Entwurf und Implementation eines FORTRAN77-Compilers auf der AEG80-60

Jens-Detlev Doll

Diplomarbeit
am
Fachbereich für Informatik
der Universität Hamburg
Januar 1981

Erstbetreuer: W. Brauer
Zweitbetreuer: H. Dilcher

* Drittbetreuer: Angela

Inhaltsverzeichnis

3. Schrittweise Übersetzung von FORTRAN 77

Einleitung

Die Sprache FORTRAN 77 ist eine fast kompatible Erweiterung von FORTRAN IV. Sie bietet bessere Kontrollstrukturen als FORTRAN IV und erweitert FORTRAN um Möglichkeiten zur Zeichenverarbeitung.

Die AEG80-60 am Deutschen Elektronen-Synchroton (DESY) soll zur Experimentekontrolle und -auswertung verwendet werden. FORTRAN verfügt über gute arithmetische Fähigkeiten und erlaubt es, Unterprogramme aus Bibliotheken relativ einfach zu Programmen hinzuzubinden. Dadurch ist FORTRAN recht gut für die Erstellung von Programmen zur Experimenteauswertung geeignet.

Der auf der AEG80-60 vorhandene FORTRAN IV-Compiler war jedoch wegen seiner hohen Compilationszeiten und seiner intensiven Benutzung der System-Ressourcen ein großes Hindernis für die Benutzer. Diese negativen Eigenschafen sowie der rudimentäre Sprachumfang von FORTRAN IV gaben den Anstoß zur Implementation eines FORTRAN 77-Compilers für die AEG 80-60.

Als Grund für die Ineffizienz des vorhandenen FORTRAN IV-Compilers ergab sich die Struktur der in einer Zweipaßübersetzung verwendeten Zwischensprache ZS. ZS stellt ein UNCOL-Konzept für die Sprachen FORTRAN IV, SL3, PEARL und etwaige weitere höhere Programmiersprachen dar. Die Baumstruktur von ZS wird in einer vollständig verzeigerten Form in Zwischendateien abgelegt und ist nicht direkt in einem Durchlauf durch die Dateien übersetzbar, sondern muß entweder vollständig eingelesen werden, oder es muß wahlfrei auf die Dateien zugegriffen werden.

Ein weiterer Grund für die Ineffizienz des Compilers war in der Konfiguration des DESY-Rechners zu finden. Die zur Zwischenspeicherung von ZS-Programmen verwendeten Dateien und der Hintergrundspeicher zur Realisierung des virtuellen Adreßraumes befanden sich auf derselben Wechselplatte, so daß es bei der FORTRAN-Compilation außerordentlich oft zu einer Neupositionierung des Wechselplattenspeichers kam.

Die wichtigsten Anforderungen an den zu implementierenden FORTRAN 77-Compiler waren deshalb ein hoher Durchsatz bzw. eine geringe Wartezeit am Terminal und eine große Compilationsgeschwindigkeit. Diese Anforderungen waren nur bei Verwendung einer Einpaßübersetzung unter effizienter Benutzung von Dateioperationen zu erreichen.

Die Sprache FORTRAN befindet sich auch nach ihrer letzten Standardisierung (FORTRAN 77) noch in der Entwicklung. Der Compiler mußte deshalb von vornherein auf eine leichte Änderbarkeit und Erweiterbarkeit der Sprache ausgelegt werden.

Eine Modularisierung des Compilers bietet die Möglichkeit, wohldefinierte Schnittstellen einzuführen, so daß Änderungen und Erweiterungen lokal durchzuführen sind. Sie bietet weiterhin den Vorteil, die Moduln bei geeigneter Wahl der Schnittstellen getrennt entwickeln und testen zu können.
In dieser Arbeit soll eine geeignete Struktur für den FORTRAN 77-Compiler, die zur Erfüllung der obigen Anforderungen führt, entwickelt werden. In Kapitel 1 sollen Quellsprache und Zielmaschine vorgestellt und deren spezielle Eigenschaften beschrieben werden.

Kapitel 2 beschäftigt sich mit der Herleitung einer geeigneten Grobstruktur für den zu entwerfenden Compiler.

Die aus Kapitel 2 entstandene Grobplanung soll dann in Kapitel 3 durch Spezifikation der zu verwendenden Strukturen und Algorithmen verfeinert werden.

In dieser Arbeit wird der Begriff Modul in verschiedenen Zusammenhängen benutzt. Ein Modul im statischen Sinne eines Programm-Moduls soll als eine Zusammenfassung von Prozedur- u. Datenobjekten mit expliziter Im-/Exportliste aufgefaßt werden (Jessen 79). Die entwickelten Moduln des FORTRAN 77-Compilers sind aus dieser Sicht Datenmoduln mit permanenten Eigendaten.

Ein Modul des FORTRAN 77-Compilers hat bezüglich seines dynamischen Verhaltens die Eigenschaften einer Koroutine. Es wird initialisiert und durchläuft bei der Übersetzung verschiedene Zustände, wobei die wechselseitige Aktivierung durch die Anweisungen der Zwischensprachen gesteuert wird.

Die verschiedenen logischen Übersetzungsphasen des FORTRAN 77-Compilers werden, wie in Kapitel 2 erläutert, direkt auf die Moduln des Compilers abgebildet.

Der in dieser Arbeit benutzte Begriff Modul vereinigt damit den statischen Begriff eines abstrakten Datentyps, den dynamischen Begriff einer Koroutine und die logische Sicht einer Übersetzungsphase. Er ist damit stets ein Maß für jeweils ein statisches, dynamisches und logisches Verhalten.

Die Grundidee bei der Entwicklung des FORTRAN 77-Compilers ist somit die Zusammenfassung des logischen, statischen und dynamischen Verhaltens des Übersetzers in abstrakte Moduln, die im folgenden aus verschiedenen Perspektiven betrachtet werden sollen.

Der Begriff syntaxgesteuerte Übersetzung soll hier in seiner allgemeinsten Form gesehen werden. Er bezeichnet hier die Erweiterung einer kontextfreien Grammatik um zusätzliche Aktionssymbole. Die mit diesen Aktionssymbolen verbundenen Attribute und Attributauswertungsregeln werden statisch und

dynamisch in einem gesonderten Modul behandelt. Es erfolgt somit lediglich eine syntaxgesteuerte Übersetzung von der Eingabesprache des Moduls Syn auf die Eingabesprache des Moduls Sem.

Die Implementation des FORTRAN 77-Compilers wurde nach Erreichen eines Sprachumfanges von ca. 90 % abgebrochen, nachdem alle grundsätzlichen Entwurfsfragen geklärt waren. Die Realisierung einer Bindemodulschnittstelle wurde nicht als essentiell angesehen, sondern nur theoretisch vollzogen. Anhand des bis jetzt fertiggestellten Compilers kann gezeigt werden, daß der Compiler die gestellten Entwurfsziele erfüllt. Der erstellte Compiler ist benutzerfreundlich. Er gibt statische Programmfehler durch Quellzeile, Position und Fehlanmeldung an. Dynamische Programmfehler werden durch Ausgabe der entsprechenden Zeilennummer und einer Rückverfolgung der Prozedurverschachtelung gemeldet.

Anfängliche Pläne zur Realisierung eines voll Bootstrap-fähigen in FORTRAN 77 geschriebenen FORTRAN 77-Compilers wurden ebenfalls nur theoretisch vollzogen. Die Realisierung eines FORTRAN 77-Compilers-Oberteils, das Code für eine virtuelle Maschine erzeugt und in FORTRAN77 geschrieben ist, wird mit relativ geringem Umcodierungsaufwand möglich.
Vorbereitende Implementationsarbeiten für dieses Projekt wurden im Herbst 1979 geleistet. In dieser Arbeit ist dann aus den erstellten Software-Werkzeugen eine gut brauchbare Installation von FORTRAN77 auf den AEG80-60 erstellt worden.

1. Untersuchung von Quellsprache und Zielmaschine

In diesem Abschnitt sollen interessante Spracheigenschaften der Programmiersprache FORTRAN 77, die sich auf die Struktur und einzelne Teile des Compilers ausgewirkt haben, vorgestellt werden. Anschließend soll die Zielmaschine im Hinblick auf ihre Eignung zur Übersetzung höherer Programmiersprachen untersucht werden.

Neben Quellsprache und Zielmaschine ist die Umgebung des zu erstellenden Übersetzers, das Programmiersystem, ein Faktor, der sich auf den Entwurf des Compilers auswirkt. Da der Compiler auf der Zielmaschine implementiert werden soll, ist das Programmiersystem sowohl Werkzeug bei der Implementation des FORTRAN 77-Compilers als auch Umgebung des fertigzustellenden Compilers.

1.1. Die Programmiersprache FORTRAN 77

FORTRAN ist auch heute noch eine der am weitesten verbreiteten Programmiersprachen. Durch umfangreiche Unterprogrammbibliotheken ermöglicht sie eine schnelle und relativ einfache Formulierung von numerischen Programmen. FORTRAN ist eine der wenigen höheren Programmiersprachen, die über komplexe Arithmetik verfügen.

Seit der letzten Standardisierung im Jahre 1966 (ANSI 66) haben auf Grund der Unzulänglichkeiten der Sprache verschiedene Implementationen eine starke Inkompatibilität verursacht (vergleiche (POOLE 74) p 430ff). Der neue FORTRAN 77-Standard erweitert die Sprache um wichtige Sprachkonstrukte unter Beibehaltung der Kompatibilität zum alten FORTRAN IV.

In (BRAINERD 78) werden die meisten dieser Neuerungen beschrieben. Hier soll lediglich die Problematik von FORTRAN IV und FORTRAN 77 an Hand einiger Beispiele verdeutlicht werden.

1.1.1. Eigenarten von FORTRAN

Die Sprache FORTRAN ist in hohem Maße kontextsensitiv. In (ANSI 78) wird der kontextfreie Teil der Sprache in Backus-Naur-Form sowie mit Hilfe von Syntaxdiagrammen beschrieben, der kontextsensitive Teil wird verbal durch eine Menge von Anforderungen und Einschränkungen abgegrenzt.

Über die normalen Kontextabhängigkeiten in anderen Programmiersprachen hinaus (Deklarationszwang von Marken und Bezeichnungen, Übereinstimmung der Typen von formalem und aktuellem Parameter etc.) sind einige Sprachkonstrukte besonders problematisch.

----DO-Loops

Eine DO-Loop besteht aus einer Folge von Anweisungen, die durch ein initiales DO-Statement und ein terminales Statement eingeschlossen werden, wobei ineinandergeschachtelte DO-Loops durchaus dasselbe terminale Statement haben können:

```
      DO 1ØØ    I = 1 , N
      ....
      DO 1ØØ    J = 1 , N
      ...
1ØØ   CONTINUE
```

Die Übereinstimmung der Marke im initialen und terminalen Statement und die mehrfache Benutzung desselben terminalen Statements lassen sich nicht in einer kontextfreien Syntax ausdrücken.

----Formelfunktionen

Die Definition einer Formelfunktion hat syntaktisch das Aussehen einer Zuweisung auf ein Array-Element:

```
A(I) = I + 2
```

Ist dabei A nicht als Feld deklariert und geht dieses Statement dem ersten ausführbaren Statement voraus und folgt allen Deklarationsanweisungen, so handelt es sich hier um eine Formelfunktion mit Namen A und formalem Parameter I. Hier muß ein FORTRAN-Parser die Tabelleneinträge für A zu Rate ziehen, um entscheiden zu können, ob es sich um die syntaktische Regel

<procid> (<parlist>) = <expr> oder

<arrayid>(<indexlist>) = <expr>

handelt.

Dieses Problem stellt sich insbesondere dem Programmierer, der durch einen Irrtum bei der Deklaration von A ein syntaktisch und semantisch völlig korrektes Programm erstellen kann, dessen Wirkung aber eine völlig andere als die von ihm beabsichtigte ist.

----Feldindizierung

Ein weiteres Beispiel ergibt sich aus dem sehr eingeschränkten Zeichensatz der Sprache. Eine Array-Referenz hat syntaktisch das gleiche Aussehen wie der Aufruf einer externen Funktion.

```
Y = F ( N )
```

bedeutet die Zuweisung eines Funktionsergebnisses der Funktion F mit aktuellem Parameter N, sofern F nicht als Array deklariert wurde. Auch hier muß ein Parser zwischen

<procid> (<aktparlist>) und

<arrayid> (<indexlist>)

unterscheiden können.

----COMMON-Blöcke

FORTRAN kennt keine globalen Variablen als Kommunikationsmittel zwischen Unterprogrammen. Der Programmierer muß diese durch COMMON-Blöcke realisieren. Ein COMMON-Block ist ein globaler Name für eine Speicherstrecke, dessen Interpretation durch Deklaration von Variablen in diesem COMMON-Block erfolgt. Dadurch können auch extern übersetzte Unterprogramme mit anderen Unteprogrammen über gemeinsame Speicherplätze kommunizieren.

Ein COMMON-Block, der in mehreren Unterprogrammen angelegt wird, ist dabei nur einmal vorhanden. Durch die Möglichkeit des externen Übersetzens von Unterprogrammen wird es für den Compiler unmöglich, die Speicherabbildung von COMMON-Blöcken vollständig durchzuführen. Er muß dies dem Binder überlassen. Die Sprache FORTRAN stellt also auch besondere Anforderungen an das Programmiersystem des Rechners.

Der FORTRAN 77-Standard hätte viele dieser Probleme beseitigen können. Leider geschah dies nicht, um Aufwärtskompatibilität zu bestehenden Programmen zu gewährleisten.

1.1.2. Interessante Sprachkonstrukte in FORTRAN 77

FORTRAN 77 kennt einen booleschen und vier arithmetische Datentypen sowie den neu eingeführten Datentyp CHARACTER, der ähnliche Eigenschaften wie ein ALGOL60-String hat.
Auf CHARACTER-Variablen sind die Operationen Zuweisung, Konkatenation und Bildung von Substrings sowie die Prädikate <, <=, =, ≠, >, >=, anwendbar, wobei die lexikalische Ordnung zugrunde gelegt wird.

FORTRAN 77 benötigt einen Übersetzungszeit-Interpreter, denn mit Hilfe eines PARAMETER-Statements hat man die Möglichkeit, konstanten Ausdrücken einen Namen zu geben. Da auch Feldgrenzen durch konstante Ausdrücke definiert werden können, ist eine Auswertung konstanter Ausdrücke zur Übersetzungszeit notwendig.

```
INTEGER KBYTE,DISK
REAL PI,RADIUS,CIRCUM
CHARACTER*(*) FIRST,LAST,NAME
PARAMETER (PI = 3.1416  ,RADIUS = 10,
           CIRCUM = 2 * PI * RADIUS,
           KBYTE = 2 ** 10, DISK = 250 * KBYTE,
           FIRST = 'WALTER'  , LAST = 'JONES'
           NAME = FIRST // ' ' // LAST )
CHARACTER PAGE  (Ø :4*KBYTE-1)
```

Die Einführung von Intrinsic-Funktionen und einer Mixed-Mode Arithmetik führen zum Problem der Operatoridentifikation, das in FORTRAN IV nicht bestand. Die Interpretation einer Intrinsic-Funktion ist vom Typ des aktuellen Parameters abhängig, Operatoren sind erst nach Auswertung ihrer Operanden zu identifizieren.

```
REAL X, Y
DOUBLE PRECISION D, YD
COMPLEX C, YC
Y = SIN(X)
YD = SIN (D)
C = 2 *Y
YC = SIN (C)
```

Die Instrinsic-Funktion SIN hat in diesem Beispiel drei verschiedene Interpretationen: einfach genauer, doppelt genauer und komplexer Sinus. Bei der Zuweisung C = 2 * Y muß die 2 zunächst in einen reellen Operanden konvertiert werden, dann ist die Operation REAL_* auszuführen. Das Ergebnis muß zu einem komplexen Operanden konvertiert werden (2*Y + i*Ø) und ist mit der Operation COMPLEX_Zuweisung auf C zuzuweisen.

Eine Möglichkeit, strukturierte Programme zu schreiben, bietet die

IF (e) THEN - {ELSE IF (e) THEN -} [ELSE -] END IF

Anweisung, die im Gegensatz zum PASCAL-IF-Konstrukt über eine eindeutige kontextfreie Syntax verfügt.

```
IF (A .EQ . Ø) THEN
...
ELSE IF (A .EQ. 2) THEN
...
ELSE IF (A . EQ. 8) THEN
...
ELSE
...
END IF
```

Die FORTRAN 77 Ein-/Ausgabe ist für den Benutzer, der nicht mit Formaten arbeiten möchte, wesentlich einfacher geworden. Bei einer listenorientierten Ein-/Ausgabe wird in einem vom Prozessor vorgegebenen Standardformat eingelesen bzw. ausgegeben. Die Prozeduren READ und WRITE sind außerdem auf interne Files vom Datentyp CHARACTER anwendbar.

```
CHARACTER * 80 BUFFER
BUFFER = 'Ø.1 ,Ø.2 ,Ø3'
READ(BUFFER,*) X,Y,Z
READ * , X
PRINT * , 'SIN(',X,')=',SIN(X)
```

Außerdem hat der Programmierer die Möglichkeit, von FORTRAN 77-Programmen aus das Dateisystem des Rechners anzusprechen und eine Datei unter ihrem logischen Namen mit einem programminternen Kanal zu verbinden.

```
REAL  A(1:100)
OPEN (UNIT=10,FILE='DATAINPUT')
READ (10,*) N
READ (10,*) (A I , I=1,N)
```

Der FORTRAN 77-Standard definiert portable Standardprogramme; es ist dem die Sprache realisierenden Sprachprozessor überlassen, Erweiterungen gegenüber der vollen Sprache anzubieten. Als sehr sinnvoll erweist sich die Einführung von WHILE- und REPEAT-Schleifen nach dem Vorbild von PASCAL. Mit geringem Aufwand läßt sich ein FORTRAN 77-Compiler um diese Kontrollstrukturen erweitern (vergleiche (BRAINERD 77)).

Syntaktisch sind bei dieser Erweiterung drei Arten von DO-Schleifen zugelassen, die sich nur durch die Bildung der Schleifenbedingungen unterscheiden:

```
                 { id = <low>,<high><step>
DO loop          { WHILE (<cond>)
                 { UNTIL (<cond>
                   ...
loop <terminal>
```

Diese zusätzlichen Kontrollstrukturen lassen sich semantisch durch ihre Äquivalenz zu vorhandenen FORTRAN 77-Strukturen definieren:

Erweiterung	FORTRAN 77-Äquivalent
DO loop WHILE (cond)	IF (.TRUE.) THEN *
	L IF (cond) THEN
....	....
loop terminal	loop terminal
	GOTO L
	END IF
	END IF
DO loop UNTIL (cond)	IF (. TRUE.) THEN *
....	L ....
loop terminal	loop terminal
	IF (.NOT.cond)GOTO L
	END IF

L : neue, eindeutige Marke
loop: Schleifenmarke
terminal: terminales Statement
cond: Schleifenbedingung
*: Die äußeren IF-Klammern sollen es dem Compiler ermöglichen, unerlaubte Sprünge in den Kontrollbereich der DO-Schleife oder eine inkorrekte Schachtelung von DOSchleifen zu erkennen.

Diese zusätzlichen Kontrollstrukturen sollen aus Kompatibilitätsgründen nur zur Implementation der Laufzeithilfen im zu erstellenden Compiler benutzt werden. Eine spätere Erweiterung des FORTRAN 77-Compilers um diese Sprachkonstrukte ist jedoch auch denkbar.

1.2. Die Zielmaschine AEG80-60

Bei der AEG80-60 handelt es sich um einen Rechner mit mikroprogrammierter Zentraleinheit, die 16 General-Purpose-Register und eine Anzahl von Maschinenregistern enthält. Der Hauptspeicherausbau des DESY-Rechners beträgt 512 KBytes, die Adressierung erfolgt byteweise über 4 offene Basisregister. Der virtuelle Adreßraum von 4 MBytes wird am DESY mit Hilfe eines Wechselplattenspeichers realisiert, so daß der Austausch von Seiten relativ langsam ist. Befehle haben eine varibale Länge von 2 bis 6 Bytes; von den 2 Operanden eines Befehls muß einer ein Register sein ("1 1/2-Adreßmaschine"). Einsatzgebiet dieses Rechners ist die Prozeßsteuerung.

1.2.1. Befehle und Speicherstruktur

Einige Befehle und Eigenschaften des Rechners sind im Zusammenhang mit der Übersetzung von höheren Programmiersprachen auf Maschinencode erwähnenswert.
Der Rechner verfügt über Befehle zur Manipulation von Byteketten der Länge 1 bis 256 Bytes. Dies gestattet insbesondere bei FORTRAN 77-Characterhandling eine wirkungsvolle Übersetzung der hochsprachlichen Konstrukte auf Maschinenbefehle. Zuweisungen

und Vergleiche von Zeichenketten lassen sich jeweils mit Hilfe eines Befehls realisieren.

Zur Feldindizierung bietet die Maschine einen Befehl, der mittels eines im Speicher angelegten "Dope-Vektors" (GRIES 71) die Relativadresse eines Feldelements berechnet und gleichzeitig die Einbehaltung der Feldgrenzen überprüft. Leider ist dieser Befehl nur bedingt benutzbar, da die Anzahl der Dimensionen auf 3 beschränkt ist, FORTRAN 77 aber Felder mit bis zu 7 Dimensionen kennt.

Sehr störend macht sich das Fehlen von Kelleroperationen bemerkbar. Selbst bei der nichtrekursiven FORTRAN-Unterprogrammtechnik ist es sinnvoll, die Rücksprungadressen und aktuellen Parameter in einem Stack anzulegen (→ 3.5). Ein logischer Befehl zur Invertierung von Operanden fehlt und muß durch die Antivalenz des Operanden mit dem logischen Wert TRUE realisiert werden.

Die AEG80-60 zeichnet sich durch ein wirksames Speicherschutzsystem aus. Ein Programm besteht zur Laufzeit aus mehreren Segmenten, denen je nach ihrer Art verschiedene Schutzeigenschaften zukommen. In ein Befehls- oder Konstanten-Segment darf nicht geschrieben werden, ein Datensegment darf nicht exekutiert werden etc. Bei einer Schutzverletzung reagiert der Rechner mit einem Alarm, der zur Unterbrechung des laufenden Prozesses führt.

Mit Hilfe dieser Eigenschaft des Rechners läßt sich verhindern, daß eine Konstante zur Laufzeit eines FORTRAN 77-Programmes verändert wird. Dies ist auf anderen Rechnern möglich, da auch Konstanten in FORTRAN als Referenz-Parameter übergeben werden und dadurch im Unterprogramm verändert werden können. Ein Erkennen solcher Fehler zur Compilationszeit ist nicht möglich. Eine Prüfung durch den Compiler zur Laufzeit ist aber dennoch möglich, wenn Konstanten in entsprechend geschützten Konstantensegmenten angelegt werden.

1.2.2 Besonderheiten des Prozeßrechner-Betriebssystems

Die Prozessorverwaltung der AEG80-60 ist einfach, aber wirkungsvoll. Sie gestattet es dem Benutzer, sich ein für seine Anwendungszwecke angemessenes System kommunizierender paralleler Prozesse zu entwerfen.

Die Vergabe der Zentraleinheit an einen von mehreren laufbereiten Prozessen wird durch Prioritäten geregelt. Ein Prozeß, der sich im Besitz der Zentraleinheit befindet, ist nur durch eigene Systemdienstaufrufe oder durch andere Prozesse höherer Priorität unterbrechbar. Die Systemdienstaufrufe für EA-Anforderungen werden im sogenannten Synchronbetrieb abgewickelt, d. h. der beauftragende Prozeß wird so lange suspendiert, bis der EA-Auftrag erfüllt wurde.

Für Compiler, die viel sequentielle Ein- und Ausgabe auf Quell- bzw. Objektdatei durchführen, ist es deshalb angebracht, im blockweisen Betrieb mehrere Sätze einer Datei zusammen einzulesen bzw. auszugeben. Dadurch wird eine häufige Neupositionierung des Wechselplattenspeichers vermieden.

Durch Benutzung einer gepufferten Ein-/Ausgabe und Bearbeitung dieser Puffer durch zusätzliche parallele Prozesse läßt sich der Durchsatz eines Compilers wesentlich steigern. Eingabeprozeß und Compiler sowie Compiler und Ausgabeprozeß spielen dabei die Rolle von Producer und Consumer. Zur Synchronisation der parallelen Prozesse bieten sich Botschaften und Semaphore an, die im Betriebssystem des Rechners realisiert sind.

1.3 Das bestehende Programmiersystem

Das Programmiersystem der AEG80-60 gliedert sich in die Komponenten Übersetzer bzw. Assembler, Programmbinder, Teilsystembinder zur Vereinigung mehrerer Programme, die eventuell konkurrent

ablaufen können, und den Lader, der Zugriffe auf die Bibliotheken für Standardprozeduren und Laufzeitunterstützungen hat. Kleinste Übergabeeinheit in diesem System ist ein Modul.

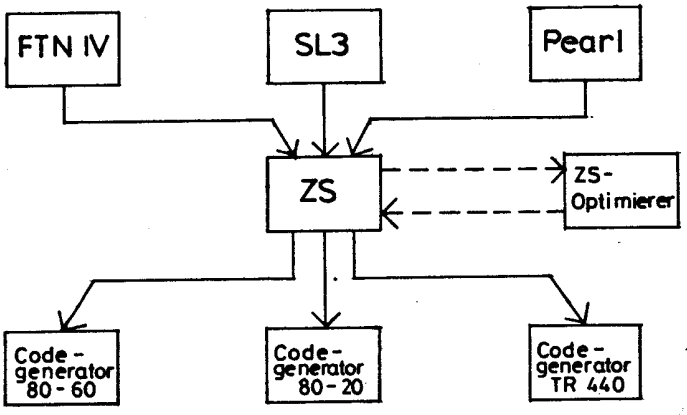
Ein Compiler erzeugt aus einem Quellsprachmodul, das aus einer Ansammlung von Daten und Prozeduren besteht, ein Bindemodul, das im allgemeinen aus mehreren Segmenten verschiedener Art besteht. Durch die Import-/Exportlisten, die globale und externe Namen, Common-Segmente und Startadressen etc. enthalten, können Beziehungen zwischen Moduln bestehen.

Aufgabe der Binder ist die Absättigung dieser Modulbeziehungen. Dabei werden unter anderem globale und externe Namen vereinigt und Common-Segmente gleichen Namens übereinandergelegt. Durch die letzte Eigenschaft ist der Binder für die Vereinigung getrennt übersetzter FORTRAN 77-Programmteile geeignet.

Beim Binde- und Ladevorgang können vom Compiler zu jedem Quellmodul zusätzlich erzeugte Testmoduln beigefügt werden, deren Aufgabe es ist, Quellmoduleigenschaften wie Adressen von Prozeduren und Variablen transparent zu erhalten. Dadurch bietet sich die Möglichkeit eines Laufzeit-Testsystems auf Quellsprachenebene.

Dem Benutzer stehen in diesem Programmiersystem drei Sprachen zur Verfügung, ein sehr einfacher Assembler ohne Makrofähigkeit, der nur für Spezialzwecke gedacht ist, und zwei höhere Programmiersprachen, FORTRAN IV und SL3, eine Systemprogrammiersprache auf der Basis von ALGOL68. (2.1)

Das vorhandene Übersetzungssystem stellt den Versuch dar, ein UNCOL-Konzept (KOSTER 74) für die Sprachen FORTRAN IV, SL3 und PEARL zu realisieren. In einer Zweipaßübersetzung werden Quellprogramme auf eine sprach- und maschinenunabhängige Zwischensprache ZS abgebildet, aus der ein Codegenerator in einem zweiten Lauf Objektcode für die AEG80-60 erzeugt.



Ein optionaler Optimierungslauf gestattet eine sprach- und maschinenunabhängige Codeoptimierung von ZS-Programmen auf Zwischensprachenniveau.

Die Zwischensprache ZS ist eine Sprache mit baumartigen Kontrollstrukturen und listenartigen Datenstrukturen, deren Syntax und Semantik durch eine zweischichtige Grammatik und eine sogenannte Makrosyntax vollständig definiert wurde (PRAHL 78). Kontroll- und Datenstrukturen sind in ihren Eigenschaften sehr stark der Sprache SL3 angepaßt, so daß sich für SL3-Programme eine sehr gute Übersetzung ermöglichen läßt.

Die Entwicklung des Übersetzungssystems erfolgte auf der TR440 mit Hilfe von BCPL und wurde von da aus durch Bootstrapping auf die AEG80-60 übertragen.

Die Kommunikation zwischen Compileroberteil und Codegenerator erfolgt über eine Anzahl Dateien, die zusammen ein Programm der Sprache ZS bilden. Bei der Entwicklung des Übersetzungssystems auf der TR440 schienen Dateien ein geeignetes Mittel zur Aufbewahrung von Zwischencode gewesen zu sein, auf der AEG80-60 macht sich dagegen die Langsamkeit des Dateisystems sehr negativ bemerkbar. Die anfänglich verwendeten 23(!) Zwischendateien (heute noch 8) führten für den Benutzer des Übersetzungssystems zu unverhältnismäßig hohen Wartezeiten am Terminal.

Ein Vorteil, den eine Zweipaßübersetzung mit sich bringt, nämlich der geringere Speicheraufwand für die einzelnen Phasen, wird bei ZS durch den relativ hohen Speicherbedarf zur Repräsentation eines baumartigen Zwischensprachenprogrammes in Compileroberteil und Codegenerator wieder vergeben. Damit ist die Realisierung des Übersetzungssystems auf kleineren Rechnern nur durch eine weitere Zerlegung der beiden Pässe möglich.

Speziell für FORTRAN hat dieses Übersetzungsverfahren einige weitere konzeptionelle Nachteile.
FORTRAN ist durch sein einfaches Prozedurkonzept und durch seine Beschränkung auf elementare Datenstrukturen eigentlich eine "maximal übersetzbare Sprache", (Jessen 79) die mit einem geringen Laufzeitsystem auskommt. Eine Übersetzung von FORTRANProgrammen auf eine ALGOL68 verwandte Zwischensprache bedeutet aber einen Rückschritt bei der Compilation, der nur durch umfangreiche Optimierung wieder auszugleichen ist.

Der Codegenerator muß zum Beispiel Eigenschaften einer FORTRAN-Prozedur wie Nichtrekursivität, kein Zugriff auf umgebende Blöcke, erkennen können, um effizienten Code zu erzeugen. FORTRAN-eigene Konstrukte wie Commonblöcke und das "assigned GOTO" müssen vom Codegenerator behandelt werden können. Für andere Sprachen sind diese Konstrukte ohne Wert. Das führt zu einer Vergrößerung des Codegenerators um einen sprachabhängigen Teil, der nur für FORTRAN von Nutzen ist.

2 Modularer Entwurf des Compilers

In diesem Kapitel soll eine geeignete Struktur für einen Einpaßübersetzer für FORTRAN 77 entwickelt werden. Dabei sind die Randbedingungen, die sich aus den Eigenschaften der Sprache FORTRAN 77 und des Programmiersystems der AEG80-60 ergeben, zu beachten. Die Zielmaschine sollte aus Gründen der Portabilität keinen Einfluß auf die zu entwickelnde Struktur haben.

Da die Entwicklung eines Compilers ein umfangreiches Programmierprojekt darstellt, ist eine Aufgliederung dieses Projektes in einzelne Moduln angebracht. Solche Moduln sollten sinnvolle logische Teilprobleme des Gesamtproblems bearbeiten. Lediglich eine genaue Spezifikation der Aufgaben eines Moduls sollte sich auf die Festlegung von Modulschnittstellen auswirken. Dadurch wird eine getrennte Entwicklung und Testbarkeit von Moduln möglich. Weiterhin tragen exakte Schnittstellendefinitionen wesentlich zur Übersichtlichkeit eines großen Programmes bei.

Da ein Compiler eine Quellsprache in eine Zielsprache überführt, liegt es nahe, auch in einem modularen Einpaßcompiler Schnittstellen zwischen dessen Moduln durch Sprachen im Sinne einer Zwischensprache für eine Mehrpaßcompilation zu definieren.

Eine übersichtliche Struktur vereinfacht eine fehlerfreie Durchführung des Programmierprojektes wesentlich. Trotzdem sind Programmier- und insbesondere Schreibfehler nicht auszuschließen, deren Erkennung und Beseitigung durch geeignete Testläufe des Compilers geschehen muß. Hierfür ist im Falle FORTRAN 77 besonders eine Pilotimplementation der Standardprozeduren und der Laufzeithilfen geeignet, die sich sehr gut in FORTRAN 77 selbst realisieren lassen.

Zunächst aber soll eine angemessene Sprache zur Durchführung dieses Programmierprojektes ausgewählt werden, die dann auch als Hilfsmittel zur Schnittstellendefinition im Compiler dienen kann. Auf der AEG80-60 bieten sich drei Sprachen an, deren Vor- und Nachteile zur Implementation eines FORTRAN 77-Compilers gegeneinander abgewogen werden müssen: Assembler, FORTRAN IV und SL3.

2.1 Die Rolle der Implementationssprache

Die Implementationssprache für ein komplexes Programmprojekt sollte zunächst einmal zwei notwendige Voraussetzungen erfüllen. Sie muß die Durchführbarkeit des Projektes gewährleisten und die Modularisierung des Programmes ermöglichen. Unter Modularisierbarkeit soll auch die Fähigkeit zum externen Übersetzen einzelner Moduln verstanden werden, denn durch strikt getrennte Entwicklung lassen sich versteckte Inter-Modulbeziehungen weitgehend vermeiden.

Zwei wichtige Anforderungen an die Entwicklung eines FORTRAN-77-Compilers sind die Portabilität des Objektes und der kleinstmögliche Programmieraufwand zur Erstellung desselben. Beide Anforderungen werden nur bei Verwendung einer höheren Programmiersprache hinreichend erfüllt, wobei der Programmieraufwand im Rahmen dieser Arbeit als das wichtigere der beiden Kriterien angesehen werden kann.

Als Nebenbedingung an die Programmiersprache ergeben sich die Effizienz in Laufzeit-und Speicheraufwand und die Fähigkeit der Sprache, die Erstellung von korrekten und verständlichen Programmen leicht zu machen.

2.1.1 Die Eignung der vorhandenen Programmiersprachen

Diese sechs Kriterien sollen nun ihrer Wichtigkeit gemäß auf die zur Verfügung stehenden Sprachen Assembler, FORTRAN IV und SL3 angewendet werden.

----FORTRAN IV

Die Wahl der Sprache FORTRAN IV als Implementationssprache für einen FORTRAN 77-Compiler hat große Vorteile für die Portabilität des Compilers. FORTRAN IV steht auf fast allen Rechenanlagen zur Verfügung und ist standardisiert. Da FORTRAN IV eine echte Teilmenge der Sprache FORTRAN 77 darstellt, läßt sich nach dem Vorbild von BCPL oder PASCAL (ELSWORTH 79) zunächst

eine Bootstrap-Version für FORTRAN 77, geschrieben in FORTRAN 77-Ø = FORTRAN IV, erstellen. Diese sollte Code für eine maschinenunabhängige virtuelle Maschine FCODE erzeugen, die, in einer beliebigen anderen Sprache realisiert, auf der AEG80-60 interpretiert werden kann. (Abbildung A) Die Sprache FORTRAN 77-Ø läßt sich dann in mehreren Zwischenschritten auf das volle FORTRAN 77 erweitern, und man erhält einen voll portablen FORTRAN 77-Compiler. Ein letzter Schritt in diesem Prozeß wäre dann die Adaptierung des Codegenerators an die AEG80-60, so daß unter Beseitigung der virtuellen FCODE-Maschine direkt Code für die AEG80-60 erzeugt würde.

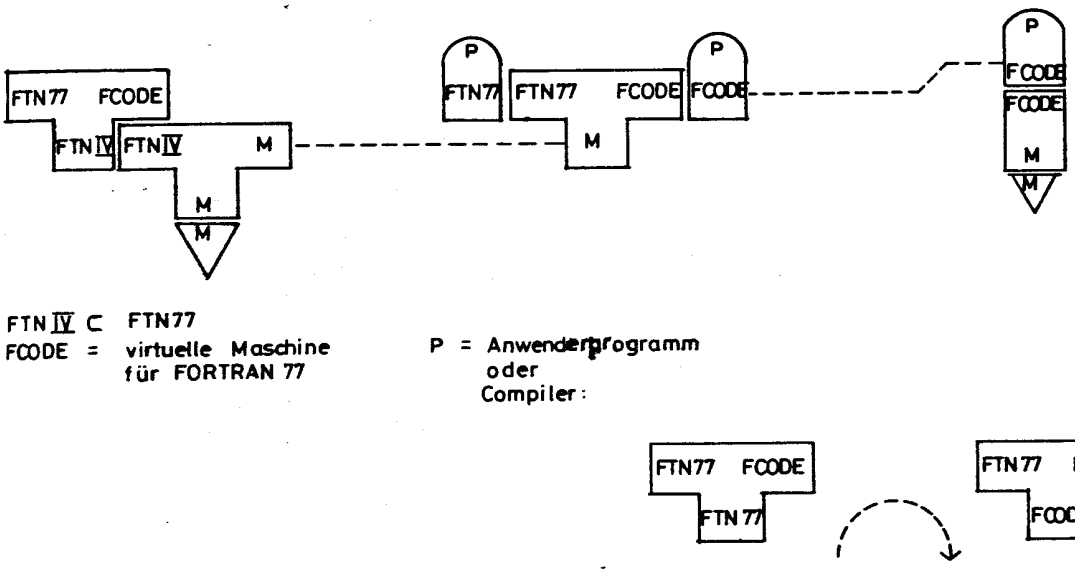


Abbildung A

Dieses ist ein denkbarer Weg zur Implementation eines FORTRAN 77-Compilers, bringt aber einen großen Programmieraufwand mit sich. Die erste Implementationssprache (FORTRAN IV) ist zur Erstellung von Compilern schlecht geeignet. Der Bootstrap-Compiler muß deshalb im zweiten Schritt in FORTRAN 77 umgeschrieben werden. Ein zusätzlicher Aufwand ergibt sich durch die Realisierung eines Interpreters und eines Codegenerators für die FCODE-Maschine.

Eine weitere Möglichkeit (Abbildung B), die sehr schnell zu einem vollen FORTRAN 77-Compiler führt, ist die Benutzung von FORTRAN IV als Zwischensprache in einem FORTRAN 77-Compiler. FORTRAN 77 kann unter Benutzung von FORTRAN IV als Implementationssprache auf FORTRAN IV übersetzt werden. Dann hat man einen Zweipaß- (auf der AEG80-60 einen Dreipaß-)Compiler, dessen Schnittstellen-Sprache FORTRAN IV ist. Dieser ist aber sehr ineffizient und bringt große Übersetzungszeiten und auch eine schlechte Codeerzeugung mit sich. Deshalb müßte, nachdem der erste Schritt vollzogen wäre, ein völlig neuer FORTRAN 77-Compiler entwickelt werden, der dann in Fortran 77 implementiert werden könnte und direkt Code für die AEG80-60 erzeugte. Damit hätte man ebenfalls einen vergrößerten Programmieraufwand gegenüber der einfachen Implementationsstrategie bei Benutzung einer einzigen Implementationssprache.

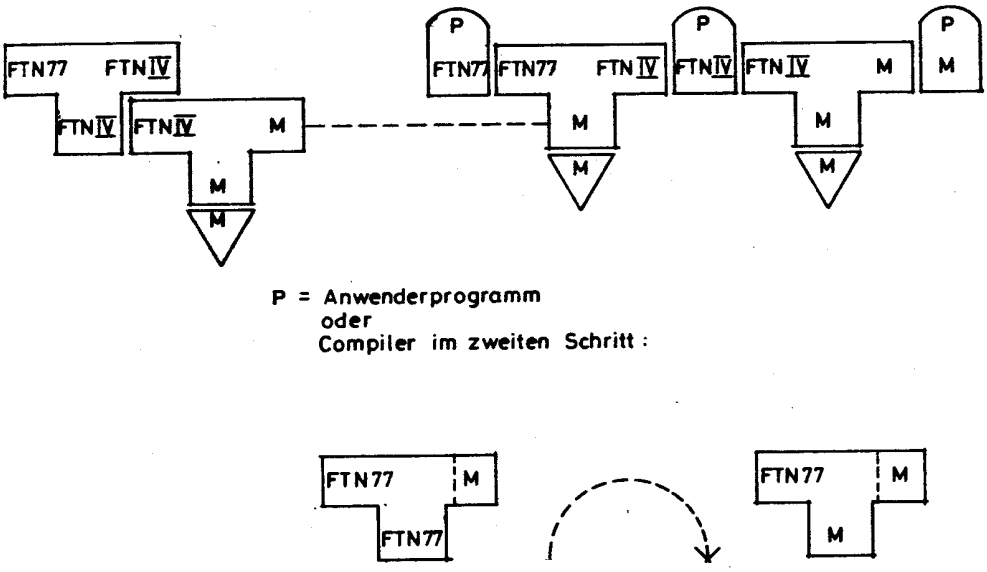


Abbildung B

Diese beiden Projekte sind nicht nur durch ihren hohen Programmieraufwand zum Scheitern verurteilt. Es erscheint zweifelhaft, ob ein solches Vorhaben überhaupt gut durchführbar wäre. FORTRAN IV hat zwar mäßige Modularitätseigenschaften - externes Übersetzen, Zusammenfassung von Routinen als Moduln, Modulschnittstellen könnten über Common-Blöcke realisiert werden -, ist aber als Implementationssprache für einen Compiler wenig geeignet. Möglichkeiten zur Zeichenverarbeitung sind in FORTRAN IV einfach nicht vorhanden und müßte unter Umgehung der Spracheigenschaften realisiert werden. Weiterhin hat FORTRAN keine höheren Datenstrukturen außer dem Feld, was zu einer übermäßig hohen Codierleistung des Programmierers führen würde. Spracheigenschaften zur Bit-Verarbeitung fehlen völlig, so daß ein Objektcode erzeugender Codegeneratur mit Hilfe von externen Assembler-Routinen zu realisieren wäre. Ein solches FORTRAN-Programm wäre aber wenig leserlich und führte wegen der geringen Fehlertoleranz (implizite Deklarationen) der Sprache auf schlechte Programme. Aus diesen Gründen muß die für die Compilertechnik interessante Möglichkeit des Bootstrapping durch FORTRAN IV leider ausscheiden.

----Assembler

Die Sprache Assembler bietet maximale Durchführbarkeitsmöglichkeiten für das Compiler-Projekt. Die Modularitätseigenschaften von Assembler-Programmen sind recht gut, wenn auch die Schnittstellendefinitionen zwischen Assembler-Moduln auf unterster Datenebene zu realisieren wären. Maximale Effizienz auf Programmierebene - nicht auf Algorithmenebene - wäre bei Benutzung des Assemblers am besten erreichbar.

Dennoch hat die Benutzung der Sprache Assembler als Implementationssprache gravierende Nachteile. Der im Assembler erstellte Compiler ist sehr schlecht portabel. Die einzige Möglichkeit, den Compiler portabel zu machen, besteht in einer Interpretation des Maschinencodes der AEG80-60 auf anderen Maschinen.

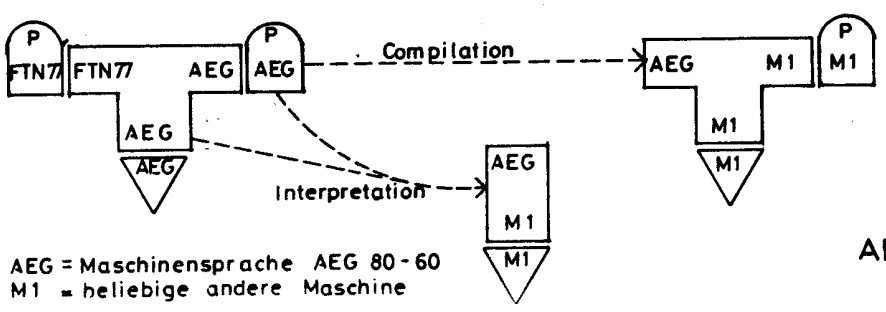


Abbildung C

Dann kann man zwar mit Hilfe eines Interpreters eine Laufumgebung für den Compiler und eine Laufumgebung für die vom Compiler erzeugten Objektprogramme schaffen, aber den Compiler selbst und die von ihm erzeugten Objektprogramme sind wenig effizient. Die virtuelle und reale Maschine AEG80-60 ist nicht der Quellsprache angepaßt, und eine Übersetzung der virtuellen Maschine AEG80-60 auf eine andere reale Maschine bringt große Probleme mit sich.

Der Programmieraufwand liegt um beträchtliches über dem, der bei der Verwendung einer höheren Programmiersprache zu leisten ist. Ein gegen Fehler unanfälliges und verständliches Programm ließe sich im Assembler nur unter großem Codieraufwand erreichen. Bei den heute zur Verfügung stehenden Programmiersprachen, die für Probleme der Systemprogrammierung geeignet sind, läßt sich aber eine ähnlich gute Effizienz wie in Assemblersprachen erreichen. Eine dieser Programmiersprachen ist die auf einer Untermenge von ALGOL68 basierende Sprache SL3.

----SL3

Die Sprache SL3 gliedert sich in zwei Teile, eine ALGOL68-Untermenge und einen Teil zur Systemprogrammierung, der unter anderem über Möglichkeiten der Ausnahme-Behandlung und der Parallelprogrammierung verfügt. Die maschinenabhängigen Konstrukte von SL3 sind dabei streng abgegrenzt, was zur Portabilität von in SL3 erstellten Programmen beiträgt. Im Rahmen einer Compiler-Implementation sollte es möglich sein, diese maschinenabhängigen Konstrukte gar nicht zu benutzen. Lediglich der Codegenerator eines Compilers erzeugt maschinenabhängige Ausgaben, die aber durch hochsprachliche SL3-Konstrukte zu realisieren sind.

Die wichtigsten und für die Implementation von Compilern am besten geeigneten Eigenschaften von SL3 sollen nun vorgestellt werden. In (DÜRR 75) wird eine Übersicht über SL3 geboten,

außerdem sei auf die entsprechenden AEG-Handbücher verwiesen. (AEG 78)

2.1.2 Nützliche Spracheigenschaften von SL3

Auf der Grundlage des ALGOL68-Typenkonzeptes stellt SL3 eine syntaktisch und semantisch sauber definierte Sprache dar.

SL3 ist wie ALGOL68 eine Ausdrucks-Sprache, jede Anweisung (SL3: jeder Abschnitt) kann wie ein Ausdruck einen Wert liefern. Dies läßt eine kompakte und durchsichtige Notation von Programmen zu.

SL3 hat die ALGOL68-Blockstruktur. Die Kontrollstrukturen IF-, CASE- und DO-Abschnitt bilden genauso wie ein BEGIN-Abschnitt eine vollständige Klammerung um einen neuen Block, der aus einer Folge von Deklarationen und Anweisungen bestehen kann. Dadurch können Namen gezielt dort eingesetzt werden, wo sie gebraucht werden, was zur Lesbarkeit eines Programmes beiträgt.

2.1.2.1 Das SL3-Modul- und Prozedurkonzept

Ein SL3-Modul ist eine Zusammenfassung von Konstanten-, Daten- und Prozedur-Objekten, die von den Modulklammern nach außen abgeschlossen sind. Ein SL3-Programm besteht im allgemeinen aus einer Anzahl von Moduln, die dem Übersetzer getrennt übergeben werden können.

Beziehungen zwischen Moduln werden explizit durch GLOBAL- und VALGLOB (=external)-Attribute bei der Objektdeklaration spezifiziert. Solche Objekte können Konstanten (GLOBAL CONST), Variablen (GLOBAL) oder Prozeduren (GLOBAL PROC) sein. Der auf die globalen Objekte zugreifende Modul spezifiziert durch ein entsprechendes VALGLOB-Attribut seine Zugriffswünsche auf Objekte in anderen Moduln.

Ein Informationsaustausch zwischen Scanner und Parser in einem Compiler kann zum Beispiel durch eine globale Variable mit dem MODE TOKEN geschehen:

```
MODE TOKEN = STRUCT (INT KIND,
                     INT ATTR);
MODULE SCANNER : GLOBAL   TOKEN SYM;
MODULE PARSER  : VALGLOB  REF TOKEN SYM;
```

Durch die Spezifikation VALGLOB REF wird der Übergabemechanismus für die Variable SYM angedeutet, übergeben wird eine Referenz auf ein Objekt der Art TOKEN, der Speicherplatz von SYM liegt im deklarierenden Modul.

Sprachgerecht wird in SL3 die Zusammengehörigkeit von GLOBAL-deklarierter und VALGLOB-spezifizierter Prozedur beschrieben. Außer dem Namen der Prozedur müssen auch noch ihre Parameter durch Typ und Anzahl im spezifizierenden Modul angegeben werden:

```
MODULE A :  GLOBAL PROC  P = (REF INT I, INT J) ...
MODULE B :  VALGLOB PROC (REF INT , INT) P;
```

SL3 ermöglicht durch drei Prozedurarten eine effiziente Unterprogramm-Technik. Die Prozedurenart PROC ist die mächtigste dieser drei Arten, aber auch die aufwendigste. Sie ist vergleichbar mit der ALGOL68PROC und benötigt umfangreiche Laufzeit-Unterstützung (dynamische Felder, Display). Eine einfachere Art ist die PROCN, die keinen Zugriff auf umgebende Prozedurvariablen hat und nur statische Felder zuläßt. Die einfachste und nichtrekursive Prozedurart bildet die SUBR, die ohne Laufzeitunterstützung auskommt. Das Zugriffsverhalten der Prozeduren ist dabei durch den Compiler abprüfbar, wodurch der Programmierer weitestgehend unterstützt wird.

2.1.2.2 SL3 - Datenstrukturen

SL3-Felder und -Strukturen lassen sich sehr gut durch die Aggregat-Notation handhaben. Statt Feld- oder Struktur-Elemente einzeln zu belegen, kann dies in einer abkürzenden Schreibweise geschehen:

```
CONST     INT  MAX  ≠:= 3;
MODE SYMBOL = ........
MODE PRODUCTION = [1:MAX]  SYMBOL;
PRODUCTION  CURRENT;
SYMBOL  EXPR , PLUSOP , TERM;

CURRENT  := [EXPR , PLUSOP , TERM] ;  (* Aggregat *)
```

Dies führt zu einer übersichtlichen Schreibweise und ermöglicht es dem SL3-Compiler, effizienten Code für eine Aggregatzuweisung zu erzeugen.

Die Vorbelegung von konstanten Tabellen zur Übersetzungszeit ist mit Hilfe dieser Aggregatnotation möglich:

```
CONST   INT   PMAX   ≠ : = 4;
/* EXPR , PLUSOP etc. seien konstante Symbole */
CONST  [1:PMAX] PRODUCTION ALLPRODUCTIONS  ≠:=
     [[EXPR , PLUSOP , TERM],
      [TERM , MULOP , FACTOR],
      [FACTOR , POWEROP , PRIMARY],
      [LEFTPAR , EXPR , RIGHTPAR]];
```

Die Handhabung von Bitfeldern auf hochsprachlichem Niveau wird durch einen Datentyp LBITn ermöglicht. Ein Objekt der Art LBITn

ist ein Feld von n Bits, auf dem die Operationen Konjektion, Disjunktion, Zuweisung etc. erlaubt sind. Damit läßt sich das Zusammensetzen von Maschinenwörtern in einem FORTRAN 77-Codegenerator in SL3 beschreiben:

```
MODE  REGREGINSTR  =  STRUCT  (  LBIT8 OPCODE,
                                 LBIT4 REGISTER1,
                                 LBIT4 REGISTER2);

MODE  REGMEMINSTR  =  STRUCT  (  LBIT8 OPCODE,
                                 LBIT4 REGISTER,
                                 LBIT4 BASE,
                                 LBIT16 ADRESS);

CONST     LBIT8     LOAD  Ø:= H'BC';    /*hexadecimal*/
CONST     LBIT8     ADD   Ø:= H'21';
CONST     LBIT4     STACK Ø:= B'1011';  /*binary*/

REGREGINSTR     ADDREG1REG2 ;
REGMEMINSTR     LOADREGFROMMEM ;

ADDREG1REG2   := [ADD, Ø , 8];      /*Add R8 to RØ */
LOAD_REGFROMMEM := [LOAD, 2, STACK, 80];
                          /*Load R2 from stack +80*/
```

2.1.3 Auswahl von SL3

Die Sprache SL3 erfüllt die Anforderungen zur Durchführbarkeit und Modularisierbarkeit des Compilerprojektes. Das Modul-Konzept gehört zur Sprache, Schnittstellen zwischen Moduln lassen sich durch sprachgerechte SL3-Konstrukte spezifizieren.

Auch den äußeren Anforderungen nach geringem Programmieraufwand und Portabilität wird SL3 gerecht. Durch höhere Daten- und Kontrollstrukturen ist SL3 sehr gut zur Implementation eines Compilers geeignet. Die Portabilität des erstellten Objektes

wird dabei durch das Übersetzungssystem der AEG80-60 sichergestellt. (→ Kap. 2.3)

Die vorhandenen höheren Kontroll- und Datenstrukturen legen eine effiziente Programmierung nahe. Damit ergeben sich leicht verständliche und weitestgehend korrekte Programme.

2.2 Aufgliederung des Compilers in Moduln

Grundlage für den Entwurf eines komplexen Programmes sollte eine Spezifikation sein, auf die hier kurz eingegangen werden soll. Sie muß außer der Definition der Aufgaben des zu erstellenden Programmes auch noch die Entwurfsziele weiter eingrenzen. (HORNING 74)

Die Aufgabenstellung ist im vorliegenden Fall durch die in (ANSI78) definierte Quellsprache und die durch entsprechende Handbücher definierte Zielmaschine (AEG80) klar gegeben.

1. Die wichtigste Randbedingung an den Entwurf des FORTRAN77-Compilers stellt die Forderung nach geringen Übersetzungszeiten dar. Da die Geschwindigkeit eines Compilers auf der AEG80-60 wesentlich durch Dateioperationen beeinflußt wird, ist eine Einpaßübersetzung angebracht. Die Sprache FORTRAN77 verfügt über keine Konzepte, die eine Einpaßübersetzung unmöglich machen. Auch die Ziel- und Implementationsmaschine ist groß genug, um eine Einpaßübersetzung zuzulassen.

2. Eine weitere Anforderung an den Compiler ist die leichte Änderbarkeit und Erweiterbarkeit der Sprache. Der Compiler muß an eine vorauszusehende neue Standardisierung von FORTRAN anpaßbar sein. Deshalb sollte der Compiler derart modular aufgebaut sein, daß sich Änderungen leicht lokal durchführen lassen.

3. Der SL3-Übersetzer macht eine wesentliche Einschränkung bezüglich der Größe der zu entwickelnden Moduln. Ein dem Übersetzer übergebener Modul sollte möglichst nicht größer als etwa 2000 Quellzeilen sein. Durch diese Bedingung ist eine modulare Struktur eine notwendige Voraussetzung zur Durchführbarkeit des Projektes.

4. Ein selbstverständliches Entwurfsziel bildet die Portabilität des Compilers und seine Adaptierbarkeit an andere Zielmaschinen. Das Programm sollte keinen maschinenabhängigen Code enthalten und eine eindeutige Schnittstelle zur Zielmaschine bieten.

2.2.1 Das Zwischensprachenmodell

Als erster Schritt zur Zerlegung des FORTRAN 77-Compilers in kleinere Teile soll ein Sprachenmodell betrachtet werden, wie es auch für Mehrpaßübersetzer verwendet wird. (JONSSON79), (HARTMANN77)

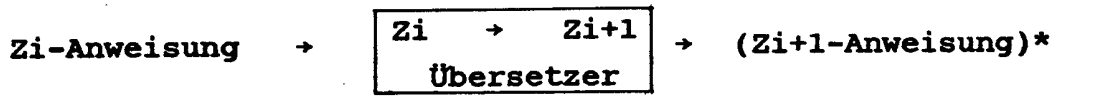
Die Abbildung Quellsprache→Zielsprache kann in eine Folge von Abbildungen

Quelle = Zø→Z1, Z1→Z2, ..., Zn-1→Zn = Zielsprache

zerlegt werden. Der Compiler zerfällt also in eine Anzahl von kleineren Moduln, die einfachere Sprachen Zi nach Zi+1 übersetzen.

Bei einer Mehrpaßübersetzung werden die Übersetzungen von Zi→Zi+1 durch sequentiell nacheinander ablaufende Phasen - (Moduln) des Compilers durchgeführt. Die Zwischensprachenprogramme Zi werden dabei auf Hilfsdateien gespeichert und der nächsten Phase übergeben. Eine Sprache, die Vorwärtskontexte zur Übersetzung bestimmter Sprachkonstrukte benötigt, ist nur in mehreren Pässen übersetzbar. Ein anderer Grund für eine Mehrpaßübersetzung ist meistens der beschränkte Speicher des verwendeten Rechners.

Auch ein Einpaßübersetzer kann diese Zwischensprachenstruktur haben. (WULF 75) Der Unterschied zur Mehrpaßübersetzung liegt nur in der Übergabeeinheit von Modul i zu Modul i+1; ein Mehrpaßübersetzer übergibt vollständige Programme, formuliert in der Zwischensprache Zi, während ein Einpaßübersetzer jeweils nur eine Anweisung der Zwischensprache Zi übergibt. Aus dieser Anweisung in Zi entsteht eine Folge von Anweisungen der Sprache Zi+1, die dann jeweils den nächsten Modul aktivieren:



Damit kommt man zu einem Modell von wechselseitig aktiven Koroutinen, die jeweils einen Teil der Übersetzung durchführen. Durch die kleine Übergabeeinheit (die Anweisung in Zi) ist eine Zwischenspeicherung von vollständigen ZiProgrammen nicht notwendig.

Im idealen Falle lassen sich die Phasen des Compilers als parallele Prozesse realisieren, die jeweils über Zi-Puffer als Producer und Consumer miteinander kommunizieren. Dieser Fall ist nur dann realisierbar, wenn vollständige Rückkopplungsfreiheit zwischen den Phasen besteht. In FORTRAN gibt es leider einige Probleme, die diese Rückkopplungsfreiheit einschränken (→2.2.3).

Das Koroutinen-Modell soll als Grundlage für die weitere Aufgliederung des FORTRAN 77-Compilers in kleinere Teile dienen. Die Koroutinen werden dabei als SL3-Moduln realisiert.

Es stellt sich die Frage, ob dieses Modell eine Einschränkung an die zu verwendenden Zwischensprachen bedeutet. Die Programme in den Zwischensprachen müssen als lineare Folge von Anweisungen darstellbar sein. Weiterhin müssen sie derart lokal übersetzbar sein, daß kein Vorwärtskontext zu ihrer Übersetzung notwendig ist.

Aus der Literatur (AHO77), (COMP74) sind drei Klassen von Zwischensprachen für Compiler bekannt, die Operatorsprachen-(n-tupel), die Kellersprachen und die baumartigen Sprachen. Operatorsprachen und Kellersprachen haben die Eigenschaft der linearen Übersetzbarkeit.

Die baumartigen Sprachen scheinen eine Ausnahme zu bilden. Doch auch eine als Baum dargestellte Zwischensprache muß zu ihrer Übersetzung linearisiert werden (WAITE74a)

Ein typisches Beispiel dazu bildet der von einem Syntaxanalysator aufgestellte Ableitungsbaum, der zu seiner Abarbeitung traversiert werden muß. Das Traversieren bedeutet eine Linearisierung des Baumes, so daß dieser linear an die nächste Phase weitergegeben werden kann, was meistens durch Keller- oder Operatorsprachen geschieht. (→Kap 3), aus denen dann der Ableitungsbaum eindeutig wieder rekonstruierbar ist.

Das vollständige Aufstellen eines Zwischensprachenbaumes ist dann notwendig, wenn eine globale Optimierung durchgeführt werden soll. Aus einer linearen Repräsentation des Baumes ist dieser in einem potentiell existierenden Optimierer aber wieder rekonstuierbar.

Die Forderung nach Modularisierung des Compilers läßt sich mit dem Zwischensprachenmodell am besten erfüllen. Jeder Zwischensprache Zi wird ein Modul zugeordnet, das Zi-Anweisungen in Zi+1-Anweisungen übersetzt.

Der Codegenerator-Modul hat die Aufgabe, Objektcode zu erzeugen. Seine Eingangssprache kann als Schnittstelle zur Zielmaschine dienen, seine inneren Eigenschaften sollten für die anderen Moduln verborgen sein. Die Anforderung an den Compiler, bezüglich der Quellsprache leicht änderbar zu sein, läßt sich durch geeignete Wahl der anderen Zwischensprachen erfüllen.

2.2.2 Ein allgemeiner Ansatz

In diesem Teilkapitel sollen die oben eingeführten Zwischensprachen Zi für den FORTRAN 77-Compiler mit Namen versehen werden. Die wichtigste Eigenschaft eines portablen und gut strukturierten Compilers ist seine Aufteilbarkeit in einen sprachabhängigen Teil und einen maschinenabhängigen Teil. Als Schnittstelle zwischen diesen beiden Teilen wird meistens eine maschinenunabhängige Zwischensprache definiert (POOLE 74).

Der FORTRAN 77-Compiler soll deshalb zunächst in einen sprachabhängigen Analyseteil, dessen Aufgabe die Analyse des eingegebenen Quelltextes und die Erzeugung einer internen Darstellung des Quellprogrammes ist, und in einen maschinenabhängigen Syntheseteil, dessen Aufgabe die Erzeugung von Objektcode ist, zerlegt werden. Die beiden Teile können über eine gemeinsame Schnittstelle, definiert durch eine FCODE genannte Sprache, miteinander kommunizieren:

Quelle → FCODE
Analyse

FCODE → Objektcode
Synthese

Die Aufgabe der Sprache FCODE ist die semantische Beschreibung des Quellprogrammes in einer kompakten Form. Sie muß deshalb möglichst alle Operatoren der Quellsprache bieten und eine Handhabung der Datenstrukturen von FORTRAN 77 ermöglichen. Sie sollte aus dem Quellprogramm möglichst einfach generierbar sein, etwa durch eine syntaxgesteuerte Übersetzung. Trotzdem muß die Sprache FCODE mächtig genug sein, um eine gute Optimierung

FCODE → FCODE
Optimierung (global)

zu ermöglichen. In Kapitel 3 wird eine Sprache vorgestellt werden, die diese Eigenschaften hat und der Sprache FORTRAN 77 angepaßt ist, so daß sich effizienter Maschinencode erzeugen läßt.

Diese Entkopplung von sprach- und maschinenabhängigem Teil hat einen weiteren Vorteil. Bei Erkennen eines fehlerhaften Quellpro-

grammes läßt sich die Codeerzeugung völlig abschalten, so daß ein fehlerhaftes Programm wesentlich schneller analysiert werden kann.

Der Sprachanalyseteil bietet eine sinnvolle Form der Zerlegung an. Die Quellsprache FORTRAN 77 ist durch ihre lexikalischen, syntaktischen und semantischen Eigenschaften definiert. Da die Quellsprache leicht änderbar sein soll, ist es angebracht, der Analyse jeder dieser Eigenschaften einen eigenen Modul zuzuordnen. Damit lassen sich Änderungen und Erweiterungen in Klassen einteilen und sind in Compiler lokal durchführbar.

Durch diese Zerlegung wird die kontextsensitive Sprache FORTRAN-77 in drei Schichten aufgeteilt: in eine lexikalische Schicht, die von einem endlichen Automaten erkannt werden sollte, in eine syntaktische Schicht, die von einem deterministischen, möglichst kontextfreien Parser analysiert werden sollte, und in eine semantische Schicht, in der kontextsensitive Elemente der Quellsprache analysiert werden. Diese drei Schichten lassen sich durch eine Folge von Sprachen beschreiben:

$$\text{Quelle} = \underset{\text{Lex}}{\text{Zlex} \rightarrow \text{Zsyn}} \qquad \underset{\text{Syn}}{\text{Zsyn} \rightarrow \text{Zsem}} \qquad \underset{\text{Sem}}{\text{Zsem} \rightarrow \text{FCODE}}$$

In den drei Moduln Lex, Syn und Sem durchläuft das Quellprogramm eine Folge von Transformationen und wird durch die Zwischensprachen immer weiter analysiert. Kontextsensitive Elemente der Sprache FORTRAN 77, wie die Zusammengehörigkeit von initialen und terminalen DO-Statement, können dabei vom Modul Syn in den Modul Sem verlegt werden (Kapitel 3). Nach erfolgreicher Analyse soll der Modul Sem ein dem Quellprogramm semantisch äquivalentes Programm in der Sprache FCODE generieren.

Ein Problem stellt die Linearisierung des Ableitungsbaumes im Modul Syn dar. Die Attribute der inneren Knoten des Ableitungsbaumes müssen berechnet und an den Modul Sem weiterge-

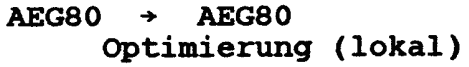
leitet werden. Bei einer Trennung von syntaktischer und semantischer Analyse ist der Modul Syn aber nicht in der Lage, die Attributauswertung vollständig durchzuführen. Der Ableitungsbaum muß also in einer abstrakten Form weitergereicht werden.

Jeder der drei Moduln Lex, Syn und Sem sollte auch im Fehlerfall nur richtige Zwischensprachenanweisungen erzeugen. Im Falle eines fehlerhaften Quellprogrammes muß eine Fehlermeldung und eine Fehlererholung des entsprechenden Moduls stattfinden.

Der Syntheseteil des Compulers ist von der zu erzeugenden Zielsprache abhängig. Für die AEG80-60 bietet sich eine Zerlegung in einen Codegenerator und einen "lexikalischen" Teil an.

Der Codegenerator soll aus den Befehlen und Objekten der Sprache FCODE Maschinencode erzeugen. Der Maschinencode muß zur Weitergabe an den Binder des Programmiersystems in eine externe Form aufbereitet werden, was in einem Modul Bin gesehen soll:

Der maschinenabhängige Teil kann noch einen zusätzlichen Optimierungsmodul enthalten, der für eine gute Ausnutzung der vorhandenen Maschinenbefehle sorgen soll. Befehlsfolgen für a:=a+1 können durch eventuell vorhandene atomare Operationen a :=: 1 ersetzt werden. Die Schnittstellensprache für den maschinenbezogenen Optimierer ist die Zwischensprache AEG80:

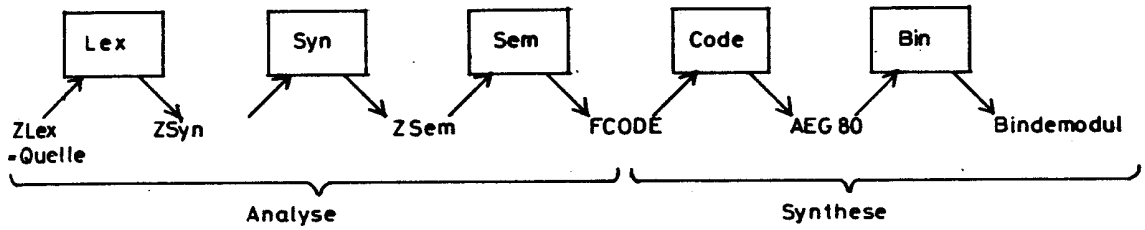


Die Schnittstellensprache AEG80 hat in der Testphase eine wichtige Bedeutung. Es ist ratsam, in einer ersten Implementation Assemblerprogramme an Stelle der Bindemoduln zu erzeugen, da diese besser lesbar sind als Maschinencode:

FCODE → AEG80 AEG80 → WASS (Assembler)
Code Ass

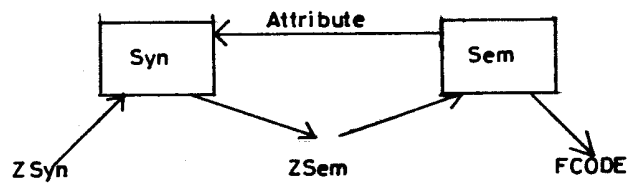
Die exakte Schnittstelle erlaubt dann ein einfaches Austauschen der Moduln Ass und Bin nach der Validierung des Compilers.

Aus diesen Überlegungen ergibt sich folgende vorläufige Struktur für den zu erstellenden Compiler:



2.2.3 Probleme bei FORTRAN 77

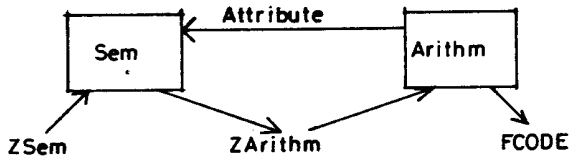
Die Syntax der Sprache FORTRAN 77 läßt sich nicht vollständig kontextfrei darstellen. Die Unterscheidung von Feldindizierung und Funktionsaufruf kann auf die semantische Analyse verschoben werden, wenn man die syntaktischen Regeln <arrayid> (<indexlist>) und <procid> (<aktparlist>) zu einer abstrakten Regel <id> (<exprlist>) zusammenfaßt (FELDMAN 79). Für eine gute syntaxgesteuerte Übersetzung ist es aber sinnvoller, die Erkennung von Feldindizierung oder Funktionsaufruf in der Syntaxanalyse durchzuführen. Der FORTRAN 77-Parser benötigt dafür Informationen, die sich aus der semantischen Analyse des Programmes ergeben. Es muß deshalb eine Rückkopplung zwischen den aufeinanderfolgenden Moduln Syn und Sem bestehen:



Da diese Rückkopplung ein Ausnahmefall ist, läßt sich der Modul Syn trotzdem getrennt entwickeln. Die Attributrückgabe läßt sich mit Hilfe einer einzigen booleschen Prozedur verwirklichen. Diese Prozedur kann bei der Entwicklung des Moduls Syn standardmäßig vorbelegt werden und später bei der Integration durch eine entsprechende Prozedur im Modul Sem realisiert werden.

Ein weiteres Problem stellt sich durch die Größe des Moduls Sem im Falle FORTRAN 77. Die Attribut-Sammlung für FORTRAN-Bezeichner ist eine umfangreiche Aufgabe. Der Modul Sem muß außerdem die Adreßvergabe für Bezeichner im COMMON-Blöcken und EQUIVALENCE-Anweisungen durchführen. Weiterhin muß eine Compilezeitrechnung für PARAMETER-Statements und zur Berechnung von Feldgrenzen stattfinden. Durch die Mixed-Mode-Arithmetik sind arithmetische Operatoren erst am Ende eines Ausdrucks festlegbar. Ebenso sind Intrinsic-Funktionen auch erst nach Auswertung ihrer Parameter zu identifizieren.

Deshalb muß der Modul Sem (-alt) in zwei Moduln Sem (-neu) und Arithm zerlegt werden. Der Modul Arithm soll die Aufgaben der Compilezeitrechnung, Bearbeitung von Mixed-Mode-Ausdrücken und Intrinsic-Funktionen sowie die Erzeugung von FCODE Befehlen übernehmen.



Auch hier besteht eine Rückkopplung zwischen den aufeinanderfolgenden Moduln, die durch die Auswertung von konstanten Ausdrücken in PARAMETER-Statements und die Berechnung von Feldgrenzen notwendig wird.

2.2.4 Kommunikation zwischen Moduln

Die Moduln des FORTRAN 77-Compilers sollen über Sprachen miteinander kommunizieren. Eine Sprache läßt sich in ihre Operatoren und ihre Operanden aufteilen. Während die Operatoren der Zwischensprachen meistens disjunkte Mengen bilden, sind die Operanden der verschiedenen Sprachen oft gleich. Vom Compiler zu behandelnde Operanden lassen sich in drei Klassen aufteilen, Konstanten, Variablen und temporäre, vom Compiler generierte Objekte. Eine vom Modul Lex generierte Konstante würde von Modul zu Modul weitergegeben werden und müßte in jedem dieser Moduln neu gespeichert werden, so daß es zu einer mehrfachen Speicherung dieser Konstanten käme. Deshalb sollte, um Speicherplatz und Laufzeit zu minimieren, eine zentrale Tabellenverwaltung die Speicherung von Zwischensprachenoperanden übernehmen.

Ein Modul Tables soll eine Speicherung von arithmetischen und textuellen Konstanten, von Bezeichnern und temporären Objekten übernehmen. Zwischensprachenoperanden sind dann durch Zeiger auf die entsprechenden Objekte zu ersetzen und im gesamten Compiler eindeutig identifizierbar. Die zentrale Tabellenverwaltung ermöglicht es, die Tabellen variabel zu konfigurieren und eventuelle Engpässe zu erkennen. In der Testphase hat man dadurch einen zentralen Zugriff auf alle im Compiler verwendeten Operanden und kann diese leicht ausgeben.

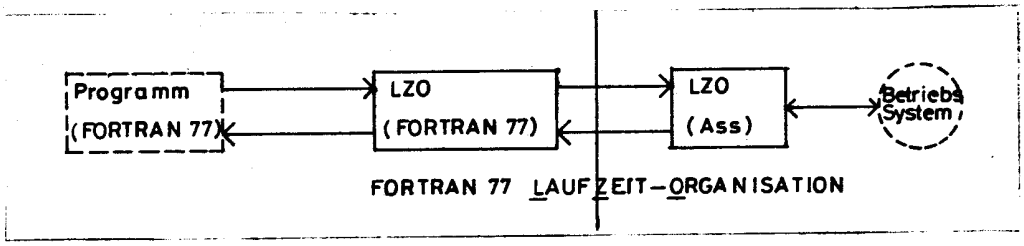
Ebenso sollte ein Modul Error existieren, das die Sammlung von Fehlermeldungen und deren Ausgabe übernimmt. Fehler lassen sich in FORTRAN oft erst weit nach ihrem Auftreten diagnostizieren. Der Sprachanalyseteil des Compilers muß deshalb zu jedem Quellsymbol Informationen über dessen Position im Quelltext mitführen. Fehler sollen dem Modul Error durch Fehleridentifikation und -position mitgeteilt werden und nach dem Ende der Analyse zusammen ausgegeben werden.

Diese Überlegungen führen zu zwei zusätzlichen Moduln Tables und Error, auf die alle anderen Moduln Zugriff haben:

2.3 Überlegungen zur Portabilität

Der in SL3 geschriebene Compiler ist durch das portable Übersetzungssystem selbst voll portabel. Für die Übertragung des FORTRAN 77-Compilers auf einen anderen Rechner müßte der Codeerzeugungsteil an die neue Maschine angepaßt werden. Die beiden Moduln Code und Bin wären dann durch entsprechende neue Moduln zu ersetzen. Damit wäre aber die Übertragung noch nicht abgeschlossen.

FORTRAN 77 benötigt für seine komfortable Ein-/Ausgabe eine umfangreiche Laufzeitunterstützung. Diese läßt sich in einen maschinenabhängigen und einen maschinenunabhängigen Teil aufgliedern. Der maschinenabhängige Teil besteht aus Routinen zum Einlesen und Ausgeben auf externe Kanäle, Routinen zum Positionieren von Dateien und zum Anschluß an das Dateisystem des Rechners. Er wäre im Assembler oder einer anderen geeigneten Sprache zu realisieren. Der maschinenunabhängige Teil soll das Konvertieren von externer in interne Darstellung und das Lesen und Schreiben von internen Dateien übernehmen. Zu seiner Realisierung ist am besten eine höhere Programmiersprache geeignet, nämlich FORTRAN 77. FORTRAN 77 bietet komfortable Möglichkeiten zur Textverarbeitung und hat im Gegensatz zu SL3 eine normale Schnittstelle zu anderen FORTRAN 77-Unterprogrammen. Damit läßt sich die Laufzeitunterstützung für FORTRAN 77 in zwei Moduln, realisiert in Assembler und FORTRAN 77, aufteilen:



LZO(FORTRAN 77) und LZO(Ass) können über die normale Schnittstelle zwischen FORTRAN 77-Unterprogrammen, nämlich Prozeduraufruf und COMMON-Blöcke, miteinander kommunizieren.

Der Modul LZO(Ass) läßt sich hinreichend klein halten, etwa 100 Befehle, so daß der FORTRAN 77-Compiler als hoch portabel angesehen werden kann.

2.4 Die Struktur des FORTRAN 77-Compilers

In einem nächsten Schritt der Verfeinerung sollen nun die Aufgaben der Moduln des Compilers näher eingegrenzt und versucht werden, geeignete Zwischensprachen zu finden. Durch die vollständige Trennung von syntaktischer und semantischer Analyse ist insbesondere die Sprache zwischen den Moduln Syn und Sem nicht konventionell. Anforderungen an die Zwischensprachen ist ihre lineare Übersetzbarkeit und die Möglichkeit, sie in einer Einpaßübersetzung schrittweise ineinander zu überführen.

2.4.1 Die Aufgaben der Moduln

Für den bisher entworfenen FORTRAN 77-Compiler sind drei Betriebsarten denkbar. Die Realisierung der Moduln als parallele Prozesse entspricht am besten dem Zwischensprachenkonzept. Zwei aufeinanderfolgende Moduln müßten dann über einen gemeinsamen synchronisierten Puffer, der die Operatoren und Operanden der jeweiligen Zwischensprache enthielte, miteinander kommunizieren. Zur Rückgabe von Attributen zwischen Sem und Syn bzw. Arithm und Sem wären zusätzliche Synchronisationsoperationen notwendig. Diese Betriebsart erscheint auf einer Einprozessormaschine nicht sinnvoll, da alle CPU-Aktivitäten zwangssequentialisiert werden. Lediglich die Ein-/Ausgabe-Operationen in den Moduln Lex und Bin wären sinnvoll als parallele Nebenaktivitäten zu realisieren. Diese können aber als verdeckte Eigenschaften der beiden Moduln angesehen werden.

Die Realisierung der Moduln als Koroutinen kann leider nur als Denkmodell dienen, da die Sprache SL3 nicht über ein Koroutinenkonzept verfügt und eine Simulierung von Koroutinen mit SL3-Sprachmitteln nur umständlich zu realisieren ist.

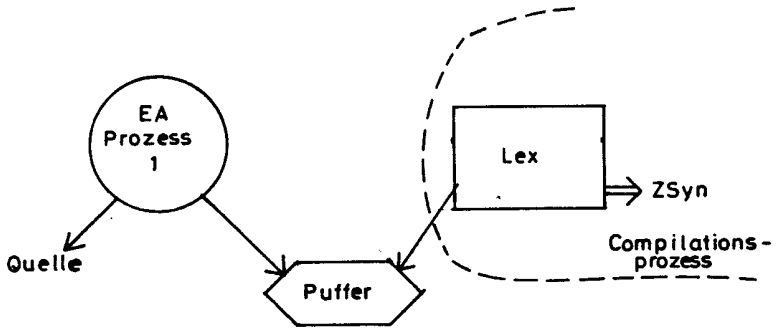
Aus diesen Gründen ist ein sequentielles Programm, das aus einem steuernden Hauptprogramm und einer Anzahl von einander aufrufenden Prozeduren besteht, die sinnvollste Betriebsart. Das Hauptprogramm ist am besten im Modul Syn zu realisieren, der eine syntaxgesteuerte Übersetzung durchführen soll. Das Zwischensprachenmodell bleibt dabei erhalten, die Moduln werden nur in einen aktiven Modul, Syn, und mehrere passive Moduln aufgeteilt. Alle Moduln sind Datenmoduln, sie enthalten permanente Daten, die ihren jeweiligen Zustand kennzeichnen. Sie können deshalb als Koroutinen angesehen werden.

Modul Syn

Syn soll eine syntaxgesteuerte Übersetzung durchführen und bei Bedarf den Scanner im Modul Lex und den Modul Sem aktivieren. Zur Realisierung des Parser-Lookaheads soll zwischen Lex und Syn ein gemeinsamer Puffer existieren, der stets hinreichend viele Lexeme enthalten soll. Syn soll den Ableitungsbaum aufstellen, diesen auf einen "abstrakten Syntaxbaum" (→ Kap 3.3) transformieren und ihn in einer sequentiellen Form an den Modul Sem weiterreichen.

Modul Lex

Die Aufgabe des Scanners ist die Verwandlung des Quellprogrammes in eine Folge von lexikalischen Einheiten, Lexeme. Wie in Kapitel 1.2 angedeutet, ist es auf der AEG80-60 angebracht, die Ein-/Ausgabe für sequentielle Dateien durch eigene Prozesse zu realisieren. Die Aktivierung eines eigenen Eingabeprozesses im Modul Lex kann als innere Eigenschaft dieses Moduls angesehen werden. Die Benutzung einer gepufferten Eingabe im Modul Lex hat keinen Einfluß auf die Umgebung des Moduls im Compiler.



(Realisierung einer gepufferten Eingabe durch
einen parallelen Eingabeprozeß)

Modul Sem

Sem soll die kontextsensitiven Elemente von FORTRAN 77 mit Ausnahme der Mixed-Mode-Ausdrücke und Intrinsic-Funktionen analysieren. Sem soll als Eingabesprache eine abstrakte, redundanzfreie Form des Ableitungsbaumes, den "abstrakten Syntaxbaum", in eine noch zu spezifizierende Ausgabesprache transformieren. Sem muß die Adreßberechnung für Variablen durchführen und weitgehend alle Tabelleneinträge vornehmen. Dazu bedarf es der im Modul Arithm berechneten Attribute von Ausdrücken, die in einer geeigneten Prozedur übergeben werden sollen.

Modul Arithm

Die Aufgabe des Moduls Arithm ist die Auswertung aller Ausdrücke und die Generierung von FCODE-Befehlen. Arithm soll den Compilezeit-Interpreter enthalten und alle konstanten Ausdrücke berechnen. Konstante Ausdrücke und Teilausdrücke sollen nicht auf PARAMETER-Statements und Feldgrenzen beschränkt, sondern bezüglich ihres Auftretens unqualifiziert sein. Das ermöglicht eine erste Optimierung im Compiler, die sogenannte Faltung:

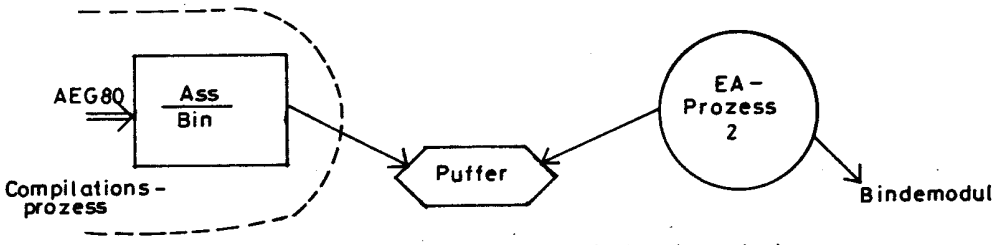
Beispiel : A = B *(5+7)=> A = B * 12

Modul Code

Der Codegenerator soll aus FCODE-Befehlen möglichst einfach, etwa durch eine makroartige Ersetzung, eine interne Repräsentation des Maschinencodes erzeugen. Es soll noch keine lokale Optimierung vorgenommen werden. Diese wird in einen noch zu realisierenden Modul Optim(lokal) verlegt. Der Codegenerator soll bei Bedarf Speicheradressen an temporäre Objekte, die bei der Berechnung von Ausdrücken entstehen, vergeben können. Die Speicherallokation für temporäre Objekte ist maschinenabhängig, also Aufgabe des maschinenabhängigen Teils. Sie hängt von der Anzahl der zur Verfügung stehenden Register der Maschine ab.

Moduln Bin, Ass

Diese beiden Moduln sollen die interne Darstellung des Maschinencodes in eine externe Form zur Übergabe an den Binder bzw. Assembler aufbereiten. Maschinencode soll zu Segmenten zusammengefaßt werden, und globale Namen für Prozeduren und COMMON-Blöcke sollen an das Programmiersystem weitergereicht werden. Die Moduln müssen auch das "Backpatching" für Adressen übernehmen, da in einer Einphasenübersetzung die Adressen von Marken erst nach ihrem Auftreten bekannt sind. Das befreit den Codegenerator von unnötigen Details. Die Moduln können sich zur Ausgabe eines zusätzlichen Prozesses bedienen:



(Realisierung einer gepufferten Ausgabe)

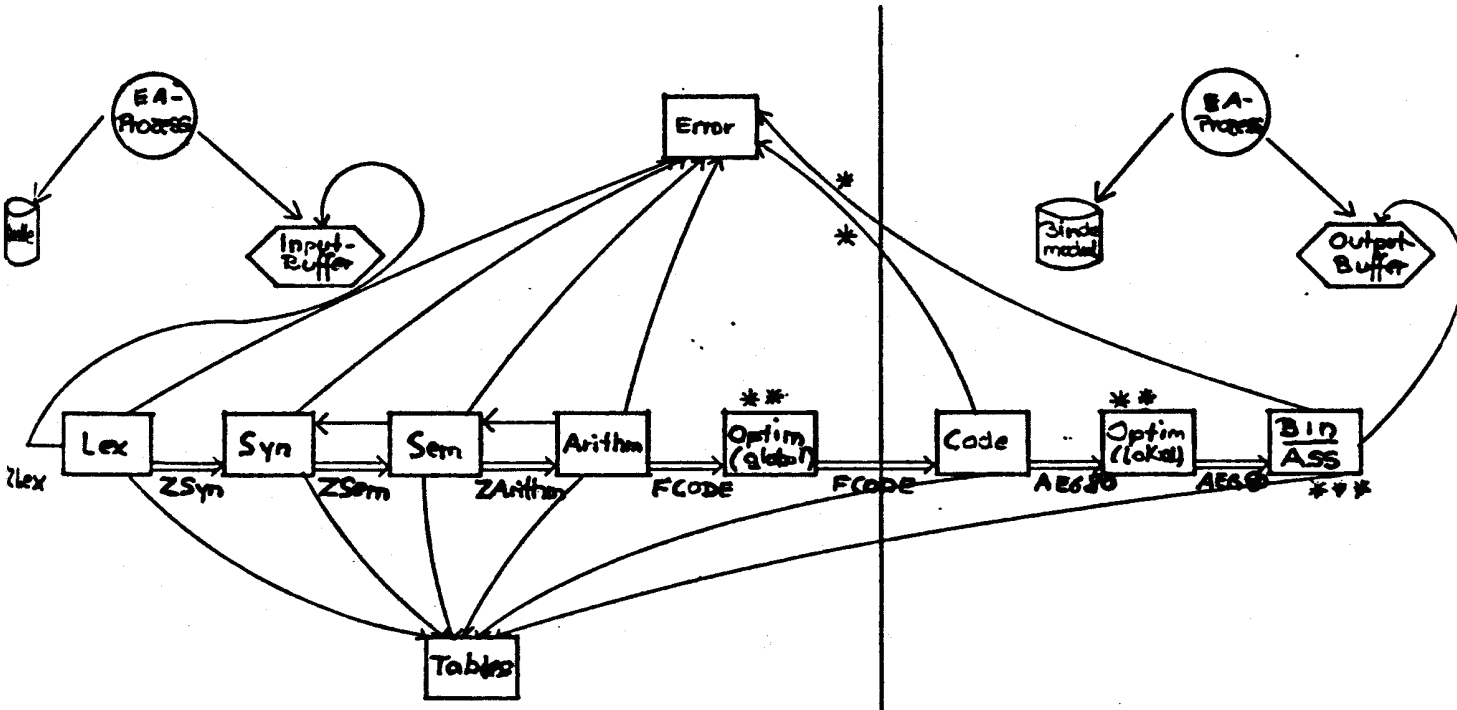
Modul Tables

Der Modul Tables übernimmt die Verwaltung von allen in den Zwischensprachen verwendeten Operanden. Er muß für jeden Operanden eine eindeutige Identifikation generieren, durch die dieser repräsentiert werden kann. Die Verwaltung von Bezeichnern durch eine Hashfunktion oder etwa durch einen binären Baum soll eine verdeckte Eigenschaft des Moduls sein.

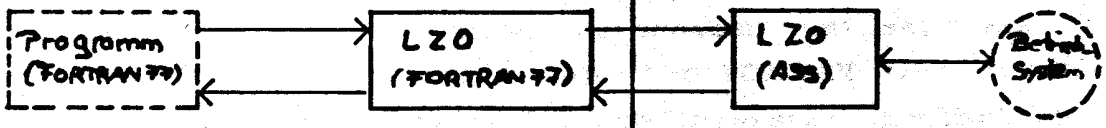
Modul Error

Error soll alle von den anderen Moduln gemeldeten Fehler sammeln und nach ihrer Position im Quelltext sortieren. Am Ende des Compilerlaufes wird er aktiviert und soll die Fehlermeldungen in benutzergerechter Form ausgeben. Die Aktivierung des Moduls Error kann als Ausnahmefall angesehen werden und bedarf keines zusätzlichen Ausgabeprozesses.

Die folgende Abbildung soll einen Überblick über die Grobstruktur des FORTRAN 77-Compilers geben:



FORTRAN 77 COMPILER



FORTRAN 77 LAUFZEIT-ORGANISATION

* nur für Compilerfehler
** nicht realisiert
*** in der Testphase Ass, dann Bin

2.4.2 Schnittstellensprachen zwischen Moduln

Es soll nun versucht werden, die Schnittstellensprachen des FORTRAN 77-Compilers zu charakterisieren. In Kapitel 3 sollen dann die Zwischensprachen und ihre Übersetzer entworfen werden.

ZLex
Diese Sprache ergibt sich aus der Definition der Quellsprache, deren Zeichensatz das Alphabet von ZLex ist. ZLex soll den Parser weitgehend unterstützen, die Regelmenge der Grammatik für ZSyn soll durch geeignete Wahl von ZLex möglichst klein werden. ZLex wird durch eine Anzahl von regulären Ausdrücken für die Lexeme Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen definiert, die als attributierte Terminalsymbole der Sprache ZSyn an die nächste Phase weitergereicht werden.

ZSyn
ZSyn ist die für die Quellsprache definierte syntaktische Sprache, deren Terminalalphabet die Menge der Lexeme ist, die in ZLex erkannt werden. ZSyn sollte von einem deterministischen kontextfreien Parser erkannt werden können. Der Parser für ZSyn soll effizient und leicht realisierbar sein, etwa durch einen Parser-Generator. Deshalb müssen kontextsensitive Regeln der Sprache auf die nächste Phase verschoben werden, was in FORTRAN 77 nicht immer sinnvoll ist. Für FORTRAN 77 empfiehlt es sich außerdem, umständlich zu definierende kontextfreie Konstrukte durch einfachere, allgemeinere Regeln zu beschreiben. Damit kommt man zunächst zu einer größeren Sprache, die in der nächsten Phase zu analysieren ist. Der ZSyn-Parser soll leicht zu einem Übersetzer erweitert werden können, der den Ableitungsbaum in einen "abstrakten Syntaxbaum" verwandelt.

ZSem
Der "abstrakte Syntaxbaum" (McKEEMAN 74) (vergl. Kap. 3.2), der sich nicht durch eine kontextfreie Grammatik beschreiben läßt, ist Grundlage der Analyse der Sprache ZSem. Er beschreibt das

Quellprogramm in einer redundanzfreien Form und geht im wesentlichen aus dem Ableitungsbaum hervor. Kettenableitungen und in dieser Phase bedeutungslose Schlüsselwörter und Begrenzer des Ableitungsbaumes sollen beseitigt sein. Am abstrakten Syntaxbaum ist die Berechnung von aufsteigenden und absteigenden Attributen durchzuführen, was er durch seine kompakte, redundanzfreie Form möglichst erleichtern soll. Arithmetische Ausdrücke können als Teilbäume angesehen werden, deren Übersetzung und Attributberechnung in der Sprache ZArithm durchgeführt wird, wodurch sich die Analyse von ZSem wesentlich erleichtert. Um eine Einpaß-Übersetzungsstrategie beibehalten zu können, soll eine vollständige Aufstellung des abstrakten Syntaxbaumes vermieden werden. Das Traversieren des Baumes muß also schon durch den ZSyn-Parser gesteuert werden können, der den Baum in einer linearen Form übergibt.

ZArithm

Das Primärziel dieser Sprache soll die Auswertung und Übersetzung von arithmetischen, logischen und textuellen Ausdrücken sein. Sie kann deshalb direkt auf diese Anforderungen zugeschnitten sein. Die einfachste Notation für Ausdrücke ist ein Präfix- oder Postfix-Code. Insbesondere Postfix-Code läßt sich sehr einfach in einer syntaxgesteuerten Übersetzung aus den vorhergehenden Sprachen generieren. Postfix-Code erleichtert eine Bottom-Up-Strategie zur Operatoridentifikation bei Mixed-Mode- und Intrinsic-Ausdrücken in FORTRAN 77.

FCODE

Leichte Optimierbarkeit und einfache Übersetzbarkeit bilden Gegensätze, zwischen denen FCODE ein guter Kompromiß sein sollte. Eine Kellersprache wie ZArithm könnte als maschinenunabhängige Zielsprache des Compileroberteils übernommen werden. Eine Kellersprache ist für die in Betracht kommenden Zielmaschinen mit mehreren Registern aber meistens nicht geeignet, speicher- und laufzeiteffizienten Code zu generieren. Operanden müssen im Gegensatz zur Kellermaschine auf der realen

Maschine explizit bekannt sein, um sie in Registern allozieren zu können. Auch ein quellbezogener Optimierer braucht zur Erkennung von identischen Teilausdrücken eine explizite Identifikation eines n-Tupels (Operator, Operand 1, ...Operand n). Für Mehradreß-Maschinen sind gute Algorithmen zur Optimierung bekannt (AHO 77), so daß FCODE aus einem Mehradreß-Code bestehen sollte. Die Übersetzung von ZArithm auf FCODE kann parallel zur Operatoridentifikation erfolgen, wodurch sich kein großer zusätzlicher Aufwand ergibt. Die Datenstrukturen der Sprache FCODE müssen sorgfältig konzipiert werden, um eine Optimierung zu ermöglichen. FORTRAN 77 kennt Adreßvariablen, die für den Programmierer nur indirekt zugänglich sind, nämlich Formalparameter und Feldreferenzen. Sie müssen in den Datenstrukturen von FCODE enthalten sein. Pseudoinstruktionen für Marken sollten ebenfalls in FCODE enthalten sein, um eine Einpaßübersetzung zu ermöglichen.

2.5 Testbarkeit des Compilers

Das Zwischensprachenmodell bietet sehr gute Möglichkeiten zum Testen des Compilers.

Fast alle Moduln sind vor der Integration in den Compiler separat testbar. Die Moduln bilden relativ einfache, kleine Programmstücke, deren Aufgabe die Sprachübersetzung von Zi nach Zi+1 ist. Das Testen der Moduln beschränkt sich auf die korrekte Durchführung dieser Aufgabe. Dazu muß ein einfaches Generatorprogramm für jeden Modul existieren, das Zi-Anweisungen erzeugt:

Zi- Quelle → Modul Mi → Zi+1-Senke

Nach der Integration lassen sich Fehler im Compiler durch die wohldefinierten Schnittstellen zwischen den Moduln gut lokalisieren. Die gemeinsame Tabellenbehandlung aller Moduln erleich-

tert dabei die mnemotechnisch günstige Ausgabe von Zwischensprachenoperatoren und -operanden in der Testphase. Ein Fehler im Compiler läßt sich dann leicht auf eine fehlerhafte Übersetzung von Zi nach Zi+1 eingrenzen, und der fehlerhafte Modul Mi kann umgebungsunabhängig korrigiert werden.

Zum Testen des FORTRAN 77-Compilers müssen geeignete Testprogramme zur Verfügung stehen. Für numerische Probleme existieren unzählige in FORTRANIV geschriebene Programme, der Bereich der Zeichenverarbeitung ist aber völlig neu in FORTRAN 77. Die Erzeugung solcher Testprogramme läßt sich mit der Entwicklung der Laufzeithilfen kombinieren. Die FORTRAN 77-Ein-/Ausgabe benötigt Routinen zur Konversion von numerischen Größen in Zeichenfolgen und umgekehrt. Diese lassen sich sehr gut in FORTRAN 77 realisieren und sind gleichzeitig gute Testprogramme. Die Ein-/Ausgabe muß dann um wenige Assemblerroutinen erweitert werden, die für die Kommunikation mit dem Betriebssystem zuständig sind.

3 Schrittweise Übersetzung von FORTRAN 77

Nachdem in Kapitel 2 eine geeignete Grobstruktur für den FORTRAN 77-Compiler entworfen wurde, sollen nun die einzelnen Moduln des Compilers verfeinert werden.

Die Kommunikation zwischen den Moduln des Compilers erfolgt mit Ausnahme der Tabellenbehandlung und der Fehlermeldung durch Zwischensprachen. Die entsprechenden Zwischensprachen müssen spezifiziert werden, und es sind entsprechende Übersetzungsalgorithmen zu entwerfen.
Das Zwischensprachenmodell ist ein logisches Modell zur Trennung der Phasen des Compilers. Es dient dazu, dem Compiler eine übersichtliche, modulare Struktur zu geben. Dadurch lassen sich alle Moduln des Compilers getrennt entwickeln und testen. Die physikalische Realisierung des Compilers entspricht jedoch

einer Einpaß-Übersetzungsstrategie, bei der eine Anzahl wechselseitig aktiver Koroutinen eine schrittweise Übersetzung der Zwischensprachen durchführen. Durch diesen Ansatz werden die Vorteile einer Mehrpaßübersetzung mit denen einer Einpaßübersetzung kombiniert.

Die Moduln Lex, Syn, Sem und Arithm bilden den Analyseteil des Compilers. Sie akzeptieren jeweils eine Eingangssprache und übersetzen diese Sprache auf die Eingangssprache des nächsten Moduls. Die Analyse der Quellsprache wird mit der Erzeugung einer maschinenunabhängigen Zwischensprache FCODE abgeschlossen. Der aus den Moduln Code und Bin bestehende Syntheseteil des Compilers generiert aus den FCODE-Befehlen Maschinencode für die zugrundeliegende Zielmaschine AEG80-60. Eine Adaptierung des Compilers an andere Zielmaschinen ist durch das Ersetzen des Syntheseteils möglich.

Auf die Moduln zur Tabellenbehandlung und Fehlermeldung soll hier nur kurz eingegangen werden. Die Tabellenbehandlung (Modul Tables) für FORTRAN 77-Programme ist relativ einfach realisierbar. Sie setzt sich aus der Konstantentabelle, der lokalen Symboltabelle, der Tabelle für temporäre Objekte und der globalen Namenstabelle zusammen, für die jeweils prozedurale Schnittstellen existieren. Der Fehlermodul (Modul Error) hat die Aufgabe, Fehlermeldungen zu sammeln, sie nach ihrer Position im Quelltext zu sortieren und am Ende des Compilerlaufes in einer benutzergerechten Form auszugeben.

Der Modul Bin, dessen Aufgabe die Erzeugung eines "Bindemoduls" für die Weitergabe an das Programmiersystem des Rechners ist, soll ebenfalls nicht näher erläutert werden.

3.1 Lexikalische Analyse

Die lexikalische Analyse soll das Quellprogramm in eine interne Repräsentation im Compiler verwandeln. Schlüsselwörter, Bezeichner, Konstanten und Sonderzeichen müssen erkannt werden und als attributierte Token an die Syntaxphase weitergereicht werden. Dazu dient meistens ein endlicher Automat, der die mit Hilfe von regulären Ausrücken beschriebenen Lexeme erkennen muß. Die Eingangssprache ZLex des Moduls Lex wird also durch die Folge

(Schlüsselwort/Bezeichner/Konstante/Sonderzeichen)*

mit den entsprechenden (regulären) Ausdrücken für Schlüsselwort etc. beschrieben.

3.1.1 Lexikalische Analyse in anderen Programmiersprachen

Für die meisten höheren Programmiersprachen ist die lexikalische Analyse der einfachste Teil des Compilers. DeRemer (DEREMER 74) unterteilt die lexikalische Analyse in "Scanning" und "Screening". "Scanning" bezeichnet das Aufbrechen des Quelltextes in Lexeme wie Wörter, Operatoren, Kommentare, Leerzeichen etc. "Screening" bedeutet die Einteilung der Lexeme in die oben aufgeführten Klassen unter Beseitigung von Kommentaren und Leerzeichen und die Erkennung von Schlüsselwörtern.

In Sprachen wie PASCAL, SIMULA oder ADA sind Schlüsselwörter reservierte Wörter, die im Programm nicht als Bezeichner verwendet werden dürfen. Einige ALGOL-Implementationen verlangen eine explizite Klammerung von Schlüsselwörtern in Hochkommata und ermöglichen dadurch auch deren Benutzung als Bezeichner. Solche Programme sind für den Benutzer schlecht lesbar, die Aufgabe des Scanners wird aber wesentlich vereinfacht. Eine Zeichenfolge 'BEGIN' ist dann stets als das Schlüsselwort BEGIN identifizierbar. In anderen Implementationen übernimmt das

Leerzeichen teilweise die Bedeutung des Hochkommas. In PASCAL oder SIMULA kann das Schlüsselwort BEGIN nur in dem Kontext

(' ' / ';') 'BEGIN' (' ')

auftreten.

Das Leerzeichen hat in solchen Sprachen eine relevante Bedeutung. Eine Folge

BEGIN A:=B END

hat nach Elimination von Blanks eine völlig andere Bedeutung:

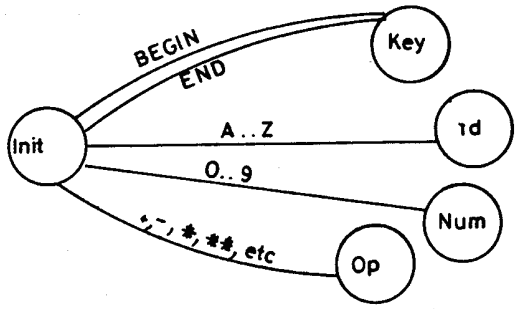
BEGINA:=BEND

und muß vom Compiler eventuell als fehlerhaft zurückgewiesen werden.

Leerzeichen und Zeilenende haben etwa in ADA nur durch ihr einmaliges Auftreten eine lexikalische Relevanz. Eine Wiederholung dieser Zeichen ist durch den "Scanner" auf ein einfaches Auftreten reduzierbar:

' ' (' ')* <=> ' ', EOL (EOL)* <=> EOL

In den erwähnten Sprachen werden Quellprogramme als eine völlig formatfreie Folge von Zeichen dargestellt, die durch Anfang und Ende der Datei begrenzt werden: TOP (Z)* EOF. Ein lexikalischer Analysator für diese Sprachen läßt sich als einfacher nichtdeterministischer endlicher Automat darstellen, dessen Kanten mit den Schlüsselwörtern und Sonderzeichen bezeichnet werden:



und ist leicht in ein Programm umsetzbar:

```
READ(CH);       WHILE   CH  ≠  EOF  DO
                CASE    CH  OF
                'A'..'Z'    :   [DORKEYWORD;
                'Ø'..'9'    :   NUMBER;
                u.s.w.
                END;
```

Wesentlich schwieriger ist das "Screening" in PL/I, das keine reservierten Wörter kennt. Das "Screening" ist vom Kontext abhängig. Zum Beispiel ist eine Zeichenfolge IF(A) noch nicht als der Beginn eines IF-Statements identifizierbar, IF könnte ein Feldbezeichner sein. Erst das Finden eines Buchstabens nach der geschlossenen Klammer legt 'IF' als das Schlüsselwort IF fest : IF(A) THEN....oder IF(A) A=B im Gegensatz zu IF(A)=Ø.

Da in einem PL/I-IF-Statement beliebig geklammerte boolesche Ausdrücke erlaubt sind, ist das "Screening" nicht mehr mit einem endlichen Automaten realisierbar. Die IF-Bedingung muß als richtig geklammerter Ausdruck erkennbar sein. Der Präfix IF (<expr>) wäre mit = Ø zu einer Zuweisung fortsetzbar, während die Fortsetzung THEN... den Präfix als den Beginn eines IF-Statements erkennbar macht.

Ein effizienter lexikalischer Analysator für PL/I ist deshalb nur im Zusammenhang mit dem Parser und der semantischen Analyse (Attributauswertung für Bezeichner: <arrayid> → 'I F') realisierbar.

3.1.2 Die Bedeutung des Scanners in FORTRAN 77

In FORTRAN läßt sich das einfache Modell des "Scanning" und "Screening" für die lexikalische Analyse nicht anwenden. Wegen der völligen Bedeutungslosigkeit von Leerzeichen und der Abwesenheit von reservierten Wörtern führt die lexikalische Sprache für FORTRAN 77, ZLex, für den allgemeinen Fall aus der Klasse der CHOMSKY-3-Sprachen, die nicht durch einen endlichen Automaten erkennbar sind, heraus.

Sehr unschön ist die völlig veraltete starre Lochkartenanordnung von FORTRAN 77-Quellzeilen. Eine 72 Zeichen lange Quellzeile wird in die ersten sechs Zeichen mit Sonderbedeutung und den Rest der Zeile aufgegliedert. FORTRAN-Statements sind auf

eine Zeile mit den entsprechend gesondert gekennzeichneten Fortsetzungszeilen beschränkt. Ein "Prescanner" muß hier zunächst eine Transformation des Statements auf

<Markenteil> <Zeichenfolge> EOS, EOS=END_of_Statement

vornehmen. Dabei ergeben sich die ersten beiden erkannten Lexeme, das End_of_Statement-Lexem und, falls vorhanden, das Statement_Label-Lexem.

Die Analyse der Zeichenfolge wird nun im Zusammenhang mit der Ermittlung der nicht reservierten Schlüsselwörter ähnlich problematisch wie in PL/I. Allerdings macht die völlige Bedeutungslosigkeit von Blanks:

{' '} <=> ε , außer in Formaten oder Strings,

es unmöglich, zunächst ein "Scanning" durchzuführen, um dann im "Screening" die Schlüsselwörter herauszufinden.

Die Notwendigkeit, für FORTRAN einen vorausschauenden buchstabierenden Automaten zu entwerfen, sei am bekannten Beispiel des DO-Statements demonstriert. Der Präfix

DO 100 I = 1

ist nach der Elimination von Blanks zunächst mit dem Präfix

DO100I=1

äquivalent. Er ist mit DO100I=1 EOS zu einer Zuweisung fortsetzbar, und mit DO 100 I=1,10 EOS bedeutet er den Anfang eines initialen DO-Statements. Im ersten Fall muß die Zeichenfolge in die Lexeme

DO100I = 1 EOS

im zweiten Fall in

DO 100 I = 1 , 10 EOS

zerlegt werden. Ein Automat, der diese beiden Fälle unterscheiden soll, muß deshalb zeichenweise vorgehen. Nach Finden des Präfixes 'DO' muß eine Vorausschau auf den Rest der Zeichenfolge stattfinden. Ein FORTRAN IV-Scanner hatte es noch relativ leicht, diese beiden Fälle zu unterscheiden. Ein DO-Statement war mit dem folgenden regulären Ausdruck zu erkennen:

```
'DO' //  NUMBER   NAME   '='   BOUND  ','  BOUND REST
NUMBER  = ('Ø' .. '9')     ('Ø' .. '9')*
NAME    = ('A' .. 'Z')     (('Ø' .. '9')/('A' .. 'Z'))*
BOUND   = (NAME / NUMBER)
REST    = (BOUND EOS / EOS)
  //    :  Vorschau-Operator[1])
```

Ebenso war eine Zuweisung auf die Feldvariable IF durch den regulären Ausdruck:

```
'IF' //  '('  INDEXLIST  ')'  '='  CHARACTERS  EOS
INDEXLIST  =  (NAME/NUMBER)  (','(NAME/NUMBER))*
CHARACTERS =  (beliebiges FORTRAN-Zeichen)*
```

von einem IF-Statement zu unterscheiden.

Einige Compiler-Generatoren stellen dem Benutzer einen Scanner-Generator zur Verfügung, in dem mittels des Vorschau-Operators [1] solche FORTRAN-Konstrukte durch reguläre Ausdrücke beschrieben werden können. Feldman (FELDMAN 79) benutzte für seine FORTRAN 77-Implementation im UNIX-System ein spezielles Programm, das eine Voranalyse von FORTRAN-Quellprogrammen durchführt.

Es ist zweifelhaft, ob solche FORTRAN IV-Scanner für FORTRAN 77 noch ausreichend sind. Die lexikalische Analyse der beiden obigen Konstrukte wird in FORTRAN 77 dadurch erschwert, daß in BOUND und INDEXLIST jeweils beliebige geklammerte Ausdrücke auftreten können:

```
BOUND      → EXPRESSION
INDEXLIST  → EXPRESSION (',' EXPRESSION)*
```

Die Vorschau nach Erkennen der Präfixe 'DO' oder 'IF' läßt sich also nicht mehr durch einen regulären Ausdruck beschreiben. Der erkennende Automat muß richtig geklammerte Ausdrücke auswerten können. Im praktischen Fall beschränkt sich dieses Erkennen auf eine Zählung der öffnenden und schließenden Klammern, um nach Auffinden der letzten schließenden Klammer festzustellen, ob der folgende Suffix den Bedingungen eines DO-Statements oder einer Zuweisung genügt:

INDEXLIST := BOUND := {weT* / $card_{'('}$ (w) =$card_{')'}$ (w)}

Da der Standard eine maximale Länge von zwanzig Quellzeilen für ein FORTRAN 77-Statement vorschreibt, ist die Zwischenspeicherung von Statements mit beschränktem Aufwand möglich. Der erkennende Automat wird durch diese Beschränkung wieder zu einem endlichen Automaten.

Eine solchermaßen umständlich zu erkennende lexikalische Sprache kann dann in eine Folge von Lexemen der Sprache FORTRAN 77 zerlegt werden. Ein Lexem soll im folgenden als ein Tripel

LEXEM = (Art, Attribut, Position)

angesehen werden. Die Art des Lexems dient der weiteren syntaktischen, das Attribut der semantischen Analyse, seine Position ist nur für Fehlermeldungen relevant. Die Attribute sollen Zeiger auf die entsprechenden Tabellen für Konstanten und Namen sein. Sofern es sich um ein Schlüsselwort oder Sonderzeichen handelt, sind die Zeiger undefiniert. Der Modul Lex soll die erhaltene Lexemfolge

(LEXEM)* EOF-LEXEM

an den Parser weitergeben, dessen Grammatik auf der Menge der Lexeme, die im Anhang I aufgeführt sind, definiert sein soll.

1) An dieser Stelle ist eine alternative lexikalische Analyse einer Zuweisung fortzuführen, falls der reguläre Ausdruck nicht erkannt wird (vergleiche (AHO 77) p 108 ff).

3.2 Geeignete Parsing-Verfahren für FORTRAN 77

Die meisten existierenden FORTRAN-Compiler machen keinen expliziten Gebrauch von einer Rahmensyntax für FORTRAN-Programme, sondern behandeln ein Programm als eine Folge von Anweisungen. Die Aufgaben des Parsers können dann schon größtenteils vom Scanner übernommen werden. Der FORTRAN IV(H)-Compiler (IBM66)

benutzt lediglich für die Auswertung von arithmetischen Ausdrücken ein formales Syntaxanalyseverfahren; ähnlich verfährt der FORTRAN IV-Compiler für die AEG80-60(AEG76). FORTRAN IV-Ausdrücke lassen sich etwa durch eine Operator-Grammatik darstellen und in polnische Notation umwandeln, die dann als Grundlage für die weitere Übersetzung dienen kann.

Der FORTRAN 77-Standard gibt im Anhang eine Rahmensyntax für FORTRAN77-Programme in Form von Syntaxdiagrammen an, die nicht direkt für formale Parsingverfahren zu verwenden ist ("The charts have been designed for human readability, not as a basis for parsing."). Es muß also erst eine Umformung der Syntaxdiagramme in Backus-Naur-Form oder eindeutige Syntaxdiagramme erfolgen, um sie in einen Parser umsetzen zu können. Eine daraus resultierende Grammatik ist sehr umfangreich und gibt teilweise den Sinn der Sprachkonstruktionen schlecht wieder. Feldman (FELDMAN 79) benutzte eine LALR(1)-Grammatik mit 270 Produktionen und 97 Terminalsymbolen, aus der ein zur Verfügung stehendes Compiler-erzeugendes System einen Parser generierte. Im folgenden sollen einige Probleme aufgezeigt werden, die sich beim Aufstellen einer Grammatik für die Sprache FORTRAN 77 ergeben.

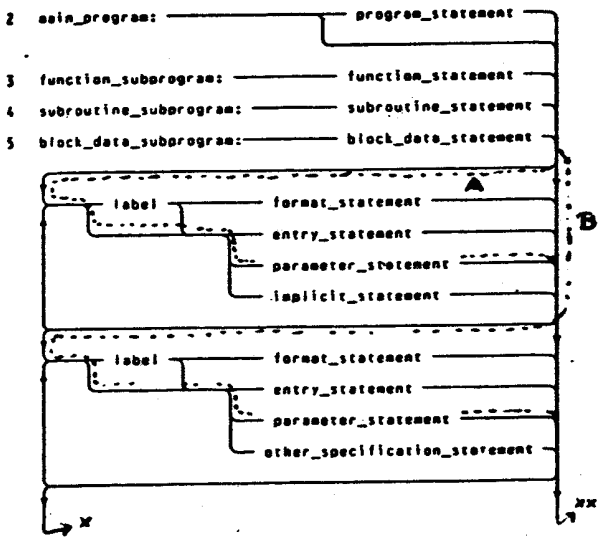
3.2.1 Die Syntax einer gewachsenen Sprache

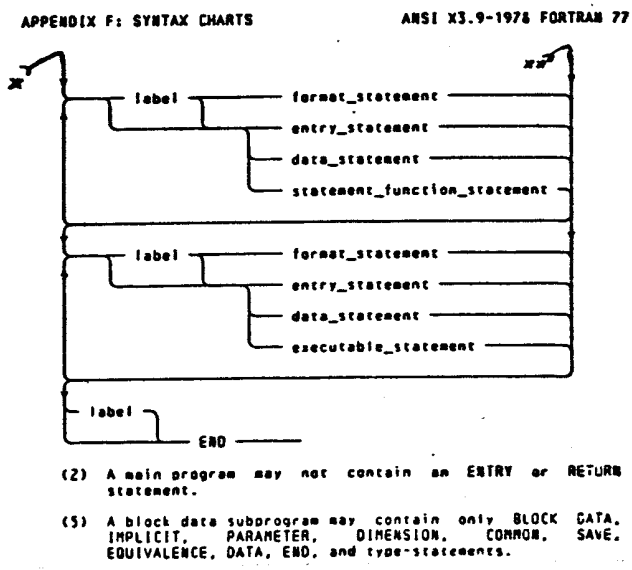
Unabhängig von der gewählten Parser-Strategie, Top-Down oder Bottom-Up, bilden die kontextsensitiven Regeln der Sprache als größte Problem der Syntaxerkennung (→1.1). Ein FORTRAN77-Parser muß entweder ganz darauf verzichten, DO-Loop-Klammerung, Funktionsaufrufe etc. zu erkennen, und eine allgemeinere Sprache zulassen oder aber einen Zugriff auf die in der semantischen Phase berechneten Attribute haben. In einem handgeschriebenen Parser lassen sich diese Zugriffe auf die Attribute von Symbolen für die Parser-Entscheidung recht gut realisieren. Bei Benutzung eines Parser-Generators kann davon kaum Gebrauch gemacht werden.

Feldman behandelt in seiner LALR(1)-Grammatik Formelfunktionen und Funktionsaufrufe überhaupt nicht und überläßt deren Analyse der semantischen Phase.

Wünschenswert wäre für FORTRAN77 ein Parsing-Verfahren auf der Basis von attributierten Grammatiken (MILTON 79), also ein deterministisches kontextsensitives Syntaxanalyseverfahren. Ein kontextfreier Parser muß eine allgemeinere Sprache als FORTRAN 77 zulassen und die weitere Analyse der semantischen Phase überlassen.

Die durch die Syntaxdiagramme definierte Sprache ist nicht eindeutig, wie der folgende Ausschnitt zeigt:





Für das einfache Beispielprogramm

```
SUBROUTINE   S
PARAMETER    (I=10)
ENTRY   A
I = 1
END
```

bestehen zwei Möglichkeiten, das PARAMETER-Statement abzuleiten, Weg A oder Weg B. Das ENTRY-Statement kann gar auf vier verschiedenen Wegen im Syntaxdiagramm erreicht werden. Diese Mehrdeutigkeit hat im obigen Fall keine semantische Bedeutung. Ein PARAMETER- bzw. ENTRY-Statement ist eine Pseudo-Anweisung,

die aus der vorhergehenden (eventuell impliziten) Deklaration von Bezeichnern eindeutig interpretierbar ist.

Die Syntax von FORTRAN 77 ist sehr umfangreich. Ein/Ausgabe-Statements haben eine syntaktisch schwer zu beschreibende Form. Ein READ-(WRITE-,OPEN-etc.) Statement kann eine Anzahl von sogenannten "io-modifiers" enthalten, die weggelassen werden können, wenn die Reihenfolge der IO-Parameter eingehalten wird, sonst sind sie obligat und dürfen nur einmal auftreten. Die folgenden Formen eines READ-Statements sind jeweils semantisch äquivalent:

```
READ (6,100)   I
READ (UNIT=6,100)   I
READ (6,FMT=100)   I
READ (UNIT=6,FMT=100)   I
READ (FMT=100,UNIT=6)   I
```

Das entsprechende Syntaxdiagramm für diese Sprachkonstrukte läßt sich nur durch eine große Anzahl von Regeln in EBNF ausdrücken (Es gibt insgesamt 22 "IO-modifier" in neun verschiedenen IO-Statements). Diese Regeln lassen sich in einer allgemeineren Grammatik besser beschreiben und müssen dann in der semantischen Phase analysiert werden.

Manche Sprachkonstrukte sind nur bei Kenntnis ihrer Umgebung zu interpretieren. Der Zeichenstring '2*3' hat in Ausdrücken die Bedeutung eines Terms:

$$\langle expr \rangle \overset{*}{\rightarrow} \langle term \rangle * \langle factor \rangle \overset{*}{\rightarrow} 2*3$$

kann aber in DATA-Statements als ein Listenelement mit der semantischen Bedeutung einer Liste '3,3' auftreten:

$$\langle list\text{-}elmt \rangle \rightarrow \langle replikator \rangle \langle const \rangle \rightarrow \langle const \rangle * \langle const \rangle \overset{*}{\rightarrow} 2*3$$

Der String '(Ø.Ø,1.Ø)' bedeutet je nach Umgebung eine komplexe Konstante oder die Parameterliste einer Prozedur:

A = (Ø.Ø,1.Ø) oder A=F (Ø.Ø,1.Ø) .

Eine Bottom-Up-Analyse von FORTRAN 77 durch einfache Präzedenzgrammatiken ist nicht möglich, denn im ersten Fall besteht ein Präzedenzkonflikt zwischen '*' und <const>, im zweiten Fall ist

die Phase (<real> , <real>) entweder zu <argument -list> oder zu <complex-const> zu reduzieren, wobei mehrere Präzedenzkonflikte auftreten. (Aus diesem Grunde sind komplexe Konstanten auch nicht durch den Scanner zu erkennen, der ja ein endlicher Automat ist.) Eine Botton-Up-Analyse von FORTRAN 77 ist deshalb nur mit einem mächtigeren Verfahren wie LALR(1)- oder SLR(1)-Analyse durchführbar.

Die Darstellung der Sprache durch eine LL(k)-Grammatik ist durch die Eingeschränktheit der LL(k)-Sprachen (LR(K)-Sprachen) auch nicht möglich. Es gibt außer den erwähnten kontextsensitiven Regeln drei Nicht-LL(k)-Konstrukte in FORTRAN 77.

Das erste tritt in CHARACTER-Ausdrücken auf, in denen ein <substring> durch folgende Produktionen beschrieben werden kann:

i : <substring>→<name> (<first>:<last>)
ii : <substring>→<array-reference> (<first>:<last>)
<array-reference>→<name> (<expr>{,<expr>})
<first>,<last> → expr / ε

Der Präfix "<name> (<expr>"ist den beiden alternativen Produktionen i und ji gemeinsam, und da für alle k :/<expr> > K ist, besteht hier eine LL(k)-Verletzung.

Ein zweiter LL(k)-Konflikt liegt bei Ein-/Ausgabe-Statements vor. Ein <io-list-elmt> kann folgende Ableitungen haben:

i : <io-list-elmt>→<expr>
ii: <io-list-elmt>→<io-do-list>
<io-do-list> → (<io list>,<name> =<do-range>)
<io-list> → <io-list-elmt>{,<io-list-elmt>}

Die Mengen $First_K$ (i) und $First_{K}$(ii) haben hier für alle K unter anderem den nichtleeren Durchschnitt ('(')^K.

Ein dritter LL(k)-Konflikt ergibt sich aus den Produktionen

i <unit-Kind> → PROGRAM <name> / ε
ii <unit-Kind> → <function-type> FUNCTION...

Die Mengen $Follow_K(i)$ und $First_K(ii)$ haben für K>3 den Durchschnitt

CHARACTER * ((^{n} .

Die Top-Down-Analyse von FORTRAN 77 ist deshalb nur als ein Verfahren mit Rücksetzung (MAYER 78) durchführbar.

3.2.2 Auswahl des Verfahrens

Nach Ersetzen der kontextsensitiven Konstrukte durch allgemeinere kontextfreie Regeln wäre also nur ein LR(1)-artiges Verfahren (SLR(1),LALR(1) oder kanonisches LR(1)) zur effizienten Syntaxanalyse für FORTRAN 77 anwendbar. Man kann aber ein Syntaxanalyseverfahren für einen Compiler nicht auswählen, ohne die anderen Teile des Compilers zu betrachten. Die Syntaxanalyse sollte einen Rahmen bilden, der den Übersetzungsvorgang steuert.

a LL - versus LR -Verfahren

Ein LR(1)-artiges Verfahren hat allgemein und im vorliegenden Fall einige Nachteile.

LR-Analysetabellen lassen sich nur schwerlich per Hand aufstellen, sondern müssen durch einen Parsergenerator erzeugt werden. Die Anzahl von Zuständen für eine FORTRAN 77-Grammatik wäre zu groß. Auf der AEG80-60 steht jedoch kein Parser-Generator zur Verfügung.

Eindeutigkeitskonflikte lassen sich in einem LR-Parser schlecht per Hand behandeln, die Analysetabellen wären an mehreren Stellen zu korrigieren.

Semantische Aktionen und Attributauswertungen sind bei der Bottom-Up-Analyse nur nach einer Reduktionsoperation durchführbar und lassen sich nicht direkt in eine Produktion einstreuen. Kontextsensitive Regeln für Formelfunktionen und Funktionsaufrufe lassen sich dann nicht direkt übersetzen. Eine Formelfunktion

F(I) = <expr>

wäre erst nach der Reduktion von <expr> in der semantischen Analyse zu erkennen. Die <expr> darf aber zu diesem Zeitpunkt noch nicht übersetzt sein, da ihre Übersetzung von der Liste der formalen Parameter der Funktion F abhängig ist. Ein formaler Parameter (hier I) ist eine lokale Variable der Formelfunktion und ist von einer eventuell existierenden anderen Variablen mit dem Namen I im umgebenden Programm zu unterscheiden. Es müssen also alle Ausdrücke zunächst zwischengespeichert werden und können erst nach Kenntnis des Kontextes übersetzt werden.

Ebenso umständlich muß die allgemeine Form für <array-reference> und <function-reference> zunächst zwischengespeichert und zu einem späteren Zeitpunkt übersetzt werden.

Diese Probleme lassen sich beseitigen, wenn der Parser einen Zugriff auf die semantischen Attribute von Bezeichnern hat.

Ein Top-Down-Parser kann seine Entscheidungen von der Auswertung dieser Attribute abhängig machen und an einer geeigneten Stelle in einer Produktion eine semantische Aktion bewirken.

Im Beispiel der Feldindizierung wäre dann zwischen

<primary> → <array-name> (<indexlist>)

und <primary> → <function-name> (<act-arg-list>)

durch die Attribute für

<array-name> <function-name>

eine kontextsensitive Parser-Entscheidung zu treffen, die sich in einem handgeschriebenen Parser leicht ermöglichen läßt.

Ein kontextsensitiver Top-Down-Parser kann sofort nach Erkennen des Präfixes "<name>(" am Anfang eines Statements die Entscheidung treffen, ob es sich hier um eine Zuweisung oder eine Formelfunktion handelt, und dann die entsprechenden semantischen Aktionen bewirken.

Die attributgesteuerte Top-Down-Analyse ermöglicht dadurch eine wirkungsvolle syntaxgesteuerte Übersetzung der obigen kontextsensitiven Konstrukte von FORTRAN 77.

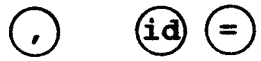
Ein deterministischer Top-Down-Parser kann jedoch die beiden Nicht-LL(k)-Konstrukte nicht erkennen. Eine Linksfaktorisierung ist in beiden Fällen nicht möglich. Die Auflösung solcher Ableitungskonflikte kann aber durch eine geeignete Lookahead-Prozedur geschehen, die eine Top-Down-Analyse mit Rücksetzen simuliert.

Im Falle des io_list_elmt müssen zur Unterscheidung der Produktionen i und ii (vergl. Kap. 3.2.1 p) die Präfixe "(<io_list_elmt>,<name>=" und "(<expr>)" unterschieden werden. Es muß also über einen richtig geklammerten Ausdruck hinausgeschaut werden. Diese Vorausschau läßt sich durch eine Wortmenge von Token beschreiben, in die die konkrete Syntax von <expr> nicht eingeht:

$$LA(i) = \{w \varepsilon T^* / w = "(u, id = " \quad , \; u \varepsilon T^* \text{ mit } card_{,(,}(u) = card_{,),}(u)\}$$

$$LA(ii) = \{w \varepsilon T^* / w \notin LA(i)\}$$

Die Lookahead-Prozedur zählt die öffnenden und schließenden Klammern und versucht nach der letzten schließenden Klammer die Tokenfolge



zu finden.

Die zweite LL(k)-Verletzung (vergleiche Kap. 3.2.1) kann durch Auswertung der Attribute von <name> aufgelöst werden. Handelt es sich um einen <arrayname>, so ist Produktion ii zu wählen, sonst Produktion i.

Ein kontextsensitiver Top-Down-Parser für FORTRAN 77 hat somit nur geringe Nachteile gegenüber der deterministischen LR(k)-Analyse. Seine Vorteile bestehen jedoch in der vereinfachten und effizienteren syntaxgesteuerten Übersetzung. Weiterhin läßt er sich wesentlich einfacher als ein LR(k)-Parser realisieren. Die syntaktische Analyse von FORTRAN 77 soll deshalb mit Hilfe eines Top-Down-Parsers durchgeführt werden.

b rekursiver Abstieg versus "predictive Parsing"

Es gibt zwei Möglichkeiten, ein Top-Down-Syntaxanalyse-Verfahren zu realisieren, den rekursiven Abstieg oder die tabellengesteuerte Analyse (bzw. "predictive parsing" (LEWIS 76)). In (MAYER 78) wird aus Gründen der Laufzeiteffizienz für die tabellengesteuerte Form der LL(k)-Analyse plädiert. In einem modular aufgebauten Compiler sprechen aber auch noch andere Gründe für die tabellengesteuerte Form (McKEEMAN 74).

Für die Methode des rekursiven Abstiegs ist eine Grammatik in erweiterter BNF erforderlich (EBNF hat gegenüber BNF den Vorteil der geringeren Anzahl von Produktionen und damit Prozeduren). Dann läßt sich ein Parser durch eine relativ kleine Anzahl von rekursiven Prozeduren einfach und schnell programmieren. In einen "recursive descent parser" müssen aber die semantischen Prozeduren zur Codeerzeugung eingeflochten werden; Syntaxanalyse und semantische Analyse lassen sich nicht gut physikalisch trennen.

Das widerspricht den in Kapitel 2 gestellten Entwurfszielen; außerdem ist ein "rekursive descent parser" mit eingeflochtenen semantischen Prozeduren für den vorliegenden Fall (Beschränkung der Größe eines Moduls) viel zu umfangreich.

Der zweite Nachteil dieses Verfahrens ist in der relativ schwierigen Fehlererholung zu sehen. Den rekursiven Prozeduren

müssen jeweils Synchronisationstoken mitgegeben werden, auf die im Fehlerfall aus einer vielfach geschachtelten Aufrufsfolge zu synchronisieren wäre.

Aus diesen Gründen ist das "predictive parsing" für den FORTRAN 77-Compiler besser geeignet. Ein "predictive parser" für eine LL(1)-Grammatik besteht aus einem Keller mit den Zugriffsprozeduren push und pop und einer LL(1)-Entscheidungstabelle, deren Einträge jeweils ein Nonterminal mit den Auswahlmengen für dessen Produktion enthalten.

In der Sprache SL3 läßt sich ein handgeschriebener "predictive parser" für die FORTRAN 77-Grammatik sehr übersichtlich und einfach realisieren.

3.2.3 Ein Top-Down-Parser für FORTRAN 77

Ein tabellengesteuerter LL(2)-Parser besteht aus einem Keller mit den Operationen push und pop und einer Kontrollstruktur in Form einer LL(2)-Entscheidungstabelle, die als eine Menge von Tripeln

$(N , select_p , p: N \rightarrow S1.....Sn)^*$

N : oberstes nichtterminales Kellersymbol

p : Produktionsnummer

$select_p$: Auswahlfunktion für Produktion p

$select_p$: TOKEN x TOKEN → (T,F)

angesehen werden kann. Schon für kleine Grammatiken können die Auswahlmengen für $select_p$, die Elemente der Potenzmenge von TOKEN x TOKEN sind, sehr groß werden. Da der FORTRAN 77 Scanner 110 TOKEN unterscheidet, ist die Realisierung einer vollständigen LL(2)-Tabelle also unmöglich.

Bei näherer Betrachtung der Grammatik ergibt sich aber, daß die Auswahlmengen für $select_p$ in den meisten Fällen die LL(1)Bedingungen erfüllen, so daß dann ein einfacher Lookahead aus-

reichend ist. Diese Eigenschaft der Grammatik läßt sich ausnutzen, um die Entscheidungstabelle des Parsers auf ein realisierbares Maß zu reduzieren. Der Parser muß dann zwischen LL(1)- und LL(2)-Produktionen unterscheiden, wobei die Anzahl der LL(2)-Produktion sehr klein ist:

$$select_p : \begin{cases} TOKEN \rightarrow (T,F) & , LL(1)\text{-Auswahl} \\ TOKENxTOKEN \rightarrow (T,F) & , LL(2)\text{-Auswahl} \end{cases}$$

Für die kontextsensitiven Regeln und die Nicht-LL(K)-Konstrukte in FORTRAN 77 muß das LL(2)-Verfahren weiter modifiziert werden. Die Funktion $select_p$ besteht in diesen Fällen aus einer Attributauswertungsfunktion oder einer speziellen Lookahead-Prozedur LA(p):

$$select_p : \begin{cases} \text{wie oben} & , LL(k)\text{ - Auswahl} \\ LA(p) \rightarrow (T,F) & , \text{für Nicht-LL(k)-Konstrukte} \\ P\_Attribute(p) \rightarrow (T,F) & , \text{für kontextsensitive Regeln} \end{cases}$$

Ein solcher Parser läßt sich in der Sprache SL3 übersichtlich und effizient realisieren.
Mit den Deklarationen

```
MODE SYMBOL =....;    /*  Nonterminals and TOKEN */
MODE PRODUCTION = STRUCT(INT LENGTH,
                         [1:PMAX] SYMBOL RIGHT-SIDE);
```

kann eine Produktion p fast wörtlich von der Grammatik in SL3 übertragen werden:

```
Grammatik : <EXPR> → <TERM> <TERMS>
            <TERMS> → PLUSOP <TERM> <TERMS>
            <TERMS> → ε

SL3:   /* EXPR*/      [2,[TERM,TERMS,NONE,..]];
       /*TERMS*/      [3,[PLUSOP,TERM,TERMS,NONE,...]];
       /*TERMS*/      EMPTY-PRODUCTION
```

Um diese günstige Eigenschaft der Sprache SL3 weiter auszunutzen, muß die Kellerprozedur push die ganze rechte Seite einer Produktion in den Keller bringen können:

```
MODE PARSESTACK = [1:SMAX] SYMBOL;
PROC PUSH = (PRODUCTION P)....;
          /*push right side of production P
            in reverse order onto stack */
PROC POP =( )...;    /* pop one symbol */
```

Eine Auswahl zwischen den beiden Alternativen

```
i  : <TERMS>→ PLUSOP <TERM> <TERMS>
ii : <TERMS>→ ε
```

wird dann in SL3 folgendermaßen realisiert:

```
.....
/*TERMS is Top-of-Stack */
POP():
IF...../* Select_i  */    THEN
     P:= [3,[PLUSOP, TERM, TERMS, NONE...]]
ELSE
     IF... /* select_ii  */    THEN
       P:= EMPTY-PRODUCTION
     ELSE
       ERROR(); RECOVER()
     FI
FI;
PUSH(P)
```

Die hauptsächliche Arbeit bei der Implementation eines solchen Parsers für FORTRAN 77 besteht also in der Berechnung der Funktionen $select_i$ für die Produktion i.

Für LL(1)- und LL(2)-Regeln müssen die Mengen $First_K(1)$ und $Follow_K(1)$, K=1 oder K=2, berechnet werden und in den booleschen Funktionen $select_p$ durch logische Ausdrücke dargestellt werden.

In den beiden Fällen der Nicht-LL(K)-Konstrukte sind die Funktionen $select_p$ durch die speziellen Lookahead-Funktionen zu ersetzen.

Für das kontextsensitive Konstrukt des Funktionsaufrufes bzw. der Array-Referenz wird die Parser-Entscheidung durch die Auswertung der Attribute von Namen getroffen:

```
/*  PRIMARY     is Top-of-Stack    */
POP();
IF .....    /* First Token is a name */    THEN
     IF    P_ARRAY_ID(FIRST_TOKEN)    THEN
           P:=[3, [NAME, INDEXLIST, SUBSTRING, NONE..]]
     ELSE
           IF    P-FUNCID(FIRST-TOKEN)    THEN
             P:=[2, [NAME,ARGLIST,NONE....]]
           ELSE
            ....
           FI
     FI
FI;
PUSH(P)
```

3.3 Erweiterung des Parsers zu einem Übersetzer

Die Aufgabe des Parsers ist neben der Analyse der durch die Grammatik definierten Eingangssprache (ZSyn) das Generieren einer für die semantische Analyse relevanten Ausgabesprache (ZSem). Im einfachsten Fall besteht ZSem aus der Menge der vom Parser erfolgreich analysierten Ableitungsbäume, die vollständig an den semantischen Analysator weitergereicht werden können. Am Ableitungsbaum können dann durch (eventuell mehrmaliges) Traversieren dieses Baumes die semantischen Attribute von nichtterminalen Symbolen berechnet werden. Der Ableitungsbaum ist aber zu diesem Zwecke viel zu redundant. Er enthält Kettenableitungen und Schlüsselwörter, die für die semantische Analyse nur implizite Bedeutung haben.

3.3.1 Der abstrakte Syntaxbaum

Die Redundanz des Ableitungsbaumes für die semantische Analyse wird am Beispiel einer einfachen Grammatik G für arithmetische Ausdrücke deutlich. Diese Grammatik wird durch die Vorrangregeln zwischen Operatoren geprägt:

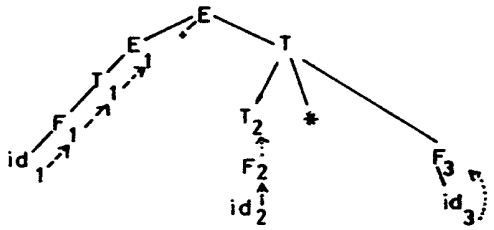
E → E + T ; E → T

T → T * F ; T → F

F → (E) ; F → id

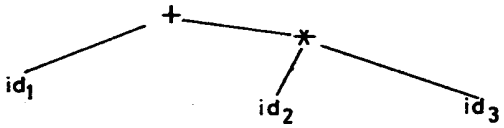
(Grammatik G = ({E,T,F},{id,+,*, (,)},{s.o.},E))

Der Ausdruck $id_1+id_2*id_3$ hat in dieser Grammatik den Ableitungsbaum



Zur semantischen Analyse dieses Ableitungsbaumes müssen die vom Scanner gelieferten aufsteigenden Attribute der Bezeichner id_i im Baum weitergereicht werden (hier durch gestrichelte Pfeile angedeutet).

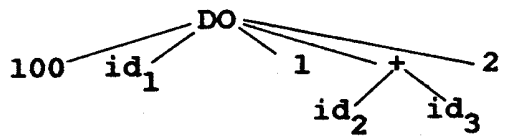
Der Baum ist in der folgenden semantisch äquivalenten Form darstellbar:



In (McKEEMAN 74) wird diese Form als der abstrakte Syntaxbaum bezeichnet. Im BCPL-Compiler (RICHARDS 79) wird der abstrakte Syntaxbaum (dort "applicative expression tree" genannt) vollständig aufgestellt und der semantischen Phase zur Analyse und Übersetzung übergeben.

Ein abstrakter Syntaxbaum (abgekürzt ASB) kann zur semantisch redundanzfreien Beschreibung vollständiger Programme dienen. Seine Blätter bestehen aus den vom Scanner generierten TOKEN,

seine Knoten aus den Operatoren der Sprache, zu denen außer den monadischen und dyadischen Operatoren in arithmetischen Ausdrücken auch die Kontrollstrukturen der Sprache zu zählen sind. Ein FORTRAN-DO-Statement läßt sich als ein Teilbaum eines ASB auffassen, dessen Unterbäume durch arithmetische Ausdrücke gebildet werden:



(ASB für DO 100 id_1=1, id_2 + id_3 , 2)

Der ASB soll für die semantische Analyse von FORTRAN 77-Programmen verwendet werden; seine Struktur wird in Kap. 3.4 und 3.5 genauer erläutert werden. Hier stellt sich zunächst die Frage, wie der ASB möglichst einfach und effizient aus dem durch den Parser implizit aufgestellten Ableitungsbaum eines Programmes zu generieren ist.

3.3.2 Die Zwischensprache ZSem

Für eine Bottom-Up-Analyse kann die in Kap. 3.3.1 angegebene Grammatik G relativ einfach zu einer attributierten Übersetzungsgrammatik erweitert werden, deren Aufgabe die Transformation des Syntaxbaumes (abgekürzt SB) in einen ASB ist. Dafür sollen die Datenstrukturen und Operationen des ASB wie folgt deklariert sein:

```
MODE  OPERATOR = .....;        /* n-adic Operator with
                                  n subtrees */

MODE  FRONTIER = .....;        /* leaf of ASB, referencing
                                  a TOKEN */

MODE TREE = .......;           /* OPERATOR u FRONTIER */
PROC LEAF =(TOKEN T)  TREE ....;
```

```
                                        /* Generate a leaf of ASB*/
PROC  NODE =(OPERATOR O, TREE T1...Tn) TREE.. ;
                                        /* Generate a Node with n
                                           subtrees */
```

Der Bottom-Up-Parser muß in einem zum Parser-Stack parallelen Semantik-Stack die Attribute der Sprachsymbole mitführen. Bei der Reduktion einer Regel kann eine semantische Routine die Auswertung der Attribute durchführen und die Übersetzung ausgeben:

$E\ a\uparrow \rightarrow E_{b\uparrow} + T_{c\uparrow}$	{ a:=NODE ("+",b,c)}
$E\ a\uparrow \rightarrow T_{b\uparrow}$	{ a:=b }
$T\ a\uparrow \rightarrow T_{b\uparrow} * F_{c\uparrow}$	{ a:= NODE ("*",b,c,)}
$T\ a\uparrow \rightarrow F_{b\uparrow}$	{ a:= b}
$F\ a\uparrow \rightarrow (\ E_{b\uparrow})$	{ a:=b}
$F\ a\uparrow \rightarrow id_{b\uparrow}$	{ a:= LEAF (b)}

(Übersetzungsgrammatik G für SB→ASB)

Für die Top-Down-Analyse muß G in eine bezüglich der Sprache äquivalente LL(1)-Grammatik G' umgeformt werden. Versieht man G' mit den entsprechenden Übersetzungsregeln zur Auswertung der Attribute, so wird die Übersetzungsgrammatik für G' wesentlich komplizierter als die ursprüngliche Grammatik für G. In G mußten nur aufsteigende Attribute behandelt werden, in G' aber auf- steigende (↑) und absteigende (↓) Attribute zusammen:

$E_{a\uparrow}$	→	$T_{b\uparrow}\ \ T'_{c\uparrow d\downarrow}$	{a:=c; d:=b}
$T'_{a\uparrow b\downarrow}$	→	ε	{a:=b}
$T'_{a\uparrow b\downarrow}$	→	$+\ \ T_{c\uparrow}\{A\}T'_{d\uparrow e\downarrow}$	A:{a:=d; e:=NODE("+",b,c)}
$T_{a\uparrow}$	→	$F_{b\uparrow}\ \ F'_{c\uparrow d\downarrow}$	{a:=c ; d:=b}
$F'_{a\uparrow b\downarrow}$	→	ε	{a:=b}
$F'_{a\uparrow b\downarrow}$	→	$*\ \ F_{c\uparrow}\{A\}T'_{d\uparrow e\downarrow}$	A:{a:=d; e:=NODE("+",b,c)}
$F_{a\uparrow}$	→	$(\ E_{b\uparrow})$	{a:= b}
$F_{a\uparrow}$	→	$id_{b\uparrow}$	{a:= LEAF (b)}

(Übersetzungsgrammatik G' für SB →ASB)

In einem "recursive descent parser" lassen sich gemischte Attribute (aufsteigende und absteigende) als Prozedurparameter verwirklichen; aufsteigende Attribute sind Ergebnis-Parameter, absteigende Wert-Parameter. Die Attributweitergabe wird dann durch den Compiler der Implementationssprache realisiert.

In einem "predictive parser" muß die Attributauswertung "per Hand" realisiert werden und ist für eine umfangreichere Grammatik als G' sehr kompliziert.

Für einen "predictive parser" gibt es aber noch eine andere Möglichkeit der Transformation Syntaxbaum → ASB, die wesentlich einfacher und effizienter als die obigen Übersetzungsgrammatiken G und G' ist.

Diese Variante benutzt zur Attribut-Auswertung einen zusätzlichen semantischen Keller, der nicht mehr parallel zum Parser-Stack ist. In diesem Stack soll der abstrakte Syntaxbaum in einer Bottom-Up-Strategie aufgebaut werden. Der semantische Stack enthält die Knoten und Blätter des ASB und ist über die Operationen push und pop zugreifbar:

```
PROC    PUSH=(TREE A)    .....;    /* put A onto Stack */
PROC    POP = () TREE ......;      /* return top-of-stack
                                      and remove one item*/
```

Die folgende Übersetzungsgrammatik G" arbeitet dann in ähnlicher Weise wie ein Verfahren zur Umformung von Infix- in Postfix-Notation. In die Regeln der Grammatik sind semantische Aktionen zur Generierung eines ASB eingefügt (bezeichnet mit {i}). Lediglich in der Regel $F \rightarrow id_{t\uparrow}$ wird die Attributauswertung eines Grammatiksymbols notwendig. Alle anderen Attribute ergeben sich aus dem Inhalt des semantischen Stacks:

```
E   →   T    T'
T'  →   ε    /   +   T  {1} T'
T   →   F    F'
F'  →   ε    /   *   F  {2} F'
```

```
F  →  ( E )   /   id_t↑ {3}
1 :{ TREE a,b,c;     c:= POP() ;  b:=POP();
       a:=NODE ("+",b,c); PUSH (a) }

2 :{ TREE a,b,c;     c:= POP() ;  b:=POP();
       a:= NODE ("*",b,c);  PUSH(a)}

3 :{ TREE a;         a:= LEAF(t)  ;  PUSH(a)}
```

(Übersetzungsgrammatik G" für SB → ASB)

G" ist aus einer Übersetzungsgrammatik zur Transformation von Infix- nach Postfix-Notation abgeleitet (vergleiche (GRIFFITHS 74) pp 76 ff)). Ersetzt man die semantischen Aktionen 1,2,3 durch die Aktionen

```
1'  : {     WRITE   ( '+')   }
2'  : {     WRITE   ('*')    }
3'  : {     WRITE   (id_t)   }
```

so ergibt sich für den Ausdruck $E \overset{*}{\rightarrow} id_1+id_2 * id_3$ folgende Linksableitungen:

$E \rightarrow TT' \rightarrow FF'T' \rightarrow id_1\{3'\}\ F'T' \rightarrow id_1\ \{3'\}\ T'$
$\rightarrow id_1\ \{3'\} + T\ \{1'\}\ T' \rightarrow id_1\ \{3'\} + FF'\ \{1'\}\ T'$
$\rightarrow id_1\ \{3'\} + id_2\ \{3'\}\ F'\ \{1'\}\ T'$
$\rightarrow id_1\ \{3'\} + id_2\ \{3'\} * F\ \{2'\}\ F'\ \{1\}\ T'$
$\overset{*}{\rightarrow} id_1\ \{3'\} + id_2\ \{3'\} * id_3\ \{3'\}\ \{2'\}\ \{1'\}$

Die Ausführung der Aktionen 1', 2' und 3' in der obigen Reihenfolge ergibt die Ausgabe

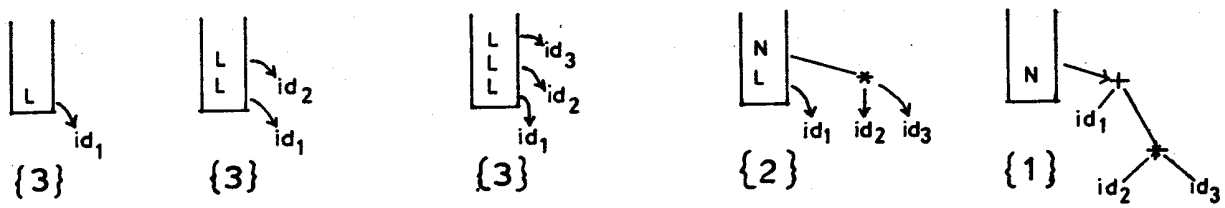
$id_1\ \ id_2\ \ id_3\ \ *\ +$,

welches die Postfix-Notation des Ausdruckes E ist.

Für denselben Ausdruck $E \overset{*}{\rightarrow} id_1 + id_2 * id_3$ generiert ein Parser für G" die Linksableitung

$E \overset{*}{\rightarrow} id_1\ \{3\} + id_2\ \{3\} * id_3\ \{3\}\ \{2\}\ \{1\}.$

Die Aktionen 1,2 und 3 ergeben in der obigen Reihenfolge die Kellerinhalte

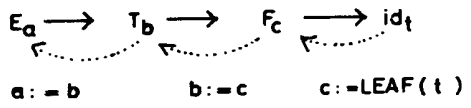


(Aufbau des ASB im semantischen Keller)

Nach der syntaktischen Analyse von E steht also der Knoten des ASB im semantischen Stack, und die Übersetzung SB → ASB ist vollendet.

Die Grammatik G" hat gegenüber G und G' den Vorteil, daß die Kettenableitungen keine Wirkung auf die semantischen Aktionen zur Attributauswertung haben. Das Attribut von E für den Ausdruck (E→T→F→id) "id_1" ergibt sich einfach aus dem Inhalt des semantischen Stacks nach der syntaktischen Analyse von E. In G oder G' hingegen müssen für solch einen einfachen Ausdruck eine ganze Anzahl von Attributen (mit stets gleichem Wert) ausgewertet werden:

z.B. G



Die Übersetzungsgrammatik G" ermöglicht es, eine syntaxgesteuerte Übersetzung durchzuführen. Syntaxanalyse und semantische Analyse lassen sich physikalisch trennen, sind jedoch logisch durch die entsprechenden Aktionen in einer syntaktischen Regel miteinander verbunden. Ein wichtiges Entwurfsziel des FORTRAN 77-Compilers wird durch diese Übersetzungsstrategie erfüllt. Die Grammatik muß zu diesem Zweck um eine dritte Art von Symbolen (zusätzlich zu Terminal und Nonterminal-Symbolen), die semantischen Aktionen, erweitert werden. Diese lassen sich in einfacher Weise in die vorhandenen Regeln der Grammatik einfügen:

```
MODE    SYMBOL = ....; /* TOKEN u Nonterminals u Actions*/
/* TERMS → PLUSOP      TERM {1}   TERMS     :   */
  PRODUCTION  : =[4, [PLUSOP,TERM,ACTION[1],TERMS,NONE..]];
  /* FACTORS→MULOP  FACTOR {2} FACTORS       :   */
   PRODUCTION := [4, [MULOP,FACTOR,ACTION [2], FACTORS,
                                NONE..]]
```

Die Grammatiksymbole ACTION[i] haben jeweils die Ableitung ACTION[i]→ε und die Nebenwirkung des Aufrufs der semantischen Aktion i bei der Ausführung dieser Ableitung. Einen Sonderfall

stellt die obige Aktion 3 der Grammatik G" dar. Hier muß neben dem Aufruf der semantischen Aktion 3 eine Attributauswertung des Terminalsymbols id_t erfolgen. Das Attribut t ist aber der Wert des dem Terminalsymbols id entsprechenden TOKENS T = (IDENTIFIER,Attribut, Position), das vom lexikalischen Analysator erzeugt wird. Der Parser muß deshalb außer der semantischen Aktion i in solchen Fällen auch noch das Attribut t↑ = T.Attribut an die semantische Routine weitergeben. Eine semantische Aktion soll deshalb als ein Paar

ACTION = (Number, Attribut)

aufgefaßt werden mit eventuell leerem Attribut, wenn dieses sich aus dem semantischen Stack ergibt.

Die Zwischensprache ZSem ergibt sich damit als eine Folge von Aktionen

ZSEM = (ACTION)* .

Die physikalische Realisierung dieser Aktionen wird in den Modul Sem verlagert. Unmittelbar nach der Ausführung einer Ableitung Aktion [i]→ε soll der Parser den Modul Sem aktivieren, welcher die entsprechende semantische Aktion ausführt.

3.4 Die semantische Analyse

Der semantische Analysator ist in seiner Aufgabenstellung dem Parser ähnlich. Er soll aus den Aktionen der Sprache ZSem einen abstrakten Syntaxbaum aufbauen, seine Zugehörigkeit zur Sprache FORTRAN 77 analysieren und bei erfolgreicher Analyse eine Übersetzung auf die Sprache FCODE ausgeben. Der kontextfreie Teil des ASB läßt sich durch eine vereinfachte Grammatik beschreiben:

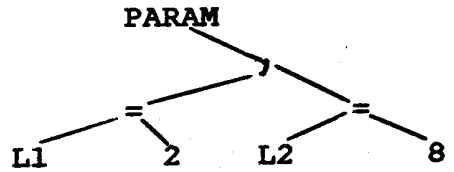
ASB	→	TREE
TREE	→	LEAF / NODE
NODE	→	{TREE} OPERATOR
OPERATOR	→	OP_ACTION
LEAF	→	LF_ACTION

OP-ACTION und LF-ACTION bezeichnen die Menge der Aktionen, die einen Knoten bzw. ein Blatt des ASB aufbauen. Die Knoten des ASB können bezüglich ihrer Wirkung auf die semantische Analyse in zwei Arten aufgeteilt werden, solche, die nur den Zustand des Moduls Sem verändern (Deklarationen, konstante Ausdrücke), und Knoten, die eine Übersetzung ergeben (nicht konstante Ausdrücke, ausführbare Anweisungen, Kontrollstrukturen). In diesem Kapitel soll zunächst nur die Analyse von ZSem betrachtet werden, in Kapitel 3.5 wird dann auf die Übersetzung ASB→FCODE eingegangen werden.

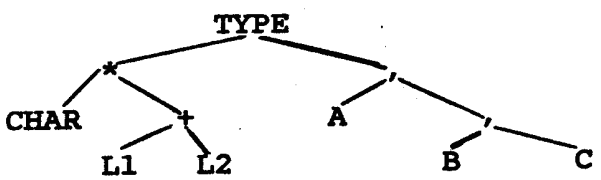
3.4.1 Attributsammlung für Bezeichner

Der FORTRAN 77-Scanner liefert für jedes Lexem eine eindeutige Identifikation in Form eines Tripels T= (Art, Attribut, Position). Der Parser gibt das Attribut des Tokens in einer Aktion an die semantische Analyse weiter. Der Wert des Attributs ist im Falle eines Bezeichners ein Zeiger auf eine Beschreibungsstruktur für den Bezeichner. Eine Aufgabe der semantischen Analyse ist die Festlegung dieser Beschreibungsstruktur und die Abprüfung ihrer Inhalte auf Konsistenz.
Zur Bearbeitung von Deklarationsanweisungen kann wieder der ASB verwendet werden. Jede Deklarationsanweisung wird in einen ASB übersetzt, der unmittelbar von der semantischen Analyse bearbeitet werden kann. Die aus Deklarationsanweisungen entstehenden Unterbäume des ASB haben nur eine zustandsverändernde Wirkung auf den Modul Sem und können nach ihrer Bearbeitung eliminiert werden.

z. B.: PARAMETER (L1 = 2, L2 = 8)



CHARACTER* (L1+L2) A,B,C



Eines der Hauptprobleme bei der semantischen Analyse von FORTRAN 77-Programmen stellt die Attributsammlung für Bezeichner dar. Im Gegensatz zu anderen Programmiersprachen ist eine Deklaration

```
INTEGER A
```

in FORTRAN 77 keine vollständige Spezifikation für den Bezeichner A. In PASCAL oder ALGOL führt die obige Deklaration zur Existenz einer Variablen A, deren Speicherallokation (relativ zum Laufzeitstack) unmittelbar festlegbar ist. In FORTRAN 77 hingegen handelt es sich hier um eine reine Typdeklaration für A. Die Speicherallokation von A kann in den folgenden Anweisungen der Programmeinheit (Programm oder Unterprogramm) implizit oder explizit weiter spezifiziert werden. Durch Zusatzanweisungen kann zunächst das Speichersegment (vergl. Kap. 3.5.2) für A festgelegt werden:

i	COMMON / ALPHA/ A	=>	Segment "ALPHA"
ii	PARAMETER (A=10)	=>	Konstantensegment der Programmeinheit
iii	EXTERNAL A	=>	Prozedursegment, A ist keine Variable sondern eine Funktion
iv	ENTRY E(A)	=>	Parametersegment, A ist eine Adresse
v	keine weitere Deklaration, Benutzung von A in einem ausführbaren Statement (z.B.:A=Ø)	=>	Standardsegment der Programmeinheit

In einer DIMENSION-Anweisung können der Speicherbedarf von A und die Zugriffsfunktion auf Elemente von A spezifiziert werden:

```
DIMENSION      A(1:10, 1:10)
```

Durch eine EQUIVALENCE-Anweisung kann A implizit in ein COMMON-Segment verlegt werden:

```
COMMON         /ALPHA/   B
EQUIVALENCE          (A (1,1) , B)
```

Eine Speicher-Äquivalenzdeklaration ist transitiv und muß durch einen aufwendigen Algorithmus bearbeitet werden:

```
INTEGER                 COLUMN     (1:10)
EQUIVALENCE     ( A(1,10) , COLUMN )
```

Die Behandlung von FORTRAN-Deklarationen ist an anderer Stelle (GRIES 71) ausführlich diskutiert worden, deshalb soll hier nicht weiter darauf eingegangen werden. Einen Sonderfall der Attributauswertung stellen die sogenannten "adjustable arrays" dar. FORTRAN kennt zwar keine dynamischen Felder, aber ein auf Parameterposition übergebenes Feld kann in einem Unterprogramm neu dimensioniert werden:

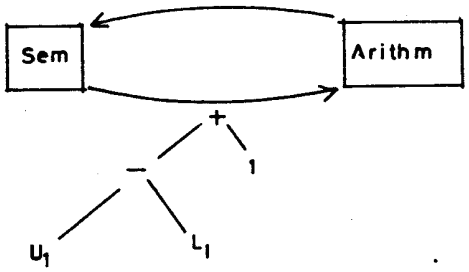
```
rufendes                    REAL   A (1:1000)
Unterprogramm :     ....
                            CALL S ( A,10)

gerufenes                   SUBROUTINE   S(B,N)
Unterprogramm :     REAL    B(1:N,1:N)
                            .....
```

Im Gegensatz zum konstant dimensionierten Feld A führt eine solche Deklaration für B zur Erzeugung von Code bei der Berechnung eines "Dope-Vektors" (GRIES 71). Zur einheitlichen Behandlung von konstanten und variablen Dope-Vektoren sollen dennoch alle Dope-Vektoren durch einen gemeinsamen Algorithmus berechnet werden.

Dies wird möglich durch den in FORTRAN 77 ohnehin vorhandenen Compilezeit-Interpreter. Die Elemente eines Dope-Vektors sind arithmetische Ausdrücke, die durch einen ASB-Unterbaum repräsentiert werden können. Sie setzen sich für $A(L_1:U_1,.....,L_n:U_1)$ aus den Ausdrücken $L_1..L_n,U_1..U_1$ zusammen, der diesen entweder interpretiert oder aber Code für ihn erzeugt. Als Ergebnis dieser Operation gibt der Modul Arithm dann entweder eine Konstante (ein Blatt eines gefalteten ASB, →3.4.3) oder einen Knoten eines ASB (eine temporäre Variable, →3.5) an den Modul Sem zurück:

z.B. $A(L_1 : U_1)$
Länge von A : $(U_1 - L_1) + 1$
i oder ii



i : L_1, U_1 sind konstant
Rückgabe : Wert des Ausdrucks

ii: L_1 oder U_1 nicht konstant
Rückgabe : temporäre Variable, die den ASB-Unterbaum repräsentiert (→3.5)

Die Elemente eines Dope-Vektors setzen sich dann einheitlich aus Objekten der Sprache FCODE (→3.5) zusammen und können in der semantischen Analyse weiter bearbeitet werden.

3.4.2 Operatoridentifikation und Intrinsic-Funktionen

Die arithmetischen Operatoren und Intrinsic-Funktionen von FORTRAN 77 sind nicht unmittelbar interpretierbar, sondern nur im Zusammenhang mit ihren Operanden bzw. Parametern. Eine Intrinsic-Funktion ist ein allgemeiner Name für eine Klasse von Funktionen, der je nach Typ des Parameters für eine spezifische Funktion steht (z. B.:SIN für real_SIN, complex_Sin etc.). Ähnlich können die arithmetischen Operatoren von FORTRAN 77 als intrinsische Operatoren angesehen werden (z. B.:+ für real_+,int_+etc.). Der Typ der Operanden bestimmt jeweils die spezifische Operation.

Die Operatoren des Ausdrucks

```
A + B * SIN(C)
```

haben je nach Deklaration von A,B,C eine andere Bedeutung:

i)	REAL	A,B,C,	real_+ , real_* , real_SIN
ii)	COMPLEX	A,B,C	complex_+,complex_*, complex_SIN

In FORTRAN 77 können auch gemischte arithmetische Datentypen in einem Ausdruck verwendet werden. Durch die Deklarationen

```
iii)    REAL A, B
        COMPLEX C
```

werden alle Operatoren des obigen Ausdrucks zu komplexen Operatoren (complex_SIN, complex_*, complex_+); A und B müssen jeweils zu komplexen Operanden konvertiert werden. Die Behandlung von Mixed-Mode-Ausdrücken soll nun am Beispiel der einfachen Übersetzungsgrammatik G" betrachtet werden. Durch G" wird ein dem syntaktisch analysierten Ausdruck E semantisch äquivalenter ASB in einer Bottom-Up-Strategie aufgebaut.

Die durch den FORTRAN77-Standard gegebenen Interpretationsregeln für arithmatisches Ausdrücke sind ebenfalls durch Bottom-Up-Regeln definiert. Der Typ einer Operation wird durch die Typen der Operanden vollständig festgelegt. Die vier arithmetischen Datentypen INTEGER, REAL, DOUBLE PRECISION und COMPLEX bilden in der angegebenen Reihenfolge eine lineare Ordnung. Sind die beiden Operanden einer Operation gemischten Typs, so wird stets der "kleinere" Typ zum "größeren" konvertiert, d. h. es soll eine größtmögliche Genauigkeit des Ergebnisses erzielt werden. Eine Mischung aus DOUBLE PRECISION und COMPLEX ist undefiniert. (Bei Einführung eines Typs DOUBLE COMPLEX ließe sich auch hier ein Ergebnistyp mit "größter" Genauigkeit definieren; die Typenreihe wäre dann abgeschlossen.)

In einem ersten Schritt zur Operatoridentifikation muß die Menge der Operatoren (Knoten) des ASB um Operatoren zur Typkonversion erweitert werden ("REAL", "DOUBLE", "COMPLEX").

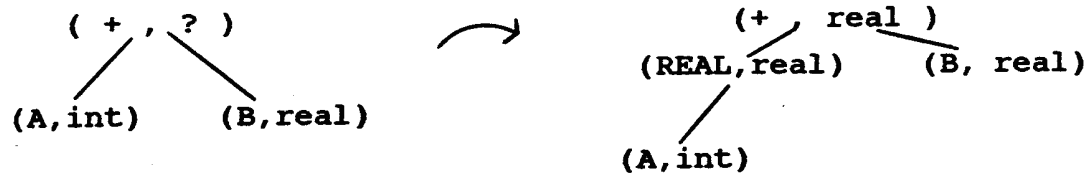
Diese Operatoren haben jeweils einen Operanden, der entsprechend konvertiert werden soll. Für die Operatoren "DOUBLE" und "COMPLEX" stellt sich sofort das Problem, daß die Operanden von unterschiedlichem Typ sein können:

DOUBLE(INT), DOUBLE(REAL), COMPLEX(INT), COMPLEX(REAL),

daß also wiederum die Operatoren nur zusammen mit ihrem Operanden interpretierbar sind. In FORTRAN 77 sind die Funktionen REAL, DBLE und CMPLX als Intrinsic-Funktionen vordefiniert.

Wollte man jeder Operator-Typ-Kombination einen spezifischen Namen zurordnen, so käme man schon für die einfache Sprache von G" auf 13 Operatoren. Die sieben arithmetischen Operatoren von FORTRAN 77 (+,-,*,/,**, "unary-",=) und Konversionsoperatoren (INT,REAL,DBLE,CMPLX) ergäben zusammen 40 spezifische Operatoren allein für arithmetische Ausdrücke.

Die Behandlung der Operatoren von FORTRAN 77 als intrinsische Operatoren ist mnemotechnisch günstiger. Jeder Operator soll durch seinen Namen und den Typ seines Ergebnisses repräsentiert werden ((+,INT),(+,real)etc.). Beim Aufbau des ASB muß dann dafür gesorgt werden, daß die Operanden eines dyadischen Operators einen übereinstimmenden Typ haben, der dem Ergebnistyp des Operators entspricht. Zu diesem Zweck werden Konversionsoperatoren in den ASB eingefügt.



Da der ASB in einer Bottom-Up-Strategie aufgebaut wird, entspricht dieses Vorgehen den Interpretationsregeln für Mixed-Mode-Ausdrücke in FORTRAN 77.
Die Anzahl der Operatoren und Intrinsic-Funktionen in FORTRAN 77 ist sehr groß. Für G" sollen hier nur die fünf Operatoren (+,int), (+, real),(*,int),(*,real) und (REAL,real) definiert sein. Ein Blatt des ASB soll das zusätzliche Attribut Typ erhalten. Die semantischen Aktionen 1,2,3 müssen dann zur Behandlung von Mixed-Mode-Ausdrücken folgendermaßen geändert werden.:

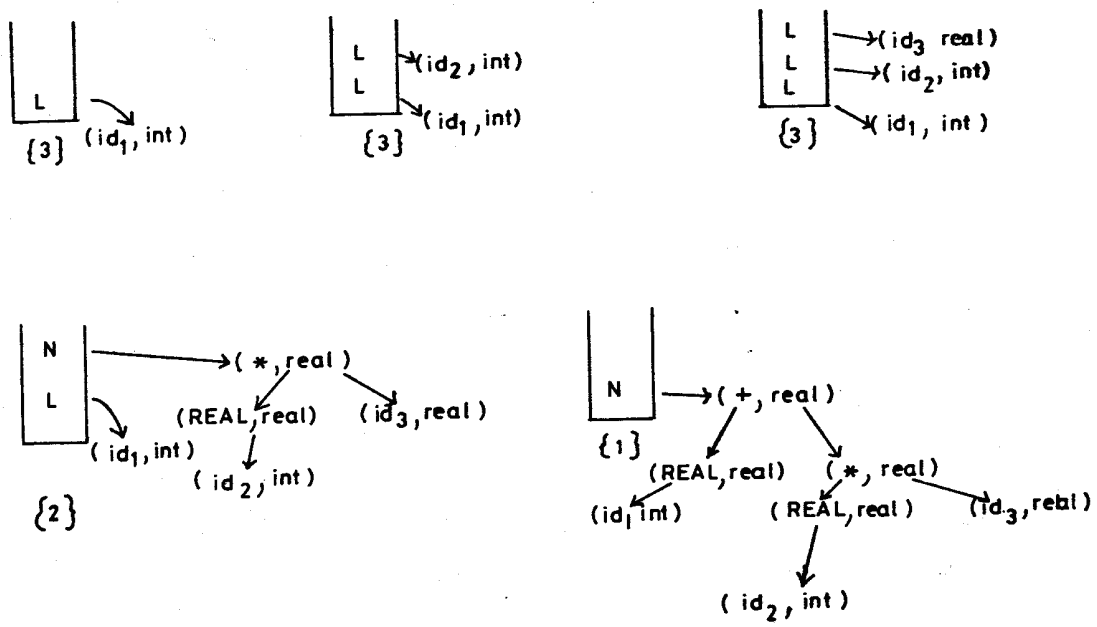
```
MODE      TYPE=  ..;      /* int or real   */
MODE      TREE1=STRUCT    (TREE tr,
                           TYPE ty);

1  : {   TREE1  a,b,c;       TYPE   g ;
         c:= POP();  b:=POP() ;
         IF  b.ty /= c.ty   THEN
           g:=real ;
           IF  b.ty  /= g  THEN
              b:=  NODE  ([ "REAL",real] ,b,NIL)
           ELSE
              c:=  NODE  ([ "REAL", real] ,c,NIL)
           FI
         ELSE
           g:= b.ty
         FI;
         a := NODE (["+",g],b, c);
         PUSH(a)  }
2 :{       ../*dito   , but replace "+" by "*"  */ }
3 :{       TREE1 a ;
           a:= LEAF  (t, typattribut (t));
           PUSH (a) }
```

Sind in dem Ausdruck $id_1 + id_2 * id_3$ id_1 und id_2 vom Typ int und id_3 vom Typ real, so ergibt sich aus der Linksableitung

$$E \overset{*}{\rightarrow} id_1\{3\} + id_2\{3\} * id_3\{3\}\ \{2\}\ \{1\}$$

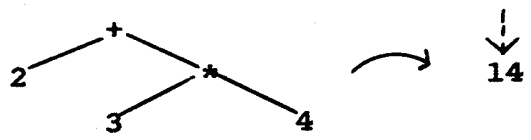
folgender ASB:



(Aufbau eines ASB für Mixed-Mode Arithmetik)

3.4.3 Compilezeit-Rechnung am ASB

In FORTRAN 77 sind Deklarationsanweisungen nur mit Hilfe einer Compilezeit-Rechnung zu bearbeiten (→1.1 ,→3.4.1). Die FORTRAN 77-Syntax macht eine Unterscheidung zwischen konstanten und nichtkonstanten Ausdrücken. Ein solches Vorgehen ist in einem Compiler aus zwei Gründen unpraktisch. Konstante Ausdrücke von FORTRAN 77 wie 'A + B' lassen sich nur mit einer kontextsensitiven Syntax beschreiben, die Attribute der symbolischen Konstanten A und B müßten in der Syntaxanalyse bekannt sein. Ein zweiter Grund ist die unnötigerweise zusätzliche Anzahl von Regeln für konstante Ausdrücke, deren kontextfreier Teil schon in der Syntax für nichtkonstante Ausdrücke enthalten ist.
Im FORTRAN 77-Parser werden deshalb konstante und variable Ausdrücke gemeinsam behandelt, und im Modul Sem soll ein gemeinsamer ASB für sie aufgebaut werden. Der Modul Sem gibt diesen ASB für einen Ausdruck direkt an den Modul Arithm weiter, dessen Aufgabe die weitere Bearbeitung des ASB's ist (vergleiche Kap. 3.4.1, Dope-Vektor). Die gleichartige Behandlung von konstanten und nichtkonstanten Ausdrücken führt dann zu einer ersten Optimierung im FORTRAN 77-Compiler, der sogenannten "Faltung" (AHO 77). Ein ASB für den Ausdruck '2+3*4' wird unabhängig von seinem Auftreten in einer Deklarationsanweisung oder in einem ausführbaren Statement dann stets gefaltet:



Die Aufgaben der Compilezeit-Rechnung soll eine Prozedur "FOLD" übernehmen, die für die Compilezeit-Operatoren +,-,*,**,"REAL","DOUBLE" etc. und deren konstante Operanden definiert ist:

```
PROC      FOLD =  (NODE  N)   FRONTIER  .... ;
          /* Compute  monadic or dyadic Operation
             and return a leaf of ASB */
```

Die semantischen Aktionen 1,2,3 von G" sind zu diesem Zweck recht einfach erweiterbar:

z.B.
```
1:{  TREE   a,b,c
     c := POP() ;  b:= POP();
     a := NODE ("+",b,c);
     IF  constant (b) AND  constant (c) THEN
         a := FOLD (a)
     FI;
     PUSH (a) }
```

In FORTRAN 77 ist die Faltungsfunktion sehr umfangreich. Sie muß außer den monadischen und dyadischen Operatoren in arithmetischen, logischen und CHARACTER-Ausdrücken auch noch die Operatoren zur Typ-Konvertierung in Mixed-Mode-Ausdrücken bearbeiten können. Der FORTRAN 77-Standard verbietet eine Benutzung von Intrinsic-Funktionen in konstanten Ausdrücken. Eine Erweiterung von FORTRAN 77 um die Compilezeit-Operatoren SIN, COS, ABS etc ist aber durchaus sinnvoll. Konstanten wie PI oder E lassen sich dann in mnemotechnisch günstiger Form in einem FORTRAN 77-Programm verwenden:

```
PARAMETER    ( PI = 4 * ASIN (1.Ø ),
               E  = ExP(1.Ø)
```

an Stelle von

```
PARAMETER    ( PI = 3.1415926535,
               E  = 2.718281828 ).
```

3.5 Die virtuelle Zwischenmaschine FCODE

Der abstrakte Syntaxbaum stellt ein gutes Entwurfskonzept dar. In einem Einpaß-Compiler ist es aber nicht notwendig, den ASB vollständig aufzustellen. Die Knoten des ASB haben entweder nur eine semantische Funktion (→3.4) oder können direkt auf eine geeignete Sprache übersetzt werden. Diese Sprache, FCODE, soll zusammen mit ihren Datenstrukturen in diesem Abschnitt betrachtet werden. Der Modul Sem übergibt mittels der Zwischensprache ZArithm, die nur eine interne Bedeutung hat, den ASB als eine Menge von Unterbäumen an den Modul Arithm. Dieser soll den ASB interpretieren oder ihn auf die Sprache FCODE übersetzen.

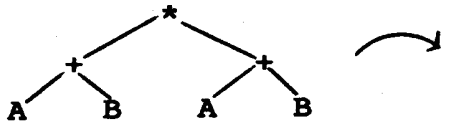
Die Sprache FCODE stellt zusammen mit ihren Datenstrukturen eine virtuelle Maschine dar, die ebenfalls FCODE genannt werden soll.

3.5.1. Beziehungen zwischen dem ASB und FCODE

Da der ASB in einer Bottom-Up-Strategie aufgebaut wird, kann er direkt auf eine in Frage kommende Zielmaschine übersetzt werden, ohne daß er vollständig aufgestellt werden muß. Im FORTRAN 77-Compiler soll aber eine wohldefinierte Schnittstelle existieren, die die in Kapitel 2 gestellten Portabilitäts- und Modularitätsanforderungen gewährleistet. Diese Schnittstelle soll eine Linearisierung des ASB sein, die Zwischensprache FCODE.

Aus einer linearen Repräsentation des ASB läßt sich in einem potentiell existierenden Optimierer der ASB, der zu einer Optimierung gut geeignet ist, wieder aufbauen. In (AHO 77) werden Optimierungsalgorithmen auf 3-Adreßmaschinen betrachtet. Die dort verwendete 3-Adreßmaschine ist aber eine Linearisierung eines einfachen ASB; die verwendeten Optimierungsalgorithmen sind Algorithmen auf Bäumen und Graphen.

Der ASB läßt sich relativ einfach auf seine lineare Form übersetzen. Zu diesem Zweck muß FCODE alle notwendigen Operatoren (Knoten) des ASB enthalten. Die Form einer Kellersprache für FCODE,



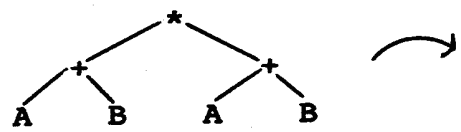
A B + A B + *

ist zu globalen Optimierungszwecken zwar geeignet (WAITE 74b), ein effizienter Codegenerator läßt sich jedoch für eine Kellersprache nur sehr umständlich realisieren. Im Codegenerator bzw. maschinenbezogenen Optimierer müssen häufig vorkommende Operationen wie A := A + 1 leicht erkannt werden können, um für sie eventuell vorhandene atomare Operationen A +:= 1 der Zielmaschine erzeugen zu können.

FCODE soll deshalb eine Mehradreß-Maschine sein. Zu jeder Knotenart des ASB soll in FCODE ein entsprechender Operator existieren. Verschiedene Inkarnationen einer Knotenart sollen in FCODE durch eine eindeutige Identifikation, NODE_ID, dargestellt werden. Zusammen mit den Operanden eines ASB-Knotens ergibt sich ein Befehl der Sprache FCODE dann als ein n-tupel

FCODE_OP = (NODE_ID,OPERATOR,OPERAND1 .. OPERANDn).

Die Operanden eines FCODE_OP's können entweder Konstanten oder Variablen der Sprache FORTRAN 77 sein, oder sie sind Knoten des ASB. Diesen Knoten des ASB entsprechen aber wiederum FCODE_OP's, die durch ihre Identifikatoren dargestellt werden können:



(1,"+",A,B)
(2,"+",A,B)
(3,"*",1,2)

Optimierung

(1,"+",A,B)
(3,"*",1,1)

Zur Übersetzung des ASB auf FCODE-Befehle soll eine Prozedur

```
PROC     NEW_FCODE = (NODE N) NODE_ID  ...;
         /* generate an FCODE_OP
            with a new identification  */
```

zuständig sein. In der Grammatik G"" lassen sich dann die Aktionen 1,2,3 zur Übersetzung auf FCODE folgendermaßen modifizieren:

```
        MODE TREE = ....; /* leaves and NODE_ID's */
1:{   TREE a,b,c;
      c := POP();  b := POP();
      a := NODE("+",b,c);
      IF constant(b) AND constant(c) THEN
        a := FOLD(a)
      ELSE
        a := NEW_FCODE(a)
      FI;
      PUSH(a)}
2:{     ... /* dito, replace "+" by "*" */ }
3:{     ... /* unchanged */ }
```

Durch die Bottom-Up-Strategie beim Aufbau des ASB existiert dann für jeden im semantischen Stack befindlichen NODE_ID schon eine Übersetzung des zugehörigen Unterbaums auf die Sprache FCODE. Der ASB muß also nicht vollständig aufgebaut werden, sondern ist parallel zu seiner Erzeugung direkt auf FCODE übersetzbar.

3.5.2 Objekte und Befehlsvorrat von FCODE

Die virtuelle Zwischenmaschine FCODE soll die Portabilität und Adaptierbarkeit des FORTRAN 77-Compilers an andere Zielmaschinen gewährleisten. Ihres Befehls- und Speicherstruktur ist deshalb der Quellsprache FORTRAN 77 angepaßt und nicht der Zielmaschine.

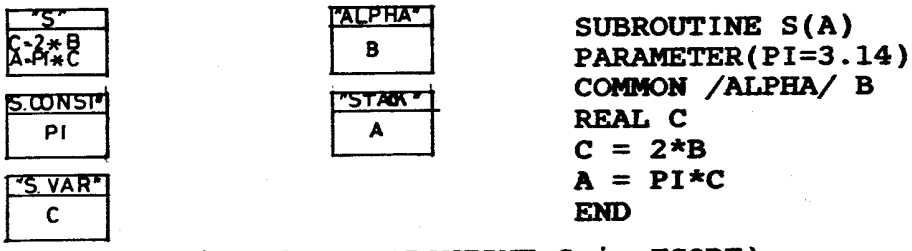
Da eine Speicherallokation für Variablen von FORTRAN 77 schon in der semantischen Analyse durchgeführt werden muß (→3.4.1), soll FCODE eine ideelle Speicherstruktur erhalten. Variablen und Konstanten von FORTRAN 77 sollen in Segmenten von FCODE angelegt werden, die der Speicherstruktur von FORTRAN 77 entsprechen. In Compileroberteil müssen dann nur relative Adressen bezüglich eines Segmentanfangs vergeben werden. Das Problem der

endgültigen Speicherallokation für COMMON-Blöcke wird dadurch auf den Codegenerator bzw. den Binder des Programmiersystems verschoben.

FCODE verfügt zu diesem Zweck über beliebig viele vom Benutzer oder vom Compiler definierte Segmente, die über einen ideellen Adressierungsmechanismus

Adress(A) = (Segment(A),Offset(A))

zugreifbar sind. Im Codegenerator muß dieser Adressierungsmechanismus dann auf eine reale Adressierung der Zielmaschine abgebildet werden. FCODE stellt damit keine spezifischen Anforderungen an die Zielmaschine und ist somit gut portabel.



(Speicherbild der SUBROUTINE S in FCODE)

Objekte von FCODE sollen außer skalaren Konstanten und Variablen von FORTRAN 77 auch temporäre Objekte, die ein Operationsergebnis darstellen, sein. FORTRAN 77 verfügt implizit über Adreßvariablen (Referenz-Parameter,Feldreferenzen), die in FCODE explizit verfügbar sein müssen. Die Klasse der temporären Objekte umfaßt deshalb Konstanten, Variablen und typgebundene Pointer. Ein temporäres Objekt von FCODE läßt sich in einer ALGOL68-Notation als

```
      MODE AMODE = ... ;
   i) TEMP_CONST = AMODE
  ii) TEMP_VAR = REF AMODE
 iii) TEMP_REF = REF REF AMODE
```

ansehen. Abkürzend wird ein temporäres Objekt von FCODE als ein Paar

```
TEMPORARY = (REF_LEVEL,AMODE)
REF_LEVEL = Ø .. 2
AMODE     = INTEGER,REAL,LOGICAL, ....
```

beschrieben. Die Nützlichkeit dieses Konzeptes wird aus der vereinfachten Übersetzung des FORTRAN 77 Statements

```
A(I) = B(J) + C
```

auf FCODE deutlich:

```
T1 = (1,"INDEX",A,I)      /* T1 is (2,REAL)  */
T2 = (2,"INDEX",B,J)      /* T2 is (2,REAL)  */
T3 = (3,"+"    ,T2,C)     /* T3 is (1,REAL)  */
     (4,"="    ,T1,T3)
```

Die aus 1 und 2 entstehenden Operationsergebnisse (T1 und T2) sind temporäre Referenzvariablen, auf die im folgenden indirekt zugegriffen werden muß. T3 hingegen ist eine einfache Variable, die direkt verfügbar ist. Zu Optimierungszwecken (lineare Adreßfortschaltung in DO-Loops) ist die FCODE-Operation "INDEX" auch für Referenz-Objekte definiert:

```
   z.B.:  T1 = (1,"INDEX",A,Ø)      /* Zero-Offset */
LOOP:     T2 = (2,"INDEX",T1,4)     /* linear Increment 4
                                       Add 4 to T1 */
          ...
               (n,"GOTO",LOOP)
```

Die Speicherallokation für temporäre Objekte sollte nicht im Compileroberteil durchgeführt werden. Sie ist maschinenabhängig und hängt von der Anzahl der zur Verfügung stehenden Register der Zielmaschine ab. Die temporären Objekte von FCODE werden deshalb in einem speziellen Segment der virtuellen Maschine FCODE angelegt, ohne eine relative Adreßvergabe durchzuführen. Der Codegenerator ist dadurch in der Lage, die Operationsergebnisse entweder in Registern oder im Speicher der Zielmaschine allozieren zu können.

Der Befehlsvorrat von FCODE (Anhang II) enthält außer den monadischen und dyadischen Operatoren von FORTRAN 77 einige zusätzliche Pseudo-Operatoren. Eine zu übersetzende FORTRAN 77-Quelleinheit, die aus mehreren Unterprogrammen bestehen kann, wird in FCODE durch Modul-Klammern umgeben:

FCODE_MODULE → MODULE_OP UNITS MODEND_OP

Der MODULE_OP dient zur Initialisierung des Codegenerators, der MODEND_OP veranlaßt ihn, globale Adreßbezüge (externe Prozeduren,COMMON-Blöcke) an den Binder weiterzugeben.

Eine Programmeinheit (Hauptprogramm,Unterprogramm) wird in FCODE von Prozedurklammern umgeben:

```
UNITS → EMPTY  /  UNIT UNITS
UNIT →  PROC_OP   FCODE_OPS   PROCEND_OP
```

Der PROC_OP dient zur Einrichtung einer neuen Programmeinheit mit ihren lokalen Segmenten (Konstanten-,Variablen-Segment); der PROCEND_OP veranlaßt den Codegenerator dann zur Ausgabe des erzeugten Codes.

FCODE enthält einen Pseudo-Befehl zur Definition von Marken. Marken können dabei sowohl benutzerdefinierte Marken sein als auch vom Compileroberteil generierte Marken, die bei der Übersetzung von Kontrollstrukturen entstehen (IF-THEN-ELSE, DO). Compilergenerierte Marken sind temporäre Objekte der Referenzstufe Ø, die nur in entsprechenden Sprungbefehlen benutzt werden dürfen. Der Pseudo-Operator LABEL_OP hat den Sinn, den Codegenerator zum "backpatching"(AHO 77) von vorwärtsdeklarierten Marken zu veranlassen.

Ein weiterer Pseudo-Operator von FCODE ist der "LINE_OP", durch den dem Codegenerator die der folgenden FCODE-Befehlssequenz entsprechende Zeilennummer im Quellprogramm mitgeteilt wird. Diese Zeilennummer kann im Codegenerator zur Realisierung eines "Rückverfolgers" ausgenutzt werden. Der Codegenerator muß zu diesem Zweck Befehle erzeugen, die zur Laufzeit des FORTRAN 77-Programmes die quellbezogene Zeilennummer verfügbar machen. Werden die Rücksprungadressen und Parameter in einem Stack angelegt, so läßt sich im Falle eines Laufzeit-Alarms die Prozeduraufruffolge bis zu diesem Alarm zurückverfolgen:

```
"ARITHMETIC ALARM IN SUBR1 , LINE 37"
"CALLED           BY SUBR2 , LINE 20"
"CALLED           BY MAIN  , LINE 12"
```

3.6 Codeerzeugen aus FCODE

Die Aufgabe des Codegenerators ist die Abbildung der virtuellen Maschine FCODE auf die reale Maschine AEG80-60. Er muß zu diesem Zweck die Speicher- und Befehlsstruktur von FCODE durch die reale Struktur der Zielmaschine ersetzen.

3.6.1 Befehlsstrukturen

Eine einfache Realisierungsform für die Operatoren von FCODE besteht in der Erzeugung von Makros. Eine direkte Ersetzung der FCODE-Befehle ist jedoch wegen der intrinsischen Bedeutung der Operatoren (→3.4) nur in einem sehr komfortablen Makrogenerator möglich.

Eine makroartige Ersetzung der FCODE-Befehle ist für eine reale Maschine mit mehreren Registern ein sehr ineffizientes Verfahren. Ein intelligenter Codegenerator, der den vorhandenen Registersatz voll ausnutzt, benötigt eine Registerverwaltung.
An Stelle der einfachen Übersetzungsstrategie

```
          T1 = A (+,INT) B
     =>      GEN("LOAD",RØ,A)
             GEN("ADD" ,RØ,B )
             GEN("STORE",RØ,T1)
```

kann mit Hilfe von geeigneten Prozeduren

```
          PROC ACCU = (OBJECT A) BOOL ... ;
               /*  <=> A is in an accu  */
          PROC GETINACCU = (OBJECT A) ACCUNR ... ;
               /* IF NOT ACCU(A) THEN
                     generate code for loading A;
                  return accu that contains A      */
          PROC UPDATE = (OBJECT A, ACCUNR AC) ... ;
               /* A is now in accu AC  */
```

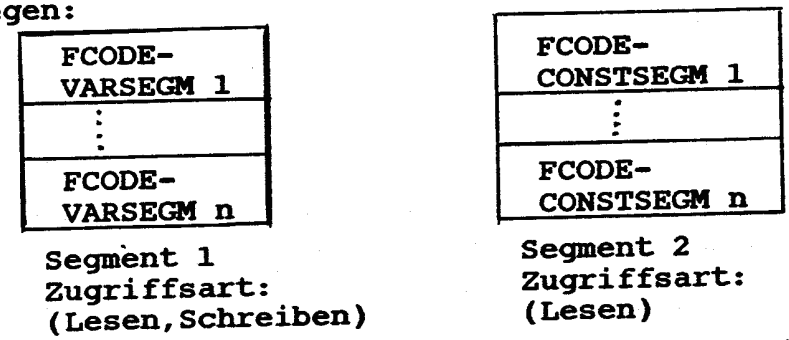
eine bessere Übersetzung für FCODE-Befehle erzeugt werden:

```
          T1 = A (+,INT) B
     =>   IF ACCU(B) THEN
            EXCHANGE(A,B)
          FI;  /* (+,INT) is commutative */
          AC1 := GETINACCU(A);
          IF ACCU(B) THEN
            AC2 := GETINACCU(B);
            GEN("ADDREG",AC1,AC2)
          ELSE
            GEN("ADD",AC1,B)
          FI;
          UPDATE(T1,AC1);
```

Die durch die Prozeduren ACCU, GETINACCU und UPDATE realisierte Registerverwaltung ähnelt in ihren Aufgaben einer Kachelverwaltung in Betriebssystemen. Die Registerverwaltung muß jeweils dafür sorgen, daß benötigte Operanden in Registern verfügbar sind. Zur Zeit nicht benötigte Operanden in Registern können in den Speicher verdrängt werden bzw. eliminiert werden, wenn es sich um ein einmalig benutzbares Zwischenergebnis handelt.

3.6.2 Speicherstrukturen

Die Segmentstruktur der FCODE-Maschine läßt sich sehr gut auf die Segmentstruktur der AEG80-60 abbilden. FCODE-Segmente gleicher Art (z.B.: Code-, Variablen- oder Konstanten-Segmente) lassen sich dichtgepackt in Segmenten der realen Maschine anlegen:



Der Adressierungsmechanismus der FCODE-Maschine wird mit Hilfe einer zusätzlichen Basisregisterverwaltung für die reale Maschine realisiert. Die Prozedur

```
PROC SEGMENT = (OBJECT A) BASEREG ... ;
     /* IF FCODE-Segment is not in baseregister THEN
          generate code for loading base ;
        return baseregister                          */
/*   use of SEGMENT : */
     GEN("LOAD",SEGMENT(A),ADRESS(A));
```

stellt mittels der vier offenen Basisregister der AEG80-60 dem Codegenerator die jeweils benötigte Adreßbasis der FCODE-Maschine zur Verfügung. Da FCODE über beliebig viele Segmente verfügt, muß bei Bedarf ein Basisregister mit einer realen Segmentsanfangsadresse geladen werden. Für einfache Programme ist aber ein einmaliges Laden der Basisregister mit den Standard-Segment-Adressen des Code-, Daten- und Parametersegments ausreichend.

4. Resultate

Anhand eines numerischen FORTRAN IV-Programmes konnte ein Vergleich zwischen dem vorhandenen FORTRAN IV-Compiler und dem in dieser Arbeit entwickelten FORTRAN 77-Compiler vorgenommen werden.

Eine Gegenüberstellung der gemessenen Werte ergab:

	FTNIV / FTN77
Länge des Quellprogramms:	70 Sätze
Compilationszeit des Oberteils:	4 sec / 4 sec
Codegenerierungszeit (Assemblieren):	8 sec / 8 sec
Länge des Objektcodes:	17 Sätze / 19 Sätze

Bei dieser Gegenüberstellung ist zu beachten, daß der erstellte FORTRAN 77-Compiler noch nicht optimiert wurde. Weiterhin ist die CPU-Zeit für das Assemblieren voll abziehbar, wenn der Modul Ass durch den Modul Bin ersetzt wird. Im Modul Ass wird der Maschinencode in einem Segment angelegt und dann in eine für den Assembler lesbare Form zurückverwandelt. Die symbolische Adressierung des Assemblers wird nicht benutzt. Lediglich zur Absättigung von globalen Adreßbezügen werden Namen im erzeugten Assemblerprogramm verwendet.

Aus diesen Überlegungen ergibt sich ein Verhältnis der Übersetzungsgeschwindigkeiten der beiden Compiler von mindestens 3 : 1 (FTN77-Compiler mit Bindemodulschnittstelle: FTNIV-Compiler).

Die Bindemodulschnittstelle wurde nicht realisiert. Weiterhin sind die Semantik der Formelfunktionen, die Erkennung von Intrinsicfunktionen, die parallelen EA-Prozesse und das "assigned go to" nicht realisiert. In der Laufzeitunterstützung wurde die formatierte Eingabe weggelassen, da sie durch eine komfortablere, listenorientierte Eingabe zu ersetzen ist. Somit wurde die Implementation bis zum Erreichen eines Sprachumfanges von ca. 90 % realisiert.

SYNTACTICAL SPECIFICATION OF FORTRAN 77 INTERMEDIATE LANGUAGE ZSYN

THE INTERMEDIATE LANGUAGE ZSYN IS THE LANGUAGE ACCEPTED BY THE FORTRAN77 PARSER.

TERMINALS

LABELTYP	INTTYP	DINTTYP	REALTYP
DOUBLETYP	LOGICTYP	CHARTYP	ID
ASSIGNSYM	BLOCKIFSYM	BLDATASYM	BACKSPSYM
CALLSYM	CASESYM	CLOSESYM	COMMONSYM
COMPLSYM	CONTINSYM	CHARACTSYM	DATASYM
DOSYM	DIMENSSYM	DEFAULTSYM	DOUBLESYM
ENTRYSYM	EXTERNALSYM	ENDFILESYM	ELSEIFSYM
ELSESYM	ENDSYM	ENDIFSYM	ENDCASESYM
EQUIVSYM	FORMATSYM	FUNCTIONSYM	GOTOSYM
INTEGSYM	INTRINSYM	INQUIRESYM	IFSYM
IMPLICITSYM	LOGICSYM	OPENSYM	PAUSESYM
PRINTSYM	PROGSYM	PARAMSYM	REWINDSYM
READSYM	RETURNSYM	REALSYM	STOPSYM
SUBRSYM	SWITCHSYM	SAVESYM	WRITESYM
TOSYM	WHILESYM	UNTILSYM	THENSYM
UNITSYM	FMTSYM	RECSYM	
ENDIOSYM			
ERRIOSYM	IOSTATSYM	FILESYM	STATUSSYM
ACCESSYM	FORMSYM	RECLSYM	BLANKSYM
EXISTSYM	OPENEDSYM	NUMBERSYM	NAMDSYM
NAMESYM	SEQUSYM	DIRECTSYM	FMTEDSYM
UNFMTEDSYM			
NEXTRECSYM			
TRUESYM	FALSESYM		
NOTSYM	ORSYM		
ANDSYM	EQVSYM	NEQVSYM	LTSYM
LESYM	EQSYM	NESYM	GESYM
GTSYM			
EOSSYM	EOFSYM	LPARENTH	RPARENTH
COMMA	COLON	PERIOD	EQUALSIGN
PLUSS	MINUSS	MULT	DIVD
POWER	CONCAT	LETTERSYM	

NONTERMINALS

PROGRAM	PROGUNITS	UNITKIND	BLOCK
CONSTP	DECLP	FUNCDATP	STMTS
FUNCTYPE	TYPE	LEN	LENEXPR
LABELP	STMT	SIMPLE	COMPOUND
RETURNMODE	GOTOMODE	IFMODE	ELSEIFST
ELSEST	CASEST	DEFAULTST	DORANGE
INCR	SUBFPARLST	FPARLST	FPARS
FPAR	SUBAPARLST	APARLST	APARS
APAR	FORMSPEC	FMTLST	KM
FMT	ONEFMT	RPT	SECLETTER
WIDTH	CONSTLST	IMPLLST	LETLST
UPLET	ARRAYLST	ADIM	BOUNDS
BOUND	UPBOUND	UPBD	DIM
DATLST	VARELMTLST	VARELMT	CONSTELMTLST
CONSTELMT	REPLIK	DATCONST	DATDOLST
DATDOELMTS	DATDOS	EQVLST	EQUIV
CBLST	CBID	COMVARLST	DECLLST
IDLST	EXPRLST	SAVELST	SAVEELMT
LBLST	IOSTATLST	AUXIOLST	IOLST
IOELMT	IOELMTS	IODOLST	CINFO
CINFOS	IOPAR	IOMOD	COND
DISJS	DISJ	LTERMS	LTERM
LFACTS	LFACT	RELEXPR	RELTERMS
RELTERM	EXPRNS	EXPR	TERMS
TERM	FACTS	FACT	PRIMS
PRIM	FUNCREF	VAR	SUBSTRING
CHARPOS	INDLST	EXPRS	CONSTANT
TAGPART	BLOCKID		

STARTSYMBOL

PROGRAM

PRODUCTIONS

SEE NEXT PAGE

THE INTERMITTED SYMBOLS I (1 2 3 ...) ARE THE SEMANTIC ACTIONS THE PARSER OUTPUTS WHEN GENERATING A LEFT DERIVATION . THEIR PURPOSE IS TO PRODUCE AN "ABSTRACT SYNTAX TREE" WHICH IS USED FOR SEMANTICAL OPERATIONS. THEIR SYNTACTIC MEANING IS

I -> EMPTY .

S T R U C T U R E O F P R O G R A M S

```
<PROGRAM>       ->   1 <PROGUNITS>   EOF 108
<PROGUNITS>     ->    EMPTY
                ->   <UNITKIND> <BLOCK> 2 <PROGUNITS>
<UNITKIND>      ->   PROGRAM ID 3 <SUBFPARLST> EOS | E      /* C1 */
                ->   SUBROUTINE ID 4 <SUBFPARLST> EOS
                ->   BLOCKDATA  <BLOCKID> 140  EOS
                ->   <FUNCTYPE> FUNCTION ID 5 ( 39 <FPARLST> ) EOS
              ----------------------------------------
<BLOCK>         ->   <CONSTP> <DECLP> 86 <FUNCDATP> <STMTS> <LABELP>
                      END EOS
              ----------------------------------------
<CONSTP>        ->    EMPTY
                ->    LABEL 46  FORMAT 47 ( <FORMSPEC> ) EOS <CONSTP>
                ->    ENTRY ID 6 <SUBFPARLST> EOS <CONSTP>
                ->    PARAMETER ( ID 48 = <COND> 49 <CONSTLST> ) EOS
                      <CONSTP>
                ->    IMPLICIT <TYPE> (  LETTER 50 <UPLET> 52
                      <LETLST> ) <IMPLLST> EOS <CONSTP>
              ----------------------------------------
<DECLP>         ->    EMPTY
                ->    LABEL 46   FORMAT 47 ( <FORMSPEC> ) EOS <DECLP>
                ->    ENTRY ID 6 <SUBFPARLST> EOS <DECLP>
                ->    PARAMETER ( ID 48 = <COND> 49 <CONSTLST> ) EOS
                      <DECLP>
                ->    EXTERNAL ID 7  <IDLST> EOS <DECLP>
                ->    INTRINSIC  ID 7 <IDLST> EOS <DECLP>
                ->    DIMENSION ID 53 <ADIM> <ARRAYLST> EOS <DECLP>
                ->    EQUIVALENCE ( ID 60 <INDLST> , ID 61
                      <INDLST> <EQUIV> ) <EQVLST> EOS <DECLP>
                ->    COMMON <CBID> ID 63 <DIM>  <COMVARLST> <CBLST>
                      EOS <DECLP>
                ->    SAVE <SAVEELMT> <SAVELST> EOS <DECLP>
                ->    <TYPE> ID 64 <DIM> <LEN> 65 <DECLLST> EOS <DECLP>
              ----------------------------------------
<FUNCDATP>      ->    EMPTY
                ->    LABEL 46  FORMAT 47 ( <FORMSPEC> ) EOS <FUNCDATP>
                ->    ENTRY ID 6 <SUBFPARLST> EOS <FUNCDATP>
                ->    DATA 130 <VARELMT> <VARELMTLST> / <CONSTELMT>
                       <CONSTELMTLST> 139 /  <DATLST> EOS <FUNCDATP>
                ->    <FUNCTION->ID 68 ( <FPARLST> ) = <COND> EOS
                      <FUNCDATP>
              ----------------------------------------
<STMTS>         ->    EMPTY
                ->    LABEL 46  FORMAT 47 ( <FORMSPEC> ) EOS <STMTS>
                ->    ENTRY ID 6 <SUBFPARLST> EOS <STMTS>
                ->    DATA 130 <VARELMT> <VARELMTLST> / <CONSTELMT>
                          <CONSTELMTLST> 139 /  <DATLST> EOS <STMTS>
                ->   70 <LABELP> <STMT> 87 EOS <STMTS>
```

D A T A T Y P E S

```
<FUNCTYPE>      ->    EMPTY ! <TYPE>
<TYPE>          ->   INTEGER 76 <LEN> ! REAL 77 <LEN> !          /* C2 */
                     DOUBLE PRECISION 78  ! COMPLEX 79 <LEN> !   /* C3 */
                     LOGICAL 80 <LEN> ! CHARACTER 81 <LEN>       /* C4 */
```

```
                  ----------------------------------------
<LEN>             ->    EMPTY  !  * <LENEXPR>
<LENEXPR>         ->    INT  82 !   ( * 83 )  !   ( <EXPR> 84 )
                  ----------------------------------------
<LABELP>          ->    EMPTY  !  LABEL 85

          E X E C U T A B L E   S T A T E M E N T S
==========================================================

<STMT>            ->   <SIMPLE>  !  <COMPOUND>
                  ----------------------------------------
<SIMPLE>          ->   <VAR> = <COND> 42
                  ->   ASSIGN 67  INT TO  ID
                  ->   GOTO <GOTOMODE>
                  ->   IF ( <COND> ) <IFMODE>
                  ->   CONTINUE
                  ->   STOP 120
                  ->   STOP <EXPR> 121
                  ->   PAUSE 122
                  ->   PAUSE <EXPR> 123
                  ->   READ 109  <IOSTATLST>
                  ->   WRITE 110  <IOSTATLST> 126
                  ->   PRINT 110  <IOSTATLST> 126
                  ->   BACKSPACE  144 <AUXIOLST>
                  ->   ENDFILE  145 <AUXIOLST>
                  ->   REWIND  146 <AUXIOLST>
                  ->   OPEN  141 ( <CINFO> <CINFOS> ) 113
                  ->   CLOSE  142 ( <CINFO> <CINFOS> ) 113
                  ->   INQUIRE  67 ( <CINFO> <CINFOS> )
                  ->   CALL  ID 10  <SUBAPARLST> 43
                  ->   RETURN <RETURNMODE>
                  ----------------------------------------
<COMPOUND>        ->   <BLOCK->IF ( <COND> ) 96 THEN EOS <STMTS>
                            <ELSEIFST> <ELSEST> <LABELP> END IF 99
                  ->   DO  INT 88  <KM> <DORANGE>
                  ----------------------------------------
<RETURNMODE>      ->    EMPTY 124   !  <EXPR> 125
                  ----------------------------------------
<GOTOMODE>        ->    INT 107
                  ->   ( 117  INT 118  <LBLST> ) <KM> <EXPR> 119
                  ->    ID  67 <KM> ( INT <LBLST> )
                  ----------------------------------------
<IFMODE>          ->    INT 102 ,  INT 103 ,  INT 104
                  ->   100 <SIMPLE> 101
                  ----------------------------------------
<ELSEIFST>        ->    EMPTY
                  ->   70 ELSE IF 97 ( <COND> ) 98 THEN EOS <STMTS>
                       <ELSEIFST>
<ELSEST>          ->    EMPTY  !  ELSE 97 EOS <STMTS>



                  ----------------------------------------
<DORANGE>         ->    93 ID 69 = <EXPR> , <EXPR> <INCR> 94
                  ->   WHILE 89 ( <COND> ) 90
                  ->   UNTIL 91 ( <COND> ) 92
                  ----------------------------------------
<INCR>            ->    EMPTY 95  !  , <EXPR>
```

S Y N T A X O F L I S T S

```
<SUBFPARLST>    ->    EMPTY 39  !  ( 39 <FPARLST>  )
<FPARLST>       ->    EMPTY     !  <FPAR> <FPARS>
<FPARS>         ->    EMPTY     !  , <FPAR> <FPARS>
<FPAR>          ->    ID 40     !  * 41
                ------------------------------------
<SUBAPARLST>    ->    EMPTY     !  ( <APARLST> )
<APARLST>       ->    EMPTY     !  <APAR> <APARS>
<APARS>         ->    EMPTY     !  , <APAR> <APARS>
<APAR>          ->   <COND> 34  !  *  INT 35  ! <ARRAY-> ID 69 34
                ------------------------------------
<FORMSPEC>      ->    EMPTY     !  <FMT> <FMTLST>
<FMTLST>        ->    EMPTY     !  <KM> <FORMSPEC>
<KM>            ->    EMPTY     !  ,
<FMT>           ->   <RPT> <FMT>!  ( <FORMSPEC> )  !  <ONEFMT>
<ONEFMT>        ->    CHARACTERS  ! / ! :
                ->    LETTER <RPT> <SECLETTER> <RPT> <WIDTH>
<RPT>           ->    EMPTY     !  INT
<SECLETTER>     ->    EMPTY     !  LETTER
<WIDTH>         ->    EMPTY     !  . INT  !  LETTER INT
                ------------------------------------
<CONSTLST>      ->    EMPTY     !  , ID 48 = <COND> 49 <CONSTLST>
                ------------------------------------
<IMPLLST>       ->    EMPTY
                ->    , <TYPE> (  LETTER 50 <UPLET> 52 <LETLST> )
                        <IMPLLST>
<LETLST>        ->    EMPTY     !  ,  LETTER <UPLET> <LETLST>
<UPLET>         ->    EMPTY     !  -  LETTER 51
                ------------------------------------
<ARRAYLST>      ->    EMPTY     !  , ID 53 <ADIM>  <ARRAYLST>
<ADIM>          ->   ( 54 <BOUND> 57 <BOUNDS> )
<BOUNDS>        ->    EMPTY     !  , <BOUND> 57 <BOUNDS>
<BOUND>         ->   * 58   !  <EXPR> 55 <UPBOUND>
<UPBOUND>       ->    EMPTY     !  : <UPBD>
<UPBD>          ->   * 59   !  <EXPR> 56
<DIM>           ->    EMPTY     !  <ADIM>
                ------------------------------------
<DATLST>        ->    EMPTY     !  <KM> 130  <VARELMT> <VARELMTLST>
                      / <CONSTELMT> <CONSTELMTLST> 139 /  <DATLST>
<VARELMTLST>    ->    EMPTY     !  , <VARELMT> <VARELMTLST>
<VARELMT>       ->   ID 131     !  <DATDOLST> 132
<CONSTELMTLST>  ->    EMPTY     !  , <CONSTELMT> <CONSTELMTLST>
<CONSTELMT>     ->   <REPLIK> <DATCONST> 138
<REPLIK>        ->    EMPTY 133 !  INT 134  *  !  ID 135  *
<DATCONST>      ->    ID 136    !  <CONSTANTS> 137
<DATDOLST>      ->   ( <DATDOELMTS> ,  ID = <EXPR> , <EXPR> <INCR> )
<DATDOELMTS>    ->   ID <DATDOS>  !  <DATDOLST> <DATDOS>
<DATDOS>        ->    EMPTY     !  , <DATDOELMTS>
```

```
<EQVLST>    ->  EMPTY  !  , ( ID 60 <INDLST> , ID 62
                       <INDLST> <EQUIV> ) <EQVLST>
<EQUIV>     ->  EMPTY  !  , ID 61 <INDLST> <EQUIV>
------------------------------------------------------
<CBLST>     ->  EMPTY
            ->  <KM> <CBID> ID 63 <DIM> <COMVARLST> <CBLST>
<CBID>      ->  EMPTY 9  !  // 9  !  /  ID 8  /
------------------------------------------------------
<COMVARLST> ->  EMPTY  !  , ID 63 <DIM> <COMVARLST>
------------------------------------------------------
<DECLST>    ->  EMPTY  !  , ID 66 <DIM> <LEN> 65 <DECLLST>
------------------------------------------------------
<IDLST>     ->  EMPTY  !  , ID 7 <IDLST>
------------------------------------------------------
<EXPRLST>   ->  EMPTY  !  , <EXPR> <EXPRLST>
------------------------------------------------------
<SAVELST>   ->  EMPTY  !  , <SAVEELMT> <SAVELST>
<SAVEELMT>  ->  EMPTY  !  ID  !  /  ID 8  /
------------------------------------------------------
<LBLST>     ->  EMPTY  !  , INT 118  <LBLST>
------------------------------------------------------
<IOSTATLST> ->  112 <CINFO> 113  <IOELMTS>
            ->  ( 111 <CINFO> <CINFOS> 113 ) <IOLST>
------------------------------------------------------
<AUXIOLST>  ->  143 <CINFO> 113
            ->  143 ( <CINFO> <CINFOS> ) 113
------------------------------------------------------
<IOLST>     ->  EMPTY  !  <IOELMT> <IOELMTS>
------------------------------------------------------
<IOELMT>    ->  <COND> 116  !  <IODOLST>  !  <ARRAY-> ID 71
<IOELMTS>   ->  EMPTY  !  , <IOELMT> <IOELMTS>
------------------------------------------------------
<IODOLST>   ->  ( 127 <IOLST> 128 , ID 69 = <EXPR> , <EXPR>
                <INCR> 129 )
------------------------------------------------------
<CINFO>     ->  <IOPAR>
            ->  <IOMOD> = <IOPAR>
<CINFOS>    ->  EMPTY  !  , <CINFO> <CINFOS>
------------------------------------------------------
<IOPAR>     ->  <EXPR> 114  !  * 115  !  <ARRAY-> ID 69 114
------------------------------------------------------
<IOMOD>     ->  UNIT ! FMT ! REC ! IOSTAT ! FILE
            ->  STATUS ! ACCESS ! FORM ! RECL ! BLANK
            ->  EXIST ! OPENED ! NUMBER ! NAMED ! NAME
            ->  SEQUENTIAL ! DIRECT ! FORM ! FORMATTED
            ->  UNFORMATTED ! RECL ! NEXTREC
```

S Y N T A X O F E X P R E S S I O N S

```
<COND>          ->   <DISJ> <DISJS>
<DISJS>         ->    EMPTY  !  .EQV. <DISJ> 12 <DISJS>  !
                      .NEQV. <DISJ> 13 <DISJS>
<DISJ>          ->   <LTERM> <LTERMS>
<LTERMS>        ->    EMPTY  !  .OR. <LTERM> 14 <LTERMS>
<LTERM>         ->   <LFACT> <LFACTS>
<LFACTS>        ->    EMPTY  !  .AND. <LFACT> 15 <LFACTS>
<LFACT>         ->   .NOT. <RELEXPR> 16  !  <RELEXPR>
<RELEXPR>       ->   <RELTERM> <RELTERMS>
<RELTERMS>      ->    EMPTY  !
                ->    .LT. <RELTERM> 17  !  .LE. <RELTERM> 18
                ->    .EQ. <RELTERM> 19
                ->    .NE. <RELTERM> 20  !  .GE. <RELTERM> 21
                ->    .GT. <RELTERM> 22
<RELTERM>       ->   <EXPR> <EXPRNS>
<EXPRNS>        ->    EMPTY  !  // <EXPR> 23 <EXPRNS>
<EXPR>          ->   <TERM> <TERMS>  !  + <TERM> <TERMS>
                ->   - <TERM> 24 <TERMS>
<TERMS>         ->    EMPTY  !  + <TERM> 25 <TERMS>
                ->    - <TERM> 26 <TERMS>
<TERM>          ->   <FACT> <FACTS>
<FACTS>         ->    EMPTY  !  * <FACT> 27 <FACTS>
                ->    / <FACT> 28 <FACTS>
<FACT>          ->   <PRIM> <PRIMS>
<PRIMS>         ->    EMPTY  !  ** <PRIM> 29 <PRIMS>
<PRIM>          ->   <CONSTANTS> 30  !  <VAR>  !  <FUNCREF>
                ->   ( <COND> <IMAGPART> )
              ------------------------------------------
<FUNCREF>       ->   (FUNCTION-)ID 11  ( <APARLST> ) 33
              ------------------------------------------
<VAR>           ->  (ARRAY-)ID 31  ( <EXPR> <EXPRLST> ) 32 <SUBSTRING>
                ->   ID 69 <SUBSTRING>
              ------------------------------------------
<SUBSTRING>     ->    EMPTY  !  ( 37 <CHARPOS> : 37 <CHARPOS> ) 38
<CHARPOS>       ->    EMPTY  !  <EXPR>
              ------------------------------------------
<INDLST>        ->    EMPTY  !  ( <EXPR> 62 <EXPRS> )
<EXPRS>         ->    EMPTY  !  , <EXPR> 62 <EXPRS>
              ------------------------------------------
<CONSTANTS>     ->    LOGICAL   !  CHARACTERS !  INT  !  DINT
                ->    REAL  !  DOUBLE
<IMAGPART>      ->    EMPTY  !  , <EXPR> 36
              ------------------------------------------
<BLOCKID>       ->    EMPTY  !  ID
              ------------------------------------------


/*  COMMENT :
      /* C1-4 */ MEANS  AN EXTENSION NOT BELONGING TO THE
                 LANGUAGE FORTRAN 77
                 THE FORTRAN 77  CONSTRUCTS WOULD BE
/* C1 */         <UNITKIND>      ->   PROGRAM ID 3 EOS  | ε
/* C2 */         <TYPE>          ->   INTEGER 76 ! REAL 77  !
/* C3 */         <TYPE>          ->   DOUBLE PRECISION 78  ! COMPLEX 79
/* C4 */         <TYPE>          ->   LOGICAL 80 ! CHARACTER 81 <LEN>

END OF COMMENT */
```

Beschreibung der FCODE-Maschine

A Speicherstruktur

Der FCODE-Speicher besteht aus einer Anzahl von Compiler- oder Benutzerdefinierten Segmenten mit einer festen Länge.

SEGMENT = (NAME,LENGTH,DESCRIPTOR)

z.B.: CODESEGM,
CONSTSEGM,
COMMONSEGM 1..n,
STACKSEGM

Ein FCODE-Objekt wird über

ACCESS = (SEGMENT,OFFSET)

referenziert.

B Objekte

Ein FCODE-Objekt hat einen Typ und eine Referenzstufe:

CONSTANT = (TYPE,OFFSET), SEGMENT=CONSTSEGM, REF_LEVEL=$\emptyset$

VARIABLE = (TYPE,ACCESS), REF_LEVEL=1

TEMPORARY= (TYPE,REF_LEVEL,ACCESS)

Der Typ eines FCODE-Objekts kann sein:

ORDERED $\in$ {INTEGER,REAL,DOUBLE PRECISION}

ARITHM $\in$ ORDERED $\cup$ {COMPLEX}

CHAR $\in$ {CHARACTER*1..n}

LOG $\in$ {LOGICAL}

ANY $\in$ ARITHM $\cup$ LOG $\cup$ CHAR

GLOBAL $=$ SEGMENT_ADRESS

C Operatoren

Ein FCODE_MODULE ist eine getrennt übersetzbare Sammlung von Unterprogrammen und höchstens einem Hauptprogramm. Die Struktur eines FCODE_MODULE wird durch folgende Syntax beschrieben:

```
⟨MODULE⟩     ->  MODULE_OP  ⟨UNITS⟩  MODEND_OP
⟨UNITS⟩      ->  ⟨UNIT⟩ ⟨UNITS⟩  !  EMPTY
⟨UNIT⟩       ->  START_OP  ⟨OPS⟩  END_OP
⟨UNIT⟩       ->  PROC_OP  ⟨OPS⟩  PROCEND_OP
⟨OPS⟩        ->  ⟨OP⟩  ⟨OPS⟩  !  EMPTY
```

⟨OP⟩kann einer der folgenden Operatoren sein:

OP	Semantik
MODULE()	/* Initialization of Codegenerator */
MODEND()	/* Static end of module */
PROC(A,B)	/* Entry of subprogram A */ /* GLOBAL A,B; A = ADR(CODESEGM) B = ADR(VARSEGM) */ A : SKIP;
PROCEND()	/* Static end of subprogram */
START(A,B)	/* Start mainprogram here */ /* GLOBAL A,B; A = ADR(CODESEGM) B = ADR(VARSEGM) */ A : SKIP;
END()	proc END = () /* static and dynamic end of program */ ... ;
ENTRY(A,B)	/* alternate entrypoint of a subroutine */ /* GLOBAL A,B; A = ADR(CODESEGM) B = ADR(VARSEGM) */ A : SKIP;
EQV(A,B,C)	proc EQV = (ref LOG A, LOG B,C) A := B eqv C ;
NEQV(A,B,C)	proc NEQV = (ref LOG A, LOG B,C) A := B neqv C;
OR(A,B,C)	proc OR = (ref LOG A, LOG B,C) A := B or C;
AND(A,B,C)	proc AND = (ref LOG A, LOG B,C) A := B and C;
NOT(A,B)	proc NOT = (ref LOG A , LOG B) A := not B;

OP	Semantik
LT(A,B,C)	proc LT = (ref LOG A, ORDERED B,C) A := B < C;
LE(A,B,C)	proc LE = (ref LOG A, ORDERED B,C) A := B <= C;
GE(A,B,C)	proc GE = (ref LOG A, ORDERED B,C) A := B >= C;
GT(A,B,C)	proc GT = (ref LOG A, ORDERED B,C) A := B > C;
EQ(A,B,C)	proc EQ = (ref LOG A, ARITHM B,C) A := B = C;
NE(A,B,C)	proc NE = (ref LOG A, ARITHM B,C) A := B /= C;
ADD(A,B,C)	proc ADD = (ref ARITHM A,ARITHM B,C) A := B + C;
SUB(A,B,C)	proc SUB = (ref ARITHM A, ARITHM B,C) A := B - C;
MUL(A,B,C)	proc MUL = (ref ARITHM A, ARITHM B,C) A := B * C;
DIV(A,B,C)	proc DIV = (ref ARITHM A, ARITHM B,C) A := B / C;
POW(A,B,C)	proc POW = (ref ARITHM A, ARITHM B,C) A := B ** C;
MIN(A,B)	proc MIN = (ref ARITHM A, ARITHM B) A := - B ;
CAT(A,B,C)	proc CAT = (ref CHAR A, CHAR B,C) A := B // C; /* concatenation */
IMAG(A,B,C)	proc IMAG = (ref COMPLEX A, ARITHM B,C) A := B + i * C; /* i=sqrt(-1) */
INT(A,B)	proc INT = (ref INTEGER A, ARITHM B) A := int(B);
REAL(A,B)	proc REAL = (ref REAL A, ARITHM B) A := real(B);
COM(A,B)	proc COM = (ref COMPLEX A, ARITHM B) A := B + i * Ø;

OP	Semantik

SAVE(A)	`proc SAVE = (ref ANY A)`
	`/* this temporary value is used more than once, an information for the codegenerator */`
	`SKIP;`

ASG(A,B)	`proc ASG = (ref ANY A, ANY B)` `A := B;`
IND(A,B,C)	`proc IND = (ref ref ANY A, ref ANY B, INT C)` `A := B[C];`
STR(A,B,C)	`proc STR = (ref CHAR A, CHAR B, INT C)` `A := B(1:C); /* a substring */`

PAR(A)	`proc PAR = (ref ANY A)`
	`/* push A onto stack */`
	`...;`

CALL(A,B)	`proc CALL = (ref ANY A, GLOBAL B)`
	`/* call function B and store result in A */`
	`A := B();`

CALL(A)	`proc CALL = (GLOBAL A)`
	`/* call subroutine A */`
	`A();`

GOTO(A)	`proc GOTO = (LABEL A)` `goto A;`
IF(A,B)	`proc IF = (LOG A, LABEL B)` `if A then goto B fi;`
IFNOT(A,B)	`proc IFNOT = (LOG A, LABEL B)` `if not A then goto B fi;`

SWITCH(A,B,C)	`proc SWITCH = (INTEGER A,B,C)`
	`if B A and A C then`
	`/* jump into the following GOTO-TABLE */`
	`...`
	`else`
	`/* jump behind GOTO-TABLE */`
	`...`
	`fi;`

LABEL(A)	`/* location of LABEL A */` `A : SKIP;`

OP	Semantik
RET(A)	proc RET = (ref ANY A) /* return from function and put result onto stack */ ...;
RET()	proc RET = () /* return from subroutine */ ...;
LEN(A,B)	proc LEN = (ref INT A, CHAR B) A := length(B);
CHECK(A,B,C)	proc CHECK = (INT A,B,C) if A B or A C then /* cause adress-alarm */ fi;
LINE(A)	proc LINE = (INT A) /* information for the codegenerator */ SKIP;

D FCODE-Makros

Die FORTRAN 77-Kontrollstrukturen werden als FCODE-Makros realisiert. Sie expandieren jeweils in elementare FCODE-Operatoren.

DO(L1,L2,V,E1,E2,E3)	proc DO = (LABEL L1,L2, ORDERED E1,E2,E3, ref ORDERED V) /* perform DO-Loop processing and gene- rate apropriate jumps */ ...;
DOWHILE(L1,L2,C)	proc DOWHILE = (LABEL L1,L2, LOG C) /* evaluate condition and generate jumps */ ...;
DOUNTIL(L1,L2,L3,C)	proc DOUNTIL = (LABEL L1,L2,L3, LOG C) /* evaluate condition and generate jumps */ ...;
COMP_GOTO(L1..Ln,E)	proc COMP_GOTO = (LABEL L1 .. Ln, INT E) /* perform computed GOTO */ ...;

Beispiel:

```
COMP_GOTO(L1,L2 .. Ln,E)
expandiert in
  SWITCH(E,1,n)
  GOTO(L1)
  ...
  GOTO(Ln)
  LABEL(BEHIND)

IF (C) THEN
  ..
END IF
wird direkt in
  IFNOT(C,LAB)
  ...
  LABEL(LAB)
übersetzt.
```

Literaturverzeichnis

AEG 76 : "FORTRAN-Modul , Plex"
AEG-Entwicklungsdokument

AEG 78 : "Einführung in SL3"
Teil 1-3
AEG80-60 Handbuch

AEG 80 : "Befehlshandbuch" und
"SVC - Handbuch"
AEG80-60 Handbücher

AHO 72 : A.V.Aho / J.D. Ullman
"The Theory of Parsing,
Translation and Compiling"
Prentice Hall
Englewood Cliffs N.J. 1972

AHO 77 : A.V.Aho /J.D. Ullman
"Principles of Compiler
Design"
Addison Wesley
Reading, Massachusetts 1977

ANSI 66 : ANSI X3.9-1966
"American National Standard
Programming Language FORTRAN"
ANS - Institute
New York 1966

ANSI 78 : ANSI X3.9-1978
"American National Standard
Programming Language FORTRAN"
ANS - Institute
New York 1978

BRAINERD 77: W.Brainerd
"A Proposal for a FORTRAN
Loop Construct"
SIGPLAN Band 12
Dezember 1977
pp 60-67

BRAINERD 78: W.Brainerd (Herausgeber)
"FORTRAN 77"
CACM Band 21
Oktober 1978
pp 806-820

BRÜGGE 78 : B. Brügge
"Übertragung von Concurrent Pascal auf das DECsystem - 10"
in
B. Brügge et al
"Entwurf einer Concurrent Pascal Netzwerkmaschine und Untersuchungen zur Realisierbarkeit"
Gruppendiplomarbeit
Fachbereich für Informatik der Universität Hamburg
Februar 1978

COMP 74 : F.L. Bauer / J. Eickel (Herausgeber)
"Compiler Construction- an Advanced Course"
Springer Lect.Notes in Computer Science Nr. 21
Berlin - Heidelberg 1974

DEREMER 74 : F.L. DeRemer
"Lexical Analysis"
in (COMP 74)
pp 109-120

DUERR 75 : D. Dürr
"SL3 - eine Systemprogrammier-sprache auf ALGOL68-Basis als Grundlage für die Prozeßrechnerlinie AEG80"
Angewandte Informatik Band 17
September 1975
pp 393-399

ELSWORTH 79: D. Elsworth
"Compilation via an Intermediate Language"
The Computer Journal Band 22
August 1979
pp 226-233

FELDMAN 79 : S.I. Feldman
"Implementation of a Portable FORTRAN 77 Compiler Using Modern Tools"
SIGPLAN Band 14II
August 1979
pp 98-106

FRIESLAND 79: G. Friesland
"Eine attributierte Übersetzungs-grammatik für den PASCAL-PCODE-Compiler"

GOOS 80 : G. Goos
"Implementation von ADA"
Kolloquiumsvortrag
Universität Hamburg
Mai 1980

GRIES 71 : D. Gries
"Compiler Construction for
Digital Computers"
Wiley
New York 1971

GRIFFITHS 74: M. Griffiths
"LL(1) Grammars and Analysers"
in (COMP 74)
pp 57-84

HARTMANN 77: "A CONCURRENT PASCAL Compiler
for Minicomputers"
Springer Lect.Notes in Computer
Science Nr. 50
Berlin 1977

HORNING 74 : J.J. Horning
"Structuring Compiler Development"
in (COMP 74)
pp 498-513

IBM 66 : " FORTRAN IV (H) Compiler
Program Logic Manual"
IBM Entwicklungsdokument 1966

JESSEN 79 : E. Jessen
"Programmiersprachen"
Vorlesungsskript SS 1979
Universität Hamburg

JENSEN 75 : K. Jensen / N. Wirth
"PASCAL User Manual and Report"
Springer Lect.Notes in Comp.Science Nr.18
Berlin 1975

JONSSON 79 : B. Jonsson / A. Kley / D. Potocan
"Methoden des Compilerbaus"
Wiss.Ber. AEG-Telefunken 1-2 1979
pp 24-30

KOSTER 74 : C.H.A. Koster
"Portable Compilers and the
UNCOL-Problem"
pp 253-264
in
W.L. van der Poel / L.A. Maarsen (Herausgeber)
"Machine Oriented Higher Level Languages"
North-Holland
Amsterdam 1974

KUPKA 77 : I. Kupka
"Makroprozessoren"
Vorlesungsskript SS 1977
Universität Hamburg

M.LEHMANN: "Compilerbau"
Vorlesungsmitschrift SS 1977
Universität Hamburg

LEWIS 76 : P.M. Lewis / D.J. Rosenkrantz /R.E. Stearns
" Compiler Design Theory"
Addison Wesley
Massachusetts 1976

MAYER 78 : O. Mayer
" Syntaxanalyse"
Bibliographisches Institut
Reihe Informatik Nr. 27
Mannheim 1978

McKEEMAN 74: W.M. McKeeman
"Compiler Construction"
in (COMP 74)
pp 1-36

MILTON 79 : D.R. Milton et al
" An All(1) Compiler Generator"
SIGPLAN Band 14II
August 1979
pp 152-157

POOLE 74 : P.C. Poole
"Portable and Adaptable
Compilers"
in (COMP 74)
pp 427-497

PRAHL 78 : U. Prahl
"ZS-Report"
AEG - Entwicklungsdokument 1978

RICHARDS 79: M. Richards
"BCPL - The Language and its Compiler"
Cambridge University Press
Cambridge 1979

SCHATZ 79 : B.R. Schatz et al
"TCOL$_{ADA}$ An Intermediate Representation for the DOD Standard Programming Language"
Dep.of Comp.Science, Carnegie Mellon University
Pittsburgh 1979

SNYDER 75 : A. Snyder
" A Portable Compiler for the Language C"
MIT Techn.Report Nr. 149
Mai 1975

WAITE 74 a : W. M. Waite
"Semantic Analysis"
in (COMP 74)
pp 157-169

WAITE 74 b : W. M. Waite
"Optimization"
in (COMP 74)
pp 549-600

WAITE 74 c : W. M. Waite
"Code Generation"
in (COMP 74)
pp 302-332

WIRTH 77 : N. Wirth
"Compilerbau"
Teubner
Stuttgart 1977

WULF 75 : W. Wulf et al
"The Design of an Optimizing Compiler"
American Elsevier
New York 1975

Hiermit bestätige ich, daß ich die Arbeit selbständig angefertigt und außer den angegebenen Quellen und Literaturzitaten keine anderen Hilfsmittel verwendet habe.

Jens-Detlev Dall