Wiederverwendung von Compiler-Komponenten bei der Konstruktion neuer Programmiersprachen

Peter Knauber
FB Informatik, Universität Kaiserslautern
Postfach 3049, 67663 Kaiserslautern
knauber@informatik.uni-kl.de

1. Einführung

Bei der Entwicklung neuer problemorientierter Programmiersprachen werden als Basis häufig Standardsprachkonstrukte verwendet, die dann um domänenspezifische Konzepte (z. B. Datenein- und ausgabe, Gerätesteuerungen) ergänzt werden. In den meisten imperativen und objektorientierten Sprachen treten z. B. ähnliche Konstrukte für Zuweisungen, bedingte und wiederholte Anweisungen, Unterprogrammaufrufe und Sprünge auf. Bei der Konstruktion einer neuen Sprache ist es daher wünschenswert, soviel existierenden Compilercode wie möglich wiederzuverwenden, um sich auf die neuen Konzepte konzentrieren zu können. Ideal wären "Bausteine", die für jeweils ein Sprachkonstrukt (unabhängig von der konkreten Syntax) die komplette semantische Analyse und die Übersetzung vornehmen, die also über den Gebrauch konventioneller Bibliotheken hinausgehen. Für die Implementierung bieten sich die Klassen der objektorientierten Programmierung an. Damit ist ein neuer Ansatz gefordert, der weg von der traditionellen Phasen-Modellierung eines Compilers (Ablauforientierung) zu einem neuen Modell führt, bei dem die Sprachkonzepte im Mittelpunkt stehen (Datenorientierung).

Mit dem OCC-System (Objectoriented Compiler Construction) kann ein Compiler aus fertigen Komponenten aufgebaut werden. In der objektorientierten Realisierung übernimmt eine Bibliotheksklasse für jeweils ein Sprachkonstrukt sowohl die semantische Analyse als auch die Codegenerierung. Mit dieser Technik lassen sich große Teiles eines Compilers sehr schnell implementieren, da für Standard-Sprachkonstrukte auf fertige und bereits getestete Komponenten zurückgegriffen werden kann. So muß nur der domänenspezifische Teil einer neuen Sprache neu entwickelt werden.

Ziel eines entsprechenden Baukastens kann es nicht sein, fertige Komponenten für jeden erdenklichen Zweck zur Verfügung zu stellen; die domänenspezifischen Konzepte müssen in jedem Fall individuell modelliert werden - unabhängig vom verwendeten Implementierungsansatz. Vielmehr sollen Komponenten zur Realisierung typischer Sprachkonstrukte in einer Bibliothek zur Verfügung stehen, die es ermöglichen, ca. 60-80% eines Compilers (d. h. den Standardteil) sehr schnell und fehlerfrei zu realisieren, die jedoch leicht für spezielle Bedürfnisse erweitert werden können.

2. Traditionelle Compiler-Entwicklung

Üblicherweise werden heute syntaxgesteuerte Compiler entwickelt, d. h. die Entwicklung eines Compilers für eine neue Programmiersprache orientiert sich an den Phasen der Analyse und Übersetzung; es sind dies:

- die lexikalische und syntaktische Analyse (Scanner und Parser),
- die semantische Analyse,
- u. U. eine Optimierungsphase und schließlich
- die Codegenerierung.

Die am meisten verbreiteten Entwicklungswerkzeuge für den Compilerbau konzentrieren sich auf die lexikalische und syntaktische Analyse sowie auf die Optimierung und Codegenerierung. Der wesentliche Aufwand bei der Compiler-Entwicklung steckt jedoch in der semantischen Analyse und nur ca 10% des Entwicklungsaufwandes fließen üblicherweise in die ersten beiden Phasen. Der überwiegende Teil des Aufwandes muß meist für die semantische Analyse und die Codegenerierung getrieben werden, je nach Bedeutung und dementsprechendem Aufwand für die Optimierung.

3. Das Baukastenprinzip

Wie in Abschnitt 2 erwähnt, unterstützen traditionelle Tools die verschiedenen Phasen eines Compilers. Günstiger ist jedoch ein Ansatz, bei dem einzelne Konstrukte durchgängig von der semantischen Analyse bis zur Codegenerierung behandelt werden und bei dem möglichst wenig Interaktionen zwischen den einzelnen Konstrukten nötig ist. Ein solches Design kann wie folgt aussehen:

- Es existiert eine Bibliothek für die wesentlichen Konzepte (z. B. Schleifen) von Programmiersprachen und häufig gebrauchte Spezialisierungen dieser Konzepte (z. B. abweisende Schleifen, akzeptierende Schleifen etc.). Die Semantik dieser Konzepte ist (textuell) beschrieben; zu jedem Konzept gehören Routinen, die die semantische Analyse entsprechend der Dokumentation durchführen und Routinen, die damit konsistenten Code produzieren. Es muß nun nur noch eine Möglichkeit gefunden werden, wie die Bibliotheksroutinen bei der Analyse und Übersetzung eines konkreten Programmes auf dessen Elemente zugreifen.

  Die Realisierung ist einfach, wenn man eine objektorientierte Darstellung wählt: Die Konzepte der

Bibliothek und ihre Spezialisierungen werden mittels Klassenhierarchien, die zugehörigen Routinen für die semantische Analyse und die Codegenerierung als Methoden organisiert. Der Zugriff auf Elemente eines konkreten Programmes kann dann über Methoden realisiert werden, die in den Bibliotheksklassen abstrakt deklariert werden. So wird z. B. jedes Objekt einer Bibliotheksklasse, die ein Sprachkonzept verkörpert, Nachrichten verstehen, die es auffordern, sich auf semantische Korrektheit zu überprüfen oder entsprechenden Code zu generieren.

- Als Ergebnis der lexikalischen und syntaktischen Analyse entsteht der abstrakte Programmbaum als Pendant des zu übersetzenden Programmes, dessen Elemente jedoch nicht mittels Knoten und Zeigern in Baumform, sondern durch geschachtelte Objekte repräsentiert werden. Die Objekte sind Instanzen von Klassen, die aus den EBNF-Produktionen der Grammatik der Sprache abgeleitet werden.
- Die Klassen der Bibliothek werden nun mittels Vererbung für eine konkrete Sprache spezialisiert, d. h. die Klassen zur Darstellung des abstrakten Programmes erben von einer jeweils geeigneten Bibliotheksklasse. Für den Zugriff auf konkrete Programmelemente werden die abstrakten Methoden der Oberklassen konkretisiert.

Ist nun ein konkretes Programm zu übersetzen, so wird daraus zunächst die abstrakte Zwischenform erzeugt. Deren semantische Analyse und die Codegenerierung übernehmen die Methoden der Bibliothek, die wegen der Vererbungsbeziehungen auf die Objekte der Zwischenform anwendbar sind. Sie bedienen sich dabei der Zugriffsmethoden, die bei der Adaption in den Unterklassen konkretisiert wurden.

In Abschnitt 4 wird nun ein System vorgestellt, das die Komponenten-Bauweise eines Compilers durch geeignete Werkzeuge unterstützt, das OCC-System.

4. Das OCC-System (Objectoriented Compiler Construction)

In diesem Kapitel wird das OCC-System kurz vorgestellt, wie es zum Teil existiert, zum Teil noch in der Implementierungsphase ist. Die wesentlichen Werkzeuge sind bereits im Einsatz, umfassende Bibliotheken sind noch in der Entwicklung. Abschnitt 4.1 bietet einen Überblick über den Arbeitsablauf mit dem OCC-System, in Abschnitt 4.2 werden dann verschiedene Implementierungsaspekte erläutert.

4.1. Überblick

Zwei Einzel-Werkzeuge des OCC-Systems, die in das Gesamtsystem integriert sind, unterstützen direkt die Entwicklung eines Compilers. Auf die Darstellung der anderen Werkzeuge kann in diesem Kontext verzichtet werden. Die Generierung eines Compilers läuft in 3 Schritten ab (vergleiche auch Abbildung 1):

- Mit dem ersten Werkzeug, *comp_gen*, wird aus der EBNF-Beschreibung der Grammatik der neuen Sprache ein Scanner und ein Parser für die lexikalische und syntaktische Analyse sowie Klassendefinitionen für die Darstellung des abstrakten Zwischencodes (als Ergebnis der Syntaxanalyse) generiert. *comp_gen* entspricht den gängigen Tools zur Unterstützung dieser Phase (vgl. Abschnitt 2), z. B. *lex* und *yacc*, und baut auch auf diesen beiden auf.
- Das zweite Werkzeug, *attribute*, bekommt als Eingabe die (von *comp_gen*) generierten Klassen, die Bibliotheksklassen sowie eine Menge von Vererbungsregeln, die die Beziehung der generierten zu den Bibliotheksklassen beschreiben. *attribute* erweitert die generierten Klassen um die Vererbungsinformation. Dann werden Methoden, die in den Bibliotheksklassen als abstrakt definiert wurden, für die generierten (Unter-) Klassen spezialisiert.
- Die von *attribute* spezialisierten Methoden werden vom Compilerbauer konkretisiert, was der Adaption der konkreten Attribute der aktuellen Grammatik an die Anforderungen der Bibliotheksklassen entspricht. Die hier notwendigen Routinen beschränken sich überwiegend auf den Zugriff auf Attribute der Zwischencodeklassen, die den Anforderungen entsprechend geeignet zu Listen oder Objekten zusammengefügt werden.

  attribute kann auch auf bereits attributierte Klassen angewendet werden; die bereits vorhandenen Attributierungen bleiben dabei erhalten.

Nach Abschluß der Konkretisierung der geerbten Methoden ist die Compiler-Konstruktion beendet, die zugehörigen Klassen und Routinen können geladen und der Compiler verwendet werden. Bei einer Änderung an der Sprache oder an der Semantik eines oder mehrerer Konstrukte kann der Ablauf ab dem ersten bzw. zweiten

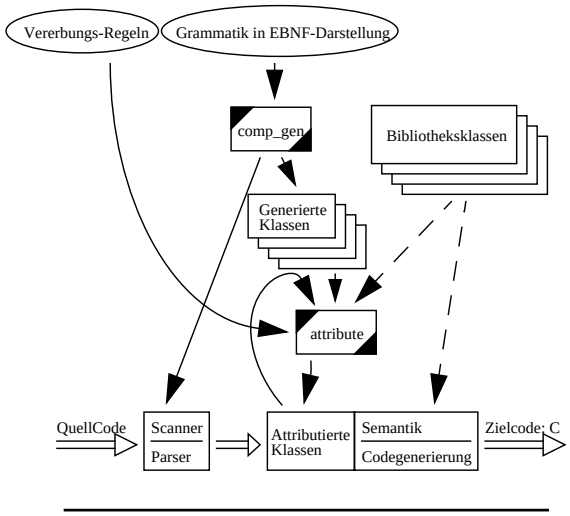

Abbildung 1: Das OCC-System

Punkt wiederholt werden. Aufgrund des objektorientierten Entwurfs zeigen Änderungen an einem Konstrukt üblicherweise nur geringe Auswirkungen auf andere Konstrukte, die nicht automatisch durch das System abgefangen werden können.

4.2. Implementierungsaspekte

Das OCC-System ist in W-Lisp [3] realisiert, einer auf Common Lisp basierenden Sprache, die sich im wesentlichen durch bessere Lesbarkeit verschiedener Konstrukte (durch Verwendung imperativer Strukturen) von Common Lisp unterscheidet. Lisp eignet sich durch seine Interpreter-Eigenschaften vorzüglich für die schnelle Entwicklung von Prototypen. Zudem steht mit CLOS (Common Lisp Object System, [2]) eine leistungsfähige objektorientierte Erweiterung zur Verfügung.

Scanner und Parser für generierte Compiler werden mit Hilfe der Standard-Unix-Werkzeuge *lex* und *yacc* realisiert. Diese sind hinreichend effizient und es ist im Rahmen dieses Projekts nicht nötig, eigene Implementierungen für die lexikalische und syntaktische Analyse zu entwickeln. Die semantischen Aktionen für Scanner- und Parsergenerator werden so gestaltet, daß ein Aufruf des Frontends den entsprechenden Zwischencode, d. h. Instanzen der von *comp_gen* generierten Klassen liefert.

Als Implementierungssprache für die Klassenbibliotheken und die Adaption der konkreten Sprachkonstrukte an deren Konzepte ist anstelle von W-Lisp auch jede andere objektorientierte Sprache denkbar, die Mehrfachvererbung sowie Multi-Methoden bietet. Das Konzept der Mehrfachvererbung ist essentiell für die Attributierung; z. B. ist die `assignment_expression` in C (vgl. [1]) eine Zuweisung, kann aber auch als Ausdruck verwendet werden, daher beerbt die zugehörige (generierte) Zwischencodeklasse mehrere Bibliotheksklassen. Die Gefahren der Unübersichtlichkeit und Undurchschaubarkeit der Vererbungsbeziehungen, die mit dem Konzept der Mehrfachvererbung üblicherweise in Verbindung gebracht werden, spielen im OCC-System jedoch keine Rolle, da es sich hier um generierten Code handelt, der nur für die Realisierung höherer Konzepte gebraucht wird. Die Multi-Methoden sind nicht unbedingt notwendig, erleichtern jedoch die modulare, seiteneffektfreie Konstruktion eines Compilers.

Zielcode der generierten Compiler ist ANSI-C, da es bei der Codegenerierung genügend Freiheiten läßt (ähnlich wie in Assembler-Sprachen gibt es viele Möglichkeiten low-level zu arbeiten), andererseits aber den Vorteil bietet, unabhängig von einer bestimmten Plattform zu sein, während die generierten Programme hinreichend schnell sind.

5. Zusammenfassung und Bewertung der Wiederverwendbarkeit

Das OCC-System ermöglicht, Sprachkonzepte und -konstrukte nach dem Baukastenprinzip miteinander zu kombinieren, ohne auf lästige und gefährliche Seiteneffenkte achten zu müssen. Es entsteht ein Compiler für die neue Sprache, der zwar nicht sonderlich effizient ist, der aber innerhalb kürzester Zeit schon zur Verfügung steht und korrekt arbeitet. Damit ist es möglich, nicht nur mit der konkreten Notation, sondern auch mit den Konzepten einer neuen Sprache zu experimentieren, bevor diese festgeschrieben und aufwendig implementiert wird.

Die mit dem OCC-System entwickelten Compiler sind sehr stabil gegenüber Änderungen an der zu übersetzenden Sprache, da die komponentenweise Konstruktion eine hohe Unabhängigkeit der einzelnen Teile gewährleistet. Das ist insbesondere bei der Entwicklung einer neuen Sprache von Vorteil.

Die Vorteile objektorientierter Sprachen bezüglich der Wiederverwendbarkeit von Komponenten sind nach wie vor umstritten. Aus den Erfahrungen beim Einsatz des OCC-Systems läßt sich aber folgern, daß zumindest im Compilerbau die Chancen für eine Wiederverwendung einmal entwickelter Komponenten recht gut stehen. Das liegt zum einen daran, daß es sich hierbei um ein recht gut erforschtes Gebiet handelt, so daß man bei der Entwicklung einer neuen Sprache nicht vor völlig neuen Problemen steht, sondern von den Erfahrungen der Vergangenheit profitieren kann. Dadurch läßt sich meistens recht gut vorhersagen, welches die wiederverwertbaren Konzepte einer Programmiersprache sind und wie diese sich günstig in eine Klassenstruktur abbilden lassen.

Zum anderen werden die wenigsten Programmiersprachen von Grund auf neu entwickelt; üblicherweise baut man auf bewährten Strukturen (z. B. im Bereich der Programm- bzw. Modulstrukturen oder der Typen) auf, für die natürlich schon recht bald gut entwickelte Klassen in der Bibliothek zur Verfügung stehen. Daran anknüpfend werden dann die neuen Aspekte der Sprache entworfen, wie z. B. spezielle Datentypen und deren Operationen oder andere Besonderheiten. Als typisches Beispiel dafür kann der Sprachumfang für ein einsemestriges Compilerbau-Praktikum für Studenten dienen. Dabei zeigt es sich, daß es in der Regel möglich ist, innerhalb weniger Tage einen funktionsfähigen und korrekten Compiler für eine neue, der Problemstellung angemessene Sprache mit dem OCC-System zu entwickeln.

Anhang: Literatur

[1] B. W. Kernighan, D. M. Ritchie,
Programmieren in C; Zweite Ausgabe ANSI-C, Carl Hanser Verlag München Wien, 1990

[2] G. S. Steele jr.,
Common Lisp, The Language, Second Edition, Digital Press, 1990

[3] H.-W. Wippermann, R. Eppler, P. Knauber, S. Vorwieger,
W-LISP - Sprachbeschreibung, Interner Bericht der Universität Kaiserslautern Nr. 237/93