4

4 Compiler

4.1 Aufbau eines Compilers

Alick E. Glennie, ein Angestellter der Universität Manchester, schrieb 1952 in seiner Freizeit den ersten Compiler für den Manchester Mark I Computer, den er AUTOCODER nannte.

Der erste Compiler für mehrere Plattformen wurde 1954-1957 von einer IBM-Gruppe unter der Führung von John W. Backus erstellt, sein Name ist „formula translator“ (FORTRAN).

Heute existieren ca. 2500 verschiedene Programmiersprachen und für jede dieser Programmiersprachen ist mindestens ein Compiler verfügbar.

4.1.1 Drei Definitionen

Die nachfolgenden Definitionen illustrieren die Herkunft und die Defintion des Begriffs Compiler.

Websters Dictionary: *Compiler – Eine Person die Enzyklopädien kompiliert (oder schreibt). Synonyme sind(engl.): encyclopaedist, encyclopedist.*

Hyperdictionary (www.hyperdictionary.com): *Ein Compiler ist ein Programm, welches ein anderes Programm von einer Source Sprache (oder Programmiersprache) in eine Maschinensprache (Object Code) übersetzt. Bestimmte Compiler erzeugen als Output Assembler, welcher anschliessend durch einen separaten Assembler in Maschinensprache umgewandelt wird.*

Aho, Seti und Ullmann [Aho et al. 1999]: *Einfach gesagt ist ein Compiler ein Programm, welches ein Programm, das in einer bestimmten Sprache geschrieben worden ist (Source Language), einliest und in seine Entsprechung in einer anderen Sprache (Target Language) übersetzt.*

4.1.2 Der Übersetzungsvorgang

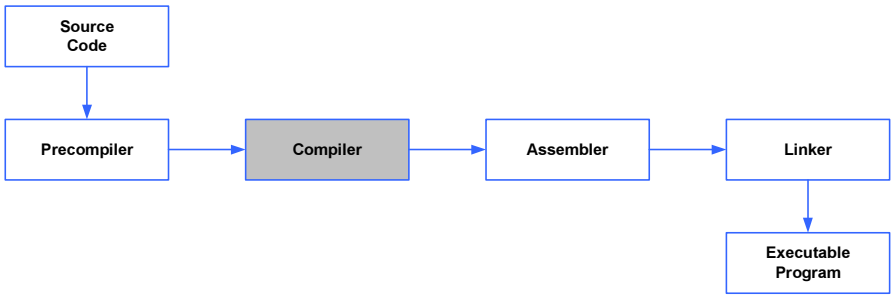

Abbildung 4.1 Der Compiler im Rahmen des Übersetzungsvorgangs

Der eigentliche Übersetzungsvorgang umfasst neben einem Compiler noch eine Reihe anderer Komponenten, wie in **Abbildung 4.1** dargestellt.

- *Precompiler:* Der Precompiler wird in einigen Fällen zur Umwandlung von Macros oder auch z.b. embedded SQL Statements verwendet. Der Präprozessor betrachtet nur die Macros oder die SQL Statements und wandelt sie in eine für den Compiler erkennbare Form um.
- *Assembler:* Ein Assembler verwendet das vom Compiler erzeugte Programm als Input und erzeugt daraus Maschinencode.
- *Loader / Linker:* Der Linker ist für die Einbindung von bereits vorhandenen und kompilierten Bibliotheken, also zur Kombination von Object Code zu einem Executable zuständig.

4.1.3 Die konzeptionelle Struktur eines Compilers

Die Konzeptionelle Struktur eines Compilers besteht immer aus den drei Bestandteilen Frontend, semantische Repräsentation und Backend. Der Compiler liest Source Code ein und produziert ausführbaren Code. Das Frontend ist für die Analyse der Sourcen zuständig, während das Backend für die Erzeugung des ausführbaren Codes verantwortlich ist. Je nach Ausprägung eines Compilers sind die einzelnen Komponenten verschieden aufgebaut. Aus konzeptioneller Sicht werden 11 Schritte durchlaufen, wie Abbildung 4.2 skizziert.

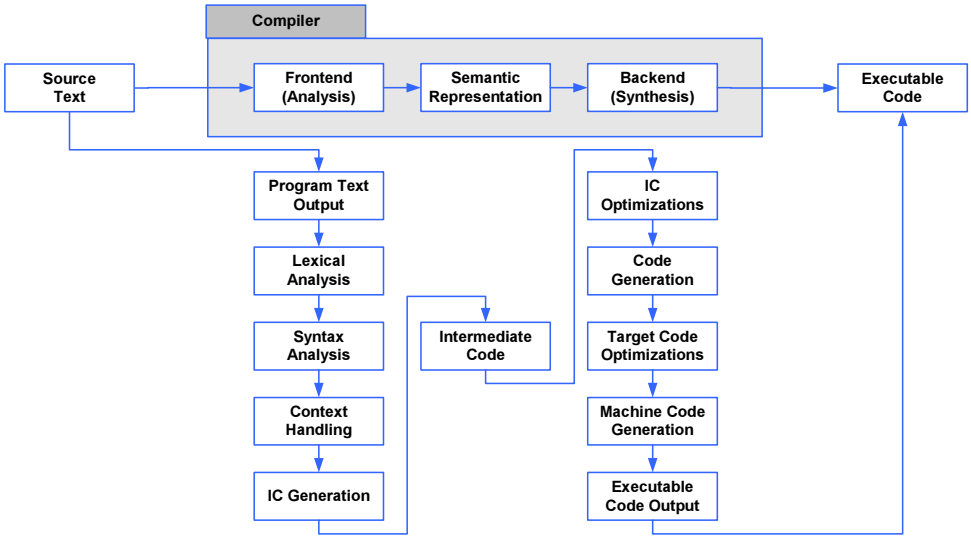

Abbildung 4.2 Konzeptioneller Aufbau eines Compilers nach [Schuldt 2003]

- *Source Text:* In einer bestimmten Programmiersprache geschriebenes Programm, welches übersetzt werden soll.
- *Frontend:* Das Frontend eines Compilers analysiert den Source Text und erzeugt die Semantic Representation des Programmes. Das Frontend ist für die Analyse des Source Codes verantwortlich, d.h. ein Source Text wird auf in Token aufgeteilt und deren Gültigkeit wird geprüft (Lexical Analysis), anschliessend wird die Regelkonformität des Source Texts in Bezug auf die verwendete Sprache geprüft (Syntax Analysis) um schliesslich eine interne Repräsentation (Intermediate Code) zu erzeugen.
- *Semantic Representation:* Eine interne Repräsentation des zu kompilierenden Programmes. Das heisst, der Source Text wird in eine meist baumartige Struktur überführt, die das Regelwerk des auszuführenden Programmes darstellt. Dieses Regelwerk beinhaltet die Bedeutung des Source Codes (Semantik).
- *Backend:* Das Backend überführt die Semantic Representation in ein auf einer bestimmten Plattform ausführbares Programm. Dabei wird auf Ebene des Intermediate Codes und auf Ebene des Target Codes in mehreren Schritten optimiert und anschliessend der ausführbare Code erzeugt. Das Backend wird auch Synthese genannt, da die Analysierte Logik eines Source Texts nun zu einem ausführbaren Programm „zusammensetzt“.
- *Executable Code:* Ausführbares Programm.

4.2 Der Compiler als Teil eines Sprachverarbeitenden Systems

4.2.1 Language Processing

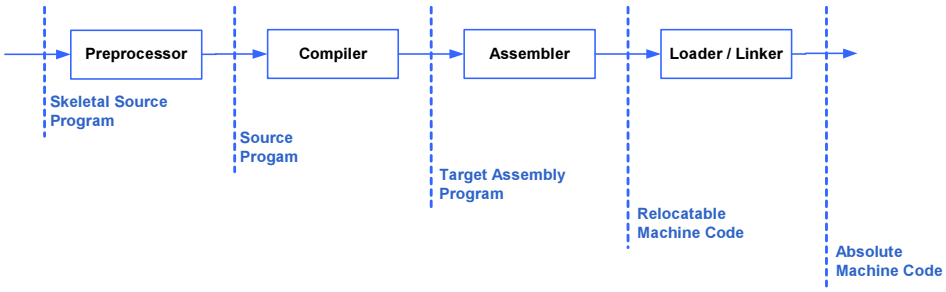

Abbildung 4.3 Language Processing System

Ein Compiler ist integraler Bestandteil eines Sprachverarbeitenden Systems (**Abbildung 4.3**). Folgende zusätzliche Komponenten sind in diesen Systemen anzutreffen.

- *Preprocessor:* Diese Komponente ist für die Vorverarbeitung von Source Code zuständig. Preprocessoren ersetzten Makros (C) oder transformieren Programmiersprachen (C++ in C).
- *Assembler:* Die Transformation des vom Compiler erzeugten Assembler Codes in Maschinen Code erfolgt durch einen Assembler.
- *Loader / Linker:* Der Linker kombiniert ein oder mehrere Dateien, die "relocatable machine code" oder auch "object code" enthalten zu einer einzigen Datei, die ausführbaren Maschinencode enthält. Es wird zwischen „static linking“ und „dynamic linking“ unterschieden. „Static linking“ kombiniert ein oder mehrere Dateien zur kompilierzeit zu einem ausführbaren Code, während „dynamic linkic“ dies erst zur Laufzeit ausführt. In vielen Fällen werden so genannte „shared libraries“ zur Laufzeit gelinkt.

4.2.2 Die Phasen eines Compilers

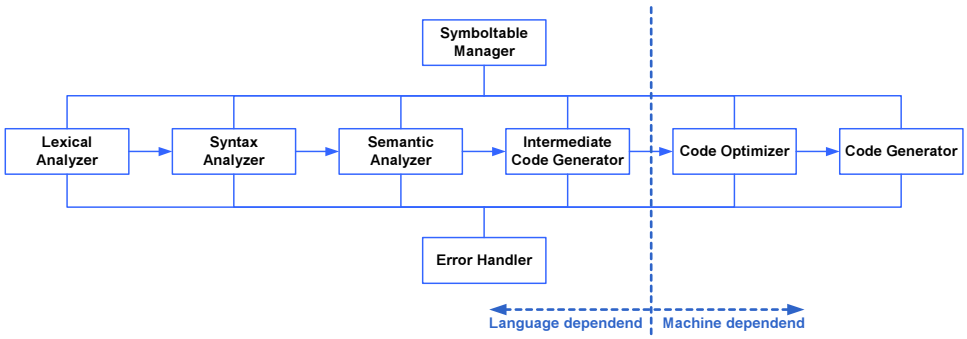

Abbildung 4.4 Phasen eines Compilers

Ein typischer Compiler arbeitet mit 6 Phasen, wie in **Abbildung 4.4** dargestellt. Die Phasen sind:

- *Lexical Analyzer:* Aufschlüsselung des Quellprogramms in einzelne Tokens, Prüfung des verwendeten Zeichenvorrates sowie die Entfernung von irrelevanten Zeichenfolgen.
- *Syntax Analyzer:* Prüfung der strukturellen Korrektheit eines Programms.
- *Semantic Analyzer:* Prüfung der Übersetzbarkeit eines Programms.
- *Intermediate code generator:* Erzeugung eines maschinenunabhängigen Zwischencodes.
- *Code Optimizer:* Optimierung des Zwischencodes aufgrund allgemeiner und Maschinenbedingter Spezifika.
- *Code Generator:* Erzeugung des Codes für die konkrete Maschine.
- *Symboltable Manager:* Verwaltung der Symbol Table, einer Datenstruktur, die sämtliche Identifiers und deren Attribute enthält.
- *Error Handler:* Zentrales Management aller Fehler, die in den anderen Phasen des Compilers auftreten.

4.2.3 Beispiel: Der Intel Composer XE

Um den Aufbau eines Compilers zu illustrieren, hier ein Auszug aus der Produktbeschreibung des Intel C++ Composer XE 2013.

4.2.3.1 Übersicht

Der Intel® Composer XE ist ein Toolpaket, das den neuesten Intel® C/C++ Compiler, Intel® C++ Compiler XE, sowie den neuesten Intel® Fortran Compiler, Intel® Visual Fortran Compiler enthält.

Zusätzlich enthält das Paket die folgenden Leistungs- und Parallelbibliotheken von Intel: Intel® Math Kernel Library (Intel® MKL), Intel® Integrated Performance Primitives (Intel® IPP) und Intel® Threading Building Blocks (Intel® TBB). Intel® Composer XE 2013 ersetzt das verbreitete Intel® Compiler Suite Professional Edition 11.1-Paket. Diese Edition enthält Unterstützung für Intel® Architecture (IA)-32 und Intel® 64 Architekturen für Windows* und Linux* Plattformen.

4.2.3.2 Anwendungen für Multicore Prozessoren

Intel® Composer XE bietet Software-Entwicklern, die C/C++ und Fortran verwenden, leistungsorientierte Funktionen, sodass diese Anwendungen für Hochleistungsumgebungen und Unternehmen auf den neuesten Intel® Architektur-Prozessoren entwickeln können. Dazu zählt auch der neue Intel® Prozessor mit dem Codenamen Sandy Bridge. Die Kombination aus branchenführenden optimierenden Compilern für die Intel® Architektur, darunter Unterstützung für den Branchenstandard OpenMP*, Innovationen wie Intel® Parallel Building Blocks (Intel® PBB) sowie erweiterte Vektorisierung, ermöglicht die einfachere und schnellere Entwicklung von vollständig optimierten Anwendungen. Der Intel Fortran Compiler implementiert Co-Array Fortran als Teil des Fortran 2008-Standards. Die einzelnen Anwendungen unterscheiden sich natürlich, aber in zahlreichen Fällen kann eine einfache Neukompilierung die Leistung um 20 oder mehr Prozent verbessern. Bibliotheken aus optimierten mathematischen Funktionen wie Intel® MKL und Funktionen aus zahlreichen anderen Bereichen wie Komprimierung, Kryptografie und Bildverarbeitung, die in Intel® IPP enthalten sind, stellen außerdem die automatische Parallelisierung sowie Leistung bereit.

Intel® Compiler unterstützen weiterhin den neuesten Standard in der OpenMP-Programmierung. Fortran-Entwicklern steht in Intel Composer XE nun Co-Array Fortran* sowie zusätzliche Unterstützung für den Fortran 2008-Standard zur Verfügung. Außerdem werden durch SIMD-Pragmas und C++ Arraynotierungen optimierte Vektorisierungsfunktionen unterstützt. Intel® PBB stellt einen Satz umfassender Modelle für die Parallelentwicklung dar und unterstützt mehrere Verfahren für Parallelismus in C++. Die Komponenten, die die Modelle enthalten, können auf einfache Weise in vorhandene Anwendungen integriert werden. Dadurch werden Ihre Investitionen in den vorhandenen Code geschützt und die Entwicklung paralleler Anwendungen unterstützt. Die Intel PBB Modelle für die parallele Programmierung stellen mehr Auswahl bereit, um die Anforderungen an die Parallelprogrammierung zu erfüllen, die Unternehmen heute und in Zukunft stellen.

Komponenten von Intel PBB sind:

- *IntelTBB* ist eine C++ Vorlagenbibliothekslösung, die für allgemeinen Parallelismus verwendet werden kann. Die Anwendung enthält skalierbare Speicherzuweisung, Lastenausgleich, eine hoch effiziente Aufgabenplanung, eine Thread-sichere Pipeline und gleichzeitige Container, anspruchsvolle Parallelalgorithmen und zahlreiche Synchronisierungsprimitive.

- *Intel® Cilk Plus* ist eine Implementierung zweier Parallelismustechnologien, die für Intel® C/C++ Compiler spezifisch sind: Intel® Cilk Plus und Arraynotierung. Die *Kombination* bietet überlegene Funktionalität, indem erweiterte Vektorisierungsfunktionen mit Arraynotierung und anspruchsvollem Loop-Datenparallelismus und -Aufgabenparallelismus kombiniert werden.
- *Intel® Array Building Blocks* (Intel® ArBB, Beta, getrennt erhältlich) stellt eine allgemeine Programmierlösung für Parallelanwendungen bereit, sodass Entwickler nicht mehr von *Parallelmechanismen* auf niedriger Ebene oder Hardware-Architekturen abhängig sind. Die Anwendung generiert skalierbare, portierbare und deterministische Parallelimplementierungen aus einer einzigen verwaltbaren und anwendungsorientierten Spezifikation der gewünschten Berechnung.

4.2.3.3 Weitere Leistungsmerkmale

- *High-Performance Parallel Optimizer (HPO)* bietet bessere Möglichkeiten zur Analyse, Optimierung und Parallelisierung von Loop Nests. Diese revolutionäre Funktion verbindet Vektorisierung, Parallelisierung und Loop-Transformation in einem einzigen Durchlauf, der schneller, effektiver und zuverlässiger als herkömmliche diskrete Phasen ist.
- *Automatic Vectorizer* analysiert Schleifen und legt fest, wann einzelne Iterationen der Schleife in sicherer und effektiver Weise parallel ausgeführt werden können. Die Vektorisierung und die automatische Parallelisierung wurden für eine breitere Anwendbarkeit, eine verbesserte Anwendungsleistung und detailliertere Einsichten in den Vektorisierer mittels des unterstützten Merkmals für die automatische Parallelisierung (Guided Auto-Parallelization, GAP) optimiert. Zusätzlich sind nun SIMDProgramme verfügbar, um Anwendern mehr Kontrollmöglichkeiten bereitzustellen.
- *Interprocedural Optimization (IPO)* steigert die Leistung kleiner oder mittelgroßer häufig verwendeter Funktionen erheblich, vor allem von Programmen, die Aufrufe innerhalb von Schleifen enthalten.
- *Loop Profiler* ist Teil des Compilers; er kann für die Erstellung von Low-Overhead-Schleifen und der Funktionsprofilierung verwendet werden, um Hotspots sowie Stellen, an denen Threads eingeführt werden sollen, aufzuzeigen.
- *Profile-Guided Optimization (PGO)* verbessert die Anwendungsleistung durch die Reduzierung des Befehlscache-Thrashing, die Reorganisierung des Codelayouts, die Reduzierung des Codeumfangs sowie die Reduzierung von Fehlvorhersagen für Verzweigungen.
- *OpenMP 3.0* wird unterstützt, um die pragmabasierte Entwicklung von Parallelität in Ihren C/C++-Anwendungen zu erleichtern.

4.2.3.4 C++ für Linux, MacOSX und Windows

Der Intel C++ Compiler existiert in Varianten für Linux, MacOSX und Windows. Obwohl Microsoft den Windows-Markt mit dem Visual C++ Compilersystem beherrscht, bietet Intel mit den gewichtigen Argumenten der besseren Performance, gezielter Unterstützung von parallelisierten Prozessen, Support von sowohl 32-Bit als auch 64-Bit-Systemen sowie der Portabilität zu anderen Betriebssystemen eine Alternative und insbesondere Ergänzung.

Intel stattet dazu seine C++ Compiler mit Optimierern aus, die spezielle Eigenschaften der Intel Prozessor-Architektur nutzen. Schwerpunkt legt Intel hier besonders auf die Möglichkeit der Entwicklung parallelisierter Programme durch Unterstützung der Hyper-Threading Technologie auf Pentium Prozessoren sowie des OpenMP Standards. Darüber hinaus ist der Intel Compiler mit einem Auto-Parallelisierer ausgestattet.

Neben den 32-Bit-Prozessoren werden auch 64-Bit-Prozessoren sowie generell die Intel® Extended Memory 64 Technology (Intel® EM64T) unterstützt (z.B. für Intel® Xeon 64-Bit Prozessoren). D.h., man ist auf entsprechend ausgestatteten Rechnern in der Lage, Programme zu entwickeln, die mehr als 4GB Speicher zu adressieren vermögen (Adressraum: bis zu 1 Terabyte). Zudem stehen 64-bit Integer zur Verfügung.

Ausgestattet ist Intel C++ mit dem Intel® Debugger. Der Intel Debugger (ab v9.0) erlaubt das Debugging auf Rechnern mit Multi-Core Architektur durch gleichzeitige Verfolgung und Überwachung mehrerer Threads.

Die Software steht durchweg auf Intel's WebSite zum Download für einen 30-Tage-Test zur Verfügung und wird durch Kauf nach Bezug einer Lizenzdatei zur unbegrenzten Nutzung freigeschaltet.

4.3 Allgemeiner Ablauf eines compile-runs

4.3.1 Übersicht

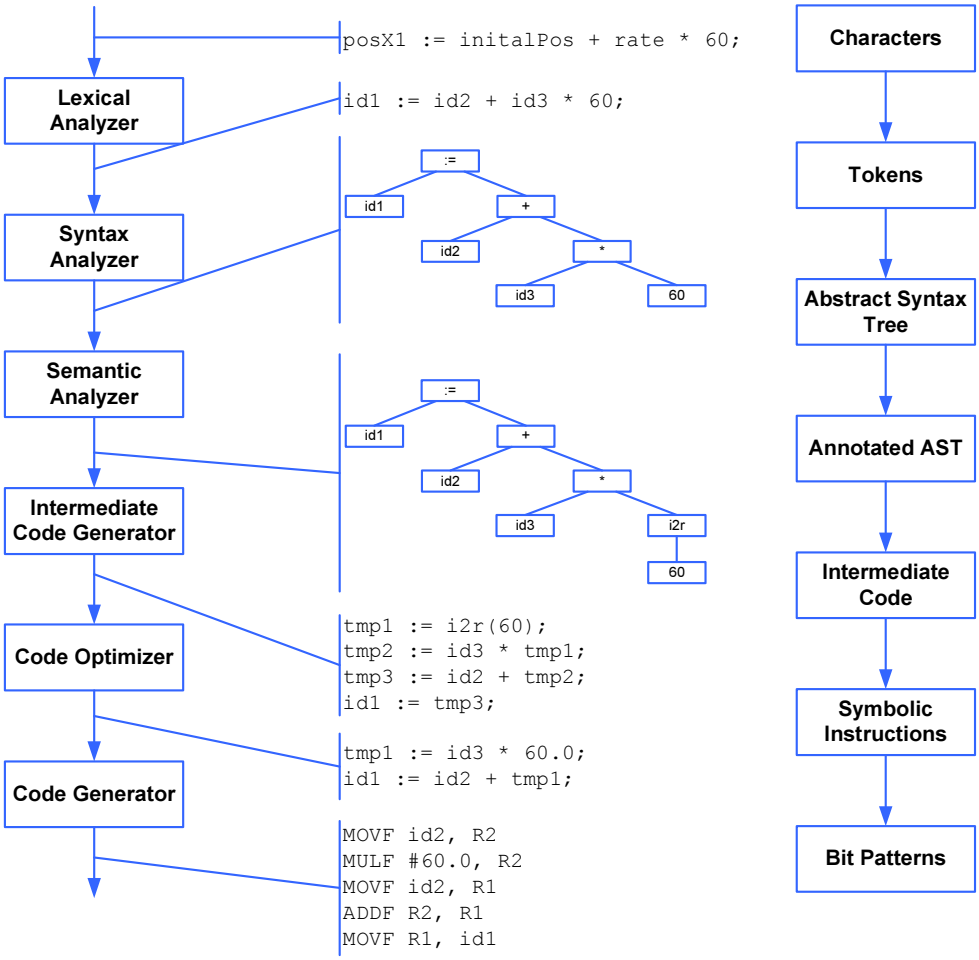

Abbildung 4.5 Umsetzung eines Statements [Aloh et al. 1999]

Ein Compiler setzt immer eine kontextfreie Sprache (Context Free Language) um. Er folgt dabei der Grammatik der entsprechenden Sprache. Ein einzelnes Statement wird gemäss **Abbildung 4.5** schrittweise vom Source Code zu einem ausführbaren Programm umgesetzt.

4.3.2 Lexikalische Analyse

- *Eingabe:* Quellprogramm in Form einer Zeichenreihe
- *Ausgabe:* Programm als Folge von Token (Folge von Zeichen, die bedeutungsmässig zusammengehören), also Identifikatoren, Zahlen, Sondersymbole. Die Symbol Table ist angelegt.

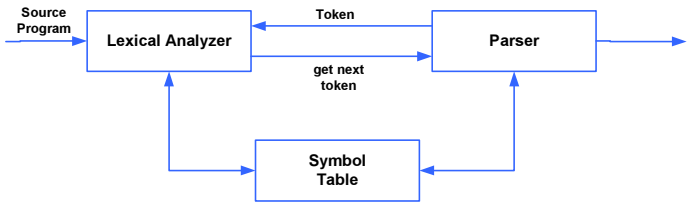

Abbildung 4.6 Interaktion zwischen lexical analyzer und parser

Die Lexikalische Analyse wird mit einem Parser durchgeführt (**Abbildung 4.6**).

Die Trennung der lexikalischen Analyse vom Parser erfolgt aus folgenden Gründen:

- *Separation of Concern:* Die Trennung von Aufgaben, wie der Entfernung von überflüssigen Zeichen oder die Zuweisung von Fehlermeldungen an bestimmte Stellen im Source code würde den Bau eines Parsers erheblich komplizieren.
- *Efficiency and Speed:* Das Einlesen des Source Codes sowie die Aufteilung in Tokens ist aufwendig, ein signifikanter Teil der Zeit eines Compile Runs wird für diese Aufgabe verwendet. Die Trennung erlaubt die Optimierung dieser Aufgabe durch spezielle Buffering Techniken.
- *Portability:* Spezifika des Inputalphabetes, sowie device-abhängige Spezifika können auf den lexical analyzer beschränkt werden.

4.3.2.1 Aufgaben der lexikalischen Analyse:

- Überprüfung, ob alle Zeichen aus dem Zeichenvorrat der Quellsprache sind.
- Erkennung von Grundsymbolen, Token.
- Entfernung von irrelevanten Zeichenfolgen (Kommentare, überflüssige Leerzeichen, Zeilenenden, Tabulatoren, etc).
- Zuweisung von Fehlermeldungen an die richtige Codezeile.

Token sind strukturiert aufgebaut in so genannten Tokenklassen. Mögliche Tokenklassen sind:

- Tok1: Identifikatoren (identifiers)
- Tok2: reservierte Wörter (keywords)
- Tok3: logische Operatoren (logical operators)

- Tok4: arithmetische Operatoren (arithmetic operators)
- Tok5: Konstanten (constants)
- Tok6: Integer-Zahlen
- Tok7: Real-Zahlen
- Tok8: Sonderzeichen
- Tok9: Kommentar

Die einzelnen Elemente jeder Tokenklasse sind durchnumeriert. Beispiel: Tok2 = (1; class, 2; begin, 3;end, 4; if, ...).

4.3.2.2 Attribute von Tokens

Der Lexical Analyzer sammelt Informationen über die analysierten Tokens und speichert diese in der Symbol Table. Diese Attribute beeinflussen die spätere Übersetzung eines Tokens.

4.3.3 Syntaxanalyse

Formale Definition Syntaxanalyse:
Die Syntaxanalyse prüft das Alphabet ∑ der kontextfreien Sprache L(G)

- *Eingabe:* Programm als Folge von Token
- *Ausgabe:* Zeichenreihe der Eingabe und ein zum Programm gehörender abstrakter Baum (Syntax *Tree*, Parse Tree), der die für die Verarbeitung relevante Struktur des Programms beschreibt. Die Knoten entsprechen Operatoren und Operanden. Sofern ein Knoten ein Element aus einer Klasse von Token repräsentiert, ist ihm eine Spezifikation zugefügt.

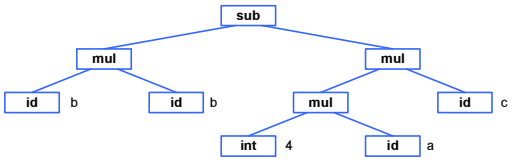

Abbildung 4.7 Vereinfachter Parse-Tree des Ausdrucks: b*b-4*a*c

Die Syntaxanalyse überprüft, ob ein syntaktisch korrektes Programm vorliegt (Überprüfung der strukturellen Korrektheit eines Programmes). Die Syntaxanalyse erfolgt durch einen Parser (Detaillierte Beschreibung siehe Kapitel 3).

4.3.4 Semantische Analyse

Formale Definition semantische Analyse:
Die semantische Analyse prüft die Produktionsregeln R der kontextfreien Sprache L(G)

- *Eingabe:* Zeichenreihe der Eingabe und ein zum Programm gehörender abstrakter Baum (Syntax Tree, Parse Tree).
- *Ausgabe:* Attributierter abstrakter Baum (Attributed Parse Tree) und sowie Informationen für Codegenerierung und Optimierung.

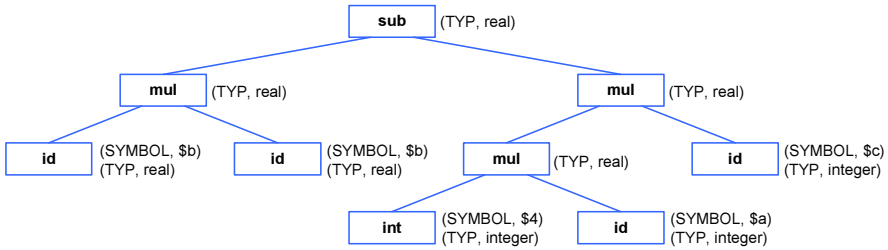

Abbildung 4.8 Attributed Parse-Tree des Ausdrucks: b*b-4*a*c

Die semantische Analyse überprüft, ob ein übersetzbares Programm vorliegt (Überprüfung der statischen Semantik). Es werden ausserdem die Kontextbedingungen überprüft.

Aufgaben der semantischen Analyse:

- Auffinden von fehlenden Deklarationen oder Mehrfachdeklarationen.
- Bestimmung des Typs von komplexen Ausdrücken.
- Typkonformität der linken und rechten Seite einer Zuweisung.
- Überprüfung der Indexgrenzen (bei statischen Grenzen).
- Bei Unterprogrammaufrufen: Überprüfung, ob die Anzahl der Argumente und die Typen der Argumente übereinstimmen.
- Überprüfung des Typs an Stellen, an denen ein boolescher Ausdruck nötig ist.

4.3.4.1 Semantische Regeln

Sämtliche Produktionsregeln einer Programmiersprache werden durch semantische Regeln abgebildet. Diese Regeln werden den Attributwerten (Informationen über Tokens) zugewiesen, um in einer späteren Phase vom Übersetzer angewendet zu werden.

Abbildung 4.9 Konzeptionelle Sicht auf die Anwendung von semantischen Regeln

Die semantischen Regeln werden durch zwei Mechanismen realisiert:

- *Syntax-directed Definitions:* Highlevel Spezifikationen für Übersetzungen (Translations).
- *Translation Schemes:* Definieren die Abfolge der Ausführung der einzelnen Syntax-directed Definitions.

Die semantischen Regeln bestimmen den Aufbau des Parse Tree.

4.3.4.2 Beispiel: Rechner

Die Produktionsregeln R sind:

E → E + T | T

T → T * F | F

F → (E) | digit

Wobei der Token digit eine Zahl zwischen 0 und 9 definiert.

Tabelle 4.1 Die Syntax-directed Definitions des Rechners sind:

Production	Semantic Rules
L →E end	Print(E.val)
E → E1 + T	E.val := E1.val + T.val
E → T	E.val := T.val
T → T1 * F	T.val := T1.val * F.val
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digit	F.val := digit.lexval

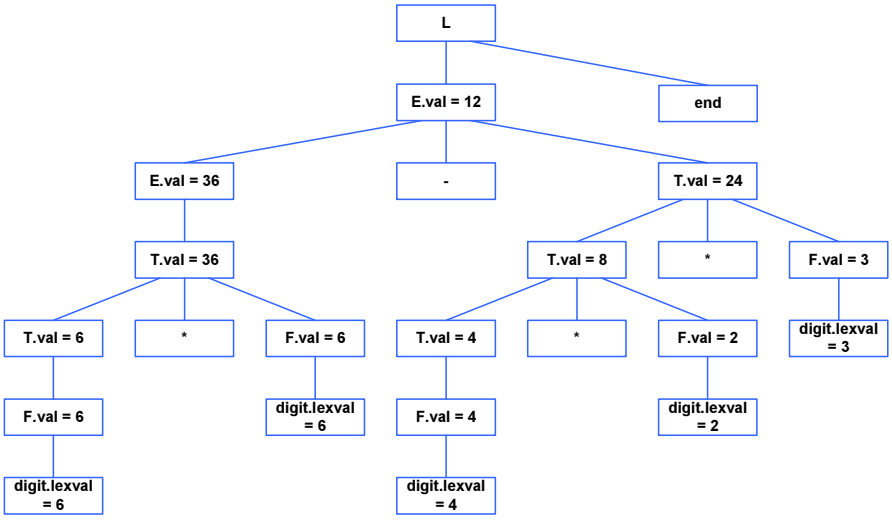

Abbildung 4.10 Annotated Parse Tree des Ausdrucks 6*6-4*2*3 end

4.3.4.3 Semantische Prüfungen

Semantische Prüfungen sind immer statische Prüfungen. Dynamische Prüfungen können nur zur Laufzeit durchgeführt werden.

Typische semantische Prüfungen sind:

- *Type checks:* Die Typenprüfung sichert die Kompatibilität von Zuweisungen. Der Operator und der Operand müssen einen passenden Typ haben.
- *Flow-of-control checks:* In den meisten Fällen bedeuten Flow-of-control checks die Prüfung der Einhaltung der vorgeschrieben Abfolge von Keywords in Konstrukten (Bsp: while do end).
- *Uniqueness checks:* In bestimmten Situation dürfen bestimmte Identifier und Labels nur einmal vorkommen. Dies wird durch uniqueness checks geprüft.
- *Name-related checks:* Die Namensgebung bestimmter Konstrukte muss konsistent sein (Bsp: Block Namen in ADA)

4.3.5 Optimierung

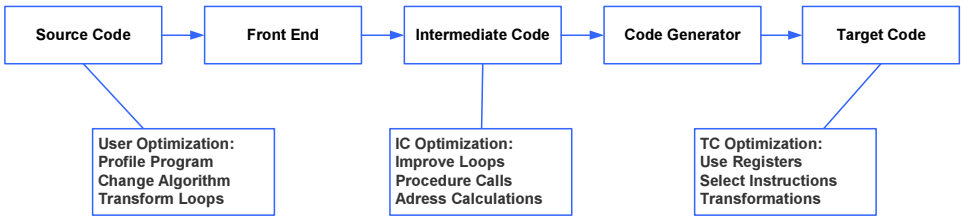

Abbildung 4.11 Potentielle Optimierungen

Die Optimierung von Code ist ein weites Gebiet, dessen Erforschung mit jeder neuen Programmiersprache und jeder neuen Hardware weitergeht [Schneck 1973]. Optimierungen können in drei Bereiche eingeteilt werden, wie in **Abbildung 4.11** dargestellt.

- *User Optimization:* Die Optimierung durch die Programmgestaltung und Architektur ist die wirkungsvollste Verbesserung. Hier können Verbesserungen in der Grössenordnung Faktor 100 bis Faktor 1000 erreicht werden.
- *IC Optimization:* Die Optimierung des Intermediate Code erreicht bestenfalls eine Verbesserung um den Faktor 1-2.
- *TC Optimization:* Die Target Code Optimierung berücksichtigt die Besonderheiten der Zielarchitektur und erreicht bestenfalls eine Verbesserung um den Faktor 2-5.

Zunehmen wichtiger wird die Optimierung einzelner Komponenten durch die Konfiguration der Komponente selbst. So stellen heute die Konfiguration und Optimierung komplexer Infrastrukturen bestehend aus Application Server, DB Server, BPEL Engine u.v.a.m. eine grosse Herausforderung dar. Diese Optimierung wird auch E2E (End to End) Optimization genannt.

4.3.5.1 Intermediate Code Optimierung

Abbildung 4.12 Die drei Schritte der IC Optimierung

Die Intermediate Code Optimierung erfolgt in drei Schritten (**Abbildung 4.12**):

- *Control-Flow Analysis:* Analyse des Kontrollflusses des Programmes.
- *Data-Flow Analysis:* Analyse des Datenflusses.
- *Transformations:* Umsetzung der Optimierungen.

4.3.6 Übersetzung und Codegenerierung

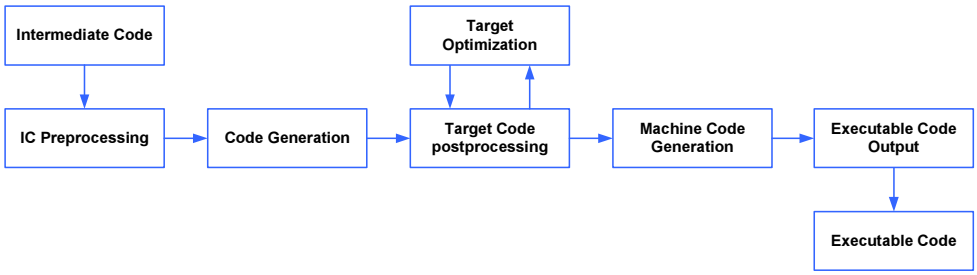

Abbildung 4.13 Ablauf Übersetzung und der Codegenerierung

Die Übersetzung und Codegenerierung kann als Sequenz wie in **Abbildung 4.13** dargestellt betrachtet werden. Die wichtigsten Schritte sind:

- *IC Preprocessing:* Der Intermediate Code wird für die Optimierung vorbereitet. Ein Vorauswahl für die Optimierung wird vorgenommen (Schlaufen, mögliche Inlinefragmente, etc).
- *Code Generation:* Aus dem Intermediate Code wird Assembler erzeugt.
- *Target Optimization:* Diese Komponente steuert die Optimierung des Codes auf eine bestimmte Zielplattform, respektive auf bestimmte Kriterien hin. Diese Kriterien können über Compiler-Switches gesteuert werden.
- *Target Code Postprocessing:* Alle Optimierungen werden auf dem bereits vorhandenen Assembler Code ausgeführt.
- *Maschine Code Generation:* Der Maschinen Code für eine bestimmte Zielplattform wird erzeugt.
- *Executable Code Output:* Der Compiler schreibt das Executable File respektive die Executable Files oder auch Object Files.

4.4 Fragen zum Kapitel

Nr	Frage
1	Welche Arten von Code Optimierung können unterschieden werden und was ist ihre Wirkung?
2	Was ist Data Prefetching und wo wird es verwendet?
3	Was versteht man unter Multi-Threading?
4	Welche Schritte durchläuft ein Compile Run?
5	Was ist eine kontextfreie Sprache?
6	Aus welchen drei Bestandteilen besteht jeder Compiler?
7	Was ist die Aufgabe eines Preprocessors?
8	Wie funktioniert die Lexikalische Analyse?
9	Was ist der Unterschied zwischen der Syntaxanalse und der Semantischen Analyse? Aus welchem Grund sind die beiden Vorgänge getrennt?
10	Was ist der Unterschied zwischen einer Produktionsregel und einer Dependenzregel?

4.5 Übung zum Kapitel

Nachfolgend finden Sie den Artikel „Techniques for Optaining High Performance in Java Programs“ von verschiedenen Autoren.

Erstellen Sie aufgrund des Artikels eine Grafik, die die im Artikel erwähnten Methoden zur Erzeugung von hochperformantem Java Code darstellt. Verwenden Sie eine bauartige Übersicht.