Werkzeuge für den Übersetzerbau

J. Grosch
H. Emmelmann

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Werkzeuge für den Übersetzerbau

Josef Grosch
Helmut Emmelmann

Feb. 7, 1990

Report No. 21

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Werkzeuge für den Übersetzerbau

J. Grosch, H. Emmelmann
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany

Übersicht

Mit Übersetzerbau-Werkzeugen lassen sich Übersetzer für Programmiersprachen weitgehend automatisch generieren. Wir stellen einen Werkzeugkasten vor, welcher die Konstruktion nahezu aller Phasen eines Übersetzers unterstützt. Die Entwurfsziele für diesen Werkzeugkasten waren praktische Brauchbarkeit, deutlich reduzierter Erstellungsaufwand für Übersetzer und hohe Qualität der generierten Übersetzer. Besonders hinsichtlich Effizienz sollten die Werkzeuge konkurrenzfähig zur Programmierung von Hand sein. Zur Zeit können mit den Werkzeugen Übersetzermodule in den Zielsprachen C und Modula-2 erzeugt werden. Erste realistische Anwendungen demonstrieren die ausgezeichnete Leistungsfähigkeit der Werkzeuge und zeigen, daß die Werkzeuge die Konstruktion von Übersetzern mit Produktionsqualität erlauben.

1. Aufbau eines Übersetzers

Ein wichtiges Hilfsmittel zur Programmierung eines Computers ist ein Übersetzer (compiler). Ein Übersetzer ist ein Programm, welches ein in einer Programmiersprache geschriebenes Programm in eine Maschinensprache übersetzt. Die Hardware versteht genaugenommen nur aus Nullen und Einsen zusammengesetzte Maschinensprach-Programme. Um einem Computer auch eine für den menschlichen Programmierer besser geeignete höhere Programmiersprache verständlich zu machen, ist eine Übersetzung nötig.

Die Konstruktion eines Übersetzers ist eine anspruchsvolle und aufwendige Aufgabe. Der Bedarf an Übersetzern ist relativ groß, da für jede Programmiersprache und jeden Computer ein eigener Übersetzer notwendig ist. Es lohnt sich daher, nach Methoden zu suchen die Erstellung von Compilern zu vereinfachen. Doch bevor wir zu unserem eigentlichen Thema kommen, nämlich der automatischen Generierung von Übersetzern mit Übersetzerbau-Werkzeugen, möchten wir kurz den Aufbau und die prinzipielle Funktionsweise eines Übersetzers erläutern. Die rechte Spalte in Abb. 1 zeigt die Phasen bzw. Module eines Übersetzers.

Die lexikalische Analyse liest das Quellprogramm zeichenweise. Sie faßt die Zeichenfolgen für Bezeichner, Zahlen und Schlüsselwörter zu Grundsymbolen zusammen und überliest Zwischenräume und Kommentare.

Die syntaktische Analyse hat als Eingabe eine Folge von Grundsymbolen. Sie überprüft das Quellprogramm auf syntaktische Fehler und rekonstruiert die Struktur des Programms, d. h. sie erkennt den Aufbau der Ausdrücke und Anweisungen sowie deren Zusammenhang. Diese Struktur wird oft in Form eines Syntaxbaums gespeichert.

Die semantische Analyse überprüft die Kontextbedingungen bzw. die Regeln der statischen Semantik und berechnet für die Codegenerierung nötige Eigenschaften. Ein Beispiel für eine Kontextbedingung ist die Vorschrift, daß alle Variablen deklariert sein müssen. Zur statischen Semantik zählen die Analyse der Gültigkeitsbereiche, die Namensanalyse, d. h. die Feststellung der zu einem Bezeichner gehörenden Deklaration, und die Typüberprüfung.

Zur Vereinfachung der gesamten Übersetzungsaufgabe wird diese häufig in zwei Schritte unterteilt. Der Syntaxbaum wird zunächst von einer Transformationsphase in eine Zwischensprache umgewandelt. Diese Zwischensprache ist meist maschinenorientiert jedoch noch maschinenunabhängig. Das niedrige Niveau der Zwischensprache erleichtert dem Codegenerator

die Erzeugung der Maschinensprache.

Zu den Aufgaben des Codegenerators zählen die Befehlsauswahl, d. h. die Abbildung der Zwischensprachanweisungen auf Maschinenbefehle, sowie die Speicher- und Registerzuteilung. Die Ausgabe ist schließlich ein binär-codiertes Maschinenprogramm.

Der folgende Abschnitt beschreibt die Vorteile, die Entwurfsziele und den Inhalt des Werkzeugkastens. Abschnitt 3 stellt die gemeinsamen Eigenschaften der Werkzeuge dar. Im Abschnitt 4 wird das von uns bevorzugte Übersetzermodell beschrieben. Der Abschnitt 5 enthält eine kurze Darstellung der einzelnen Werkzeuge. Abschnitt 6 berichtet von den Erfahrungen des Einsatzes der Werkzeuge in zwei realistischen Anwendungen. Abschnitt 7 enthält eine Zusammenfassung und beschreibt weiterführende Arbeiten.

2. Werkzeugkasten

Die Erstellung eines Übersetzers von Hand ist eine sehr anspruchsvolle und aufwendige Aufgabe. Durch den Einsatz von Übersetzerbau-Werkzeugen läßt sich dieser Aufwand reduzieren. Im folgenden stellen wir einen Werkzeugkasten zur Übersetzer-Generierung vor, welcher für nahezu jede Übersetzerphase Werkzeuge enthält. Diese sind für den Einsatz in realistischen Übersetzerprojekten konzipiert.

Im allgemeinen akzeptieren die Werkzeuge als Eingabe eine Spezifikation, die in einer werkzeug-spezifischen Sprache geschrieben ist. Sie produzieren als Ausgabe ein Programm-Modul in einer Zielsprache (C oder Modula-2). Deshalb kann ein Werkzeug als generische Lösung eines Teilproblems in einem Übersetzer gesehen werden, wobei mit Hilfe einer Spezifikation eine konkrete Lösung gewonnen wird.

Die Benutzung von Werkzeugen hat gegenüber der Programmierung von Hand mehrere Vorteile: Der zur Konstruktion eines Übersetzers notwendige Aufwand wird wesentlich verringert. An Stelle eines Programms wird eine viel kürzere Spezifikation entwickelt. Die Werkzeuge können eine Spezifikation in vielfacher Weise auf Konsistenz überprüfen. Das Schreiben automatisch prüfbarer Spezifikationen verringert die Anzahl möglicher Fehler und erhöht so die Zuverlässigkeit des resultierenden Übersetzers.

Die wichtigsten Entwurfsziele für den Werkzeugkasten waren:

- praktische Brauchbarkeit für realistische Programmiersprachen
- automatische Generierung von Übersetzern mit Produktionsqualität
- wesentliche Steigerung der Übersetzerbau-Produktivität
- mit Handprogrammierung vergleichbare Qualität der erzeugten Übersetzer

Mit dieser Zielsetzung sollte die praktische Einsatzfähigkeit des Werkzeugkastens in realistischen Übersetzerbauprojekten erreicht werden. Daher wurde auch die Konkurrenzfähigkeit zur Handprogrammierung betont. Wir meinen, daß die hohe Produktivität und Zuverlässigkeit nicht durch eine geringere Codequalität oder Effizienz des resultierenden Compilers erkauft werden muß.

Der Werkzeugkasten enthält folgende Werkzeuge:

Rex	Generator für lexikalische Analysatoren
Lalr	LALR(1) Zerteilergenerator
Ell	LL(1) Zerteilergenerator
Ast	Generator für abstrakte Syntaxbäume
Ag	Generator für Attributauswerter
Estra	Transformation abstrakter Syntaxbäume
Beg	Generator für Codegeneratoren
Reuse	Bibliothek wiederverwendbarer Module

Alle Werkzeuge wurden ursprünglich in Modula-2 programmiert und laufen unter dem Betriebssystem UNIX. Unter Verwendung des Modula-2 nach C Übersetzers *Mtc* [Mar90] (siehe Abschnitt 6.1), konnte von den Programmen automatisch eine C-Version erstellt werden. Zur Zeit erzeugen die meisten Werkzeuge Module in den Zielsprachen C und Modula-2.

3. Gemeinsame Eigenschaften

Unsere Entwurfsziele führten zu einigen für alle Werkzeuge gemeinsamen Entwurfsentscheidungen. Nahezu jedes Werkzeug benötigt eine Programmiersprache, mit der der Benutzer gewiße Aktionen, Bedingungen oder Berechnungen spezifizieren kann. Das trifft offensichtlich für Attributgrammatiken zu, aber auch der Codegenerator-Generator muß Attribute und Bedingungen auswerten. Sogar die Zerteilergeneratoren brauchen eine solche Sprache zur Spezifikation semantischer Aktionen.

Wir entschieden uns dafür direkt die Zielsprache (nämlich C oder Modula-2) zu verwenden. Deshalb können Spezifikationen Abschnitte mit Zielsprachanweisungen enthalten. Abgesehen von geringfügigen Ersetzungen wird dieser Text unverändert in die erzeugten Module kopiert. Der Nachteil dieser Methode ist, daß die in der Zielsprache geschriebenen Teile nicht vollständig von den Werkzeugen überprüft werden können. Zum Beispiel kann das Attributgrammatik-Werkzeug nicht überprüfen, ob Attributberechnungen keine Seiteneffekte haben. Andererseits wird damit eine sehr große Flexibilität erreicht, da die volle Ausdruckskraft der Zielsprache zur Verfügung steht. Ebenso wird die praktische Brauchbarkeit drastisch erhöht, da die Einbeziehung anderer, eventuell handgeschriebener Komponenten leicht möglich ist. Schließlich führt es zu einfachen Werkzeugen und einfachen Spezifikationssprachen.

Unsere Erfahrung mit früheren Werkzeugen zeigte, daß während der Konstruktion realistische Übersetzer eine Reihe kleiner Sonderprobleme auftritt, die nicht mit den Werkzeugen gelöst werden können. Deswegen sind Schlupflöcher nötig, also Möglichkeiten, die es dem Werkzeugbenutzer erlauben leicht handgeschriebene Teile einzufügen. Diese Schlupflöcher tragen auch dazu bei die Werkzeuge einfach zu machen, da man nicht gezwungen ist, für jedes Problem sofort eine Lösung bereitzustellen. Das Schlupfloch kann benutzt werden solange bis eine wirklich gute Lösung gefunden wird, welche man in ein Werkzeug einbauen kann.

Die Werkzeuge sind größtenteils von einander unabhängig. Dies wird dadurch erzielt, daß keines der erzeugten Module eine festgelegte Ausgabe besitzt. Stattdessen wird diese Ausgabe mittels Anweisungen der Zielsprache spezifiziert und kann somit beliebig gewählt werden. Die Unabhängigkeit der Werkzeuge sorgt für große Freiheiten beim Übersetzerentwurf. Eine Ausnahme bilden die Werkzeuge *Ag* und *Estra*, denn sie basieren auf den mit *Ast* spezifizierten Syntaxbäumen. Deshalb hängen diese Werkzeuge von *Ast* ab, und alle drei Werkzeuge sind für Übersetzer zugeschnitten, die einen attributierten abstrakten Syntaxbaum benutzen.

4. Übersetzer-Modell

Obwohl die Werkzeuge kein bestimmtes Übersetzer-Modell erzwingen, möchten wir das von uns bevorzugte Modell vorstellen. Wir meinen, daß dieses am besten von den Werkzeugen unterstützt wird. Wir betrachten die semantische Analyse nach wie vor als den schwierigsten

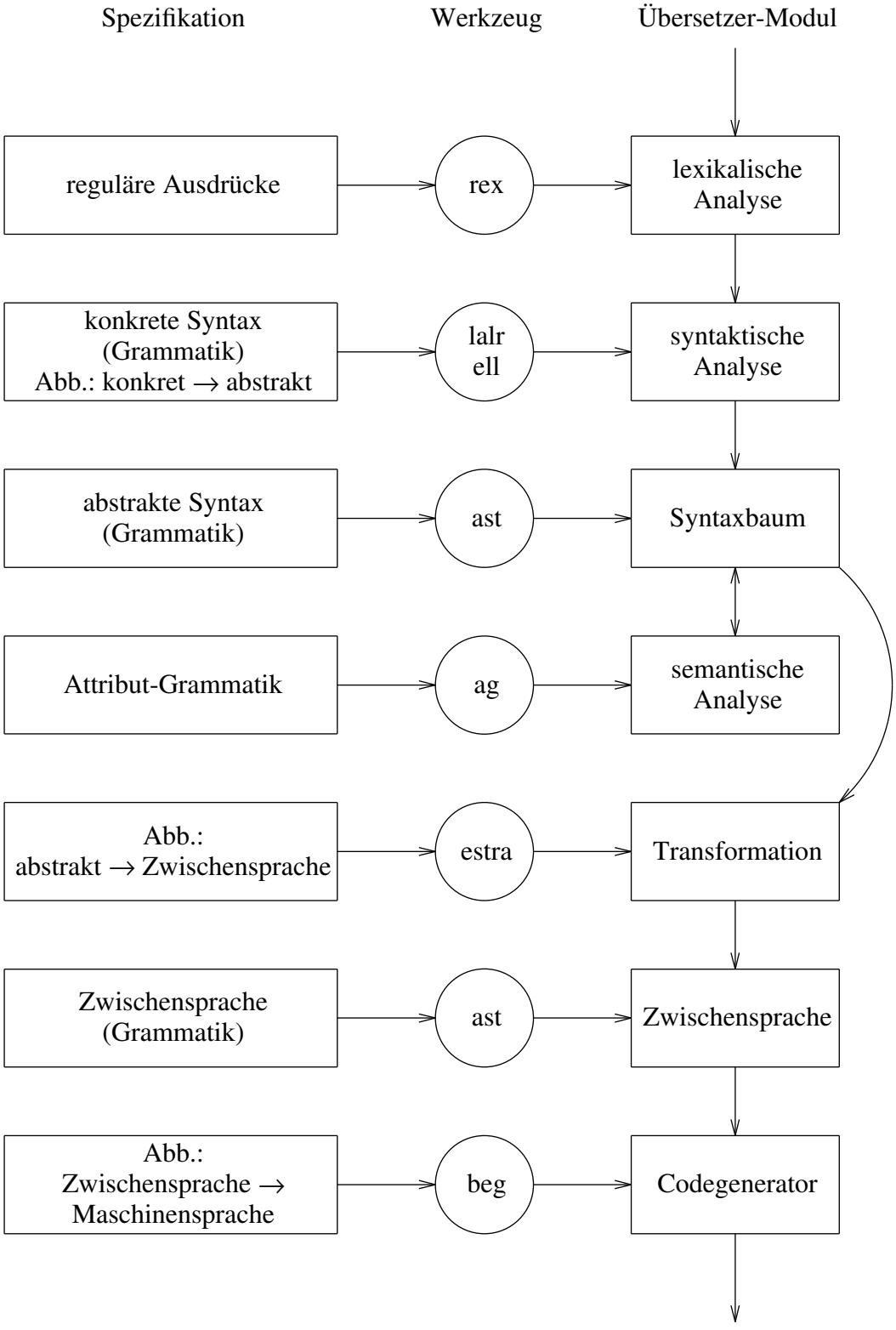

Abb. 1: Übersetzer-Modell

Teil eines Übersetzers. Deshalb gehen wir für die semantische Analyse und die Erzeugung einer Zwischensprache von der abstrakten Syntax aus. Wir bauen den abstrakten Syntaxbaum explizit

auf, welcher während der semantischen Analyse eventuell mit Attributen ergänzt wird. Neben der abstrakten Syntax, welche als erste, hohe Zwischensprache betrachtet werden kann, bevorzugen wir die Verwendung einer zweiten, niederen Zwischensprache als Schnittstelle zum Codegenerator. Dies bringt Vorteile in der Optimierung und der mustergesteuerten Codeauswahl mit sich.

Abbildung 1 zeigt das von uns bevorzugte Übersetzermodell. Die rechte Spalte enthält die wichtigsten Module eines Übersetzers. Die linke Spalte zeigt die dazu notwendigen Spezifikationen. Die dazwischen liegenden Werkzeuge werden von den Spezifikationen gesteuert und erzeugen die einzelnen Module. Die Pfeile stellen den Datenfluß dar, teils zur Generierungszeit und teils zur Übersetzungszeit.

5. Die Werkzeuge

Die folgenden Abschnitte stellen kurz die einzelnen Werkzeuge des Werkzeugkastens vor. Wir beschreiben nur die Eigenschaften der Werkzeuge. Für weitere Einzelheiten, die Spezifikationstechniken oder für Beispiele sei der Leser auf die existierenden, werkzeug-spezifischen Dokumente verwiesen.

5.1. Rex

Rex (*r*egular *ex*pression tool) ist ein Generator für lexikalische Analysatoren [Gro87a, Gro88, Gro89a]. Seine Spezifikationen basieren auf regulären Ausdrücken und beliebigen semantischen Aktionen, die in einer der Zielsprachen C oder Modula-2 geschrieben werden. Immer wenn in der Eingabe des erzeugten lexikalischen Analysators eine einem regulären Ausdruck entsprechende Zeichenkette erkannt wurde, werden die zugehörigem Aktionen ausgeführt. Falls zur eindeutigen Erkennung der Symbole der Kontext betrachtet werden muß, so kann der rechte Kontext durch einen zusätzlichen regulären Ausdruck spezifiziert werden, und der linke Kontext wird mit sogenannten Start-Zuständen behandelt. Falls mehrere reguläre Ausdrücke auf die aktuelle Eingabe zutreffen, so wird der Ausdruck mit der längsten Zeichenkette bevorzugt. Falls es immer noch mehrere Möglichkeiten gibt, so wird der zuerst in der Spezifikation stehende Ausdruck gewählt.

Die erzeugten lexikalischen Analysatoren berechnen automatisch Zeile und Spalte in der Quelle für jedes erkannte Symbol und enthalten einen Mechanismus für Include-Dateien. Bezeichner und Schlüsselwörter können effizient in Groß- oder Kleinbuchstaben normalisiert werden. Es gibt vordefinierte Regeln um Leerstellen, Tabulatoren und Zeilenwechsel zu überlesen. Die generierten lexikalischen Analysatoren sind tabellengesteuerte, deterministische endliche Automaten. Die Tabellen werden mit der sogenannten Kammvektortechnik komprimiert [ASU86].

Die herausragende Eigenschaft von *Rex* ist seine Geschwindigkeit. Die lexikalischen Analysatoren verarbeiten nahezu 200.000 Zeilen pro Minute ohne Hashing von Bezeichnern und 150.000 Zeilen pro Minute mit Hashing. Dies ist die vierfache Geschwindigkeit gegenüber mit *Lex* [Les75] generierten lexikalischen Analysatoren. In typischen Fällen besitzen mit *Rex* generierte Analysatoren ein Viertel der Größe derer von *Lex*. Normalerweise benötigt *Rex* nur 1/10 der Zeit von *Lex* zum Generieren eines lexikalischen Analysators.

5.2. Lalr

Lalr ist ein LALR(1) Zerteiler-Generator der Grammatiken, die in erweiterter BNF geschrieben sind, verarbeitet [GrV88, Gro88]. Die Grammatikregeln können mit semantischen Aktionen versehen werden, die direkt in einer Zielsprache formuliert sind. Immer wenn der erzeugte Zerteiler eine Grammatikregel erkennt, wird die zugehörige semantische Aktion ausgeführt. Der Generator stellt einen Mechanismus zur S-Attributierung zur Verfügung, d. h.

synthetisierte Attribute können während der Zerteilung berechnet werden.

Im Falle von LR-Konflikten liefert *Lalr* nicht wie andere Generatoren nur Information über aus Mengen von Situationen bestehende Zustände, sondern druckt einen Ableitungsbaum, der wesentlich nützlicher zur Analyse des Konflikts ist. Konflikte können durch die Angabe von Priorität und Assoziativität für Operatoren und Produktionen gelöst werden. Die generierten Zerteiler beinhalten eine automatische Fehlerbehandlung mit Fehlermeldungen und -reparatur. Zur Fehlerbehandlung wird die vollständige, rücksetzungsfreie Methode von Röhrich [Röh76, Röh80, Röh82] verwendet. Die Zerteiler sind tabellengesteuert und wie im Falle von *Rex* werden die Tabellen mit der Kammvektortechnik komprimiert. Der Generator verwendet den in [DeP82] beschriebenen Algorithmus zur Berechnung der Vorschaumengen.

Mit *Lalr* erzeugte Zerteiler sind zwei bis drei mal schneller als mit *Yacc* [Joh75] erzeugte. Sie erreichen eine Geschwindigkeit von 580.000 Zeilen pro Minute ohne Berücksichtigung der lexikalischen Analyse. Die Größe der Zerteiler ist gegenüber *Yacc* leicht erhöht, denn für die Geschwindigkeit muß ein kleiner Preis bezahlt werden.

Die Eingabesprachen von *Rex* und *Lalr* sind hinsichtlich der Syntax gegenüber *Lex* und *Yacc* lesbarer gestaltet. Mit Hilfe zweier, hier nicht näher beschriebener Präprozessoren können *Rex* und *Lalr* auch Eingaben für *Lex* und *Yacc* verarbeiten. Dadurch sind unsere Werkzeuge in Bezug auf die Benutzerschnittstelle kompatibel mit den UNIX-Werkzeugen.

5.3. Ell

Ell ist ein LL(1) Zerteiler-Generator, der die gleiche Spezifikationssprache wie *Lalr* verarbeitet, mit dem Unterschied, daß die Grammatiken die LL(1)-Eigenschaft besitzen müssen [GrV88, Gro88, Gro89b]. Während der Zerteilung kann eine L-Attributierung ausgewertet werden. Die erzeugten Zerteiler beinhalten eine automatische Fehlerbehandlung mit Fehlermeldungen und -reparatur wie *Lalr*. Die Zerteiler sind nach dem Verfahren des rekursiven Abstiegs implementiert und erzielen eine Geschwindigkeit von 900.000 Zeilen pro Minute.

5.4. Ast

Ast ist ein Generator für abtrakte Syntaxbäume [Gro89d, Gro89e]. Er generiert Programm-Module oder abstrakte Datentypen zur Bearbeitung attributierter Bäume. Neben Bäumen können auch attributierte Graphen bearbeitet werden. Den Knoten dieser Datenstrukturen können beliebig viele Attribute von beliebigem Typ zugeordnet werden. Die Spezifikationen für dieses Werkzeug basieren auf erweiterten Baum-Grammatiken. Sie können als gemeinsame Notation sowohl für konkrete und abstrakte Syntax als auch für attributierte Bäume und Graphen betrachtet werden. Ein Erweiterungsmechanismus stellt einfache Vererbung zur Verfügung. Intern werden die Bäume durch verzeigerte Verbunde gespeichert. Zahlreiche Operationen für Bäume und Graphen können auf Anforderung von *Ast* erzeugt werden: Sogenannte Knotenkonstruktoren kombinieren Aggregatschreibweise mit Speicherverwaltung. Lese- und Schreibeprozeduren übertragen Graphen aus/in Dateien in lesbarem ASCII- oder internem Binärformat. Die Reihenfolge von Teilbäumen in einer Liste kann umgekehrt werden. Es werden Prozeduren für häufig benutzte Traversierungsstrategien wie *top down* oder *bottom up* zur Verfügung gestellt. Ein interaktiver *Graph-Browser* erlaubt die Inspektion von Graphen in lesbarer Weise und unterstützt so den Programmtest.

5.5. Ag

Ag ist ein Generator für Attributauswerter [Gro89c, Gro90]. Er verarbeitet geordnete Attributgrammatiken (OAGs) [Kas80] und sogenannte *higher order* Attributgrammatiken (HAGs) [VSK89]. Er basiert auf der abstrakten Syntax oder besser gesagt auf den von *Ast* erzeugten Baummodulen. Deshalb ist die Baumstruktur völlig bekannt. Den Terminalen und

Nichtterminalen können beliebig viele Attribute zugeordnet werden. Diese werden mit den Typen der Zielsprache getypt. Dabei sind auch baumwertige Attribute möglich. *Ag* erlaubt regellokale Attribute und bietet einen Erweiterungsmechanismus an, welcher einfache Vererbung für Attribute und Attributberechnungen zur Verfügung stellt. Dieser gestattet ebenfalls die Elimination von Kettenregeln. Die Attributberechnungen werden in der Zielsprache formuliert und sollten in einem funktionalen Stil gehalten sein. Es ist möglich externe Funktionen von getrennt übersetzten Modulen aufzurufen. Die Verwendung nicht-funktionaler Anweisungen und von Seiteneffekten ist möglich, verlangt allerdings sorgfältige Überlegung. Die Syntax der Spezifikationssprache ist im Hinblick auf die Unterstützung kompakter, modularer und lesbarer Dokumente entworfen worden. Eine Attributgrammatik kann aus mehreren Modulen bestehen, wobei die kontextfreie Grammatik nur einmal spezifiziert wird. Es gibt Kurzschreibweisen für Kopierregeln and gefädelte Attribute womit viele triviale Attribut-Berechnungen weggelassen werden können. Die erzeugten Attributauswerter sind sehr effizient, da sie unter Verwendung von rekursiven Prozeduren direkt codiert sind. Die Speicherung der Attribute wird optimiert indem Attribute als lokale Variable und Prozedurparameter implementiert werden, wenn ihre Lebenszeit innerhalb eines Besuches liegt.

5.6. Estra

Estra ist ein Generator für Transformatoren von abstrakten Syntaxbäumen [Vie89]. Die erzeugten Transformations-Module haben als Eingabe einen attributierten Baum und bilden diesen auf eine Ausgabe beliebiger Art ab. Die Ausgabe kann ein neuer Baum sein, eine lineare Zwischensprache wie z. B. P-Code, ein Quellprogramm z. B. in Pascal oder eine Folge von Prozeduraufrufen. In jedem Fall bleibt der Eingabebaum unverändert, um das Problem der Reattributierung aus Konsistenzgründen zu umgehen. Jedoch können Teilbäume des Eingabebaums zur Konstruktion eines Ausgabebaums wiederverwendet werden. Die beabsichtigten Anwendungsgebiete für die Transformationen sind die Erzeugung von Zwischensprachen aus abstrakten Syntaxbäumen und Optimierer für interne Baumstrukturen jeden Niveaus. *Estra* arbeitet mit dem Werkzeug *Ast* zusammen in der Art, daß die Eingabebäume mittels von *Ast* erzeugten Modulen erstellt werden.

Die Spezifikation einer Transformation ist regelbasiert. Eine Regel besteht aus einem Muster, welches ein Baumfragment beschreibt, und einer Aktion. Aktionen bestehen aus Anweisungen der Zielsprache. Es können mehrere Transformationen spezifiziert werden. Die Teilbäume eines Musters können in beliebiger Reihenfolge transformiert werden. Sie können mehrmals mit der selben oder mit verschiedenen Transformationen bearbeitet werden. Die Aktionen haben Lesezugriff auf die Attribute des Eingabebaums. Zusätzliche abgeleitete oder vererbte Attribute können während der Transformation berechnet werden. Die Anwendung von Regeln läßt sich durch Bedingungen einschränken. Mehrdeutigkeiten werden mittels Kostenangaben aufgelöst.

Zwei Implementierungen für den Algorithmus zum Mustervergleich können gewählt werden: Ein direkt codierter dynamischer Programmierungs-Algorithmus oder ein tabellen-gesteuerter Baummuster-Vergleicher. In beiden Fällen besitzt eine Transformation zwei Phasen. Während die Erste die mit minimalen Kosten passenden Muster bestimmt, führt die Zweite die zugehörigen Aktionen aus.

5.7. Beg

Beg (*b*ack *e*nd *g*enerator) erzeugt Module zur Codeauswahl und zur Registerzuteilung [Emm89a, Emm89b]. Codeauswahl wird mittels Baummuster-Vergleich durchgeführt. Die Maschinenbefehle werden mit Regeln beschrieben welche Baummuster enthalten. Der erzeugte Codegenerator hat als Eingabe eine baumförmige Zwischensprache. Ein Eingabebaum wird abgebildet durch die Überdeckung des Baums mit Mustern und der anschließenden Ausgabe der

zugehörigen Maschinenbefehle. Die Regeln sind mit Kosten versehen, wodurch der Codegenerator eine Überdeckung mit Regeln mit minimalen Kosten auswählen kann.

Der Benutzer beschreibt auf eventuell mehrdeutige Art und Weise die Abbildung bestimmter Konstrukte der Zwischensprache. Er braucht keinen Algorithmus zu programmieren, der die beste Abbildung eines Eingabebaums auswählt. Es ist günstig bei der Entwicklung einer Codegenerator-Beschreibung erst einen Teil der Maschinenbefehle zu spezifizieren, der groß genug ist, um die ganze Sprache zu übersetzen. Dies führt zu einem funktionsfähigen Übersetzer, welcher eventuell ineffizienten Code erzeugt. Später können nach und nach weitere Regeln hinzugefügt werden, was schließlich zu einem Übersetzer führt, der guten Code erzeugt.

Beg implementiert die Bestimmung einer Überdeckung mit minimalen Kosten unter Verwendung einer direkt codierten Version des dynamischen Programmierungs-Algorithmus [Emm88, ESL89].

Die Generierung eines Registerzuteilers ist von besonderer Bedeutung, da hier Handprogrammierung ziemlich schwer und lästig ist und weil Fehler in der Registerzuteilung manchmal schwer zu finden sind. Innerhalb der Regeln können die Eigenschaften eines Maschinenbefehls hinsichtlich der Registerzuteilung spezifiziert werden: die erlaubten Register für jeden Operanden, die durch Seiteneffekte veränderten Register und ob es sich um einen Zweiadreßbefehl handelt oder nicht. Zusätzlich wird der Registersatz der Zielmaschine beschrieben. Sogar das Doppelregister-Problem (wie z. B. auf IBM 370) kann behandelt werden.

Zwei Arten von Registerzuteilung sind möglich: Die *on the fly* Registerzuteilung kann nur einfache Registersätze behandeln. Sie ist jedoch sehr effizient und liefert für viele Maschinen zufriedenstellende Ergebisse. In manchen Fällen ist der allgemeine Registerzuteiler nötig, welcher eine Art von Vorschau durchführt und deshalb einen zusätzlichen Paß benötigt.

5.8. Reuse

Reuse ist eine Bibliothek wiederverwendbarer Module hauptsächlich für den Einsatz im Übersetzerbau [Gro87b]. Sie enthält Module oder abstrakte Datentypen, die fast in jedem Übersetzer gebraucht werden:

- eine dynamische Speicherverwaltung
- ein Modul für dynamische und flexible Felder
- ein Modul zur Speicherung variabel langer Zeichenketten
- ein Modul zur Zeichenkettenbearbeitung
- eine Bezeichnertabelle, welche Zeichenketten unter Verwendung eines Hashverfahrens eindeutig auf ganze Zahlen abbildet
- Module für oft verwendete Datenstrukturen wie Mengen von ganzen Zahlen oder binäre Relationen zwischen ganzen Zahlen ohne Beschränkung des Definitionsbereichs.

6. Erfahrungen

Dieser Abschnitt berichtet über die Erfahrungen des Einsatzes des Werkzeugkastens für zwei realistische Anwendungen.

6.1. Modula nach C Übersetzer

Die bisher größte Anwendung des Werkzeugkastens war die Generierung eines Modula-2 nach C Übersetzers [Mar90]. Das *Mtc* genannte Programm übersetzt Modula-2 Programme in lesbaren C Code ohne Einschränkung (sogar geschachtelte Prozeduren und Module). Es ist weitgehend automatisch generiert und folgt dem in Abschnitt 4 vorgeschlagenen Übersetzer-Modell. Anstelle einer Zwischensprache erzeugt *Mtc* C Code und benötigt deshalb keinen Codegenerator

zur Ausgabe von Maschinencode. Es enthält so viel von der semantischen Analyse wie für die Aufgabe gebraucht wird. Die semantische Analyse ist ziemlich vollständig und enthält die Behandlung der Gültigkeitsbereiche, Namensanalyse und Typbestimmung. Es fehlt die Überprüfung von Kontextbedingungen, da davon ausgegangen wird, daß nur korrekte Programme übersetzt werden. Tabelle 1 enthält die Größen der Spezifikationen und der generierten Quell-Module. Der Entwurf und die Implementierung von *Mtc* wurden im Rahmen einer Diplomarbeit mit einem Aufwand von 6 Mannmonaten durchgeführt. Das Programm ist stabil und hat bereits mehr als 100.000 Zeilen Modula-2 nach C übersetzt.

Phase	Spezifikation			Quell-Modul			Werkzeug	
	formal	Code	Summe	Def.	Impl.	Summe	Name	Referenzen
Lex. Analyse	392	133	525	56	1320	1376	Rex	[Gro87a, Gro88]
Zerteiler	951	88	1039	81	3007	3088	Ell	[GrV88, Gro88]
Syntaxbaum	189	51	240	579	2992	3571	Ast	[Gro89d]
Symboltabelle	115	938	1053	413	1475	1888	Ast	[Gro89d]
Sem. Analyse	1886	151	2037	9	3288	3297	Ag	[Gro89c]
Codegenerator	2793	969	3762	47	7309	7356	Estra	[Vie89]
Wiederverw.	-	-	-	819	2722	3541	Reuse	[Gro87b]
Sonstiges	-	-	-	698	3153	3851		
Summe	6326	2330	8656	2702	25266	27968		

Tabelle 1: Umfang der Spezifikationen und der Quellmodule von *Mtc*

Die Größe des Binärprogramms beträgt 300 K Bytes. Es läuft mit einer Geschwindigkeit von 810 Grundsymbolen (*tokens*) pro Sekunde oder 167 Zeilen pro Sekunde auf einem SUN/3 Arbeitsplatzrechner (MC 68020 Prozessor). Diese Zahlen berücksichtigen nur die Größe der übersetzten Module. Wenn man zusätzlich die (transitiv) importierten Definitionsmodule berücksichtigt, die ebenfalls lexikalisch, syntaktisch und semantisch analysiert werden, so erreicht man 1320 Grundsymbole pro Sekunde oder 418 Zeilen pro Sekunde. Zum Vergleich die Zahlen für zwei Übersetzer des Rechner-Herstellers: Der C-Übersetzer läuft mit einer Geschwindigkeit von 97 Zeilen pro Sekunde (ohne *Header*-Dateien) bzw. 163 Zeilen pro Sekunde (mit *Header*-Dateien) und der Modula-2-Übersetzer mit 48 Zeilen pro Sekunde. Die Laufzeit von *Mtc* ist folgendermaßen verteilt:

lex. + syn. Analyse + Baumaufbau	42 %
semantische Analyse	33 %
C Codegenerierung	25 %

Die semantische Analyse verbringt 95% der Zeit mit der Berechnung von Attributen mittels vom Benutzer spezifizierten Anweisungen und nur 5% für die Baumtraversierung bzw. für Besuchsaktionen. Für 11 Knotentypen sind fünf Besuche notwendig.

Mtc braucht ungefähr 360 K Bytes dynamischen Speicher pro 1000 Quellzeilen zur Speicherung des abstrakten Syntaxbaums, der Attribute und der Symboltabelle ohne Optimierung der Attributspeicherung. Weitere 600 K Bytes benötigt der von *Estra* generierte Transformator. Dies ist bei den heutigen Speicherkapazitäten erträglich. Es zeigt, daß im Gegensatz zu der in der Literatur vertretenen Meinung möglich ist, alle Attribute im Baum zu speichern. Wir tun dies sogar mit dem sogenannten Umgebungsattribut. Dies wird möglich, indem wir die Symboltabelle als abstrakten Datentyp in der Zielsprache programmieren. Die Implementierung ist zeit- und speichereffizient durch die Ausnutzung von Zeigersemantik und *structure sharing*.

6.2. Codegenerator für Modula-2-Übersetzer

In einer anderen Anwendung wurde der ursprünglich handgeschriebene Codegenerator des GMD Modula-2-Übersetzers *Mocka* durch zwei mit *Beg* erzeugte Versionen ersetzt. Die Zielmaschine war ein MC 68020 Prozessor. Die Spezifikation des Codegenerators umfaßt 1625 Zeilen und enthält 187 Regeln und 99 Zwischensprach-Operatoren. Zum Vergleich der Qualität des erzeugten Maschinencodes haben wir die Laufzeiten für eine Sammlung von Benchmark-Programmen gemessen (siehe Tabelle 2). Dabei ist *Mocka* der GMD Modula-2-Übersetzer mit handgeschriebenem Codegenerator, *Begra* und *Begfly* sind die mit *Beg* erzeugten Versionen mit dem allgemeinen bzw. mit dem *on the fly* Registerzuteiler, und *Sun* der SUN Modula-2-Übersetzer Version 1.0. Im Durchschnitt ist der Code von mit *Beg* erzeugten Codegeneratoren genau so gut, wie der des handgeschriebenen Codegenerators.

Tabelle 3 vergleicht die Übersetzungszeiten für ein 1116 Zeilen langes Programm. Alle Messungen wurden auf einer SUN 3/60 durchgeführt, die gemessenen Zeiten waren *user* Zeiten. Die Größe des Codegenerators nahm von 5140 Zeilen (17,117 Grundsymbole) für die handgeschriebene Version auf 9574 Zeilen (27,044 Grundsymbole) zu.

7. Zusammenfassung und Ausblick

Wir haben einen Werkzeugkasten mit Übersetzerbau-Werkzeugen vorgestellt, womit sich Übersetzer für Programmiersprachen weitgehend automatisch generieren lassen. Die Übersetzerbau-Werkzeuge unterstützen die Konstruktion nahezu aller Übersetzerphasen. Die Werkzeuge sind sehr mächtig, flexibel und weitgehend unabhängig von einander. Besonders hervorzuheben sind die praktische Brauchbarkeit der Werkzeuge, der deutlich reduzierte Erstellungsaufwand für Übersetzer und die hohe Qualität der generierten Übersetzer. Von der Effizienz her sind die Werkzeuge konkurrenzfähig zur Programmierung von Hand. Sie unterstützen einen breiten Bereich von Übersetzerstrukturen und erlauben die Konstruktion von Übersetzern mit Produktionsqualität. Erste realistische Anwendungen zeigen die ausgezeichnete Leistungsfähigkeit der Werkzeuge.

Die Übersetzerbau-Werkzeuge eignen sich für viele Aufgabenstellungen, die über die Konstruktion von reinen Übersetzern hinausgehen. Sie gestatten beispielsweise die Implementierung von Präprozessoren, die Spracherweiterungen und Sprachdialekte auf Standardsprachen abbilden. Wie eines unserer Anwendungsbeispiele zeigte, lassen sich Umsetzer von einer Quellsprache in eine andere erstellen. Weiterhin ist etwa die Generierung von Prüfprogrammen für Programmierkonventionen möglich.

	Perm	Towers	Queens	Intmm	Mm	Puzzle	Quick	Tree	Bubble	FFT	Summe
Mocka	40	58	37	53	103	285	32	72	56	152	888
Begra	42	57	35	54	104	297	32	58	56	153	888
Begfly	42	57	34	54	102	299	33	56	56	151	884
Sun	67	90	28	83	101	263	41	81	63	150	967

Tabelle 2: Vergleich der Codequalität (Laufzeiten in *ticks* = 1/60 Sek.)

Mocka	2.9
Begfly	3.2
Begra	3.9
Sun	25.4

Tabelle 3: Vergleich der Übersetzungszeiten (in Sek.)

Die Optimierung der Attributspeicherung des Werkzeugs *Ag* werden wir verbessern, damit Attribute gegebenenfalls auch als globale Variable und globale Keller implementiert werden können. Außerdem sollte die Transformation von Grammatiken, die nicht die OAG-Eigenschaft besitzen, in OAG-Grammatiken automatisiert werden.

Für das Werkzeug *Estra* ist ein Redesign geplant. Die Spezifikationssprache läßt sich vereinfachen, und die Integration des Werkzeug mit *Ast* kann verbessert werden. Die Effizienz der generierten Transformations-Module läßt sich sowohl hinsichtlich Laufzeit als auch hinsichtlich Speicherverbrauch verbessern.

Die Optimierungsphase eines Übersetzers sollte selbstverständlich auch unterstützt werden. Dies kann entweder durch einen wiederverwendbaren sprachunabhängigen Optimierer, durch einen parameterisierbaren Optimierer oder durch einen Optimierergenerator geschehen.

Das Werkzeug *Beg* wird in folgende Richtungen erweitert werden: Generierung eines globalen Registerzuteilers, Unterstützung der Befehlsumordnung (*instruction scheduling*) und einer Einrichtung zur direkten Generierung von binärem Objektcode.

Danksagung

Wir danken allen die zur Entstehung des Werkeugkastens und dieses Aufsatzes durch aktive Mitarbeit oder durch ihre Ideen beigetragen haben: Michael Besser, Carsten Gerlhof, Bob Gray, Eduard Klein, Rudolf Landwehr, Matthias Martin, Thomas Müller, F. W. Schröer, Dirk Schwartz-Hertzner, Doris Vielsack, Bertram Vielsack und William M. Waite.

Literatur

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[DeP82] F. DeRemer and T. Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 615-649.

[Emm88] H. Emmelmann, Automatische Erzeugung effizienter Codegeneratoren, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1988.

[ESL89] H. Emmelmann, F. W. Schröer and R. Landwehr, BEG - a Generator for Efficient Back Ends, *SIGPLAN Notices 24*, 7 (July 1989), 227-237.

[Emm89a] H. Emmelmann, Automatische Erzeugung effizienter Codegeneratoren, GMD-Studie Nr. 158, GMD Forschungsstelle an der Universität Karlsruhe, 1989.

[Emm89b] H. Emmelmann, BEG - A Back End Generator - User Manual, Arbeitspapier Nr. 420, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1989.

[Gro87a] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[Gro87b] J. Grosch, Reusable Software - A Collection of Modula-Modules, Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1987.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[Gro89a] J. Grosch, Efficient Generation of Lexical Analysers, *Software—Practice & Experience 19*, 11 (Nov. 1989), 1089-1103.

[Gro89b] J. Grosch, Efficient and Comfortable Error Recovery in Recursive Descent Parsers, Compiler Generation Report No. 19, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1989.

[Gro89c] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro89d] J. Grosch, Ast - A Generator for Abstract Syntax Trees (Revised Version), Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro89e] J. Grosch, Tool Support for Data Structures, Compiler Generation Report No. 17, GMD Forschungsstelle an der Universität Karlsruhe, Nov. 1989.

[Gro90] J. Grosch, Object-Oriented Attribute Grammars, in *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmanci and E. Gelenbe (ed.), Cappadocia, Nevsehir, Turkey, Oct. 1990, 807-816.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13*, 3 (1980), 229-256.

[Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[Röh76] J. Röhrich, Syntax-Error Recovery in LR-Parsers, in *Informatik-Fachberichte*, vol. 1, H.-J. Schneider and M. Nagl (ed.), Springer Verlag, Berlin, 1976, 175-184.

[Röh80] J. Röhrich, Methods for the Automatic Construction of Error Correcting Parsers, *Acta Inf. 13*, 2 (1980), 115-139.

[Röh82] J. Röhrich, Behandlung syntaktischer Fehler, *Informatik Spektrum 5*, 3 (1982), 171-184.

[Vie89] B. Vielsack, Spezifikation und Implementierung der Transformation attributierter Bäume, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, June 1989.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

Inhalt