# WELCHER ART ERGEBNISSE ERWARTET DER COMPILERBAU VON DER THEORETISCHEN INFORMATIK?[+)]

H. LANGMAACK

Wohl in allen Naturwissenschaften und in vielen Ingenieurwissenschaften kennen wir die Aufspaltung in praktische und theoretische Forschung. Es gibt z.B. die Experimentalphysik und die theoretische Physik, es gibt die experimentelle Chemie und die theoretische Chemie, wir haben die praktische und die theoretische Elektrotechnik. Auch in der Informatik, die wir als Ingenieurwissenschaft verstehen müssen - gewiß als recht eigenartige Ingenieurwissenschaft -, haben wir die praktisch orientierte und die theoretische Informatik. Unter praktisch orientierter Informatik ist nicht angewandte Informatik zu verstehen, die Methoden der Informatik auf andere Disziplinen anwendet. Die praktisch orientierte Informatik kann durchaus in der engeren Informatik verharren. Arbeitsbereiche wie die Programmiersprachenentwicklung, der Übersetzerbau und die Betriebssystemkonstruktion zählen dazu, Bereiche, die sich also darum bemühen, die wirklichen Rechenanlagen einzusetzen und kunstvoll mit ihnen umzugehen. Die theoretische Informatik befaßt sich dagegen mit gedanklichen Modellen von Rechenanlagen und Rechenprozessen.

Seitdem die Veranstalter dieser Tagung mich gebeten haben, über das angekündigte Thema zu sprechen, habe ich mit zahlreichen Experten für

[+)] Der Autor möchte Herrn Prof. O.J.Dahl, Oslo, für die zahlreichen Gespräche danken.

Programmierung, Programmiersprachen und Compilerbau diskutiert, gewissermaßen, um Stoff zu sammeln. Um es vorweg zu sagen: Die Diskussionen sind nicht ergiebig gewesen. Zumindest hat man mir kaum konkrete Probleme nennen können, die sich zum unmittelbaren Gebrauch des Theoretikers eignen. Von einigen, sehr ernst zu nehmenden Kollegen ist sogar ganz stark bezweifelt worden, ob die Theoretiker den Praktikern in der Form von Arbeitsteilung überhaupt zur Hilfe kommen könnten. Unter Arbeitsteilung ist zu verstehen: Der praktische Übersetzerkonstrukteur formuliert seine Probleme im Rahmen präziser, mathematischer Modelle, und der theoretische Informatiker führt die Beweise.

An dieser kritischen Auffassung ist natürlich sehr vieles richtig. Wenn man als Übersetzerbaufachmann zu einer abgerundeten Problemlösung kommen will, in der die Antworten ein mit präzisen Definitionen und Beweisen versehenes Gebäude bilden, dann besteht in der Tat die Hauptarbeit darin, aus dem Wust von Techniken, verschwommenen Vorstellungen und ungenauen Redeweisen ein klares Modell herauszupräparieren. Jetzt noch den Theoretiker herbeizurufen, ist häufig überflüssig. Ein Praktiker, der sogar in der Lage ist, klare Modelle zu formulieren, kann auch den Rest der Arbeit erledigen.

Wenn in dem idealen Sinne der Arbeitsteilung die Hilfe des Theoretikers gar nicht möglich ist, dann bleibt die Frage offen, ob der Praktiker dem Theoretiker nicht Gebiete nennen könnte, wo eine Modellbildung sinn- und hoffnungsvoll wäre. Da ist natürlich die Frage angebracht, ob dieses Vorgehen den landläufigen Theoretiker nicht überforderte. Er müßte regelrecht in die Programmiertechniken des Praktikers voll einsteigen, um daraus das Abstrahieren zu versuchen. Dieser Theoretiker wäre gar kein Theoretiker mehr in dem Sinne, daß er es nur nötig hätte, auf vorfabrizierte Probleme zu antworten. Die Frage ist auch, ob der Theoretiker das überhaupt will; er wird sich überlegen, ob er nicht auf einfachere Weise ohne Umwege über harte praktische Probleme schneller zu Resultaten gelangt, die der Veröffentlichung wert sind. Dem Theoretiker Arbeitsgebiete vorzuschlagen, legt dem Ratgeber außerdem eine große Verantwortung auf. Denn das Risiko des Mißerfolgs ist natürlich groß, wenn nicht einmal das Modell feststeht, in dem die Antworten gesucht werden.

In dem kürzlich erschienenen und sehr informativen Band "Compiler Construction, an Advanced Course" /1/, an dem zahlreiche international bekannte Spezialisten des Compilerbaus mitgewirkt haben, findet man kaum an Theoretiker gerichtete Aufforderungen zu neuen Forschungen. Wenn

aufgefordert wird, dann eher zu Experimenten, um ein Gespür zu bekommen, in welche Richtung Compilertechniken weiterentwickelt werden sollten. Solche Experimente sind natürlich in erster Linie Praktikern vorbehalten.

Trotz aller dieser Bedenken will ich jedoch versuchen, Gebiete anzuführen, auf denen eine Zusammenarbeit von Praktikern mit Theoretikern lohnend erscheint. Man möge mir verzeihen, daß die Auswahl an persönlichen Interessen orientiert sein wird. Von den Zweiflern am Unterfangen ist mir vorhergesagt worden, daß es schwer fallen dürfte, bereits allgemein akzeptierte Problemkreise zu nennen.

Um der im Titel genannten Frage näher zu rücken, sollte man sich zunächst einmal fragen: Wo hat die theoretische Informatik bisher schon Ergebnisse geliefert, die für den Compilerbau von besonderem Nutzen und Interesse gewesen sind? Hier ist als Musterbeispiel die Theorie der Analyseverfahren für kontextfreie Grammatiken zu nennen. Diese Theorie wurde durch Erscheinen des ALGOL 60-Berichts /2/ angeregt. Die Grundaufgaben wurden Anfang der sechziger Jahre durch E.T.Irons /3/ und M.Paul /4/ formuliert, die ihre Ideen aus der praktischen Konstruktion von ALGOL 60-Übersetzern empfangen haben. Weiterentwickelt worden ist die Theorie dann mit den Operatorpräzedenzgrammatiken von R.W.Floyd /5/, mit den kontextbeschränkten Grammatiken von J.Eickel /6/ und R.W.Floyd /7/, den von links nach rechts analysierbaren (LR (k) ) Grammatiken von D.E.Knuth /8/, den Präzedenzgrammatiken von N.Wirth und H.Weber /9/ und den von links nach rechts im top down-Verfahren analysierbaren (LL (k) ) Grammatiken von J.M.Foster /10/, P.M.Lewis und D.J.Stearns /11/. In dem Nachschlagewerk von A.V.Aho und J.D.Ullman /12/ ist die gesamte Theorie Anfang der siebziger Jahre in schöner Weise zusammengefaßt worden.

Für den praktischen Compilerbau werden nur wenige Grammatiktypen von Bedeutung bleiben. Dazu gehören die LR (0)-, die LL (1)-, die (einfachen) SLR (1)-, die (look ahead-) LALR (1)- und die LR (1)- Grammatiken. Das Auftreten der einfachen /13/ und der look ahead- LR (k)- Grammatiken /14/ in der Theorie ist dem Einwirken der Praxis zu verdanken. Sie sind bei dem Bemühen entstanden, parser-Generatoren für LR (k)- Grammatiken zu erzeugen, wobei man einige sehr aufwendige Programmteile kurzgeschlossen hat.

Ich meine, daß man heute nicht mehr zu viel Arbeit in die Theorie der Analyseverfahren für kontextfreie Grammatiken investieren sollte. Wenn,

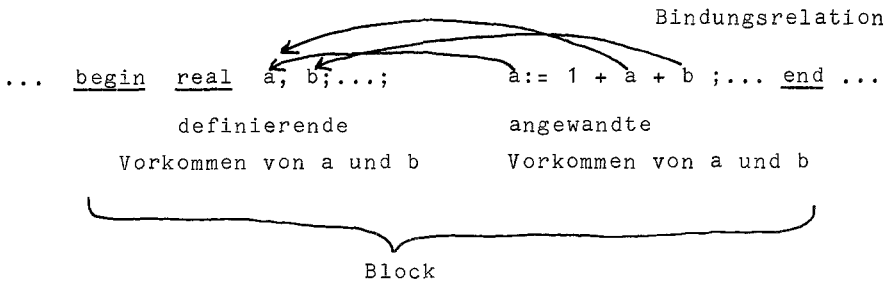
dann bestenfalls für Grammatiktypen innerhalb des LR (1)- Bereichs.Ich möchte vor allem zwei Gründe näher ausführen: Einmal ist die Entwurfstechnik für Programmiersprachen weitergeschritten - man legt nicht mehr bloß kontextfreie Grammatiken zugrunde -, zum anderen hat die Theorie den schwierigen Komplex der Codeerzeugung bislang nur sehr stiefmütterlich behandelt.

Bei den neueren Programmiersprachenentwürfen ist vornehmlich ALGOL 68/15/ zu nennen. Ihm liegt eine van Wijngaarden-Grammatik zugrunde. Solche zweistufigen Grammatiken lassen sich zwar als "pseudo"-kontextfrei mit unendlich vielen Produktionen ansehen, die bekannte Theorie der Analyseverfahren läßt sich aber trotzdem nicht leicht ausdehnen. Deshalb sind bislang nur wenige Arbeiten über automatische Analyseverfahren für zweistufige Grammatiken erschienen; ich kann hier nur C.H.A.Koster /16,17/, H.J.Bowlden /18/ und D.A.Watt /19/ erwähnen.

Ebenso existieren zur Theorie der Codeerzeugung nur wenige Arbeiten. Als wichtigste Publikation pflegt man gern T.R.Wilcox /20/ zu zitieren, der 1971 den Versuch gemacht hat, den Vorgang der Codeerzeugung generell und prinzipiell zu erfassen. Dabei ist das Schwierige nicht, daß sich die Struktur von Rechenanlagenund Maschinenbefehlen gewandelt hat; auch T.R.Wilcox legt die klassische Struktur zugrunde. Er sieht die Codeerzeugung im Prinzip als zweistufigen Prozess, er erkennt einen Übersetzungs- und einen anschließenden Codierungsprozess. Der Übersetzer geht von zwei Eingaben aus, einem in linearer Form dargestellten abstrakten Programmbaum und einer Symbolliste, beide gewonnen durch syntaktische Analyse in einem Vorlauf. Der Übersetzer erzeugt Befehle einer sog. Quellsprachenmaschine, die vom Codierer in Maschinencode verwandelt werden. Diese Sicht zeigt deutlich, daß der Strukturbaum eines Programms, der ihm durch eine kontextfreie Grammatik aufgeprägt ist, hinter den abstrakten Programmbaum zurücktritt; der Strukturbaum ist zu sehr mit Künstlichem behaftet, als daß er für das Aufhängen der Semantik adäquat erscheint. Dementsprechend ist in der Übersetzerbaupraxis auch die Scheu gewichen, die durch Sprachdefinitionen mitgegebenen Grammatiken umzuschreiben, um bequemer analysieren und um besseren Code gewinnen zu können. Die Praxis verlangt also nicht mehr um jeden Preis, mit dem Analyseproblem einer gegebenen Grammatik im strengen Sinne fertig zu werden.

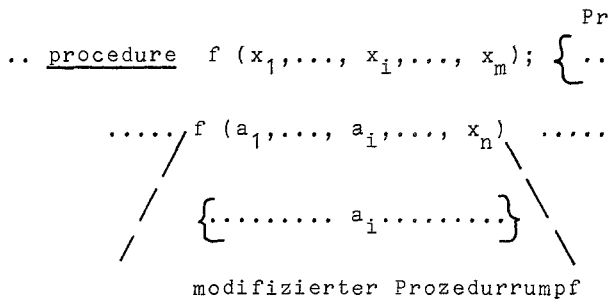
Ein anderer Themenkomplex, den die Theorie mehr beachten sollte, ist der der Bindung angewandter Vorkommen an definierende Vorkommen von Identi-

fikatoren in Programmen. Die Bindungsrelation in ALGOL 60 z.B. dürfte jedem Informatiker bekannt sein.




Um übersetzbares Programm zu sein, genügt nicht, daß es vermöge der kontextfreien Grammatik des ALGOL 60-Berichts auf das Axiom <program> reduziert werden kann. U.a. muß es auch folgenden Forderungen genügen: Jeder angewandt vorkommende Identifikator hat ein zugehöriges definierendes Vorkommen, und Doppeldefinitionen pro Block sind verboten.

Probleme wirft die Bindungsrelation vor allem in Zusammenhang mit Prozeduraufrufen auf. Um die Kopierregel von ALGOL 60 nicht naiv




sondern vernünftig anzuwenden, ist zuvor eine zulässige Umbenennung von Identifikatoren vorzunehmen, so daß das Ausgangsprogramm ausgezeichnet wird, d.h. daß verschiedene Vorkommen von Identifikatoren durch verschiedene Identifikatoren bezeichnet sind. Die Kopierregel hat ihr Vorbild im $\lambda$-Kalkül von A.Church /21/.

Für eine Prozedur f interessiert nun häufig das Problem: Ist f aktuell rekursiv? D.h. gibt es Eingabedaten, so daß f ein wiederholtes Mal aktiviert wird, ohne daß eine vorangegangene Aktivierung von f beendet ist? Der Übersetzerkonstrukteur ist nämlich an nicht aktuell rekursiven Prozeduren stark interessiert, weil diese eine wesentlich günstigere

Implementierung auf Rechenanlagen erlauben. Leider ist das Problem algorithmisch nicht generell lösbar. Schuld daran ist die Arithmetik, die Bestandteil fast aller realistischen Programmiersprachen ist.

Die Frage ist darum: Kann man die Arithmetik in irgendeiner Weise überspielen? Etwa indem man die Programme formal oder schematisch betrachtet? Dann eröffnet sich das neue Problem: Ist f formal rekursiv? Die Frage ist natürlich, ob dieses Abweichen vom ursprünglichen Problem sinnvoll ist. Es ist deshalb sinnvoll, weil die Aussage "f ist nicht formal rekursiv" impliziert: "f ist nicht aktuell rekursiv". Wenn also die formale Rekursivität entscheidbar sein sollte und wenn durch den Entscheidungsalgorithmus festgestellt wird, daß f nicht formal rekursiv ist, dann weiß man auch, daß f nicht aktuell rekursiv ist, womit man die gewünschte Information in der Hand hat. Leider ist auch das neue Problem algorithmisch nicht generell lösbar. Schuld daran ist jetzt die in der Kopierregel verlangte zulässige Umbenennung von Identifikatoren /22/. Die formal rekursiven Prozeduren sind zwar effektiv aufzählbar, die interessanteren nicht formal rekursiven Prozeduren sind es dagegen nicht.

Um der Entscheidbarkeit näher zu rücken, kann man daran denken, die Kopierregel naiv anzuwenden. Das führt zu dem weiteren Problem: Ist f naiv formal rekursiv? Dieses Problem ist algorithmisch generell lösbar, und das zugehörige Entscheidungsverfahren kann in nützlicher Weise vom Compiler eingesetzt werden.

Die formale, schematische Betrachtungsweise muß natürlich gerechtfertigt werden. M.S. Paterson /23/ rechtfertigt seine Untersuchungen über Programmschemata mit der Antwort, daß für realistische Programmiersprachen fast alle interessanten Eigenschaften effektiv unentscheidbar seien. Dazu zählen etwa die Korrektheit, die Äquivalenz und das Terminieren von Programmen und die Erreichbarkeit und Rekursivität von Unterprogrammen.

Man kann die schematische Vorgehensweise aber noch viel schlagender rechtfertigen. Denn der Übersetzerkonstrukteur betrachtet zu übersetzende Programme auch nur schematisch. Die Übersetzung ist nämlich nicht durch die Definition der Programmiersprache eindeutig festgelegt. Vielmehr kann eine Anweisung des Quellprogramms in verschiedene Anweisungen des Zielprogramms übertragen werden, die in verschiedener Weise effizient sind. Weil es algorithmisch unentscheidbar ist, ob eine Anweisung häufig, selten oder gar nicht angesprochen wird, muß auch der praktische Übersetzerkonstrukteur den aktuellen Lauf der Rechnung vernachlässigen.

Eine Kopierregel wie für Prozeduraufrufe in ALGOL 60 gibt es auch bei Makroprozessoren, eine Umbenennung von Identifikatoren wird aber nicht verlangt. Trotzdem ist für einige Prozessoren algorithmisch unlösbar, ob die jeweilige Makroexpansion abbrechen wird. Die Umbenennungsvorschrift ist also nicht in jedem Falle schuld an der Unentscheidbarkeit. Das gilt insbesondere für den general purpose macro generator GPM von C. Strachey /24, 25/. Eine Makrodefinition für GPM hat die Gestalt

§DEF, ..... , .... [.... ] ;

Makro-name — Makro-rumpf

während ein Makroaufruf so aussieht:

§ ..... , ..... , ..... , ..... ;

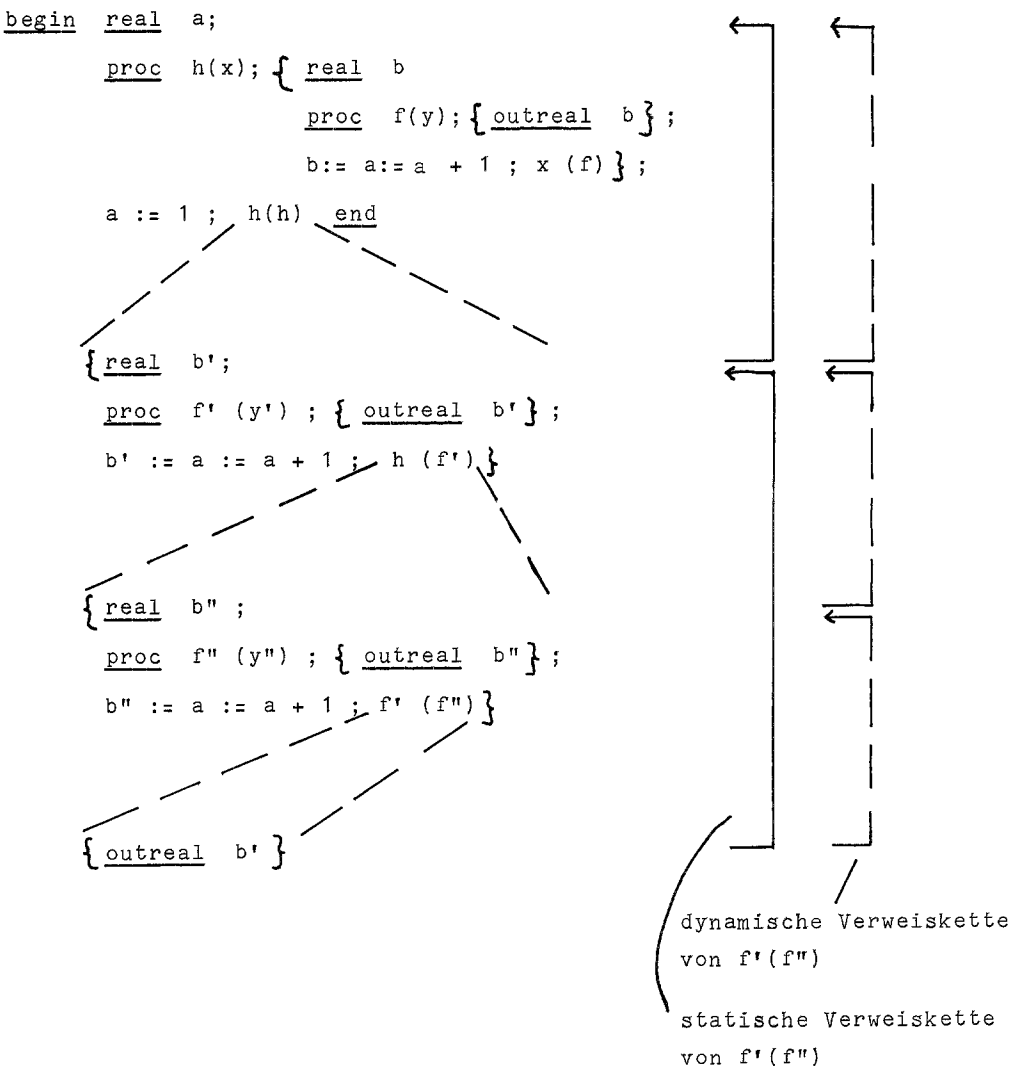
Makro-name — erster aktueller Parameter — letzter aktueller Parameter

i-te formale Parameter in einem Makrorumpf werden durch ~i dargestellt, und zusammengehörige eckige Klammern [ ] auf äußerstem Niveau in aktuellen Parametern sind beim Kopierprozeß fortzulassen.

Ähnliche Untersuchungen ließen sich auch für andere Makroprozessoren anstellen, z.B. für STAGE 2 von W.M. Waite /26/. Die Unentscheidbarkeit des Abbruchs der Makroexpansion ist gewiß unbefriedigend, und man sollte daher nach entscheidbaren Expansionsmechanismen suchen. Wählt man die Mechanismen zu einfach, dann vermindern sich allerdings die Steuerungsmöglichleiten für Selbstaufrufe. Gleicht man den Verlust durch Makrovariable, Makrowertzuweisungen und bedingte Makroanweisungen aus /27/, dann gerät man wegen der verwendeten Arithmetik unversehens wieder in die Unentscheidbarkeitszone.

Die Überlegungen zur "most recent"-Eigenschaft von Programmen stellen ein sehr schönes Beispiel theoretischer Untersuchungen dar, die aus dem unmittelbaren Studium von Implementierungstechniken für übersetzte ALGOL 60-Programme angeregt worden sind. In einer frühen Arbeit /28/ hat

E.W.Dijkstra behauptet, alle ALGOL 60-Programme hätten die genannte Eigenschaft. Worum geht es? Wir betrachten das folgende Programm P.




Dieses Programm besitzt nicht die "most recent"-Eigenschaft, weil die statische Verweiskette des Eintrags des Aufrufs f'(f") (activation record) der Prozedur f nicht auf den jüngsten Eintrag h(f') der Prozedur h sondern auf das vorangehende h(f) zeigt. Die korrekte Ausgabe ist 2. P ist in gewissem Sinne das kleinste Programm, das die "most recent"-Eigenschaft nicht erfüllt, wie man aus einer Arbeit von C.L.McGowan/29/ entnimmt. Während die aktuelle "most recent"-Ei-genschaft allein schon

wegen der beteiligten Arithmetik algorithmisch unentscheidbar ist, hat P.Kandzia /30/ die formale "most recent"-Eigenschaft als entscheidbar nachgewiesen, obwohl die Umbenennung von Identifikatoren beim Kopierprozeß verlangt wird.

Die Laufzeitsysteme von sehr vielen ALGOL 60-Compilern sind so konstruiert, als ob alle Programme die "most recent"-Eigenschaft hätten. Auf obiges Programm P angewendet, würden sie die inkorrekte Ausgabe 3, den Wert von b" statt von b', liefern. Die "most recent"-Eigenschaft ist nun nicht nur theoretisch interessant, sondern hat auch praktische Bedeutung. Denn die Speicherbereiche für Aufrufeintragungen einer gewissen Prozedur werden dann nur kellerartig beansprucht. Wenn bei Arbeitsspeicherüberlauf momentan nicht beanspruchte Eintragungen auf Hintergrundspeicher verdrängt werden, etwa bei virtuellem Speicherbetrieb, denn verhält sich ein Programm mit "most recent"-Eigenschaft offensichtlich günstiger als eines ohne die Eigenschaft, weil letzteres häufigeren Seitenwechsel verlangt. Wenn man ferner bedenkt, daß Benutzer nur selten gegen die in dieser Weise fehlerhaften Laufzeitsysteme protestiert haben, kann man die Idee verfolgen, die "most recent"-Eigenschaft grundsätzlich für statthaft zu erklären. Die zugehörige "most recent"- formale Rekursivität von Prozeduren und auch andere Programmeigenschaften dürften sich als entscheidbar erweisen. Man erkennt, wie eine mehr oder minder starke Manipulation an der Semantik von Programmiersprachen zu erheblichen Vorteilen für das Übersetzen und Codeerzeugen führen kann.

Übrigens gibt es verschiedene "most recent"-Eigenschaften, die alle entscheidbar sind. Es ist denkbar, daß sich in vernünftiger Weise eine ganze Hierarchie schwächer werdender "most recent"-Eigenschaften einführen läßt, womit man das allgemeine Programmverhalten approximieren könnte.

Um das Ziel besserer Übersetzungen zu erreichen, kann man auch den Weg verfolgen, Teilklassen von Programmen abzugrenzen und dann zu beweisen, daß sie wünschenswerte Eigenschaften besitzen. Eine solche Teilklasse ist z.B. die der monadischen Programme, wo für jeden Prozeduraufruf $a_o(a_1,\ldots, a_n)$ gilt: wenn $a_i$ und $a_j$ formale Parameter sind, dann sind sie gleich. Eine andere Teilklasse ist die der Programme mit endlichen Arten im Sinne von ALGOL 68, d.h. der Programme, die ohne den mode-Deklarator geschrieben werden können. Die formale Erreichbarkeit und Rekursivität von Prozeduren, die Makroprogrammeigenschaft und die formale Äquivalenz von Programmen sind in diesen Teilklassen vermutlich ent-

scheidbare Eigenschaften; die bisherigen Untersuchungen /22, 25, 31/ deuten darauf hin. Die bislang betrachteten Fälle haben zu Entscheidbarkeitsfragen für reguläre Systeme /32/, stack-Systeme /33/ und Baum - grammatiken /34/ geführt. Es dürfte eine Theorie nützlich sein, die stack-Systeme auf Bäume ausdehnt, ähnlich wie die regulären Systeme auf Bäume erweitert worden sind.

Für das Abgrenzen von Programmteilklassen ist wichtig, daß die Definitionstexte für Anwender einfach und verständlich sind. Wenn die Regeln effizienten Programmierens kompliziert sind, sinkt erfahrungsgemäß ihr Gebrauchswert.

Ein ganz wichtiger Problemkreis ist der der hängenden Bezüge (dangling references), wie P.Wegner /35/ ihn nennt. In ALGOL 60 tritt er gar nicht auf, weil ALGOL 60 keine Bezüge, Marken oder Prozeduren als Inhalte von Variablen und Werte von Prozeduren kennt. Als Inhalte und Werte gibt es nur die problemlosen Daten: ganze Zahlen, reelle Zahlen und Wahrheitswerte. Dementsprechend ist die Speicherverwaltung zur Laufzeit einfach: Bei Block- oder Prozedurende können die reservierten Speicherbereiche ohne Vorbehalte freigegeben werden, das Laufzeitsystem arbeitet kellerartig, es darf die deletion-Strategie /36/ verfolgen.

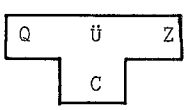
Anders in den Programmersprachen LISP /37/, EULER, ALGOL 68, GEDANKEN /38/ und OREGANO /39/, in denen auch Bezüge, Marken und Prozeduren wie Daten behandelt werden. Dort können hängende Bezüge auftreten, wenn Bezüge, Marken oder Prozeduren an Variablen oder Prozeduren zugewiesen werden, so daß erstere eine kleinere Reichweite (scope) als letzere besitzen. J.C.Reynolds /38/ und D.M.Berry /39/ vertreten daher den Standpunkt, daß Speicherplätze erst dann wieder freizugeben seien, wenn nicht mehr auf sie zugegriffen werden kann, und daß man sinnvollerweise die retention-Strategie verfolgen müsse. Abgesehen davon, daß vollständige Sprachen /38/ dafür sorgen, daß Laufzeitfehler weniger häufig gemacht werden, erhebt sich die Frage, ob die retention-Strategie mächtiger als die deletion-Strategie ist. M.J.Fischer /40/ hat bewiesen, daß beide Strategieen gleichmächtig sind, sogar in dem starken Sinne, daß es ein effektives Verfahren gibt, womit jedes Programm in ein anderes verwandelt wird, welches auch mit deletion-Strategie korrekt ausgeführt wird.

Dies wird nicht die einzig mögliche Aussage über den Mächtigkeitsvergleich sein. Da die Programmiersprachenentwicklung stark zu den vollständigen Sprachen tendiert, sollte man sich überlegen, ob und in welchem

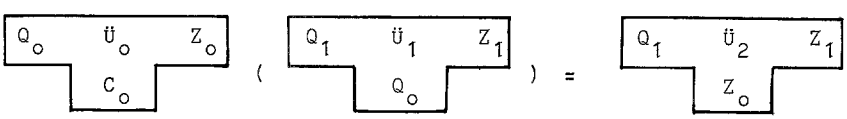
Sinne retention echt mächtiger als deletion ist, ähnlich wie man in den letzten Jahren die Wirksamkeit des goto theoretisch beleuchtet hat /41, 42/.

Zum Abschluß möchte ich die Aufmerksamkeit auf die bootstrapping-Technik im Compilerbau lenken. Sie ist seit langem bekannt, die Programmiersprache PASCAL /43, 44/ ist in dieser Technik auf vielen Rechenanlagen erfolgreich implementiert worden, jedoch fehlt bisher eine ansprechende Theorie. Um eine Programmiersprache S zu implementieren, konstruiert man eine aufsteigende Folge von Teilsprachen $S_o$, $S_1$,..., $S_n$ = S, wobei in jedem Schritt neue Sprachelemente hinzutreten. Es soll in Maschinensprachen $M_o$, $M_1$,..., $M_n$ übersetzt werden, wobei der Einfachheit halber alle $M_i$ von derselben Rechenanlage R verstanden werden mögen. $M_{i+1}$ werde als Verbesserung von $M_i$ angesehen, z.B. durch lokale Optimierung, die zu neuen Kombinationen von Maschinenbefehlen führt, oder durch globale Optimierung, etwa durch Vorwegziehen identischer Programmstücke oder durch rekursive Adressenberechnung. Dabei kann das Vorhandensein zusätzlicher Akkumulatoren, Adreß- und Indexregister ausgenutzt werden.

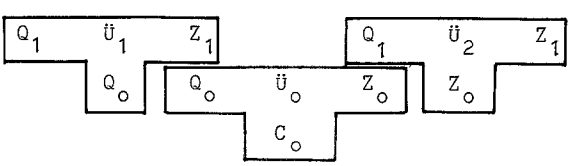
Einen Übersetzer Ü, geschrieben in der Sprache C, der Programme der Quellsprache Q in die Zielsprache Z überträgt, stellt man nach W.M.Mc-Keeman /45/ im T-Diagramm



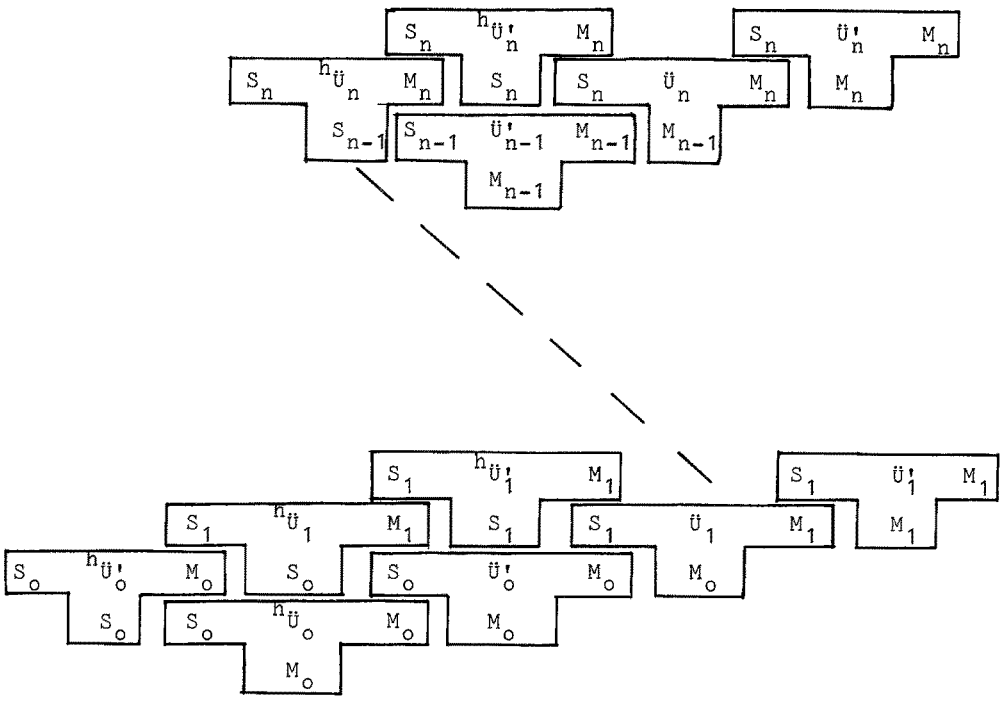

dar, und eine Gleichung (Übersetzung eines Übersetzers)




repräsentiert man so:




Die Generierung eines Übersetzers von $S_n$=S nach $M_n$, geschrieben in $M_n$, kann nach folgendem Schema geschehen:




Dieses Schema geht von einer Sequenz handgeschriebener Übersetzer $^{h}Ü_{o}$, $^{h}Ü'_{o}$, $^{h}Ü_{1}$, $^{h}Ü'_{1}$,..., $^{h}Ü_{n}$, $^{h}Ü'_{n}$ aus, und es erlaubt, ein komplexes Übersetzerprojekt übersichtlich zu organisieren. In den Übersetzern $^{h}Ü_{i}$ treten jeweils neue Quellsprachelemente hinzu und verbesserter Maschinencode wird erzeugt, in den $^{h}Ü'_{i}$ wirken sich diese Verbesserungen rekursiv auf die Übersetzungsprozesse selbst aus. Eine mathematische Theorie über die bootstrapping-Technik und deren Effizienzverbesserung wäre sehr interessant und wünschenswert.

Die bootstrapping-Technik bietet außerdem einen guten Ansatz, um die Korrektheit von Übersetzern zu beweisen, ein Problem, dessen Bedeutung wohl nicht erst betont zu werden braucht. Weil in jedem Schritt $^{h}Ü_{i}$, $^{h}Ü'_{i}$ geringe Modifikationen hinzutreten, dürften die Beweise auch von Hand mit vertretbarem Aufwand durchzuführen sein. Bezüglich der Methoden und der Strenge sollte man sich von Numerikern leiten lassen, die ja auch stets Eigenschaften über ihre Algorithmen beweisen müssen und dies per Hand tun. Die Genauigkeit automatischer Beweismethoden scheint nicht in jedem Falle erforderlich zu sein.

Literatur:


/1/ F.L.Bauer, J. Eickel (Ed.): Compiler Construction, an advanced Course. Lecture Notes in Computer Science 21, Springer, Berlin-Heidelberg-New York 1974

/2/ P.Naur (Ed.): Report on the algorithmic Language ALGOL 60. Num. Math. 2, 106-136 (1960)

/3/ E.T. Irons: A syntax-directed Compiler for ALGOL 60. Comm.ACM 4, 51-55 (1961)

/4/ M.Paul: ALGOL 60 Processors and a Processor Generator. Information Processing 1962, 439-447, North Holland, Amsterdam 1963

/5/ R.W.Floyd: Syntactic Analysis and Operator Precedence. J. ACM 10, 316-333 (1963)

/6/ J.Eickel: Generation of Parsing Algorithms for Chomsky 2-Type Languages. Math. Inst. Techn. Univ. München, Bericht 6401, 1964

/7/ R.W.Floyd: Bounded Context syntactic Analysis. Comm. ACM 7, 62-67 (1964)

/8/ D.E.Knuth: On the Translation of Languages from left to right. Inform. and Control 8, 607-639 (1965)

/9/ N.Wirth, H.Weber: EULER - a Generalisation of ALGOL and its formal definition, Parts 1 and 2. Comm. ACM 9, 13-23, 89-99 (1966)

/10/ J.M.Foster: A Syntax improving Device. Computer J. 11, 31-34 (1968)

/11/ P.M.Lewis, R.E.Stearns: Syntax directed Transduction.J. ACM 15, 464-488 (1968)

/12/ A.V.Aho, J.D.Ullman: The Theory of Parsing, Translation, and Compiling. Vol.I and II, Prentice Hall, Englewood Cliffs 1972 and 1973

/13/ F.L. De Remer: Simple LR(k) Grammars. Comm.ACM 14, 453-460 (1971)

/14/ W.R. La Londe, E.S.Lee, J.J.Horning: An LALR(k) Parser Generator. Proc. IFIP Congress 71, TA-3, North Holland, Amsterdam 1971

/15/ A. van Wijngaarden (Ed.): Report on the algorithmic Language ALGOL 68. Num. Math. 14, 79-218 (1969)

/16/ C.H.A.Koster: Affix Grammars. In: J.E.L.Peck (Ed.): ALGOL 68 Implementation. North Holland, Amsterdam 95-109 (1971)

/17/ C.H.A.Koster: Using the CDL Compiler-Compiler. In /1/, 366-426 (1974)

/18/ H.J.Bowlden: Cascaded SLR(k) Parser in ALGOL 68. In: Zweite GI-Fachtagung über Programmiersprachen, Saarbrücken 1972. GMD, St. Augustin, 189-200 (1972)

/19/ D.A.Watt: Analysis-oriented two-level Grammars. Ph.D. Thesis, Glasgow 1974

/20/ T.R.Wilcox: Generating Machine Code for high-level Programming Languages. Ph.D. Thesis, Cornell Univ. 1971

/21/ A.Church: The Calculi of Lambda-Conversion. Princeton Univ. Press 1941

/22/ H.Langmaack: On correct Procedure Parameter Transmission in higher Programming Languages. Acta Informatica 2, 311-333 (1973)

/23/ M.S.Paterson: Decision Problems in computational Models. SIGPLAN Notices 7, 1, 74-82 (1972)

/24/ C.Strachey: A general Purpose Macrogenerator. Comp. Journal 8, 225-241 (1965/66)

/25/ H.Langmaack: On Procedures as open Subroutines I, II. Acta Informatica 2, 311-333 (1973), 3, 227-241 (1974)

/26/ W.M.Waite: The mobile Programming System: STAGE 2. Comm. ACM 13, 415-421 (1970)

/27/ G. Seegmüller: Einführung in die Systemprogrammierung. Bibl.Inst. Zürich 1974

/28/ E.W.Dijkstra: Recursive Programming. Num. Math. 2, 312-318 (1960)

/29/ C.L.McGowan: The "most recent" Error: its Causes and it Correction. SIGPLAN Notices 7, 1, 191-202 (1972)

/30/ P.Kanzia: On the "most recent" property of ALGOL-like programms. In: J.Loeckx (Ed.): Automata, Languages and Programming. 2nd Coll. Univ. Saarbrücken 1974, Lecture Notes in Comp. Science 14, Springer, Berlin-Heidelberg-New York 97-111 (1974)

/31/ W.Lippe: Entscheidbarkeitsprobleme bei der Übersetzung von Programmen mit einparametrigen Prozeduren. In: W.Frielinghaus, B.Schlender (Ed.): Dritte GI-Fachtagung über Programmiersprachen, Kiel 1974, Lecture Notes in Comp. Science 7, Springer, Berlin-Heidelberg- New York 11-24 (1974)

/32/ J.R.Büchi: Regular canonical Systems. Arch. Math. Logik u. Grundlagenforsch. 6, 91-111 (1964)

/33/ S.Ginsburg, S.A.Greibach, M.A.Harrison: Stack-Autoamta and Compiling. J. ACM 14, 172-201 (1967)

/34/ W.C.Rounds: Mappings and Grammars on Trees. Math. Systems Theory 4, 257-287 (1970)

/35/ P.Wegner: Data Structure Models for Programming Languages. SIGPLAN Notices 6, 2, 1-54 (1971)

/36/ J.B. Johnston: The Contour Model of Block structured Processes. SIGPLAN Notices 6, 2, 55-82 (1971)

/37/ J. McCarthy et al.: LISP 1.5 Programmer's Manual. The M.I.T. Press, Cambridge, Mass. (1962)

/38/ J.C.Reynolds: GEDANKEN, a simple typeless Language based on the Principle of Completeness and the Reference Concept. Comm. ACM 13, 5, 308-319 (1970)

/39/ D.M.Berry: Introduction to OREGANO. SIGPLAN Notices 6, 2, 171-190 (1971)

/40/ M.J.Fischer: Lambda Calculus Schemata. SIGPLAN Notices 7, 1, 104-109 (1972)

/41/ D.E.Knuth, R.W.Floyd: Notes on Avoiding "goto" Statements. Inform. Proc. Letters 1, 23-31 (1971)

/42/ E.Ashcroft, Z.Manna: Inform. Proc. 71, North-Holland, Amsterdam 250-255 (1972)

/43/ N.Wirth: The Programming Language PASCAL. Acta Informatica 1, 35-63 (1971)

/44/ U.Ammann: Die Entwicklung eines PASCAL-Compilers nach der Methode des strukturierten Programmierens. Diss. ETH 5456, Zürich (1975)

/45/ W.M.McKeeman: Compiler Construction. In /1/, 1-36 (1974)

Hans Langmaack

Institut für Informatik

und Praktische Mathematik

der Universität Kiel

23 K i e l 1

Olshausenstr. 40-60