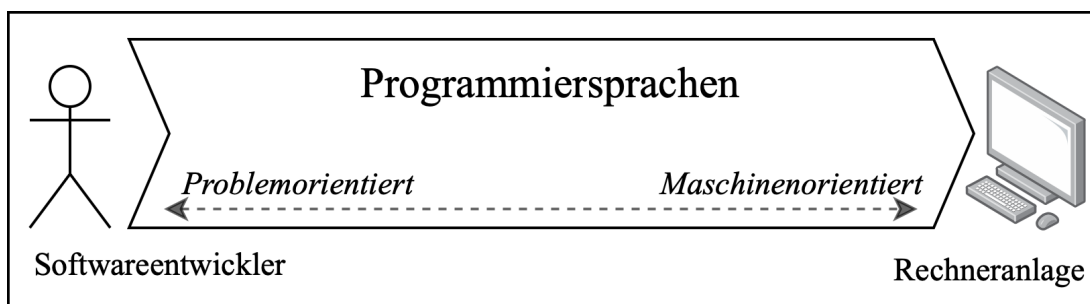


Abbildung 2.1: Programmiersprachen als Schnittstelle

Für die Ausführung einer in einer problemorientierten Programmiersprache geschriebenen Anwendung ist es notwendig, die Sprache in eine maschinenorientierte Form zu überführen.¹¹ Bereits im Jahre 1952 stellte Rutishauser fest, dass Computer in der Lage sind, diesen Übersetzungsvorgang selbst durchzuführen.¹² Durch die Möglichkeit zur automatischen Übersetzung von problemorientierten Programmiersprachen konnten Hochsprachen entwickelt werden, die menschenfreundliche Sprachelemente anstatt Maschineninstruktionen verwenden.¹³

¹⁰Vgl. Schneider 1975, S. 15.

¹¹Vgl. Schneider 1975, S. 15.

¹²Vgl. Rutishauser 1952, S. 312.

¹³Vgl. Wagenknecht und Hielscher 2014, S. 47.

2.1 Grundbegriffe

Diese historische Einführung zeigt, dass Software zur automatisierten Übersetzung schon seit der Mitte des letzten Jahrhunderts thematisiert wurde, so hat sich in der Wissenschaft eine einheitliche Definition ergeben. Ullman et al. beschreibt die sogenannten Compiler im Jahre 2008 wie folgt:¹⁴

Definition 1: Compiler

Ein Compiler ist ein Programm, welches ein anderes Programm aus einer Quellsprache in ein gleichwertiges Programm einer Zielsprache übersetzen kann.

Aus dieser Definition lässt sich ein für diese Arbeit relevanter Fakt ableiten: Compiler sind nicht ausschließlich Übersetzer zwischen problemorientierten und maschinenorientierten Programmiersprachen. Sie sind ausschließlich für die Übersetzung von einer Quellsprache in eine Zielsprache verantwortlich. Auch wenn der Begriff Programm für jedermann geläufig ist, kann es passieren, dass von verschiedenen Repräsentationen gesprochen wird. So können alle drei der folgenden Begriffe als Programm bezeichnet werden: Der Quelltext, das ausführbare Programm und der laufende Prozess auf einem Computer. Für das weitere Verständnis dieser Arbeit ist mit dem Begriff Programm die ausführbare Anwendung auf den Smartphones des Anwenders gemeint.

Neben der Übersetzung von problem- zu maschinenorientierter Sprache gibt es ebenfalls Compiler, die andere Ziele verfolgen. Dazu gehören zum Beispiel die sogenannten Binärübersetzer, die den Binärcode eines Programmes für andere Rechner übersetzen, sodass er auf diesen ausgeführt werden kann.¹⁵ Ein Source-to-Source(S2S) Compiler, häufig auch als "Transpiler" bezeichnet, ist ebenfalls eine besondere Ausprägung eines Compilers, die sich wie folgt definieren lässt.¹⁶

Definition 2: Source-to-Source Compiler

Ein Source-to-Source-Compiler ist ein Compiler, bei dem sowohl die Quellsprache als auch die Zielsprache eine Hochsprache ist.

Der Begriff Hochsprache ist dabei ein Synonym für die bereits eingeführten problemnahen Sprachen wie zum Beispiel C++, Java, C# oder Dart und damit für den Menschen in einer lesbaren und änderbaren Form geschrieben sind.¹⁷

¹⁴Vgl. Ullman et al. 2008, S. 1.

¹⁵Vgl. Ullman et al. 2008, S. 27.

¹⁶Vgl. Rohit, Aditi und Hardikar 2015, S. 1629.

¹⁷Vgl. Eisenecker 2008, S. 9.

2.2 Compiler Struktur

Zur Übersetzung von Programmen bearbeiten Compiler zwei Teilaufgaben, die Analyse und die Synthese. Während der Analyse wird das Programm in seine Bestandteile zerlegt und mit einer grammatikalischen Struktur versehen. Diese wird anschließend verwendet, um eine Zwischendarstellung zu generieren. Dabei wird überprüft, ob das Programm syntaktisch und semantisch fehlerfrei ist oder ob der Programmierer Änderungen vornehmen muss.¹⁸ Der Begriff Syntax beschreibt den Aufbau eines Programmes, sie legt fest wie Sprachelemente aus anderen Sprachelementen zusammengesetzt sind. Im Gegensatz dazu beschreibt die Semantik die Bedeutung der Programmen und regelt die Bedeutung von Sprachelementen.¹⁹ Außerdem werden bei der Analyse Informationen über das Quellprogramm gesammelt und in der so genannten Symboltabelle abgelegt. Die Synthese konstruiert aus der Zwischendarstellung und den Informationen aus der Symboltabelle das gewünschte Zielprogramm. Der Teil des Compilers, der sich mit der Analyse befasst wird oft als Front-End bezeichnet, derjenige der für die Synthese zuständig ist als Back-End.²⁰

Der Vorgang des Kompilierens lässt sich basierend auf diesen zwei Teilaufgaben nach Ullman et al. in mehrere Phasen unterteilen, die in Abbildung 2.2 grafisch dargestellt sind und in diesem Abschnitt detailliert beschrieben werden.²¹

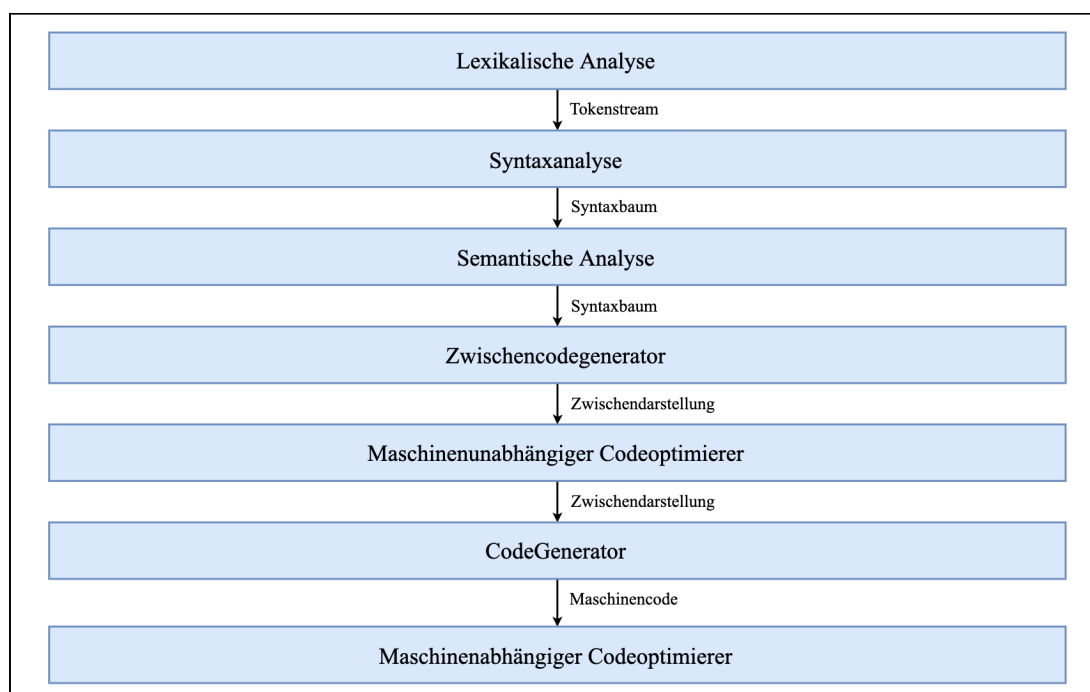


Abbildung 2.2: Phasen eines Compilers²²

¹⁸Vgl. Ullman et al. 2008, S. 6f.

¹⁹Vgl. Schneider 1975, S. 36.

²⁰Vgl. Ullman et al. 2008, S. 6f.

²¹Vgl. Ullman et al. 2008, S. 6.

²²Abbildung in Anlehnung an Ullman et al. 2008, S.6.

2.3 Lexikalische Analyse

Die erste Phase eines Compilers ist die lexikalische Analyse, die den Quelltext in Lexeme untergliedert. Ein Lexem ist die Folge von Zeichen im Quellprogramm, die als Instanz eines Tokens erkannt wurden. Dabei ist ein Token ein Paar aus Namen und einem optionalen Attributwert, wobei der Name zum Beispiel ein bestimmtes Schlüsselwort, oder eine Folge von Eingabezeichen sein kann und der Attributwert auf einen Eintrag in der Symboltabelle verweist.²³ In Tabelle 2.1 werden einige beispielhafte Tokens aufgeführt sowie die Information darüber, aus welchen Lexemen diese extrahiert werden.

Token	Beschreibung	Lexem
if	Zeichen i,f	if
comparison	Vergleichsoperatoren	<=
id	Buchstaben	pi
number	Numerische Konstanten	3.14159

Tabelle 2.1: Token-Beispiele²⁴

Der Teil eines Compilers, der die Lexikalische Analyse durchführt, wird als Lexer bezeichnet. Basierend auf der beschriebenen Arbeitsweise ist in 2.3 ein Beispiel dargestellt, das zeigt wie der Lexer aus einer Zeichenfolge mehrere Tokens mit den optionalen Attributwerten extrahiert.

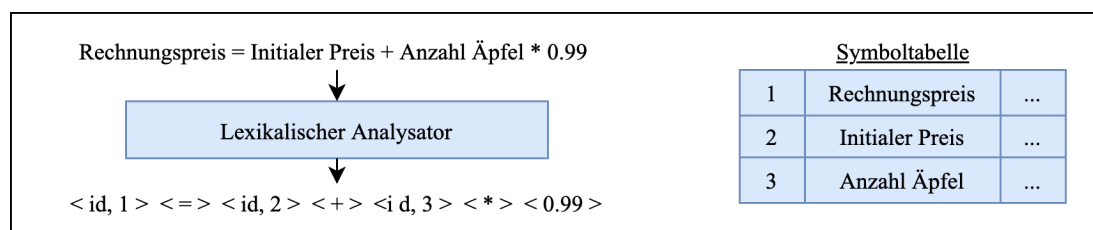


Abbildung 2.3: Lexer Beispiel²⁵

Der Lexer interagiert mit anderen Komponenten eines Compilers, dies wird in Abbildung 2.4 visualisiert. Klassischerweise wird der Lexer über den sogenannten Parser, welcher im nächsten Abschnitt eingeführt wird, zur Übermittlung von Tokens aufgefordert, dies wird in der Abbildung mit dem Aufruf "getNextToken" dargestellt.²⁶

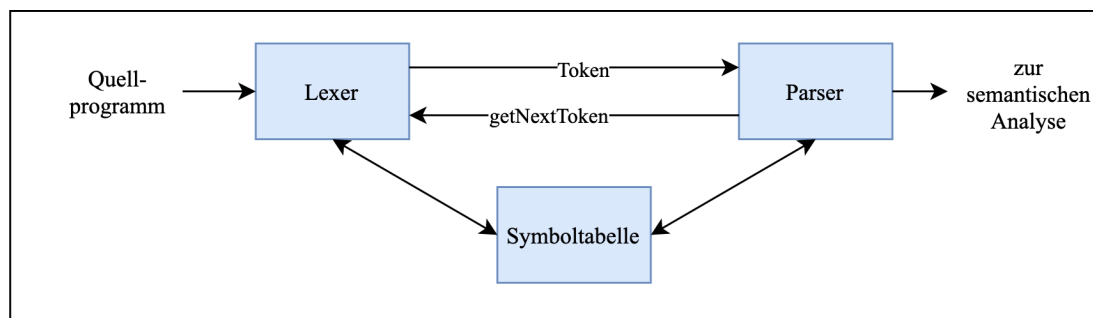
Da der Lexer derjenige Teil des Compilers ist, der den Quelltext liest, kann er neben der Identifikation von Lexemen auch weitere Aufgaben übernehmen. So eignet er sich ideal zum Streichen von Kommentaren im Quelltext und zum Entfernen von Leerstellen, wie

²³Vgl. Ullman et al. 2008, S. 135 f.

²⁴Vgl. Ullman et al. 2008, S. 137.

²⁵Abbildung in Anlehnung an Ullman et al. 2008, S.10.

²⁶Vgl. Ullman et al. 2008, S. 135.

Abbildung 2.4: Interaktion zwischen Lexer und Parser²⁷

Leerzeichen und Tabulatoren. Zudem kann er gefundene Fehler den entsprechenden Zeilennummern zuordnen und dem Entwickler während der Kompilierung so einen genauen Hinweis auf den Ort des Fehlers geben.²⁸ Häufig werden Lexer daher in zwei kaskadierende Prozesse unterteilt, einen für das Löschen von Kommentaren und Zusammenfassung von Leerraumzeichen und einen für die eigentliche lexikalische Analyse.²⁹

2.4 Syntaxanalyse

In der zweiten Phase, der Übersetzung, der Syntaxanalyse, werden durch den bereits erwähnten Parser auch syntaktischer Analysator genannt, die vom Lexer ausgegebenen Tokens in eine baumartige Zwischendarstellung überführt, die die grammatikalische Struktur der Tokens zeigt. Diese Darstellung wird basierend auf ihrem Aussehen häufig als Syntaxbaum bezeichnet. Die Knoten im Syntaxbaum stehen für eine Operation und die Kindknoten für die Argumente dieser Operation. Die Anordnung der Operationen stimmt mit üblichen arithmentischen Konventionen überein, wie zum Beispiel dem Vorrang der Multiplikation vor Addition.³⁰ Abbildung 2.5 zeigt die Erstellung eines Syntaxbaumes aus den Tokens der Abbildung 2.3. Anhand des Knotens <id, 1> ist jederzeit über die Symboltabelle bekannt, dass das Ergebnis der Rechnung an den Speicherort des Bezeichners Rechnungspreis abgelegt werden muss.³¹

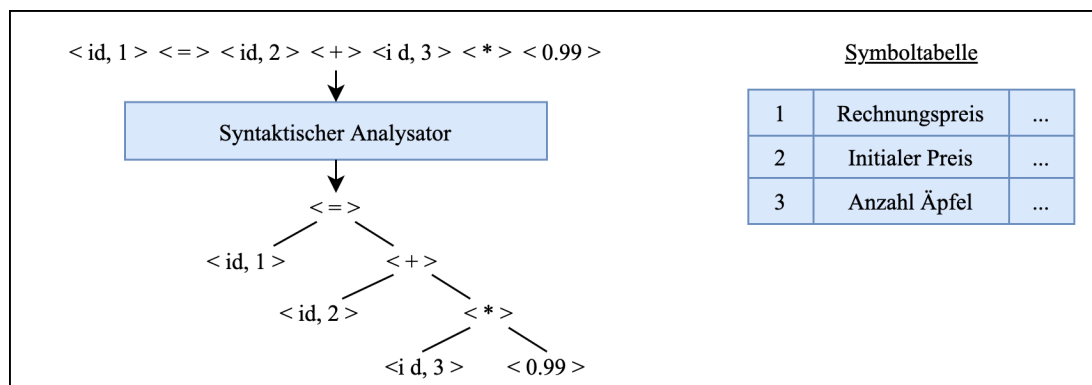
²⁷ Abbildung in Anlehnung an Ullman et al. 2008, S.135.

²⁸ Vgl. Ullman et al. 2008, S. 135.

²⁹ Vgl. Ullman et al. 2008, S. 136.

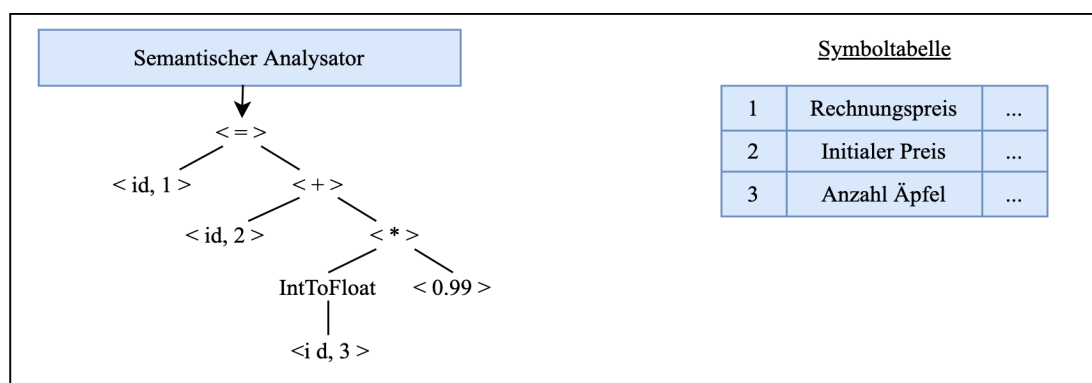
³⁰ Vgl. Ullman et al. 2008, S. 9.

³¹ Vgl. Ullman et al. 2008, S. 9.

Abbildung 2.5: Syntaxbaum³²

2.5 Semantische Analyse

Bei der semantischen Analyse wird der Syntaxbaum als Aufgliederung der Programmstruktur, zusammen mit den Informationen aus der Symboltabelle verwendet, um das Quellprogramm auf semantische Konsistenz mit der Sprachdefinition zu überprüfen.³³ Zudem werden hier Typinformationen gesammelt und zur späteren Verwendung im Syntaxbaum oder der Symboltabelle hinterlegt. Auch findet eine Typüberprüfung statt die analysiert, ob jeder Operator die passenden Operanden hat. So wird beispielsweise validiert, ob ein Index eine Ganzzahl ist. Es besteht die Möglichkeit, innerhalb des Baums Typkonvertierungen zu deponieren. So wurde in dem bisherigen Beispiel die Anzahl Äpfel als Ganzzahl behandelt und wird für die Berechnung des Preises in Abbildung 2.6 zu einer Fließkommazahl konvertiert.³⁴

Abbildung 2.6: Typüberprüfung³⁵

³²Abbildung in Anlehnung an Ullman et al. 2008, S.10.

³³Vgl. Wilhelm, Seidl und Hack 2012, S. 157.

³⁴Vgl. Ullman et al. 2008, S. 9ff.

³⁵Abbildung in Anlehnung an Ullman et al. 2008, S.10.

2.6 Zwischencodeerzeugung

Während der Übersetzung eines Programms kann der Compiler mehrere Zwischendarstellungen in unterschiedlichsten Formen, zum Beispiel wie die eines Syntaxbaums, erstellen. Nach der semantischen Analyse stellen viele Compiler eine maschinennahe Zwischendarstellung auf niedriger Abstraktionsebene her, die eigentlich für maschinenabhängige Aufgaben wie Befehlsauswahl geeignet ist. Eine Zwischendarstellung, die

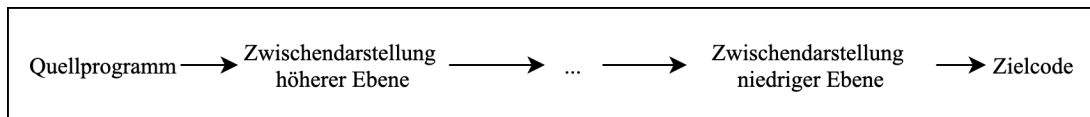


Abbildung 2.7: Zwischendarstellungen³⁶

von Compiler zu Compiler in Auswahl oder Entwurf unterschiedlich ist, kann entweder eine tatsächliche Sprache sein, oder aus internen Datenstrukturen bestehen, die von den Phasen des Compilers gemeinsam verwendet werden. Auch wenn C eine Programmiersprache ist, wird sie häufig als eine Zwischenform verwendet, da sie flexibel ist, zu effizientem Maschinencode kompiliert werden kann und ihre Compiler weitgehend verfügbar sind.³⁷

2.7 Codeoptimierung

In dieser Phase wird der Code so optimiert, dass sich daraus ein besserer, das heißt schnellerer oder ressourcenschonender Zielcode ergibt. Der Umfang der Codeoptimierung schwankt dabei von Compiler zu Compiler erheblich.³⁸ Die Codeoptimierung, die ein Compiler vornimmt, ist im Laufe der Zeit wichtiger und komplexer geworden. Grund für die zunehmende Komplexität sind die immer komplexeren Prozessorarchitekturen, die mehr Gelegenheiten bieten, die Ausführung des Codes zu verbessern. Die gestiegene Bedeutung ergibt sich Beispielsweise aus der steigenden Anzahl an Kernen in modernen Computern und der Möglichkeit, Programme parallel auszuführen.³⁹

³⁶Abbildung in Anlehnung an Ullman et al. 2008, S.433.

³⁷Vgl. Ullman et al. 2008, S. 433.

³⁸Vgl. Ullman et al. 2008, S. 11f.

³⁹Vgl. Ullman et al. 2008, S. 20.

2.8 Codeerzeugung

Die Überführung aus der Zwischendarstellung in die Zielsprache nennt man Codeerzeugung. Hierbei muss die semantische Bedeutung des Quellprogramms erhalten und hochwertig dargestellt sein. Die größte Herausforderung ergibt sich aus der nicht komplett mathematischen Berechenbarkeit aller Prozesse bei der Überführung. Ein Beispiel wäre die Vergabe von Registern, die nicht effizient berechenbar sind. In der Praxis müssen heuristische Techniken ausreichen- die guten, aber nicht unbedingt optimalen Code liefern. Die Codeoptimierungs- und Codeerzeugungsphasen können mehrfach durchlaufen werden, bevor das Zielprogramm finalisiert ist.⁴⁰

2.9 Der .NET Compiler Roslyn

Für die Arbeit mit der Programmiersprache C# steht mit Roslyn ein Compiler zur Verfügung, der sich aus modularen Bibliotheken zusammensetzt. Durch die Referenzierung dieser Bibliotheken können Programme auf den Funktionsumfang von Roslyn zugreifen. So ist es möglich, den Compiler zu verwenden, ohne das Ziel zu haben, die Programmiersprache C# in plattformnahen Code zu übersetzen. Dabei stehen die Bibliotheken über den Paketmanager Nuget für die Einbindung in eigene Projekte zur Verfügung. Um diese Funktionalität zu gewährleisten, unterteilt Roslyn die Übersetzung in mehrere Phasen, welche wiederum einige der in diesem Kapitel beschriebenen Phasen zusammenfassen. Die erste Phase ist die Erstellung des Syntaxbaums, die zweite Phase ist die semantische Analyse gefolgt von der letzten Phase der Ausgabe der so genannten Intermediate Language als Zielsprache.⁴¹

⁴⁰Vgl. Ullman et al. 2008, S. 618f.

⁴¹Vgl. Albahari und Johannsen 2020, S. 1017.

Literaturverzeichnis

Gedruckte Quellen

- Albahari, Joseph und Eric Johannsen (2020). *C# in a Nutshell. The Definitive Reference*. Sebastopol: O'Reilly Media Inc.
- Eisenecker, Ulrich (2008). *C++: der Einstieg in die Programmierung. Strukturiert und prozedural programmieren*. Witten: W3L.
- Joorabchi, Mona Erfani (Apr. 2016). „Mobile App Development: Challenges and Opportunities for Automated Support“. Diss. Vancouver: University of British Columbia.
- Keist, Nikolai-Kevin, Sebastian Benisch und Christian Müller (2016). „Software Engineering für Mobile Anwendungen. Konzepte und betriebliche Einsatzszenarien“. In: *Mobile Anwendungen in Unternehmen*. Hrsg. von Thomas Barton, Christian Müller und Christian Seel, S. 91–120.
- Rohit, Kulkarni, Chavan Aditi und Abhinav Hardikar (2015). „Transpiler and it's Advantages“. In: *International Journal of Computer Science and Information Technologies* 6.2.
- Rutishauser, Heinz (1952). „Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen“. In: *Journal of Applied Mathematics and Physics (ZAMP)* 3, S. 312–313.
- Schneider, Hans-Jürgen (1975). *Compiler. Aufbau und Arbeitsweise*. Berlin: Walter de Gruyter.
- Ullman, Jeffrey D. et al. (2008). *Compiler. Prinzipien, Techniken und Werkzeuge*. 2. Aufl. München: Pearson Studium.
- Vollmer, Guy (2017). *Mobile App Engineering. Eine systematische Einführung - von den Requirements zum Go Live*. 1. Aufl. Heidelberg: dpunkt.
- Wagenknecht, Christian und Michael Hielscher (2014). *Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch für Grundstudium*. 2. Aufl. Wiesbaden: Springer.
- Wilhelm, Reinhard, Helmut Seidl und Sebastian Hack (2012). *Übersetzerbau. Syntaktische und semantische Analyse*. Bd. 2.

Wissel, Andreas, Chrsitian Liebel und Thorsten Hans (2017). „Frameworks und Tools für Cross-Plattform-Programmierung“. In: *iX – Magazin für professionelle Informatik* 2.

Online Quellen

Hunter, Scott (2020). *Introducing .NET Multi-platform App UI*. URL: <https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/> (besucht am 27.02.2021).

Sells, Chris (2021). *What's New in Flutter 2*. URL: <https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65> (besucht am 27.02.2021).

Sneath, Tim (2020). *Flutter Spring 2020 Update*. URL: <https://medium.com/flutter/flutter-spring-2020-update-f723d898d7af> (besucht am 27.02.2021).

Stack Exchange Inc. (2019). *Developer Survey Results 2019*. URL: <https://insights.stackoverflow.com/survey/2019> (besucht am 27.02.2021).

– (2020). *Developer Survey Results 2020*. URL: <https://insights.stackoverflow.com/survey/2020> (besucht am 27.02.2021).

Eidesstattliche Erklärung

Studierender: Julian Pasqué
Matrikelnummer: 902953

Hiermit erkläre ich, dass ich diese Arbeit selbstständig abgefasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

.....
Ort, Abgabedatum

.....
Unterschrift (Vor- und Zuname)

