

---

**Universität Stuttgart  
Institut für Informatik  
Abteilung Programmiersprachen**

Prüfer: Prof. Erhard Plödereder  
Betreuer: Bernd Holzmüller  
CR-Nummern: D.3.4, D.3.3, D.4.2, D.1.5

Begonnen am: 1. Januar 1997  
Beendet am: 31. Mai 1997

Diplomarbeit-Nr. 1484

**Implementierung eines  
Eiffel-Compilers  
für SUN/SPARC**

Fridtjof Siebert

Fridtjof Siebert  
Nobileweg 67  
70439 Stuttgart  
fridi@studbox.uni-stuttgart.de

## Aufgabenstellung

Aufgabe ist die Implementierung eines Compilers für die Programmiersprache Eiffel V3 [Meyer92]. Der Compiler selbst soll dabei in Eiffel implementiert werden. Der Compiler soll zunächst einen maschinenunabhängigen Zwischencode erzeugen, aus dem dann durch ein *Backend* Code für eine SUN/SPARC erzeugt wird.

## Einschränkungen beim Sprachumfang

Es sollen keine Einschränkungen gemacht werden, die es unnötig schwer machen, später den Compiler zu einem effizienten Compiler für den gesamten Eiffel-Sprachumfang weiterzuentwickeln. Um das Projekt jedoch in dem beschränkten Zeitrahmen der Diplomarbeit realisierbar zu machen, sollen folgende Einschränkungen gemacht werden:

- *Once-* und *external-*Routinen (zum Aufruf von C-Code) müssen zunächst nicht unterstützt werden.
- *Pre-* und *Postconditions*, Klasseninvarianten, Schleifenvarianten und Invarianten und *Check-*Anweisungen brauchen zunächst nicht implementiert zu werden. Insbesondere kann die Implementierung der *Old-* und *Strip-*Ausdrücke weggelassen werden.
- Die *Debug-* und die *Retry-*Anweisung brauchen nicht implementiert zu werden.
- *Manifest-Arrays* brauchen zunächst nicht implementiert zu werden.
- Exceptions brauchen zunächst nicht unterstützt zu werden.
- *Bit Types* brauchen nicht unterstützt zu werden.
- Die volle Implementierung der Standardbibliothek wird nicht verlangt.
- Die Überprüfung der *System Validity* braucht zunächst nicht implementiert zu werden.
- Es soll ausführlich die mögliche Implementierung eines inkrementellen Garbage-Collectors beschrieben werden, auch wenn dieser zunächst nicht realisiert wird. Der Compiler braucht zunächst nur die Allokierung von Objekten zu erlauben, ohne automatische oder explizite Freigabe zu ermöglichen.

## Literatur

- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertfordshire, 1992

---

# Inhalt

<b>1. Einleitung</b>	<b>1</b>
1.1 Ziel der Arbeit	1
1.2 Bezeichnungen	2
1.3 Diagramme	3
<b>2. Datenstrukturen</b>	<b>4</b>
2.1 Grundüberlegungen	4
2.2 Listen	5
2.2.1 Unsortierte Listen	5
2.2.2 Sortierbare Listen	5
2.3 Teilweise sortierte Felder	6
2.3.1 Eigenschaften der Partly Sorted Arrays	6
2.3.2 Definition des Partly Sorted Arrays	6
2.3.3 Einfügen von Elementen	7
2.3.4 Aufwand des Einfügens im <i>PS_ARRAY</i>	8
2.3.5 Finden von Elementen	9
2.3.6 Sortieren des Feldes	10
2.4 Zeichenketten	11
<b>3. Der Scanner</b>	<b>12</b>
3.1 Die Klassen des Scanners	12
3.1.1 Die Klasse <i>POSITION</i>	13
3.1.2 Die Klasse <i>SCANNER_SYMBOL</i>	13
3.1.3 Die Klasse <i>SCANNER</i>	13
3.1.3.1 Die Terminale Symbole <i>and then</i> und <i>or else</i>	14
3.1.3.2 <i>INTEGER</i> -Konstante gefolgt von <i>double dot</i>	14
3.1.3.3 Bezeichner und Schlüsselwörter	14
3.2 Parserunterstützung	15
3.2.1 Symbole holen	15
3.2.2 Überprüfen bestimmter Symbole	15

3.2.3 Besondere Attribute von Scannersymbolen	15
3.2.4 First-Mengen	15
<b>4. Der Parser</b>	<b>16</b>
4.1 Parsingverfahren	16
4.1.1 Arbeitsweise der Parser	16
4.1.2 Operator-Präzedenz-Parser für Ausdrücke	16
4.2 Der abstrakte Syntaxbaum	17
4.2.1 Struktur des abstrakten Syntaxbaumes	17
4.2.2 Abstrakte Klassen und Vererbung im Syntaxbaum	18
<b>5. Validity-Überprüfung</b>	<b>20</b>
5.1 Ausgabe der Fehlermeldungen	20
5.2 Bestimmung der Schnittstellen der Klassen	21
5.3 Conformance	23
5.4 System Validity	23
<b>6. Die Struktur des Zielcodes</b>	<b>24</b>
6.1 Aufbau der Objekte	24
6.1.1 Expandierte Objekte	24
6.1.2 Referenzobjekte	25
6.1.3 Objekte bei Mehrfachvererbung	25
6.1.4 Objekte generischer Klassen	26
6.1.5 Position des Typdeskriptorzeigers	26
6.2 Featureaufrufe und dynamisches Binden	27
6.2.1 Implementierung der dynamischen Bindung in anderen Compilern	27
6.2.1.1 <i>SmallEiffel</i> von Dominique Colnet	27
6.2.1.2 Smalltalk-80 System von Deutsch und Schiffman	28
6.2.1.3 C++ Implementierungen	28
6.2.2 Unqualifizierte Aufrufe der Features von <i>Current</i>	31
6.2.3 Qualifizierte Featureaufrufe	31
6.2.4 Ein Beispiel	33
6.2.5 Aufwandsvergleich	35
6.3 Generische Klassen	37
6.4 Parameter und lokale Variablen	39
6.4.1 Parameterübergabe	39
6.4.2 Funktionsergebnisse	39
6.4.3 Lokale Variablen	40
6.5 Assignment-Attempt	40
6.6 Once-Routinen	40
<b>7. Zwischencodeerzeugung</b>	<b>42</b>
7.1 Lokale Variablen	42
7.1.1 Typen lokaler Variablen	42
7.1.1.1 <i>Reference</i>	43
7.1.1.2 <i>Pointer</i>	43
7.1.1.3 <i>Integer</i> , <i>Character</i> und <i>Boolean</i>	43
7.1.1.4 <i>Real</i> und <i>Double</i>	43
7.1.1.5 Expandierte Typen	43
7.1.2 Arten von lokalen Variablen	43
7.1.2.1 Argumente	44

7.1.2.2 Lokale Werte	44
7.2 Zwischencodebefehle	44
7.2.1 Zuweisung	44
7.2.2 Zuweisung einer Konstanten	45
7.2.3 Speicherzugriffe	45
7.2.4 Berechnungen	46
7.2.5 Adressbestimmung	46
7.2.6 Aufrufe	46
7.3 Kontrollstrukturen	47
7.3.1 Kein Nachfolger	47
7.3.2 Reihung	47
7.3.3 Bedingte Anweisung	48
7.3.4 Multi_branch	49
7.4 Zwischencodeerzeugung für Ausdrücke	50
7.4.1 Werte	50
7.4.1.1 Konstanten	51
7.4.1.2 Lokale Variablen	51
7.4.1.3 Speicheradressen	51
7.4.1.4 Boolesche Werte	51
7.4.2 Aufrufe	53
7.4.2.1 Unqualifizierter Routinenaufruf	53
7.4.2.2 Qualifizierter Routinenaufruf	53
7.4.2.3 Unqualifizierter Zugriff auf Attribute	54
7.4.2.4 Qualifizierter Zugriff auf Attribute	54
<b>8. Zielcodeerzeugung</b>	<b>55</b>
8.1 Objektorientiertes Design	55
8.2 Phasen der Zielcodeerzeugung	56
8.2.1 Erste Expandierungsphase	57
8.2.2 Kopie-Propagierung	58
8.2.3 Registervergabe	58
8.2.3.1 Bestimmung der Lebenszeiten	58
8.2.3.2 Zuweisungen an tote Variablen entfernen	59
8.2.3.3 Bestimmung der Konfliktmatrix	59
8.2.3.4 Registerallokation	60
8.2.4 Stapelallokation	61
8.2.5 Zweite Expandierungsphase	61
8.2.6 Instruction Scheduling	62
8.2.7 Maschinencodeerzeugung	62
8.3 Textliche Trennung der maschinenabhängigen Teile	63
8.4 Erzeugung der Objektdatei	63
<b>9. Systemerzeugung</b>	<b>65</b>
9.1 Systemstart	65
9.2 Globale Informationen	65
9.3 Typdeskriptoren	66
<b>10. Laufzeitsystem</b>	<b>67</b>
10.1 Allozierung von Speicher für Objekte	67
10.2 Typüberprüfungen	68
10.3 Integerarithmetik	69

---

<b>11. Standardklassen</b>	<b>70</b>
11.1 Expandierte Standardklassen	70
11.2 Die Klasse <i>GENERAL</i>	71
11.3 Die Klasse <i>ARRAY</i>	72
11.3.1 Zugriff auf Elemente des Feldes	72
11.3.2 Verwaltung des Speichers für die Feldelemente	73
11.4 Die Klasse <i>STRING</i>	73
<b>12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer</b>	<b>74</b>
12.1 Grundüberlegungen	74
12.2 Algorithmus	75
12.2.1 Der Collectorzyklus	75
12.2.2 Aufgaben des Mutators	76
12.2.2.1 Zeigerzuweisungen	76
12.2.2.2 Erzeugen von Objekten	76
12.2.2.3 Globale Variablen	77
12.2.3 Lokale Variablen	77
12.2.4 Implementierung	78
12.3 Aktivierung des Collectors und Zeitaufteilung	78
12.3.1 Wechsel zwischen Mutator und Collector an Sollbruchstellen	79
12.3.2 Wechsel zum Collector nur beim Allozieren	80
12.3.2.1 Konstantes $u$	80
12.3.2.2 Variables $u(n)$	80
12.3.2.3 Objekte unterschiedlicher Größe	83
12.3.2.4 Sweep-Phase des Collectors	84
12.4 Größenklassen	84
12.5 Kompaktierung	86
12. Anhang A: Implementierung des Collectors	87
12.A.1 Objektstruktur	87
12.A.2 Der Collector-Zyklus	88
12.A.3 Zeigerzuweisungen	90
12.A.4 Aktivierungsblöcke	91
12. Anhang B: Wechsel zwischen Mutator und Collector	93
12. Anhang C: <i>Allocate</i> und <i>Free</i>	93
12. Anhang D: Kompaktierer	95
<b>13. Benutzeranleitung</b>	<b>97</b>
13.1 Installation	97
13.2 Aufruf des Compilers	98
13.3 Arbeitsweise des Compilers	98
13.4 Beispielcompilation	99
13.5 Standardbibliotheken	100

*Writing an Eiffel compiler is not a piece of cake. Other compiler authors would probably confirm that it is unrealistic to produce something decent in less than two years' calendar time. [...] I won't go into the details of why the task is difficult; it is not a matter of complexity but one of intellectual challenge – the need to reconcile high-level concepts such as repeated inheritance, disciplined exception handling, automatic memory management, typing etc. with the constraints of hardware performance.*  
– Bertrand Meyer

---

# 1. Einleitung

Zunächst sollen in diesem Kapitel einige grundsätzliche Anmerkungen zu dieser Arbeit, deren Aufbau und den verwendeten Begriffen und Notationen gemacht werden.

## 1.1 Ziel der Arbeit

Bisherige Eiffel-Implementationen benutzen als Zielsprache meist C-Quelltext. Im Vergleich zur direkten Erzeugung von Maschinencode führt dies nicht nur zu einer umständlicheren und langsameren Übersetzung der Quelltexte. Vielmehr ist dadurch auch die Erzeugung effizienten Codes für Besonderheiten der Sprache erschwert, etwa der für einen dynamisch gebunden Aufruf nötige Code oder die für eine effiziente Implementierung des Garbage-Collectors zu beachtenden Besonderheiten.

Es war Ziel dieser Arbeit, bei konsequenter Verwendung objektorientierter Techniken, einen Compiler zu entwickeln. Die Beschreibung des objektorientierten Aufbaus des Compilers nimmt daher auch einen großen Teil dieser Ausarbeitung ein. Die Reihenfolge der Kapitel entspricht dabei der fortschreiten Übersetzung in den einzelnen Phasen des Compilers: Scanner, Parser, Überprüfung der Gültigkeit, Zwischencodeerzeugung, Zielcodeerzeugung und Systemerzeugung. Schließlich befassen sich noch zwei Kapitel mit Besonderheiten des Laufzeitsystems und den Standardklassen.

Dies soll jedoch nicht nur eine Arbeit über die Implementierung eines Compilers sein: Eiffel besitzt eine Reihe an leistungsfähigen Konstrukten, wie die flexible Mehrfachvererbung und Generizität, deren effiziente Implementierung eine gute Wahl der im Zielcode benutzten Strukturen verlangt. Bevor die Zwischencodeerzeugung beschrieben wird, ist daher ein großes Kapitel nur diesen Strukturen gewidmet.

Schließlich ist ein Garbage-Collector in Eiffel unerlässlich. Auch wenn er im Rahmen dieser Arbeit nicht implementiert werden konnte, so befasst sich doch ein eigenes Kapitel ausführlich mit der detaillierten Beschreibung einer effizienten Implementierung der Speicherverwaltung. Dieses Kapitel ist jedoch weitgehend unabhängig vom Rest dieser

Arbeit geschrieben, so dass es auch getrennt betrachtet werden kann. Es besitzt sogar eigene Anhänge.

Schließlich enthält das letzte Kapitel noch eine Benutzeranleitung für den implementierten Compiler. Da die Benutzung des Compilers kein Wissen über dessen Implementierung voraussetzt, ist auch dieses Kapitel unabhängig vom Rest der Arbeit gehalten und kann einzeln betrachtet werden.



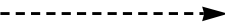


## 1.2 Bezeichnungen

In [Meyer92] verwendete englische Bezeichnungen werden wie folgt ins Deutsche übersetzt oder im Englischen belassen. Diese Ausdrücke werden in [Meyer92] definiert, die Definitionen sind zum Teil komplex und aufwendig und werden daher hier nicht wiederholt.

<u>Englische Bezeichnung</u>	<u>Deutsche Übersetzung</u>
heir(-class)	Tochter(-klasse)
parent(-class)	Vater(-klasse)
descendant	Nachfolger
ancestor	Vorgänger
client	Klient
type	Typ
feature	Feature
deferred	abstrakt
effective	konkret
attribute	Attribut
routine	Routine
function	Funktion
instruction	Anweisung
expression	Ausdruck
creation	Erzeugung
validity	Gültigkeit, Validity
validity constraint	Validity-Constraint
to conform	passen
conformance	Conformance
multiple inheritance	Mehrfachvererbung
repeated inheritance	wiederholte Vererbung
renaming	Umbenennung
joining	Joining
sharing	Sharing
redefinition	Redefinition
origin	Origin
seed	Seed
(feature-)call	(Feature-)Aufruf
(call-)target	Zielobjekt (eines Aufrufs)
(un-)qualified call	(un-)qualifizierter Aufruf
(generic) derivation	(generisches) Derivat
once-routine/-function	Once-Routine/-Funktion

## 1.3 Diagramme

Diagramme, die die Beziehungen zwischen Klassen beschreiben, halten sich an die in [Meyer92] gemachten Konventionen. Es werden die im folgenden beschriebenen Symbole benutzt:

<u>Symbol</u>	<u>Bedeutung</u>
	Die konkrete Klasse <i>PARSE_CLASS</i>
	Die abstrakte Klasse <i>EXPRESSION</i>
	„Benutzt“-Beziehung zwischen Klassen
	„Erbt-von“-Beziehung zwischen Klassen. „Ist-Referenz-auf“-Beziehung zwischen Speicherzellen.
	Speicherzelle oder Speicherbereich mit dem Wert <i>Attribut 1</i>

## Literatur

- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertfordshire, 1992

*Dynamic data structures expand and contract like an accordion. They can be collapsed to nothing or expanded continually until the memory capacity of the computer is exhausted. Widely varying lists, such as compiler cross-reference tables, recursion stacks, and simulation queues, often fit the loose mold of dynamic data structures.*

*– Modula-2: A Seafarer's manual and Shipyard guide, Edward J. Joyce*

---

## 2. Datenstrukturen

Ein Compiler benötigt eine Reihe an effizienten Datenstrukturen, wie sie in anderen Arten von Programmen weniger häufig vorkommen.

### 2.1 Grundüberlegungen

Die nötigen Datenstrukturen sind vor allem Container-Strukturen wie Listen, über deren Elemente oft eine Totalordnung definiert werden kann (d. h. zwei unterschiedliche Elemente sind stets vergleichbar, eines kommt vor dem anderen). So sind zum einen Listen wichtig, die möglicherweise später sortiert werden. Zum anderen sind, beispielsweise für Bezeichner, Strukturen nötig, in die nach und nach weitere Elemente eingefügt werden, bei denen jedoch auch während des Aufbaus der Struktur ein schnelles Finden von Elementen wichtig ist.

Zum Speichern von Elementen in Listen oder Bäumen ist es gewöhnlich nötig, dass die Elemente von einem bestimmten Knotentyp abgeleitet sind. So müssen beispielsweise die Elemente einer typischen Liste in Eiffel als Nachfolger einer Klasse wie *LINKABLE* deklariert werden. Dies erscheint aus zwei Gründen als unschön: Zum einen ist es so nicht möglich, dasselbe Element in mehreren solchen Strukturen gleichzeitig zu speichern (da es Attribute wie *next* nur einmal besitzt) – wird es versehentlich doch in mehreren Strukturen gleichzeitig gespeichert, so sind die Folgen katastrophal. Zum anderen ist die Definition der Elemente nicht unabhängig von der Datenstruktur, in der sie gespeichert werden sollen. Eine spätere Änderung der Struktur erzwingt die Änderung des *inherit*-Teils der Klassen aller in ihr gespeicherten Objekte und damit die Änderung eines großen Teils des Quelltextes.

Es soll aus diesen Gründen von den Elementen eines Containers also nicht verlangt werden, selbst Informationen zu enthalten, die nur für die Speicherung im Container nötig sind.

Lediglich muss im Fall von sortierter Speicherung von den Elementen verlangt werden, dass sie eine Vergleichsoperation zur Verfügung stellen. Dies geschieht in Eiffel gewöhnlich durch die Definition als Tochterklasse von *COMPARABLE*. Ein direkter Vergleich zweier Elemente erscheint meist nicht sinnvoll (die Features *one* und *infix "\*"* können nicht miteinander verglichen werden), sondern lediglich ein Vergleich eines Schlüssels der Elemente (der Name des Features *one* kann mit dem des Features *infix "\*"* verglichen werden). Daher soll von den Elementen, die sortiert gespeichert werden sollen, verlangt werden, dass sie als Nachfolger der generischen Klasse *SORTABLE* definiert sind. Diese Klasse hat als generischen Parameter den Typ des Schlüssels, der von *COMPARABLE* abgeleitet sein muss. Die Klasse ist sonst trivial: Sie stellt lediglich das Attribut *key* von diesem generischen Typ bereit:

```
class SORTABLE [KEY -> COMPARABLE]
  feature
    key: KEY;
  end -- SORTABLE
```

## 2.2 Listen

### 2.2.1 Unsortierte Listen

Die häufigste Container-Struktur sind Listen von Elementen. Die generische Klasse *LIST* wurde hierfür definiert. Der generische Parameter unterliegt keiner Typbeschränkung. Die wichtigsten Operationen auf Listen sind das Einfügen von Elementen an das Listenende und die Iteration über die Elemente der Liste.

Die Verwendung eines *Cursors* für die Iteration einer Liste, wie er in [Meyer88] vorgeschlagen wird, ist bei der rekursiven Struktur eines Compilers gefährlich. Es müsste hier sichergestellt werden, dass während der Iteration einer Liste keine andere aufgerufene Routine dieselbe Liste iteriert und damit den *Cursor* verändert. Um dieser Gefahr auszuweichen, soll auf die einzelnen Elemente der Liste mit einem *Integer*-Index zugegriffen werden, so wie gewöhnlich auf die Elemente eines Feldes zugegriffen wird.

Implementiert werden Listen derzeit mit Hilfe der Klasse *ARRAY*. Um nicht unnötig Speicher zu verschwenden, aber auch hohe Effizienz bei großen Listen zu garantieren, wird beim Einfügen neuer Elemente die Größe des Feldes jeweils verdoppelt, wenn es für das neue Element nicht mehr ausreichend freien Platz bietet.

### 2.2.2 Sortierbare Listen

Eine spezielle Variante der Liste wird definiert für Datenstrukturen, die während Ihres Aufbaus unsortiert sein können, die danach aus Effizienzgründen jedoch nach einem Schlüssel sortiert werden sollen. Nötig ist dies beispielsweise bei der Auswertung einer *Multi\_Branch*-Anweisung, um zu prüfen, dass keine der angegebenen Alternativen doppelt vorkommt.

Sortierbare Listen werden implementiert durch die Klasse *SORTABLE\_LIST*, eine Tochterklasse von *LIST*, deren generischer Parameter auf Nachfolgerklassen von *SORTABLE* beschränkt ist. *SORTABLE\_LIST* führt die Features *sort* und *find* ein, mit denen die Liste

sortiert werden kann und in der sortierten Liste nach einem bestimmten Schlüssel gesucht wird.

## 2.3 Teilweise sortierte Felder

Schwieriger sind die Datenstrukturen für sortierbare Elemente, in denen schon während des Aufbaus der Struktur nach bestimmten Schlüsseln gesucht werden muss. Dieser Fall tritt z. B. bei der Bestimmung der Features einer Klasse auf: Geerbte und neue Features können nicht einfach hinzugefügt werden, sondern zunächst muss geprüft werden, ob ein gleichnamiges Element bereits existiert, das dann nach den Eiffel-Regeln (*Joining, Sharing, Redefinition*) mit dem neuen verschmolzen werden muss. Die Datenstruktur muss also ein schnelles Einfügen und Suchen nach Elementen erlauben.

Hierfür werden gewöhnlich binäre Baumstrukturen verwendet, wie beispielsweise AVL-Bäume [Sedgewick91]. Diese Strukturen sind jedoch schwer zu implementieren, wenn die Elemente nicht von dem Knotentyp des Baumes abgeleitet sind, was aus oben genannten Gründen nicht geschehen soll. Eine Allokation eines gesonderten Knotenobjektes für jedes Element des Baumes erscheint als unverhältnismäßig speicherintensiv.

### 2.3.1 Eigenschaften der Partly Sorted Arrays

Statt der Baumstruktur wird hier eine speziell für diesen Compiler entwickelte Feldstruktur, *Partly Sorted Array* oder *PS\_ARRAY*, verwendet werden, die in der untersuchten Literatur nicht erwähnt wird. In dieser Struktur sollen Elemente in durchschnittlich logarithmischer Zeit eingefügt und gesucht werden können. Außerdem soll nach der Erzeugung der kompletten Struktur ein vollständig sortiertes Feld in linearer Zeit erzeugt werden können.

### 2.3.2 Definition des Partly Sorted Arrays

Die Idee: Die Elemente werden in einem gewöhnlichen Feld  $f$  gespeichert. Dieses Feld hat dabei immer eine Größe von  $2^n - 1$  Elementen. Damit  $e$  Elemente gespeichert werden können, muss also  $e < 2^n$  gelten. Das Feld wird als eine Reihe von Teilfeldern  $t_0$  bis  $t_{n-1}$  betrachtet, wobei jedes  $t_i$  genau  $2^i$  Elemente speichert und in  $f$  den Indexbereich  $2^i$  bis  $2^{i+1} - 1$  belegt. *Bild 1* zeigt ein Feld  $f$  mit seinen Teilfeldern.

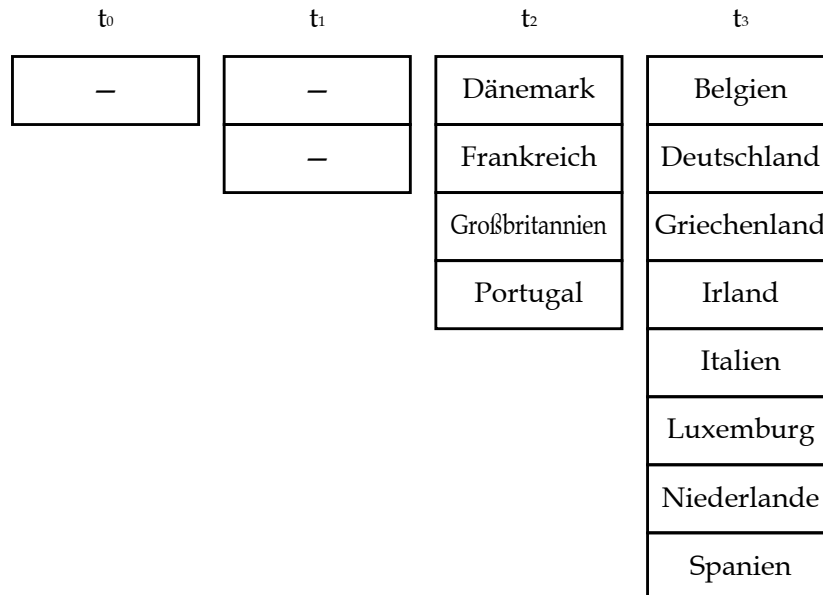
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t_0$	$t_1$	$t_2$				$t_3$								

**Bild 1:** Feld  $f$  mit 15 Elementen und den Teilfeldern  $t_0$  bis  $t_3$ .

Sind in einem *PS\_ARRAY*  $e$  Elemente gespeichert, so sind alle  $t_i$  gültig, für die  $(e/2^i) \bmod 2 = 1$  ist, d. h. in der Binärdarstellung der Zahl  $e$  gibt es für jede Eins ein Teilfeld  $t_i$ , dessen Größe gerade der Wertigkeit der Position der Eins entspricht. Alle gültigen Teilfelder sind komplett mit Elementen gefüllt, alle ungültigen Teilfelder enthalten dagegen keine Daten.

In einem *PS\_ARRAY* muss nun stets gelten: Die Elemente in jedem gültigen Teilfeld sind sortiert. *Bild 2* zeigt als Beispiel die Namen der früheren 12 Mitgliedsstaaten der

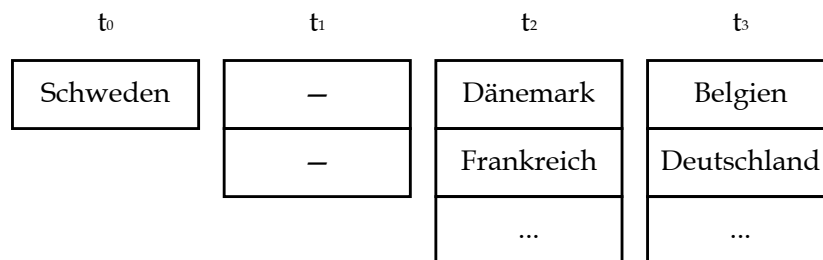
EU, alphabetsich nach ihren Namen sortiert. Die Teilfelder sind hier senkrecht angeordnet. Lediglich  $t_2$  und  $t_3$  enthalten gültige Elemente, die innerhalb der Teilfelder alphabetisch sortiert sind.



**Bild 2:** Mögliches  $PS\_ARRAY$  für die alte EU

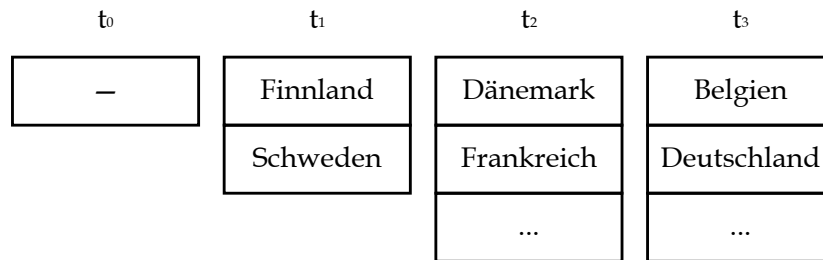
### 2.3.3 Einfügen von Elementen

Beim Einfügen erhöht sich die Zahl der Elemente des Feldes von  $e$  auf  $e'+1$ . Ist  $e$  gerade, so bedeutet dies lediglich, dass das zunächst ungültige  $t_0$  nun gültig wird. Da  $t_0$  nur ein einziges Element enthält, ist es auf jeden Fall sortiert. Das neue Element kann also einfach als das einzige Element von  $t_0$  eingefügt werden. Dies geschieht in unserem Beispiel dann, wenn auch Schweden in die EU aufgenommen wird, wie *Bild 3* veranschaulicht.



**Bild 3:**  $PS\_ARRAY$  aus *Bild 2* um Schweden erweitert ( $t_2$  und  $t_3$  unverändert).

Ist  $e$  dagegen ungerade, so gibt es eine Reihe an gültigen Teilfeldern  $t_0$  bis  $t_m$  mit  $m \geq 0$  und  $t_{m+1}$  ungültig (diese Teilfelder entsprechen in der Binärdarstellung von  $e$  den niederwertigen Einsen vor der ersten Null). Für  $e'$  werden  $t_0$  bis  $t_m$  ungültig,  $t_{m+1}$  dagegen gültig.  $t_0$  bis  $t_m$  haben zusammen  $2^{m+1}-1$  Elemente,  $t_{m+1}$  speichert  $2^{m+1}$  Elemente. Es können in  $t_{m+1}$  also alle Elemente aus  $t_0$  bis  $t_m$  zusammen mit dem neu einzufügenden Element gespeichert werden. Allerdings muss sichergestellt sein, dass die Elemente von  $t_m$  danach sortiert sind. In unserem Beispiel muss beim Eintritt von Finnland in die Staatengemeinschaft das eine Element von  $t_0$  zusammen mit dem neuen Element in  $t_1$  sortiert gespeichert werden, so wie *Bild 4* zeigt.

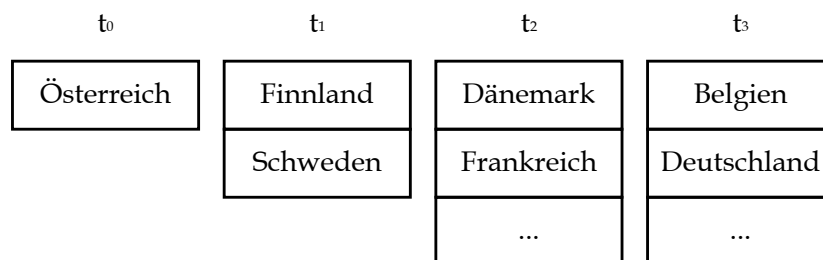


**Bild 4:** *PS\_ARRAY* aus *Bild 3* um Finnland erweitert ( $t_2$  und  $t_3$  unverändert).

Für die Sortierung der Teilfelder  $t_0$  bis  $t_m$  und des neuen Elementes nützen wir aus, dass zwei gleich große, sortierte Felder in linearer Zeit zu einem sortierten, doppelt so großen Feld zusammengefügt werden können, indem jeweils das kleinere der beiden ersten Elemente der ursprünglichen Felder an die nächste Position des neuen Feldes kopiert wird. Hat das große Feld  $n$  Elemente, so sind hierfür  $n$  Kopien und maximal  $n-1$  Vergleiche nötig.

Auf diese Weise können wir nun das neue Element und die ungültig werdenden Teilfelder sortieren. Um unnötiges Kopieren zu vermeiden, füllen wir  $t_{m+1}$  von hinten: Zunächst wird das neue Element an das Ende von  $t_{m+1}$  kopiert. Dann wird dieses zusammen mit  $t_0$  sortiert und das Ergebnis an die letzten zwei Elemente von  $t_{m+1}$  gespeichert. Nun wird dies für alle  $t_i$  mit  $0 < i \leq m$  wiederholt: Es wird mit steigendem  $i$  aus  $t_i$  und den hintersten  $2^i$  Elementen von  $t_{m+1}$  ein sortiertes Feld mit  $2^{i+1}$  Elementen erzeugt und im hinteren Bereich von  $t_{m+1}$  gespeichert. Für  $i=m$  sind alle Elemente von  $t_{m+1}$  belegt und sortiert, dafür wurden  $2^{m+2}-1$  Elemente kopiert (dies ergibt sich aus dem zunächst eingefügten neuen Element, das mit  $t_0$  dann ein Feld mit zwei Elementen, mit  $t_1$  vier Elementen, usw. ergibt, es werden also  $1+2+4+\dots+t_{m+1} = 2^{m+2}-1$  Elemente kopiert).

In dem Beispiel ist die nächste neue Mitgliedschaft von Österreich wieder ohne Sortieraufwand möglich, das Ergebnis zeigt *Bild 5*. Würde die Gemeinschaft nun noch um Norwegen erweitert, ist der eben beschriebene Sortieraufwand der Felder  $t_0$  bis  $t_3$  für  $m=3$  nötig, die sortierten Staaten landeten allesamt in  $t_4$ , wie in *Bild 6* gezeigt.



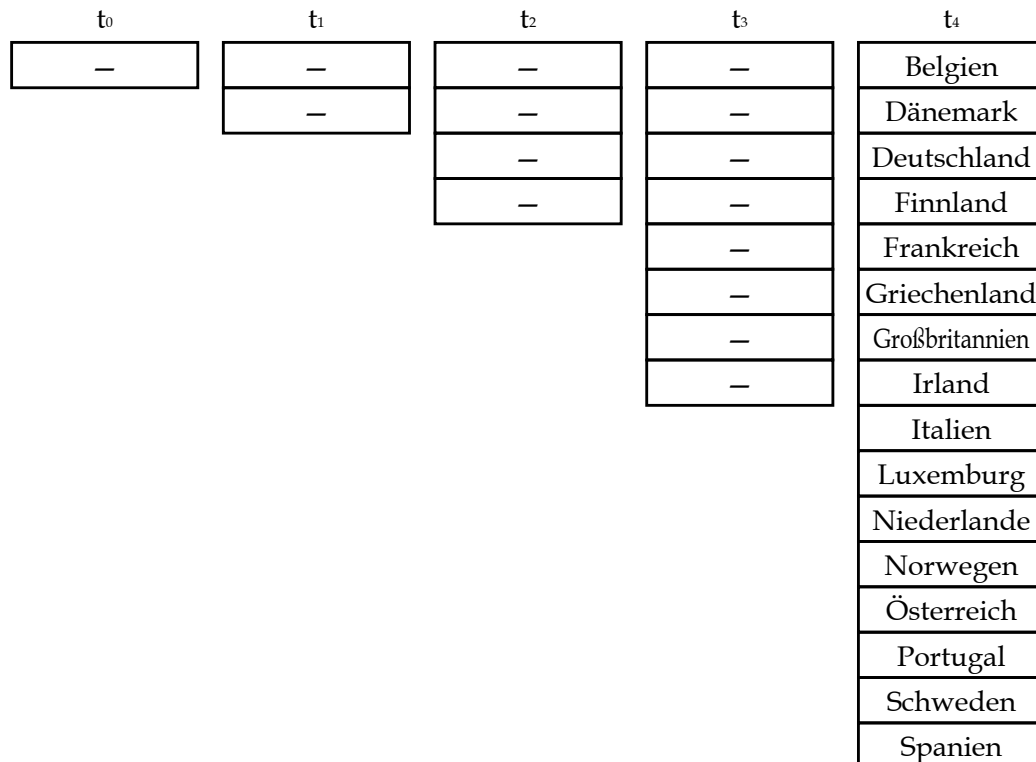
**Bild 5:** *PS\_ARRAY* aus *Bild 4* um Österreich erweitert ( $t_2$  und  $t_3$  unverändert).

### 2.3.4 Aufwand des Einfügens im *PS\_ARRAY*

Der Aufwand hängt von der Zahl und der Größe der zu sortierenden Teilfelder ab. Wie eben gezeigt, kann das Zusammenfügen der Teilfelder zu einem neuen Teilfeld  $t_{m+1}$  mit  $2^{m+2}-1$  Kopien erfolgen. Das Teilfeld  $t_m$  wird bei jedem  $2^{m+1}$ -ten Einfügen eines Elementes gefüllt, so dass sich der durchschnittliche Aufwand bei  $e$  Elementen zu

$$\begin{aligned}
\text{Aufwand}(e) &= \sum_{m=0..log(e)} (2^{m+2}-1)/(2^{m+1}) \\
&\leq \sum_{m=0..log(e)} 2 \\
&= 2 \cdot (\log(e)+1) \\
&\in O(\log(e))
\end{aligned}$$

ergibt. Der durchschnittliche Aufwand ist also logarithmisch, der Aufwand zur Erzeugung der gesamten Struktur liegt in  $O(e \cdot \log(e))$  und entspricht daher gewöhnlichen Sortierverfahren.



**Bild 6:** Wäre auch Norwegen dabei, so wäre dies das *PS\_ARRAY* der EU-Staaten

### 2.3.5 Finden von Elementen

Um in einem *PS\_ARRAY* ein Element zu finden, beginnt man am besten mit einer binären Suche im größten gültigen Teilfeld. Ist die Suche erfolglos, so fährt man mit dem nächstkleineren Teilfeld fort. Unter der Annahme, dass nach allen Elementen gleich häufig gesucht wird, kann der durchschnittliche Aufwand für eine erfolgreiche Suche wie folgt bestimmt werden:

Das größte Teilfeld hat zumindest  $e/2$  Elemente, höchstens  $e$  Elemente. Die Suche ist also mit mindestens 50-prozentiger Wahrscheinlichkeit erfolgreich und hat den Aufwand  $\log(e/2)$ . War die Suche bisher erfolglos, so gilt für den Rest der Suche der Aufwand der Suche in einem *PS\_ARRAY* ohne das untersuchte größte Teilfeld. Dieses *PS\_ARRAY* hat höchstens  $e/2$  Elemente, also

$$\text{Aufwand}_{\text{find\_success}}(e) \leq \log(e) + 50\% \cdot \text{Aufwand}_{\text{find\_success}}(e/2)$$

Der Suchaufwand für die trivialen Fälle ist natürlich

$$\text{Aufwand}_{\text{find\_success}}(e \leq 1) = 1$$

zusammen mit der Abschätzung  $\log(e/2^i) \leq \log(e)$  ergibt sich dann

$$\begin{aligned} \text{Aufwand}_{\text{find\_success}}(e) &\leq \log(e) + 50\% \cdot \text{Aufwand}_{\text{find\_success}}(e/2) \\ &\leq \log(e) \cdot (1 + 1/2 + 1/4 + \dots) \\ &\leq 2 \cdot \log(e) \\ &\in O(\log(e)) \end{aligned}$$

Der Aufwand für eine erfolgreiche Suche ist somit im Mittel logarithmisch in der Anzahl der Elemente. Für den Fall, dass ein Objekt nicht gefunden wird, müssen alle Teilfelder durchsucht werden. Dies ist auch der schlechteste Fall bei der erfolgreichen Suche. Hier ist der Aufwand

$$\begin{aligned} \text{Aufwand}_{\text{find\_failure}}(e) &\leq \log(e) + \text{Aufwand}_{\text{find\_failure}}(e/2) \\ &\leq \log(e) + \log(e/2) + \log(e/4) + \dots + 0 \\ &\leq \log(e) + \log(e) + \log(e) + \dots + \log(e) \quad (\log(e) \text{ Summanden}) \\ &= \log(e) \cdot \log(e) \\ &\in O(\log^2(e)) \end{aligned}$$

denn es müssen nun maximal  $\log(e)$  Teilfelder durchsucht werden, die jeweils höchstens  $e$  Elemente enthalten. Der Aufwand für die Suche eines Elementes, das in der Datenstruktur nicht enthalten ist, ist für *PS\_ARRAYs* also möglicherweise schlechter als bei der Verwendung von Binärbäumen oder komplett sortierten Feldern.

### 2.3.6 Sortieren des Feldes

Sobald keine neuen Elemente dem *PS\_ARRAY* zugefügt werden, dieses aber weiterhin häufig für die Suche nach Elementen benötigt wird, so sollte es vollständig sortiert werden, damit das Finden von Elementen effizienter wird. Das Sortieren des gesamten Feldes kann in linearer Zeit geschehen, wenn die Teilfelder, begonnen mit dem kleinsten Teilfeld, nacheinander zusammengefügt werden. Sind in einem *PS\_ARRAY* alle Teilfelder bis auf das größte zusammengefügt, so ist der Aufwand für das letzte Zusammenfügen  $e$ . Da das letzte Teilfeld mindestens  $e/2$  Elemente besitzt, kann der Aufwand der Sortierung der kleineren Teilfelder mit  $\text{Aufwand}_{\text{sort}}(e/2)$  abgeschätzt werden:

$$\text{Aufwand}_{\text{sort}}(e) \leq e + \text{Aufwand}_{\text{sort}}(e/2)$$

Mit dem trivialen Aufwand für ein einziges Element

$$\text{Aufwand}_{\text{sort}}(1) = 1$$

ergibt sich der Gesamtaufwand zu

$$\begin{aligned} \text{Aufwand}_{\text{sort}}(e) &\leq e + e/2 + e/4 + \dots + 1 \\ &\leq 2 \cdot e \\ &\in O(e) \end{aligned}$$

und ist linear in der Anzahl der Elemente.

## 2.4 Zeichenketten

Ein Compiler muss sehr viel mit Zeichenketten arbeiten. Diese kommen direkt im Quelltext als Bezeichnernamen vor, werden aber auch in den erzeugten Objektdateien für Symbole benötigt. Sehr häufig werden sie in den oben beschriebenen Datenstrukturen als Schlüssel verwendet, es ist daher sehr wichtig, dass Zeichenketten schnell miteinander verglichen werden können.

Damit hier die höchste Effizienz erreicht wird, wird jeder Zeichenkette eindeutig eine *INTEGER*-Zahl zugeordnet, der sogenannten *id* der Zeichenkette. Alle bei der Übersetzung vorkommenden Zeichenketten werden global in einem Feld gespeichert, mit dem *id* als Index kann über dieses Feld direkt der zugehörige String bestimmt werden. Zudem werden die Zeichenketten sortiert in einem *PS\_ARRAY* gehalten, so dass in logarithmischer Zeit zu jeder Zeichenkette der passende *id* ermittelt werden kann.

Damit das Arbeiten mit Zeichenketten dennoch so einfach wie möglich bleibt, wird die Klasse *ANY* um ein *once*-Feature *strings* erweitert, mit dem leicht *ids* in Zeichenketten und umgekehrt transformiert werden können: *strings # text* liefert den *id* der Zeichenkette *text*, während *strings @ id* den *id* zugeordneten String liefert.

In allen Datenstrukturen, in denen Zeichenketten vorkommen, werden jetzt lediglich die *Integer*-Werte der *ids* gespeichert, und sortierte Datenstrukturen werden nicht alphabetisch, sondern nach den Werten der *ids* sortiert. Damit kommen Vergleiche von Zeichenketten ohne einen Funktionsaufruf aus.

Ein weiterer Vorteil der Verwendung von *ids* liegt im Speicherbedarf: Mehrere gleiche Zeichenketten werden nun nur noch einmal gespeichert. Bei einer Testcompilation des Compilerquelltextes wurden über 30 000 Zeichenketten benutzt, von denen jedoch lediglich etwa 5 000 unterschiedlich waren.

## Literatur

- [Meyer88] Bertrand Meyer: „Object-oriented Software Construction“, Prentice-Hall Int., Hertfordshire, 1988
- [Sedgewick91] Robert Sedgewick: „Algorithms“, Addison-Wesley, Reading, Mass., 1991

The first letter was a 'w', the second an 'e'. Then there was a gap. An 'a' followed, then a 'p', an 'o' and an 'l'. Marvin paused for a rest. After a few moments they resumed and let him see the 'o', the 'g', the 'i', the 's' and the 'e'. The next two words were 'for' and 'the'. The last one was a long one, and Marvin needed another rest before he could tackle it. It started with an 'i', then 'n' then a 'c'. Next came an 'o' and an 'n', followed by a 'v', an 'e', another 'n' and an 'i'. After a final pause, Marvin gathered his strength for the last stretch. He read the 'e', the 'n', the 'c' and at last the final 'e', and staggered back into their arms.  
– So long, and Thanks for all the Fish,  
Douglas Adams

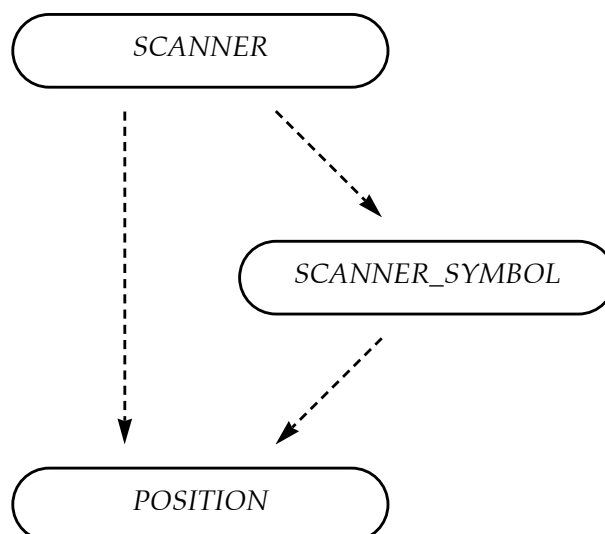
---

### 3. Der Scanner

Der Scanner ist derjenige Teil eines Compilers, der direkt mit dem Quelltext arbeitet, so wie er vom Programmierer eingegeben wurde. Die Hauptaufgabe des Scanners besteht in der Bestimmung der Terminalsymbole, wie sie vom Parser leicht verarbeitet werden können. Da der Scanner als einziger Teil des Systems direkten Zugriff auf den Quelltext hat, ist es auch seine Aufgabe, Referenzen auf Positionen im Quelltext zu verwalten. Diese werden nicht nur für Fehlermeldungen benötigt, sondern sind auch für Aufgaben wie dem Source-Level-Debugging oder Profiling nötig.

#### 3.1 Die Klassen des Scanners

Der Scanner läuft als eigene Phase der Übersetzung und zerlegt den gesamten Quelltext in Terminalsymbole, bevor diese vom Parser weiterverarbeitet werden. Die in *Bild 1*



**Bild 1:** Klassen des Scanners

gezeigten Klassen sind Teile des Scanners, wobei die Routinen der Klasse *SCANNER* den größten Teil der Aufgaben des Scanners übernehmen.

### 3.1.1 Die Klasse *POSITION*

Objekte der Klasse *POSITION* speichern Positionen im Quelltext. Sie enthalten Attribute für Zeile und Spalte im Text und auch einen Verweis auf den Namen der Quelltextdatei. Diese Objekte sollen innerhalb aller Teile des Compilers verwendet werden, um Verweise auf Quelltextpositionen weiterzugeben und Meldungen zu erzeugen, die auf diese Positionen verweisen.

Auch wenn meist die betroffene Klasse und somit der betroffene Quelltext aus dem Kontext klar ist, enthält jede Position eine Referenz auf den Quelltextnamen. Dies erhöht den Speicherbedarf etwas, vereinfacht jedoch den Rest des Compilers und vermeidet Fehler, da jedes *POSITION*-Objekt alle nötigen Informationen einer Quelltextposition enthält.

### 3.1.2 Die Klasse *SCANNER\_SYMBOL*

Der Scanner erzeugt ein großes Feld von Objekten dieser Klasse. Die Scannersymbole bestehen dabei meist nur aus einem *INTEGER*-Wert, ein *unique*-Wert, der dem Terminalsymbol (*Token*) entspricht. Diese Werte sind in der Klasse *SCANNER* definiert. Einige Terminalsymbole benötigen zusätzliche Informationen, beispielsweise den Wert einer numerischen Konstanten oder den Namen eines Bezeichners. Für diese *Tokens* enthält *SCANNER\_SYMBOL* ein zusätzliches Attribute *special*, das den nötigen Wert entweder direkt enthält oder ein Index in eines der Felder *strings* oder *reals* ist, die in *SCANNER* definiert sind.

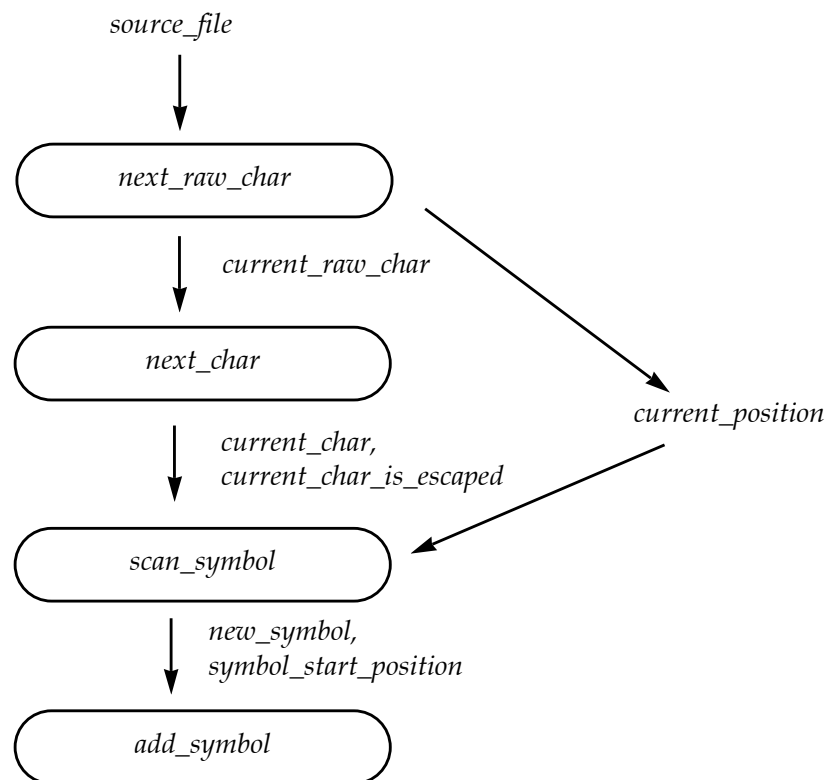
### 3.1.3 Die Klasse *SCANNER*

Der Zeichenstrom, wie er von der Eingabedatei kommt, wird in verschiedenen Stufen vom Scanner untersucht. Die einzelnen Abschnitte werden von den in *Bild 2* gezeigten Routinen bearbeitet.

Es wird zunächst von *next\_raw\_char* die Eingabe zeichenweise eingelesen und dabei die Position im Quelltext bestimmt. Die Zeichen werden dann von der Routine *next\_char* nach Sonderzeichen untersucht, die mit der %-Notation im Quelltext enthalten sind. In Eiffel können an jeder Stelle im Quelltext mit Hilfe des Prozentzeichens Sonderzeichen eingefügt werden, so dass dies am besten ausgefiltert wird, noch bevor der Scanner mit der Zerlegung in die Terminalsymbole des Parsers beginnt. Innerhalb von Zeichen- und Zeichenkettenkonstanten ist die Unterscheidung von gewöhnlichen Zeichen und solchen, die über die %-Notation entstanden sind, wichtig. Daher gibt *next\_char* nicht nur das aktuelle Zeichen über *current\_char* weiter, sondern zeigt mittels *current\_char\_is\_escaped* zusätzlich an, ob dieses über die %-Notation entstanden ist.

Nun endlich werden von *scan\_symbol* die Zeichen in die entsprechenden Terminalsymbole zerlegt. Für komplexere Terminalsymbole ruft *scan\_symbol* noch Zwischenroutinen auf, die diese behandeln. Einfache Terminalsymbole werden direkt mittels *add\_symbol* in die Symbolliste eingetragen.

Die meisten Terminalsymbole werden am ersten oder an den ersten beiden Zeichen erkannt. Besondere Beachtung muss einigen Terminalsymbolen gewidmet werden:



**Bild 2:** Entstehung der Scannersymbole aus den Quelltextzeichen

### 3.1.3.1 Die Terminalsymbole *and then* und *or else*

Diese beiden Terminalsymbole werden zunächst falsch als *and* bzw. *or* erkannt. Bevor der Scanner eines der Symbole *then* oder *else* in die Symbolliste einträgt, wird dann jedoch geprüft, ob diese zusammen mit dem zuletzt erkannten Symbol *and then* bzw. *or else* ergeben. In diesem Fall wird das zuletzt erkannte Symbol entsprechend geändert. Die Erkennung dieser zusammengesetzten Operatoren wird bereits vom Scanner erledigt, da so der Operator-Präzedenz-Parser für Ausdrücke vereinfacht wird.

### 3.1.3.2 INTEGER Konstante gefolgt von double dot

Ein weiteres Problem ergibt sich, wenn einer *INTEGER*-Konstanten direkt zwei Punkte („..“) folgen, wie dies beispielsweise in einer *inspect*-Anweisung vorkommen kann. Hier wird nun zunächst eine *REAL*-Konstante erkannt. Erst wenn der zweite Punkt gelesen wird, ist klar, dass es eine Konstante gefolgt von zwei Punkten ist. Hier erzeugt der Scanner erst dann die beiden korrekten Symbole.

### 3.1.3.3 Bezeichner und Schlüsselwörter

Schlüsselwörter werden nicht wie die anderen Symbole direkt erkannt, sondern zunächst als Bezeichner betrachtet. Bevor nun ein Bezeichnersymbol erzeugt wird, wird geprüft, ob es möglicherweise ein Schlüsselwort ist. Dazu wird es mittels binärer Suche mit den Einträgen in einer sortierten Liste der Schlüsselwörter verglichen.

Auf diese Weise wird im Vergleich zum direkten Erkennen von Schlüsselwörtern mit einem endlichen Automaten die Zahl der Zustände drastisch verringert.

## 3.2 Parserunterstützung

Eine Reihe an Routinen der Klasse *SCANNER* erledigen Aufgaben, die eigentlich dem Parser zuzuordnen sind, sich jedoch nur auf das aktuelle *Tokens* beziehen und so logisch besser dem Scannerobjekt zugeordnet werden.

### 3.2.1 Symbole holen

Die am häufigsten vom Parser benutzten Features des Scanners sind das Attribut *current\_symbol* und die Routine *next\_symbol*, die das aktuelle Symbol liefern bzw. dieses auf das nächstfolgende Symbol setzen.

Für die Fälle, in denen die Eiffel-Grammatik nicht LL(1) ist, gibt es noch das Attribut *current\_symbol\_index*, welches die Nummer des aktuellen Symbols angibt, und die Routine *reset\_current\_symbol*, mit der auf den Index eines früher gelesenen Symbols zurückgesprungen werden kann.

### 3.2.2 Überprüfen bestimmter Symbole

Oft erwartet der Parser ein ganz bestimmtes Symbol, etwa das Schlüsselwort *then* oder eine schließende Klammer. Jedes andere Symbol bedeutet, dass der Quelltext fehlerhaft ist. Um solche erwarteten Symbole zu prüfen, stellt der Scanner eine Reihe an Routinen mit den Namen *check\_...* gefolgt von dem Namen des erwarteten Symbols zur Verfügung.

Diese Routinen versuchen im Fehlerfall möglichst sinnvoll zu reagieren. Beispielsweise wird bei einem fehlenden Komma nicht nur eine Fehlermeldung erzeugt. Ist das aktuelle Symbol etwa ein Strichpunkt, so wird es zudem überlesen, ist es dagegen ein Schlüsselwort, so bleibt es im Symbolstrom.

Auf diese Weise ist ein großer Teil der Fehlerbehandlung nicht verteilt im Parsercode, sondern komprimiert in diesen Prüfungsroutinen.

### 3.2.3 Besondere Attribute von Scannersymbolen

Eine Reihe an Funktionen des Scanners, deren Bezeichner mit *get\_...* beginnen, liefern die mit einigen *Tokens* verbundenen besonderen Daten, wie die Werte von *INTEGER*-Konstanten oder die Namen von Bezeichnern.

### 3.2.4 First-Mengen

Für den LL(1)-Parser werden an vielen Stellen Tests benötigt, die feststellen, ob das gefundene Symbol in den *First*-Menge einer bestimmten Produktion der Grammatik liegt [Aho86]. Dafür stellt die Klasse *SCANNER* eine Reihe an Routinen *first\_of\_...* bereit, deren Name jeweils um den Namen der betroffenen Produktion erweitert ist und die als boolesches Ergebnis liefern, ob das aktuelle Symbol in der jeweiligen *First*-Menge liegt.

## Literatur

- [Aho86] Alfred V. Aho, R. Sethi, J.D. Ullman: „COMPILERS: Principles, Techniques and Tools“, Bell Telephone Laboratories, 1986

---

## 4. Der Parser

Die Aufgabe des Parser besteht in der Erzeugung eines abstrakten Syntaxbaums aus der vom Scanner gelieferten Folge von Symbolen.

### 4.1 Parsingverfahren

Wie in [Blach92] gezeigt, lässt sich die Eiffel-Grammatik mit wenigen Ausnahmen leicht in eine LL(1)-Grammatik umwandeln. Die einfache Grammatik rechtfertigt nicht den Aufwand, der mit dem Einsatz eines leistungsfähigeren Parsingverfahrens und eines Parsergenerators verbunden ist.

#### 4.1.1 Arbeitsweise der Parser

Es wird nach dem Verfahren des rekursiven Abstiegs [Wirth86] geparkt. Dabei werden direkt die Objekte des abstrakten Syntaxbaums erzeugt. Diese Objekte besitzen jeweils eine Erzeugungsroutine namens *parse*, die die Attribute des Objekts mit den Daten aus dem Quelltext belegt.

Produktionen der Grammatik, die verschiedene Alternativen besitzen, werden meist durch eine abstrakte Klasse realisiert, die für jede Alternative einen konkreten Nachfolger besitzt, so gibt es beispielsweise die abstrakte Klasse *INSTRUCTION* mit konkreten Nachfolgern wie *ASSIGNMENT*. Um diese Produktionen zu parsen, werden spezielle Routinen benutzt, die das jeweilige Objekt der im untersuchten Quelltext gefundenen Alternative erzeugt. Dies wird für Anweisungen durch die Routine *parse\_instruction* der Klasse *PARSE\_INSTRUCTION* getan.

#### 4.1.2 Operator-Präzedenz-Parser für Ausdrücke

Die große Zahl an unterschiedlichen Präzedenzen für binäre Operatoren lässt die Benutzung eines rekursiven Abstiegsparser für deren Untersuchung nicht sinnvoll er-

scheinen. Stattdessen werden binäre Ausdrücke durch einen einfachen Operator-Präzedenz-Parser untersucht [Aho86].

Die in [Meyer92] festgelegten Präzedenzen für binäre Operatoren sind alle geringer als diejenigen für unäre Operatoren. Dies erlaubt es, für die unären Operatoren weiterhin den rekursiven Abstiegsparser zu verwenden. Die Eiffel-Grammatik wurde so verändert, dass der Operator-Präzedenzparser lediglich die Produktion

*Expression* => {*Anything\_but\_binary\_expression Binary\_Operator ...*}+.

untersuchen muss (die hier verwendete Notation für die Produktionsregeln ist die aus [Meyer92]). Der Parser hierfür sieht wie folgt aus:

```
op_prec_expression(s: SCANNER; last_binary: INTEGER) is
  local
    left,right: EXPRESSION;
    op_name: STRING;
    op: INTEGER;
  do
    from
      parse_anything_but_binary_expression(s);
    until
      not is_binary_operator(s) or else
      higher_precedence(last_binary,s.current_symbol.type)
    loop
      left := expression;
      op := s.current_symbol.type;
      op_name := binary_operator_name(s);
      s.next_symbol;
      op_prec_expression(s,op);
      right := expression;
      !CALL!expression.make_binary(left,op_name,right);
    end;
  ensure
    expression /= Void;
  end; -- op_prec_expression
```

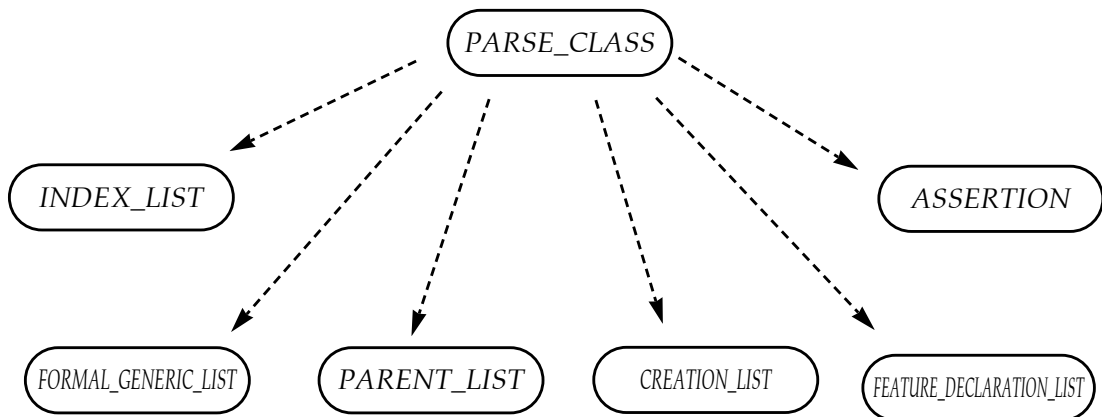
*op\_prec\_expression* analysiert einen Ausdruck, bis entweder kein binärer Operator mehr gefunden wird (*not is\_binary\_expression(s)*), oder bis der nächste binäre Operator eine geringere Präzedenz als der zuletzt vor diesem Ausdruck gefundene Operator hat (*higher\_precedence(last\_binary,s.current\_symbol.type)*).

Die Funktion *higher\_precedence(l,r)* vergleicht die Präzedenzen der beiden gegebenen Operatoren *l* und *r*. Sie liefert *true*, wenn der erste Operator *l* eine höhere Präzedenz hat, falls er links vom zweiten Operator *r* steht. Dies kann bei unterschiedlichen Operatoren einfach durch Vergleich der Einträge in der Präzedenztabelle aus [Meyer92] geschehen. Haben die Operatoren gleiche Präzedenz, so muss ihre Assoziativität beachtet werden. Alle Operatoren bis auf „^“ sind linksassoziativ, so dass hier das Ergebnis nur dann *false* wird, wenn beide Parameter „^“ sind.

## 4.2 Der abstrakte Syntaxbaum

### 4.2.1 Struktur des abstrakten Syntaxbaumes

In weiten Bereichen entspricht der abstrakte Syntaxbaum dem durch die Grammatik vorgegebenen Parsebaum. *Bild 1* zeigt dies exemplarisch für die Wurzelklasse *PARSE\_CLASS* des Baumes.

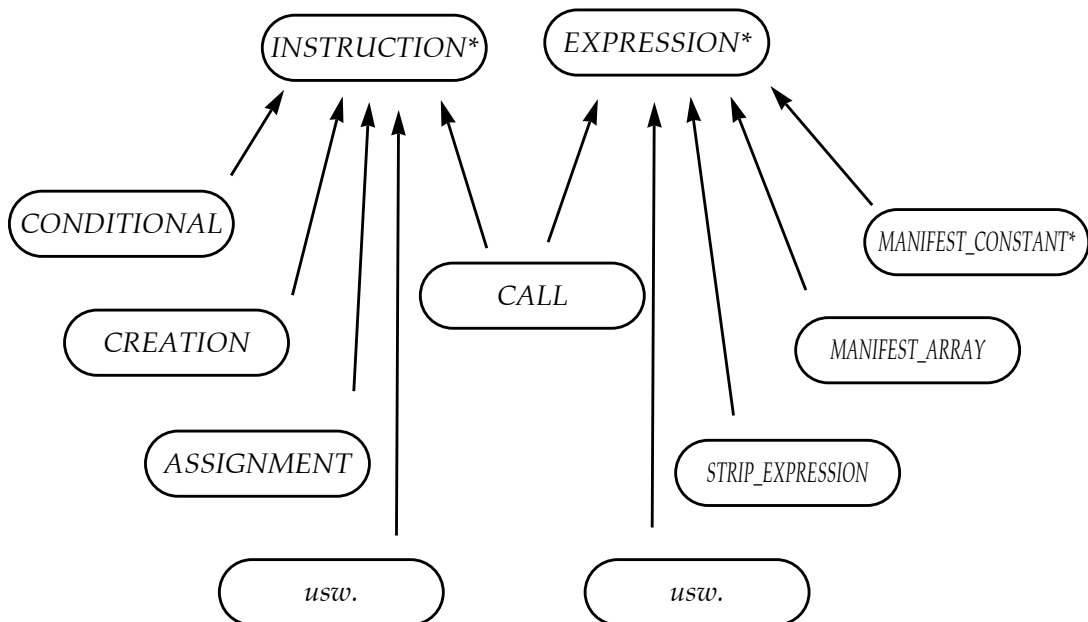


**Bild 1:** Struktur der Hauptklasse des Abstrakten Syntaxbaums

#### 4.2.2 Abstrakte Klassen und Vererbung im Syntaxbaum

Für Konstrukte, die aus mehreren Alternativen bestehen, wie Ausdrücke, Anweisungen oder Features, werden abstrakte Klassen *EXPRESSION*, *INSTRUCTION* oder *FEATURE\_VALUE* definiert, die für die verschiedenen Alternativen konkretisiert werden, also beispielsweise als *CALL*, *CONDITIONAL*, *ASSIGNMENT* usw. als konkrete Nachfolger von *INSTRUCTION*.

Eine Sonderstellung nimmt hierbei das Konstrukt *CALL* ein, das sowohl als Alternative für einen Ausdruck als auch für eine Anweisung verwendet wird. Zudem werden für Ausdrücke der Formen *UNARY\_EXPRESSION* und *BINARY\_EXPRESSION* im abstrakten Syntaxbaum *CALL*-Objekte erzeugt: Im abstrakten Syntaxbaum werden Aufrufe



**Bild 2:** Vererbungsstruktur von *INSTRUCTION* und *EXPRESSION*

fe in Punktnotation oder mittels unärer oder binärer Operatoren nicht mehr unterschieden und können von den folgenden Teilen des Compilers gleich behandelt werden. Wie *Bild 2* zeigt, ist die Klasse *CALL* sowohl als Nachfolger von *EXPRESSION* als auch von *INSTRUCTION* definiert, so wie dies auch von der Grammatik vorgeschlagen wird. Auf diese Weise können bei der Codeerzeugung die möglichen Vorkommen von Aufrufen gleich von derselben Routine behandelt werden. Es ist lediglich eine Unterscheidung nötig, ob ein Ergebnis zurückgeliefert werden muss oder nicht.

## Literatur

- [Aho86] Alfred V. Aho, R. Sethi, J.D. Ullman: „COMPILER Principles, Techniques and Tools“, Bell Telephone Laboratories, 1986
- [Blach92] Rüdiger Blach: „Ein LL(1)-Parser für Eiffel“, German Chapter of the ACM, Berichte 35: Eiffel, Hans-Jürgen Hoffmann (Hrsg.), Fachtagung am 25. und 26. Mai 1992 in Darmstadt.
- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertfordshire, 1992
- [Wirth86] Niklaus Wirth: „Compilerbau“, Teubner Studentbücher Informatik, Teubner Verlag, Stuttgart, 1986

---

## 5. Validity-Überprüfung

Bevor mit der Codeerzeugung begonnen werden kann, muss geprüft werden, ob die Quelltexte ein gültiges System bilden. Dies geschieht in Eiffel durch die Überprüfung der in [Meyer92] beschriebenen *Validity-Constraints*. Diese beschreiben u. a. die erlaubten Fälle der Vererbung und Redeklaration, die korrekte Verwendung von Typen und die Korrektheit von Deklarationen, Ausdrücken und Anweisungen.

Die in [Meyer92] definierten *Validity-Constraints* sind so gehalten, dass sie nicht von den aktuellen generischen Parametern der Klasse abhängen, deren Gültigkeit geprüft wird. Daher ist es ausreichend, für jede Klasse die Prüfung nur einmal durchzuführen, selbst wenn verschiedene generische Derivationen der Klasse im compilierten Eiffel-System verwendet werden.

Jeder vom Parser eingelesenen Klasse wird hierfür ein Objekt der Klasse *CLASS\_INTERFACE* zugeordnet. Dieses Objekt koordiniert die Überprüfung der *Validity-Constraints* dieser Klasse und liefert gleichzeitig die Schnittstelle für den Zugriff auf diese Klasse, die für die Überprüfung der Klienten dieser Klasse nötig ist.

### 5.1 Ausgabe der Fehlermeldungen

Der Compiler wäre wenig benutzerfreundlich, wenn er pro Compileraufruf nur einen einzigen Fehler im Quelltext finden könnte. Soll er jedoch mehrere Fehler anzeigen, so muss der Compiler auch bei fehlerhafter Eingabe sinnvoll weiterarbeiten können. Es ist kaum zu verhindern, dass der Compiler auch Folgefehler anzeigt, jedoch sollte der Anteil der Folgefehler nicht übermäßig hoch sein.

Um diese Ziele zu erreichen, werden pro Übersetzung nur die in einer Klasse gefundenen Fehler angezeigt, nämlich die Fehler der Klasse, in der der erste Fehler gefunden wurde. Fehler in Klienten dieser Klasse sind dann sehr wahrscheinlich nur Folge der Benutzung der fehlerhaften Klasse und werden nicht angezeigt.

Da manche Fehler im Quelltext das korrekte Arbeiten des Compilers unmöglich machen können, wird die Übersetzung an bestimmten Stellen gestoppt, falls Fehler aufgetreten sind. Dies geschieht jeweils nach dem Scannen, Parsen, der Prüfung der Vererbung auf Zyklen und dem Abschluss der Validity-Überprüfung einer Klasse.

## 5.2 Bestimmung der Schnittstellen der Klassen

Die Schnittstelle einer Klasse besteht aus der Liste an Features, die diese Klasse ihren Klienten zur Verfügung stellt, und einer Liste aller Vorgängerklassen dieser Klasse. Diese Liste ist für die Überprüfung der *Conformance* zwischen Typen nötig. Jede Vorgängerklasse wird durch ein Objekt der Klasse *ANCESTOR* beschrieben. Ein *ANCESTOR*-Objekt repräsentiert eine Klasse, deren generische Parameter entweder direkt angegeben sind, oder von den aktuellen generischen Parametern der Klasse abhängen, deren Vorgänger beschrieben wird.

Eiffel-Klassen können zyklische Klienten voneinander sein, und dies ist auch sehr häufig der Fall: Beispielsweise liefert die Funktion *count* der Standardklasse *STRING* einen *INTEGER*-Wert während *out* von *INTEGER* ein *STRING*-Objekt erzeugt. Jede Klasse kann sogar ihr eigener Klient sein.

Bei der Prüfung der Gültigkeit einer Klasse darf der Compiler daher nicht verlangen, dass die Gültigkeit aller Klassen, deren Klient diese Klasse ist, vorher bestimmt werden können. Er muss jedoch auf die Schnittstellen dieser Klassen zugreifen können.

Um die Schnittstelle einer Klasse *C* zu bestimmen, werden zunächst die Schnittstellen aller Vorgängerklassen dieser Klasse benötigt. Das *CLASS\_INTERFACE* von *C* wird während der Bestimmung der Schnittstellen der beerbten Klassen mit einem booleschen Flag *getting\_inherited* markiert, um hierbei eine endlose Rekursion bei einer fehlerhaften zyklischen Vererbung zu verhindern.

Mit den Schnittstellen der Vorgängerklassen können nun die Mengen der Vorgängerklassen und der geerbten Features bestimmt werden. Die Vorgänger von *C* werden bestimmt aus den Listen der Vorgänger aller direkten Vaterklassen *P* von *C* (das sind diejenigen, die im Quelltext in der *Parent\_list* aufgelistet sind), indem jedes Vorkommen eines formalen generischen Parameter von *P* durch die im Quelltext von *C* angegeben aktuellen generischen Parameter ersetzt werden. Diese aktuellen generischen Parameter können wiederum von den formalen generischen Parametern von *C* abhängen. Zudem wird die Klasse *C* mit ihren formalen generischen Parametern selbst in die Liste ihrer Vorgängerklassen eingetragen (die Vorgängerbeziehung wurde in [Meyer92] reflexiv definiert). Vorgängerklassen, die bei diesem Algorithmus doppelt gefunden werden, werden dennoch nur einmal in die Liste der Vorgängerklassen eingetragen. Ein Beispiel soll dies veranschaulichen: Für die generische Klasse

```
class A[G]
end -- A
```

enthält die Menge der Vorgängerklassen außer *A[G]* nur den impliziten Vorgänger *ANY*, es ist also die Menge  $\{A[G], ANY\}$ . Für eine neue Klasse *B* mit dem Quelltext

```
class B[H,I]
inherit
    A[A[I]];
end -- B
```

wird die Vorgängermenge aus der Menge  $\{B[H, I]\}$  und den Vorgängern von  $A$  erzeugt, wobei der generische Parameter  $G$  durch den in der *inherit*-Anweisung angegebenen aktuellen generischen Parameter  $A[I]$  ersetzt wird. Die Vorgängermenge von  $B$  ergibt sich also zu  $\{B[H, I], A[A[I]], ANY\}$ . Schließlich können in einer Klasse wie

```
class C
inherit
    B[INTEGER, CHARACTER];
    A[BOOLEAN];
end -- C
```

auch die generischen Parameter der Vorgängerklassen echte Klassen sein, in diesem Fall wäre diese Menge  $\{C, B[INTEGER, CHARACTER], A[A[CHARACTER]], A[BOOLEAN], ANY\}$ .

Zur Bestimmung der Features einer Klasse  $C$  werden die Featurelisten der direkten Vorgänger von  $C$  genommen und, so wie im *Feature\_adaption* Teil des Quelltextes von  $C$  angegeben, umbenannt, als undefiniert, redefiniert oder selektiert markiert und mit einer neuen Klientenliste (*New\_exports*) versehen. Die unter demselben Namen geerbten Features werden zunächst mehrfach in die Liste der Features von  $C$  eingetragen, für jeden Eintrag wird ein Objekt der Klasse *FEATURE\_INTERFACE* erzeugt.

Alle gleichnamigen Einträge in der Menge der Features müssen durch *Sharing* oder *Joining* [Meyer92] zu einem Feature zusammenfallen. *Sharing* trifft immer dann zu, wenn mehrere Features das gleiche *Seed* und *Origin* besitzen, also aus derselben Klasse und derselben Deklaration stammen und ihre Signaturen, die durch unterschiedliche generische Derivate der Ursprungsklasse unterschiedlich sein können, zueinander passen.

*Joining* trifft für alle Paare von geerbten abstrakten Features zu, die in der neuen Klasse zu einem einzigen abstrakten Feature zusammenfallen.

Schließlich darf von den übrigen gleichnamigen Features jeweils nur eines konkret (*effective*) sein, alle gleichnamigen abstrakten (*deferred*) Features werden mit diesem verschmolzen.

Dies ergibt die Menge der geerbten Features (*inherited features*), in der jedes Feature einen eindeutigen Namen hat. Für übriggebliebene gleichnamige Features wird eine entsprechende Fehlermeldung erzeugt.

Zu dieser Menge werden nun alle in  $C$  neu eingeführten Features hinzugefügt. Hat eines dieser Features denselben Namen wie eines der geerbten Features, so muss es sich um eine Redeklaration handeln, deren Gültigkeit geprüft wird. Schließlich ergibt die resultierende Menge die Menge der Features von  $C$  und die Bestimmung der Schnittstelle ist somit abgeschlossen.

Mit Hilfe der eben bestimmten Schnittstellen der Klassen und den vom Parser erzeugten abstrakten Syntaxbäumen können die verbliebenen *Validity-Constraints* meist direkt geprüft werden. Die vielen zu überprüfenden Regeln werden daher hier nicht im Einzelnen erläutert.

## 5.3 Conformance

Eine bei der Prüfung der Gültigkeit einer Klasse oft benutzte Beziehung zwischen Typen ist die *Conformance*. Ein Typ bezeichnet dabei einen Ausdruck der Produktion *Type* der Eiffel Grammatik, also einen *Class\_type*, *Class\_type\_expanded*, *Anchored\_type*, *Formal\_generic\_name* oder *Bit\_type*. Typen werden im abstrakten Syntaxbaum durch Nachfolgerklassen der abstrakten Klasse *TYPE* dargestellt, es sind Objekte der Klassen *CLASS\_TYPE*, *ANCHORED*, *FORMAL\_GENERIC\_NAME* oder *BIT\_TYPE*.

Die Conformance wird von der Routine *is\_conforming\_to* der abstrakten Klasse *TYPE* getestet. Diese Routine benutzt zu ihrer Auswertung die abstrakten Routinen *conforms\_directly\_to* und *conforms\_recurisively*, die in den Klassen der verschiedenen Typen gemäß [Meyer92] implementiert sind. Bei der Auswertung der rekursiven Conformance wird u.U. *is\_conforming\_to* rekursiv aufgerufen, dann allerdings mit einem Typ einer Vaterklasse des gerade überprüften *Class\_type*. Da schon bei der Schnittstellenbestimmung rekursive Vererbung abgefangen wurde, ist sichergestellt, dass diese Rekursion endet.

## 5.4 System Validity

Durch die Möglichkeit der kovarianten Typanpassung bei der Vererbung in Eiffel kann es zu Typfehlern kommen, die bei der Betrachtung einzelner Klassen nicht erkannt werden können. Durch die Prüfung der sogenannten *System Validity*, wie sie in [Meyer92] beschrieben wird, kann für ein Eiffel-System sichergestellt werden, dass trotz kovarianter Typanpassung alle Aufrufe typkorrekt sind. Hierfür wird für alle Referenzvariablen und -attribute über eine systemweite Datenflussanalyse die Menge der Typen aller möglichen referenzierten Objekte bestimmt. Mit dieser Information kann dann für jeden Aufruf geprüft werden, ob er typkorrekt ist. Die Prüfung der *System Validity* wird von dieser Implementierung bisher nicht vorgenommen.

## Literatur

- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertforshire, 1992

*Redefinition implies that a qualified routine reference, say `h.tax`, may have many different interpretations depending on the value of `h` at run-time. A run-time search for the appropriate routine would carry a heavy performance penalty, especially with multiple inheritance. The Eiffel implementation solves this problem through a technique that always finds the appropriate routine in constant time, with only a small penalty over a standard procedure call, and no significant space overhead.*  
– Bertrand Meyer

---

## 6. Die Struktur des Zielcodes

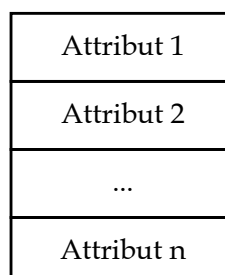
Hier sollen die verwendeten Datenstrukturen und Algorithmen beschrieben werden, die für eine effiziente Implementierung der Eigenschaften von Eiffel benutzt werden, insbesondere für die Vererbung, dynamische Bindung und Generizität.

### 6.1 Aufbau der Objekte

Objekte sind Konstrukte ähnlich den in Sprachen wie *Pascal* oder *C* bekannten *Records* oder *structs*. Allerdings erfordert die Vererbung und die dynamische Bindung, dass u.U. zusätzliche Informationen über den Typ und die Struktur eines Objekts in diesem gespeichert werden.

#### 6.1.1 Expandierte Objekte

Expandierte (*expanded*) Objekte, wie sie als Instanzen von expandierten Klassen oder durch explizite Deklaration definiert werden können, benötigen keine zur Laufzeit zu speichernden Typinformationen. Es ist bei Ihrer Verwendung der Typ jederzeit zur Übersetzungszeit bekannt und die Zuweisungssemantik stellt sicher, dass keine Referenzen auf expandierte Objekte erzeugt werden können. Expandierte Objekte werden innerhalb von anderen Objekten auf der Halde alloziert oder kommen als lokale Variablen oder Parameter von Routinen vor, die auf dem Stapel angelegt werden. *Bild 1* zeigt die Struktur expandierter Objekte.

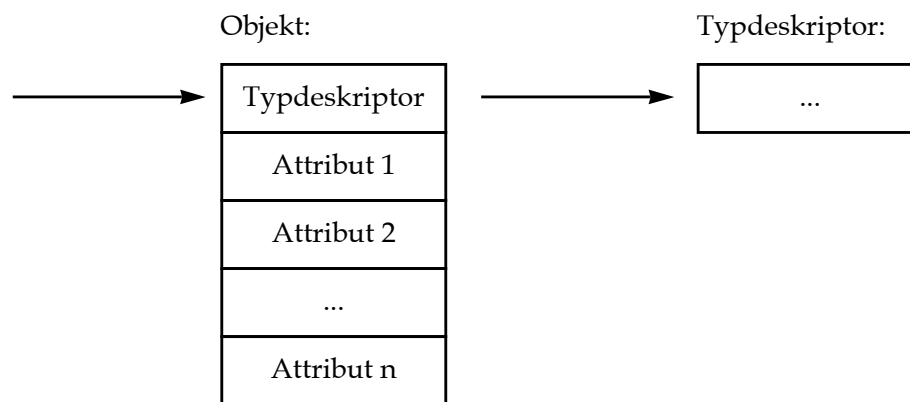


**Bild 1:** Struktur expandierter Objekte

### 6.1.2 Referenzobjekte

Alle Objekte, die nicht expandiert sind, werden auf der Halde gespeichert. Zugriffe erfolgen über Referenzen. Speziell können diese Referenzen auch in Attributen oder lokalen Variablen gespeichert werden, deren statischer Typ einer Vaterklasse des aktuellen (dynamischen) Typs des Objektes entspricht. Hier wird es also nötig, zusätzlich Informationen über den dynamischen Typ in jedem Objekt zu speichern.

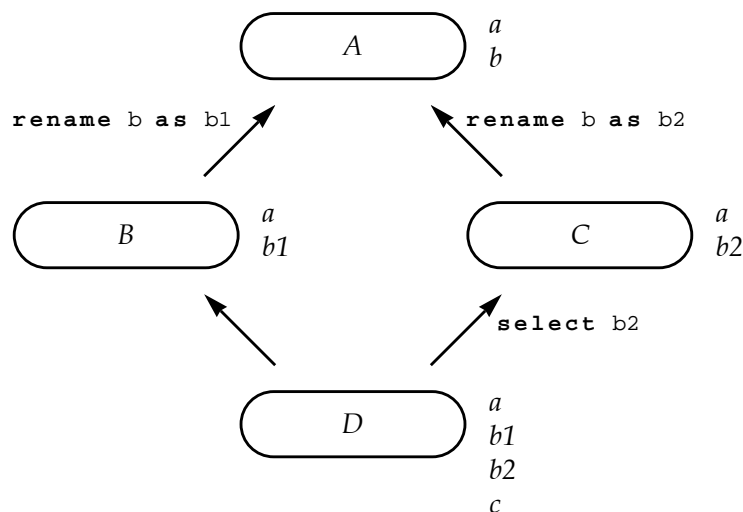
Da gewöhnlich mehrere Objekte desselben Typs auf der Halde erzeugt werden, ist es nicht sinnvoll, alle Informationen über den Typ in diesen Objekten zu speichern, sondern lediglich eine Referenz auf einen Typdeskriptor, der für alle Objekte desselben Typs gleich ist. *Bild 2* zeigt die Struktur eines Referenzobjekts.



**Bild 2:** Struktur eines Referenzobjekts

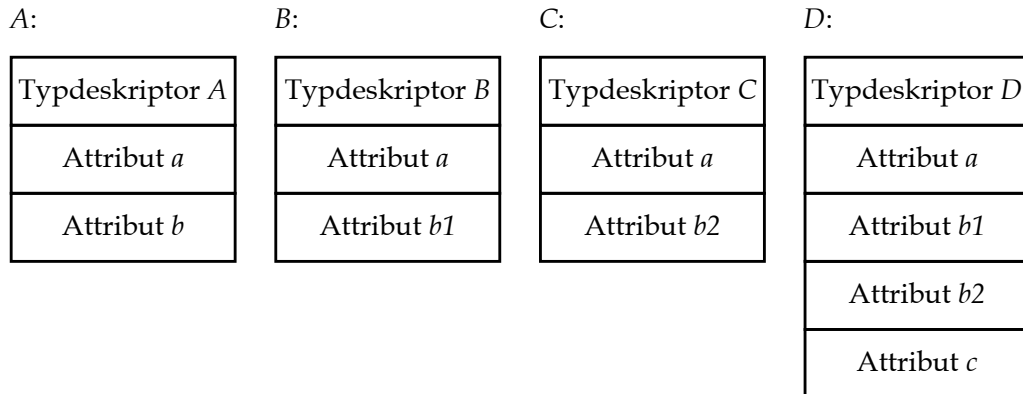
### 6.1.3 Objekte bei Mehrfachvererbung

Die flexible Vererbungsstruktur in Eiffel erschwert im Vergleich zu Sprachen wie C++ [Strou87] den Aufbau der Objekte bei Mehrfachvererbung. In C++ ist es möglich, bei Mehrfachvererbung die Objekte der Vaterklassen als gleich aufgebaute Teilobjekte des Nachfolgerobjektes zu implementieren. Dies ist in Eiffel nicht einfach möglich, wie das



**Bild 3:** Mehrfachvererbung von Attributen

folgende Beispiel veranschaulicht: *Bild 3* zeigt eine in Eiffel typische Vererbungsstruktur. Die angegebenen Features  $a$ ,  $b$ ,  $b1$ ,  $b2$  und  $c$  seien Attribute der jeweiligen Klassen. *Bild 4* zeigt den möglichen Aufbau der Eiffel-Objekte der Klassen  $A$ ,  $B$ ,  $C$  und  $D$  aus *Bild 3*. Die in Klasse  $D$  von den Klassen  $B$  und  $C$  geerbten Attribute können nicht so an-



**Bild 4:** Struktur der Objekte aus *Bild 3*

geordnet werden, wie sie in den Vaterklassen  $B$  und  $C$  vorkommen (es sei denn, das Attribut  $a$  würde in  $D$  verdoppelt, das würde jedoch unnötig Speicher verschwenden und den Aufwand beim Ändern des Attributes erhöhen). Der für den Zugriff auf die Attribute erzeugte Code muss die unterschiedlichen Objektstrukturen berücksichtigen. Dies wird im Abschnitt über Featureaufrufe weiter unten genauer erläutert.

#### 6.1.4 Objekte generischer Klassen

Der Aufbau und die Größe der Objekte einer generischen Klasse hängt von den aktuellen generischen Parametern der Klasse ab, falls die Klasse Attribute besitzt, deren Typ von einem formalen generischen Parameter abhängt. Objekte einer generischen Klasse mit unterschiedlichen aktuellen generischen Parametern haben daher im Allgemeinen einen unterschiedlichen Aufbau und unterschiedliche Typdeskriptoren.

Eine Ausnahme bildet der häufige Fall von Referenzen als aktuelle generische Parameter: Objekte einer generischen Klasse, deren aktuelle generische Parameter sich nur dadurch unterscheiden, dass sie unterschiedliche Referenztypen sind, sollen gleich aufgebaut sein und sich nur durch unterschiedliche Typdeskriptoren unterscheiden. Da in Eiffel-Systemen aktuelle generische Parameter sehr häufig Referenzen sind, kann so der häufigste Fall optimiert werden – es muss für diese generischen Derivate nicht mehrmals Code erzeugt werden.

#### 6.1.5 Position des Typdeskriptorzeigers

Es ist hierbei zu überlegen, den Typdeskriptorzeiger nicht direkt an der durch die Objektreferenz angesprochenen Adresse zu speichern, sondern beispielsweise an einem festen, negativen Offset. Beispielsweise in dem Wort direkt vor der Objektadresse. Dadurch würde ein Austausch mit Objekten anderer Programmiersprachen erleichtert, die auf Typinformationen verzichten. Allerdings ergeben sich dadurch möglicherweise Effizienzeinbußen durch den Prozessor-Cache: für Referenzen werden gewöhnlich nur Vielfache der Größe einer Seite im Cache verwendet. Bei kleinen Objekten führt somit ein Zugriff auf den Typdeskriptor, wie er bei einem Routinenaufwurf wie unten beschrieben nötig wird, zum Kopieren des ganzen Objektes in den Cache. Ist der Typdeskriptor

jedoch an einer negativen Adresse gespeichert, so wird lediglich die vor dem Objekt liegende Seite in den Cache geladen, der erste Zugriff auf ein Attribut stellt also möglicherweise einen zusätzlichen Cache-miss dar.

## 6.2 Featureaufrufe und dynamisches Binden

Das bei weitem häufigste Konstrukt in Ausdrücken und Anweisungen ist der Aufruf (*Call*). Dabei wird entweder auf eine lokale Variable, ein Attribut oder eine Routine zugegriffen. Es ist also besonders wichtig, Aufrufe so effizient wie möglich und speziell die am häufigsten vorkommenden Arten von Aufrufen auch am effizientesten zu machen.

Der Zugriff auf lokale Variablen und Parameter von Routinen, einschließlich des meist impliziten Zugriffs auf *Current*, ist am häufigsten und muss am effizientesten erledigt werden. Dies wird weiter unten genauer beschrieben.

Sehr häufig ist auch der Zugriff auf Attribute und Routinen des aktuellen Objekts *Current*. Auch wenn durch Mehrfachvererbung die Position der Attribute im aktuellen Objekt und die aufzurufende Routine nicht zur Compilationszeit feststehen, soll in diesem Fall direkt, d. h. ohne dynamische Berechnung der Adresse, auf die Attribute zugegriffen werden können bzw. eine Routine statisch gebunden aufgerufen werden.

Schließlich muss der Zugriff auf Attribute und Routinen anderer Objekte als des aktuellen Objektes effizient sein, speziell wie in [Meyer88] gefordert soll der Zugriff in konstanter Zeit erfolgen. In diesen Fällen ist eine statische Bindung oft nicht möglich (in großen Eiffel-System müssen bis zu 60% aller Aufrufe dynamisch gebunden sein, siehe [Meyer97], S. 511).

### 6.2.1 Implementierung der dynamischen Bindung in anderen Compilern

Bevor die Implementierung der dynamischen Bindung für den hier entwickelten Eiffel-Compiler genauer beschrieben wird, sollen einige existierende Compiler oder für andere Systeme vorgeschlagene Implementierungen aufgezeigt werden.

#### 6.2.1.1 SmallEiffel von Dominique Colnet

Der für das Bootstrapping verwendete Compiler *SmallEiffel* [Colnet96] erzeugt für jedes Feature jeder statischen Klasse eine Dispatch-Routine. Diese Routine bestimmt mittels binärer Suche nach einer Integer-Zahl, die dem dynamischen Typ entspricht, die dynamisch gebundene Routine und ruft diese dann auf.

Der Zeitaufwand für einen dynamisch gebunden Aufruf erhöht sich um einen zusätzlichen Routinenaufruf für die Dispatch-Routine. Für die binäre Suche ist zudem logarithmischer Aufwand in der Zahl der Nachfolgerklassen der jeweiligen statischen Klasse nötig. Damit ist der Zeitaufwand nicht wie in der Sprachdefinition [Meyer92] gefordert konstant, sondern steigt mit der Komplexität der Vererbungsstruktur. Auch wenn der Aufwand nur logarithmisch, also sehr langsam, mit der Anzahl der Nachfolgerklassen steigt, so wird doch durch diese Implementierung die Verwendung der Vererbung mit Leistungseinbußen bestraft, was nicht im Sinne von Eiffel ist.

### 6.2.1.2 Smalltalk-80 System von Deutsch und Schiffman

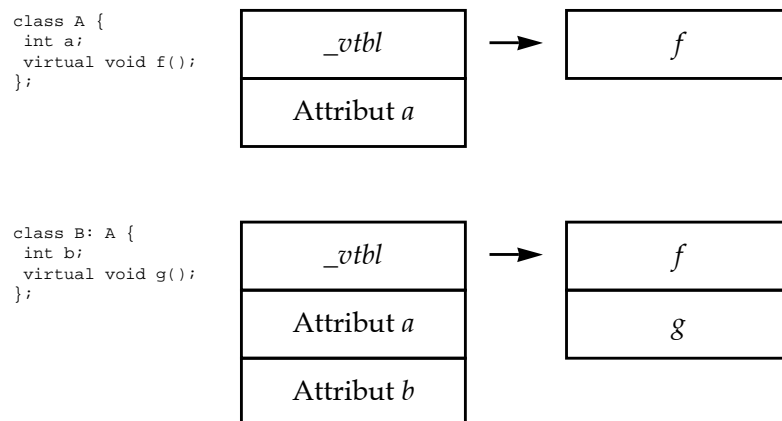
Deutsch und Schiffmann können den Aufwand für einen dynamischen Aufruf in ihrem Smalltalk System durch sogenanntes *in-line caching* im Mittel erheblich senken [Deut84], [Chamb92]. Dieses System nutzt die Tatsache aus, dass meist der Typ des Zielobjektes zwischen zwei Aufrufen gleich bleibt. Um einen Aufruf zu realisieren wird zunächst eine Dispatch-Routine aufgerufen, die wie in der eben beschriebenen Eiffel-Implementierung von Dominique Colnet nach der aufzurufenden Routine sucht.

Danach wird der Aufruf der Dispatch-Routine umgelenkt auf die gefundene Routine, so dass zukünftige Ausführungen des Aufrufs direkt diese Routine anspringen und die Dispatch-Routine übergangen wird. Natürlich ist dieser umgelenkte Aufruf nicht korrekt, wenn sich der Typ des Zielobjektes ändert. Damit dieses Verfahren auch dann noch funktioniert, muss jede Routine vor ihrer Ausführung prüfen, ob das Zielobjekt vom richtigen Typ ist. Dieser Test kann als einfacher Vergleich implementiert werden und in konstanter Zeit erfolgen. Ist er Typ nicht korrekt, so muss wiederum die Dispatch-Routine benutzt werden.

Verglichen mit einem statisch gebundenen Aufruf sind für diesen Test vier oder fünf Maschinenanweisungen und zwei oder drei Speicherzugriffe zusätzlich nötig. Bleibt der Typ des Zielobjektes gleich, so ist der Aufwand für den Test also klein und konstant. Im schlechtesten Fall, wenn sich die Typen der Zielobjekte bei jedem Aufruf ändern, ist der Aufwand jedoch größer als der direkte Aufruf der Dispatch-Routine. Es kann mit diesem Verfahren keine konstante Ausführungszeit garantiert werden.

### 6.2.1.3 C++-Implementierungen

Das von den meisten C++-Implementierungen verwendete Verfahren zur Realisierung der dynamischen Bindung wird in [Chamb92] und [Wilhelm96] beschrieben. [Wilhelm96] beschreibt sogar eine mögliche Anwendung dieses Verfahrens in einem Eiffel-Compiler.



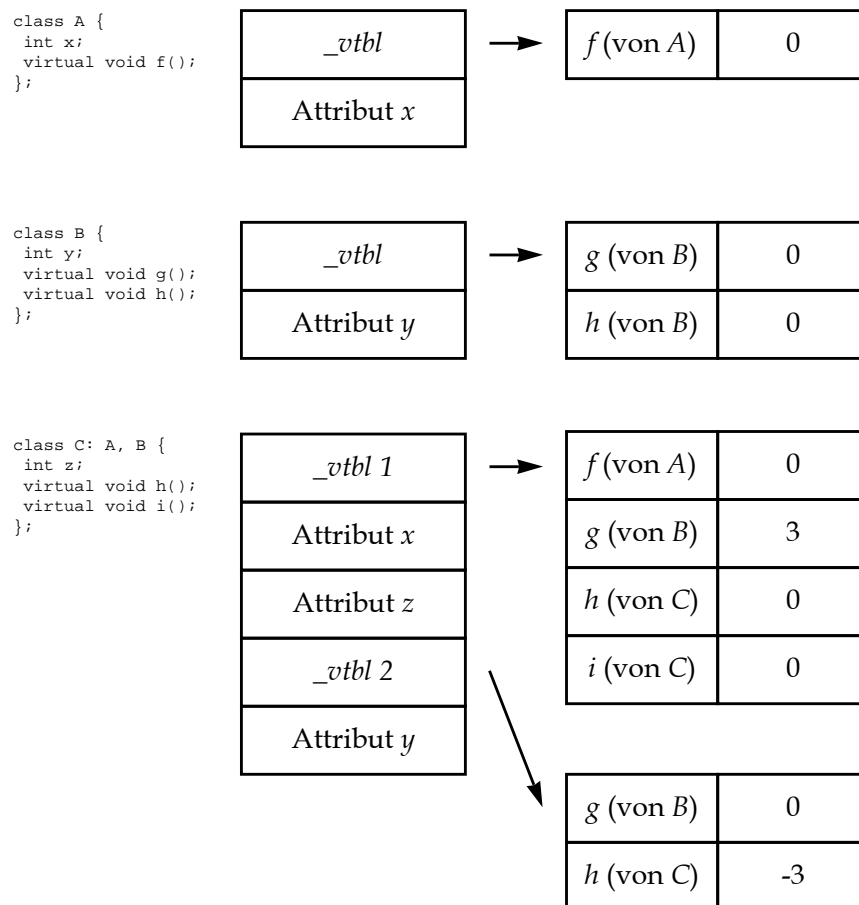
**Bild 5:** Struktur von C++ Objekten bei Einfachvererbung

Die Sprache C++ wurde zunächst ohne Mehrfachvererbung konzipiert [Strou86]. Die ersten Implementierungen mit Einfachvererbung benutzten einen Zeiger pro Objekt, der auf eine Liste der virtuellen Routinen der Klasse des Objektes zeigt, der „*virtual function table*“, kurz `_vtbl`. Die Positionen der Attribute im Objekt verändern sich in Nachfolgeklassen nicht, lediglich neue Attribute können hinzugefügt werden, sie werden an das Ende des Objektes angefügt. Entsprechend sind die Indizes der virtuellen

Routinen im `_vtbl` konstant, in Nachfolgeklassen zugefügte Routinen bekommen neue Indizes am Ende der Liste zugeordnet. *Bild 5* zeigt die Struktur zweier Objekte mit ihren `_vtbls`.

Die in [Strou87] eingeführte Mehrfachvererbung ist nicht so flexibel wie der Eiffel-Mechanismus: Mehrfach geerbte Attribute werden stets verdoppelt, der in *Bild 3* und *Bild 4* gezeigte Fall, bei dem sich die zwei Vaterklassen *B* und *C* das Attribut *a* teilen, kann hier nicht auftreten. Bei Mehrfachvererbung wird das neue Objekt durch Aneinanderhängen der Objekte der direkten Vaterklassen erzeugt. Dabei enthält das neue Objekt alle `_vtbl`-Zeiger der Objekte der direkten Vaterklassen.

Damit die Routinen, die für die Vaterklassen definiert wurden, auch weiterhin benutzt werden können, werden ihnen Zeiger auf das entsprechende Teilobjekt übergeben, selbst wenn dieses mitten im neuen Objekt liegt. Bei Zugriffen auf den `_vtbl` Eintrag greifen diese Routinen also auf den jeweiligen, für das Teilobjekt gültigen `_vtbl`-Zeiger zu. *Bild 6* veranschaulicht einen einfachen Fall von Mehrfachvererbung in C++.



**Bild 6:** Struktur von C++ Objekten bei Mehrfachvererbung

Objekte der neuen Klasse *C* sind zusammengesetzt aus zwei Objekten der Klassen *A* und *B*. Dabei wurde das Objekt *A* noch um das in *C* neue Attribut *z* erweitert, und der `_vtbl` von *A* wurde um die Einträge der in *A* nicht vorhandenen virtuellen Routinen von *B* und *C* erweitert. Die von *B* geerbten virtuellen Routinen bekommen dabei in *C* einen neuen Index im `_vtbl`.

Die *\_vtbls* enthalten jetzt für jede virtuelle Routine zwei Einträge: Neben dem Zeiger auf die aufzurufende Routine zusätzlich eine Integer-Konstante. Die Konstante muss vor dem Aufruf einer virtuellen Routine jeweils zu dem Zeiger auf das Objekt addiert werden, um einen Zeiger auf das korrekte Teilobjekt zu erhalten.

Bei einem Aufruf *c.g*, wobei *c* vom statischen Typ *C* und auf ein Objekt der Klasse *C* zeigt, wird die im ersten *\_vtbl* von *C* eingetragene Routine *g* aufgerufen, die in *B* implementiert wurde. Der Zeiger *c* wird jedoch nicht direkt an *g* übergeben, sondern vorher noch um den neben *g* im *\_vtbl* eingetragenen Wert 3 erhöht. Damit zeigt er auf den Anfang des zweiten Teilobjektes.

Wird nun innerhalb von *B* die in *C* neu implementierte virtuelle Routine *h* aufgerufen, so wird deren Adresse über den zweiten *\_vtbl* bestimmt und der Zeiger mit der Konstanten -3 wieder angepasst, er zeigt dann also wieder auf den Anfang des gesamten Objektes.

Der Aufwand für den Aufruf einer dynamisch gebundenen Routine beträgt bei dieser Implementierung drei Speicherzugriffe und eine Addition, im Vergleich zu einem statischen Aufruf werden in etwa vier zusätzliche Maschinenanweisungen benötigt. Unabhängig von der Komplexität der Vererbung ist dieser Aufwand konstant. Zusätzlich wird noch die Zuweisung von Zeigern unterschiedlichen statischen Typs aufwendiger, da mit zwei Speicherzugriffen und einer Addition der Zeiger auf das jeweilige Teilobjekt bestimmt werden muss.

Die Implementierung eines Garbage-Collectors wird durch die bei dieser Implementierung vorhandenen Zeiger auf Teilobjekte erschwert, der Collector wird dadurch weniger effizient.

Der Speicheraufwand für die Listen der virtuellen Routinen pro Klasse ist  $O(2n+m)$ , wobei *n* die Anzahl aller geerbten virtuellen Routinen ist, und *m* die Zahl der neu eingeführten. *n* geht hier doppelt ein, da bei Mehrfachvererbung geerbte Routinen zwei Einträge erhalten können, eines in der Liste der neuen Klasse und eines in der des Teilobjektes, das der zugehörigen Vaterklasse entspricht.

Ein ganz entscheidender Nachteil dieser Implementierung ist jedoch der Speicheraufwand pro Objekt: Bei der Mehrfachvererbung wird für jede direkte Vaterklasse ein zusätzlicher *\_vtbl* Eintrag nötig, alle bereits in Vaterklassen existierenden *\_vtbls* müssen zudem übernommen werden. Der Speicheraufwand pro Objekt steigt also mit der Breite der Mehrfachvererbung linear an. Damit steigt entsprechend auch der Zeitaufwand für das Erzeugen und Initialisieren von Objekten.

Mehrfachvererbung wird in den meisten C++-Implementierungen also mit erheblichem zusätzlichem Speicherbedarf bestraft. Dies wäre für Eiffel unhaltbar, da in Eiffel Mehrfachvererbung sehr häufig verwendet wird und nicht durch schlechte Effizienz vom Compiler bestraft werden darf. Ein Beispiel aus der Eiffel-Standardbibliothek [ELKS95] macht dies deutlich: Die Klasse `INTEGER_REF` stellt gewöhnliche `INTEGER`-Zahlen zur Verfügung, die als Objekte auf der Halde alloziert werden können. Diese Klasse erbt von den abstrakten Klassen `NUMERIC`, `COMPARABLE` und `HASHABLE`. Damit müsste auf einer 32-Bit-Architektur jedes `INTEGER_REF` Objekt neben seinem Wert, der 4 Bytes Speicher beansprucht, noch 12 Bytes für die Zeiger auf die *\_vtbls* speichern.

### 6.2.2 Unqualifizierte Aufrufe der Features von Current

Bevor der für diese Implementierung gewählte Mechanismus für das dynamische Binden beschrieben wird, soll noch ein Optimierung des häufigsten Aufrufs beschrieben werden: Der unqualifizierte Aufruf von Features des aktuellen Objektes *Current*.

Der Zugriff auf Attribute und Routinen des aktuellen Objektes kann nicht einfach statisch erfolgen, da sich die Position der Attribute, wie oben beschrieben, in Nachfolgerklassen ändern kann und Routinen redefiniert werden können.

Die einfachste Lösung wäre der Zugriff über den Typdeskriptor, wie er im nächsten Abschnitt beschrieben wird. Es wäre jedoch sehr wünschenswert, wenn dieser häufigste Aufruftyp optimiert werden kann. Hierfür wird in Kauf genommen, dass mehr Programmcode erzeugt wird:

Für jede Routine, die in einer Klasse neu eingeführt oder redefiniert wird, wird Maschinencode erzeugt. Dabei wird auf Attribute von *Current* statisch zugegriffen, also mit dem Offset der Attribute in Objekten genau dieser Klasse. Routinen von *Current* werden entsprechend statisch gebunden aufgerufen.

Der Code aller geerbten Routinen, die auf Attribute von *Current* zugreifen, deren Offset in der neuen Klasse anders ist, oder die Routinen von *Current* benutzen, die in der neuen Klasse redefiniert werden, werden dupliziert. D.h. für sie wird in der neuen Klasse nochmals Code erzeugt, der auf Attribute an ihren neuen Offsets zugreift und die redefinierten Routinen aufruft. Entsprechend werden auch all diejenigen geerbten Routinen dupliziert, die Routinen von *Current* aufrufen, die ihrerseits (rekursiv) dupliziert werden müssen.

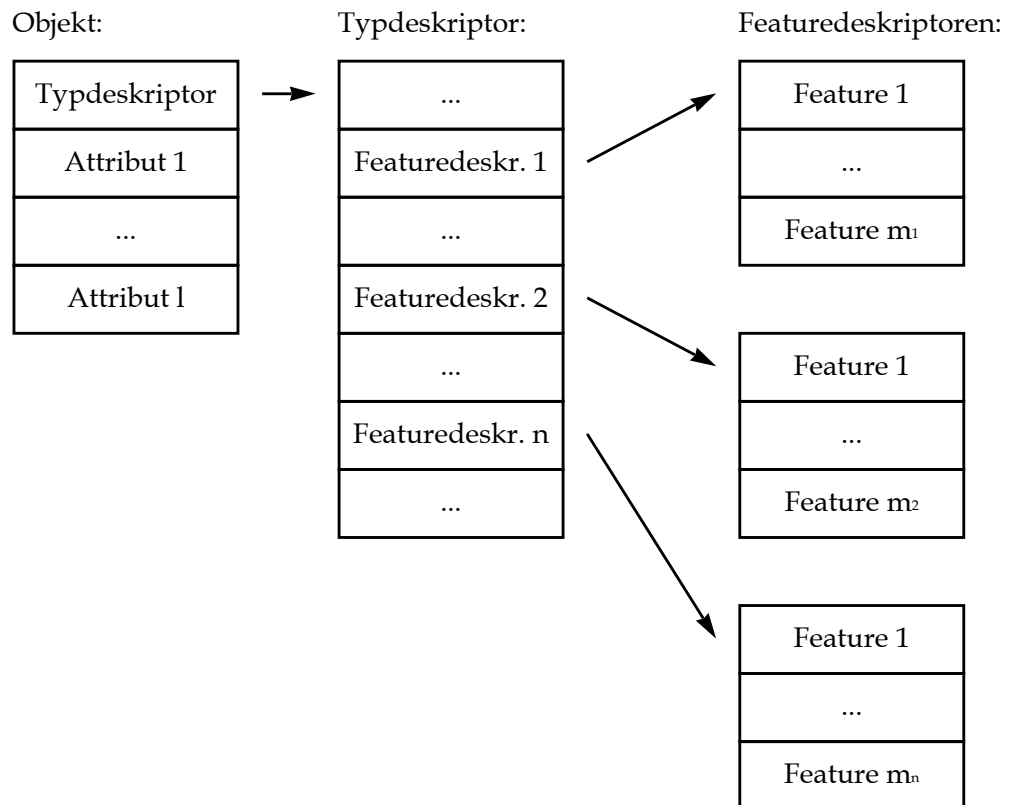
Alle übrigen geerbten Routinen greifen auf keine in der neuen Klasse veränderten Attribute oder Routinen zu, für sie kann also die geerbte Implementierung verwendet werden.

Auf diese Weise wird der Code einiger Routinen automatisch vom Compiler kopiert, speziell geschieht dies in den Fällen der wiederholten Vererbung (*repeated inheritance*), in denen dies wie in [Meyer88] beschrieben ohnehin unumgänglich ist. Andererseits wird hierdurch der zusätzliche Zeitaufwand der dynamischen Bindung und der Mehrfachvererbung beim Zugriff auf Attribute und Routinen von *Current* im Vergleich zur statischen Bindung auf null reduziert.

### 6.2.3 Qualifizierte Featureaufrufe

Beim Zugriff auf Features eines Objekts über eine Referenz, die in einer lokalen Variablen oder einem Attribut von *Current* gespeichert ist, führt nun nichts an einer etwas aufwendigeren Lösung der dynamischen Bindung vorbei.

Wie schon oben im Abschnitt über die Objektstruktur beschrieben, sollen Objekte neben den Attributen nur einen einzigen zusätzlichen Eintrag haben, einen Zeiger auf Ihren Typdeskriptor. Über diesen Typdeskriptor müssen die von den verschiedenen statischen Typen abhängigen unterschiedlichen Sichten auf die Features der Klasse erreichbar sein, ähnlich den unterschiedlichen *virtual function tables* (*\_vtbl*) in der oben be-



**Bild 7:** Objekt mit Typ- und Featuredeskriptoren

schriebenen C++-Implementierung, die unterschiedlichen statischen Typen entsprechen.

Der Typdeskriptor soll daher aus einer Liste von Zeigern bestehen, ein Zeiger für jede mögliche statische Sicht auf das Objekt, also für jede Vorgängerklasse ein Eintrag. Diese Zeiger weisen wiederum auf Featuredeskriptoren, die für jedes Feature der statischen Klasse einen Eintrag enthalten. *Bild 7* zeigt den Aufbau eines Objektes mit seinen Typ- und Featuredeskriptoren.

Die einzelnen Einträge in den Featuredeskriptoren sind für Routinen Zeiger auf den Code der entsprechenden Routine, und für Attribute der Offset des Attributs im Objekt. Die Features jeder statischen Klasse können einfach durchnummeriert werden, die jedem Feature zugeordneten Nummern sind die Indizes in den Featuredeskriptoren.

Etwas schwieriger ist die Verteilung der Indizes im Typdeskriptor: Diese Indizes sind innerhalb eines Eiffel-Systems für jede Klasse konstant. Die Indizes müssen für zwei unterschiedliche Klassen *A* und *B* unterschiedlich sein, wenn ein Typdeskriptor einer anderen Klasse *C* Einträge für die Sichten *A* und *B* benötigt. Die Indizes müssen also unterschiedlich sein, wenn *A* und *B* in dem Eiffel-System eine gemeinsame Nachfolgerklasse *C* besitzen. Diese Eigenschaft kann nur bei Betrachtung des gesamten Systems geprüft werden.

Die einfachste Möglichkeit, Indizes für die Einträge in die Typdeskriptoren zu verteilen, ist die schlichte Durchnummerierung aller Klassen eines Systems mit unterschiedlichen Indizes. Dann haben unterschiedliche statische Sichten stets unterschiedliche Indizes in den Typdeskriptoren. Allerdings wird in einem System mit *n* Klassen für jede Klasse ein

Typdeskriptor mit  $n$  Einträgen benötigt (von denen viele leer sind), und der Speicherbedarf des Systems ist somit  $O(n^2)$ . Für große Systeme ist dies kaum akzeptabel.

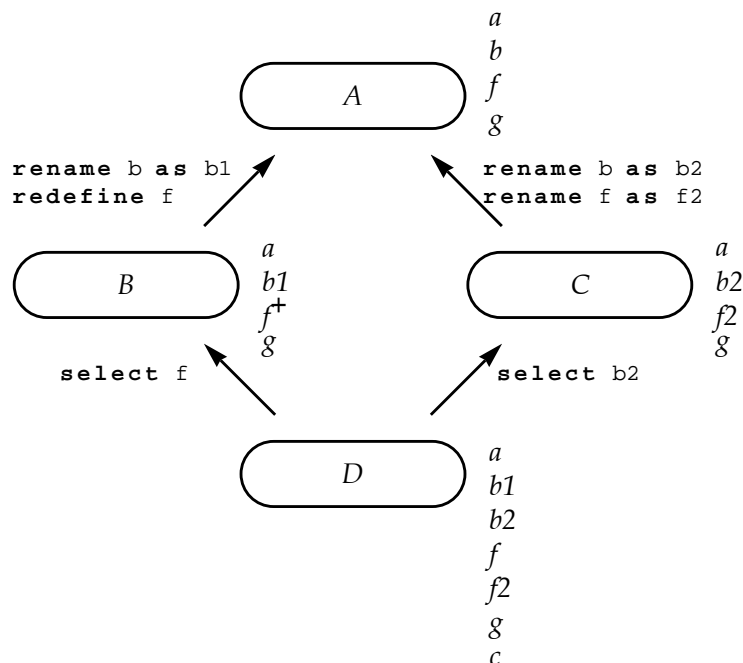
Eine bessere Lösung erhält man, indem man versucht, Klassen, die keinen gemeinsamen Nachfolger besitzen, denselben Indexwert zuzuordnen. Dann kann die Verteilung der Indizes als das folgende Graphfärbungsproblem beschrieben werden:

Es gilt, den Graphen  $G=(V,E)$  zu färben mit  $V$  gleich der Menge aller im System vorkommenden Klassen und  $(v1,v2) \in E$  falls ein  $v3 \in V$  existiert mit  $v3$  ist Nachfolger von  $v1$  und  $v2$ . Die Färbungszahl  $c(G)$  ist die Anzahl der in den Typdeskriptoren nötigen Einträge und eine Färbung  $color$  von  $G$  mit den natürlichen Zahlen 1 bis  $c(G)$  ordnet jeder Klasse  $v \in V$  den Index  $color(v)$  in den Typdeskriptoren ihrer Nachfolger zu.

Der Speicheraufwand verringert sich auf diese Weise zu  $O(n \cdot c(G))$ , im schlechtesten Fall, falls beispielsweise in einem System eine Klasse existiert, die Nachfolger aller anderen Klassen des Systems ist, ist  $c(G)=n$  und damit der Aufwand weiterhin  $O(n^2)$ . Es ist jedoch zu erwarten, dass die Färbungszahl gewöhnlich weit geringer als  $n$  sein wird.

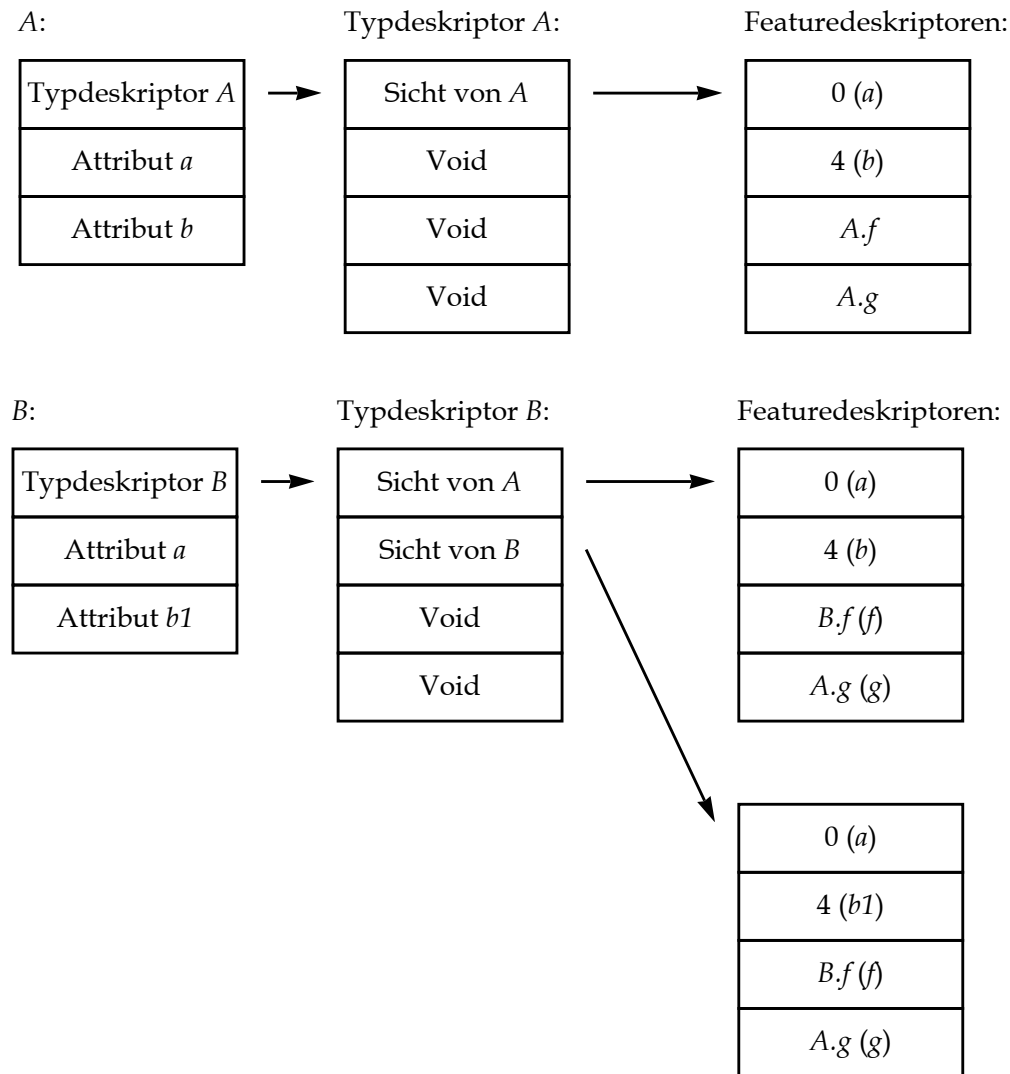
### 6.2.4 Ein Beispiel

Als Beispiel für den Aufbau der Typdeskriptoren sollen das in *Bild 3* eingeführte Klassensystem wie in *Bild 8* gezeigt noch um die Routinen  $f, f2$  und  $g$  erweitert werden. Die Abbildung lehnt sich an die Darstellungsweise in [Meyer92] an, so bedeutet das Pluszeichen in „ $f^+$ “, dass dieses Feature in der Klasse  $B$  redefiniert wird.



**Bild 8:** Mehrfachvererbung von Attributen

Die Objekte sehen dann beispielsweise wie in den folgenden Bildern *Bild 9*, *Bild 10* und *Bild 11* gezeigt aus. Dabei wurde angenommen, dass die Sichten von  $A, B, C$  und  $D$  die Indizes 1, 2, 3 und 4 zugeordnet bekommen haben und das System sonst keine Klassen enthält (wäre dies der Fall, so hätten die Typdeskriptoren möglicherweise weitere *Void-*



**Bild 9:** Objekte der Klassen A und B aus Bild 8 mit Typ- und Featuredeskriptoren (Einträge). In den Featuredeskriptoren sind in Klammern jeweils die Featurenamen der jeweiligen Sicht angegeben.

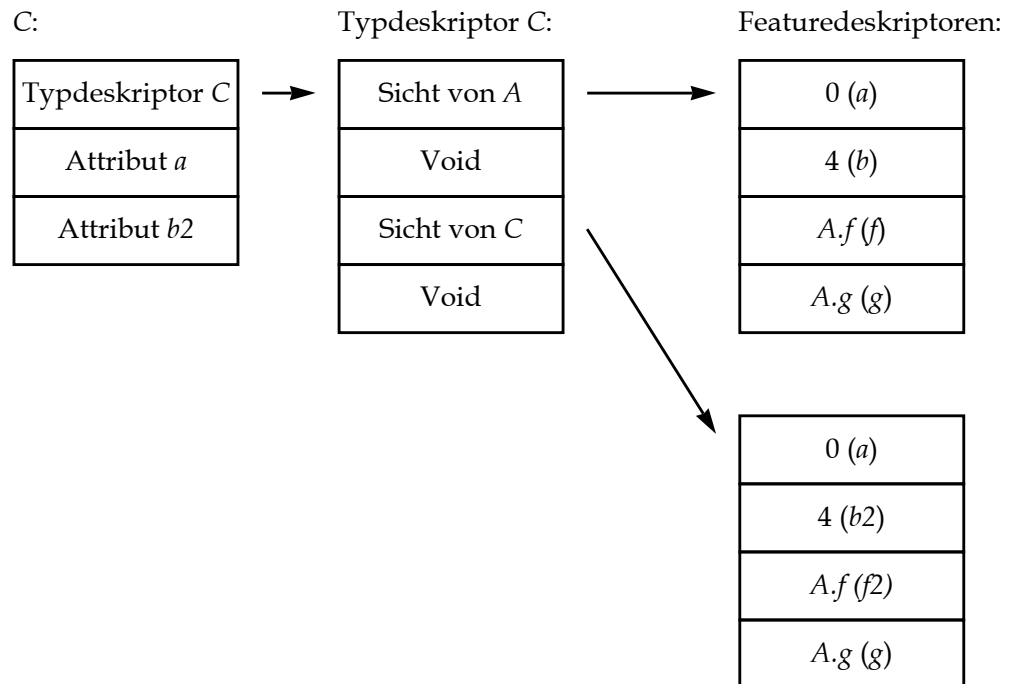
Bei den Einträgen für die Attribute geben die Zahlen die Byte-Offsets der Attribute im Objekt an. Dabei wurde davon ausgegangen, dass der Typdeskriptor einem negativen Offset besitzt, so dass das erste Attribut jedes Objektes den Offset null hat. Es wurde angenommen, dass *INTEGER*-Werte 4 Bytes belegen.

Der für einen Attributzugriff der Form  $q := a.b$  nötige Code (dabei sei  $a$  vom statischen Typ  $A$ ) sieht in einer Modula-2-ähnlichen Syntax wie folgt aus:

$$q := (a.^{\text{typdeskriptor}[\text{index}(A)]^{\wedge}.\text{feature}[\text{feature\_num}(b)] + a}^{\wedge}$$

Je nach dem dynamischen Typ von  $a$  wird hier das Attribut an Offset 4 (für die Klassen A, B und C) oder an Offset 8 (bei Klasse D) gelesen.

Im Vergleich zum Zugriff mit festem Offset sind also drei zusätzliche Speicherzugriffe und eine Addition nötig, dies kann in vier zusätzlichen Maschinenbefehlen durchgeführt werden.



**Bild 10:** Objekt der Klasse C aus Bild 8 mit Typ- und Featuredeskriptor

Ähnlich wird bei einem Routinenaufruf  $a.f$  vorgegangen. Hier ist Code der Form

```
CALL (a^.typdescriptor[index(A)]^.feature[feature_num(f)])
```

nötig. Je nach dem dynamischen Typ von  $a$  wird hier die Routine  $A.f$  (für  $A$  und  $C$ ) oder  $B.f$  (für  $B$  und  $D$ ) aufgerufen.

Im Vergleich zum statischen Aufruf sind dies drei zusätzliche Speicherzugriffe, die in drei zusätzlichen Maschinenbefehlen gemacht werden können.

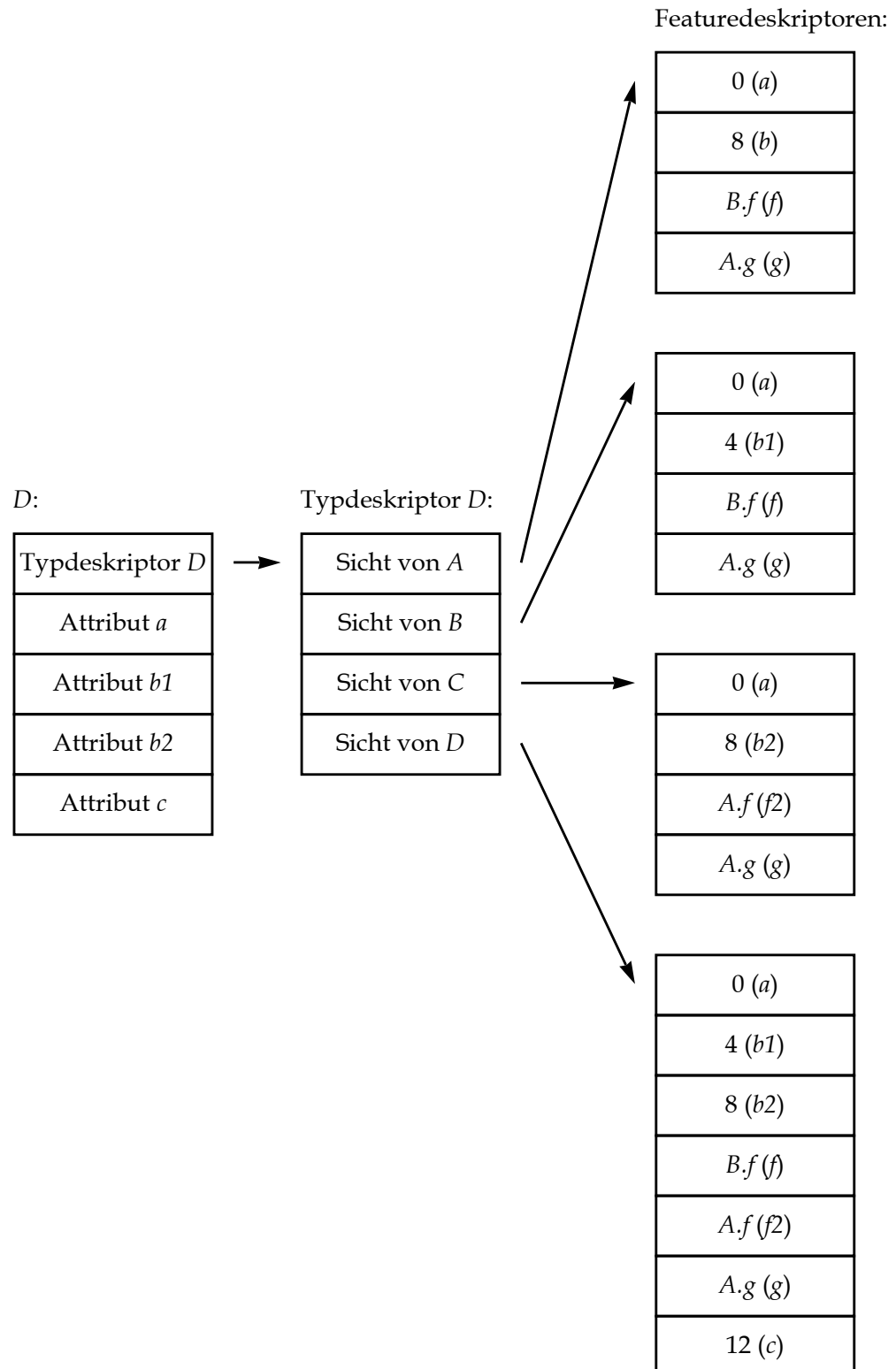
### 6.2.5 Aufwandsvergleich

Wie bei dem Beispiel eben schon beschrieben, beträgt der Aufwand für einen dynamisch gebundenen Routinenaufruf bei der vorgeschlagenen Implementierung drei zusätzliche Maschinenbefehle. Dies ist sogar weniger als die für C++ nötigen vier zusätzlichen Anweisungen.

Auch der konstante Speicheraufwand von einer Referenz pro Objekt ist dem mit der Komplexität der Mehrfachvererbung anwachsendem Aufwand der C++-Implementierungen deutlich überlegen. Die Ausführungszeit und die Objektgröße bleiben von der Komplexität der Klassenhierarchie bei dieser Implementierung völlig unbeeinflusst.

Der Speicheraufwand für die Featuredeskriptoren ist vergleichbar mit dem Aufwand für die *virtual function tables* in C++, allerdings steigt deren Anzahl pro Vererbungsstufe um eine Liste pro Klasse mehr an als die der *\_vtbls* in C++: Dort bleibt bei Einfachvererbung die Anzahl der Listen gleich, während sie in Eiffel auch hier um eins ansteigt.

Der Speicheraufwand für die Featuredeskriptoren kann abgeschätzt werden, wenn wir annehmen, dass in einem System mit  $n$  Klassen, welche jeweils durchschnittlich  $f$  Fea-



**Bild 11:** Ein Objekt der Klasse *D* aus *Bild 8* mit Typ- und Featuredeskriptoren

tures besitzen, jede Klasse im Schnitt  $v$  Vorgängerklassen besitzt. Dann ist der Speicheraufwand in  $O(n \cdot f \cdot v)$ . Nehmen wir weiter an, dass auch in beliebig großen Systemen jede Klasse weniger als 500 Features und höchstens 100 Vorgängerklassen besitzt (komplexere Klassen scheinen aus Software-Engineering Gründen nicht mehr handhabbar), so

steigt der Speicherbedarf für die Featuredeskriptoren asymptotisch lediglich mit der Größe des Systems, also  $O(n)$ . Allerdings ist für kleinere und mittlere Systeme der Speicherbedarf für die Featuredeskriptoren doch merklich.

Der schlimmstenfalls quadratisch mit der Anzahl der Klassen anwachsende Speicherbedarf für die Typdeskriptoren kann für große Systeme die gravierenderen Probleme bereiten. Hier muss auf jeden Fall mit der oben beschriebenen Graphfärbung gearbeitet werden. Es ist jedoch schwer, die Färbungszahl für beliebige Vererbungsstrukturen vorherzusagen. Wenn eine Reihe an großen, existierenden Klassensystemen untersucht werden könnte, sollten deren Färbungszahlen bestimmt werden, damit wenigstens empirisch die Vermutung belegt werden kann, dass die Färbungszahl auch bei großen Systemen deutlich unter der Anzahl der Klassen bleibt.

### 6.3 Generische Klassen

Es wäre möglich, für jede generische Klasse nur ein einziges Mal Code zu erzeugen, der dann für alle aktuellen generischen Derivate verwendet würde. Objekte derselben generischen Klasse würden sich dann lediglich in ihren Typ- und Featuredeskriptoren unterscheiden: Falls die aktuellen generischen Parameter unterschiedliche Größe haben, so unterscheiden sich die Einträge für die Offsets der Attribute in den Featuredeskriptoren.

Allerdings ergeben sich hierbei eine Reihe an Schwierigkeiten und Ineffizienzen: Stackframes müssten flexible Größe haben, da die Größe von lokalen Variablen und Parametern, deren aktuelle Typen durch einen generischen Parameter beschrieben werden, nicht feststehen. Zudem könnten viele Zugriffe auf Attribute von *Current* nicht mehr wie oben beschrieben statisch erfolgen, sondern müssten den Typdeskriptor verwenden.

Auf Kosten von zusätzlichem Programmcode soll daher spezieller Code für unterschiedliche aktuelle generischen Parameter erzeugt werden. Um dies genauer zu erklären werden im folgenden Text einige Begriffe benötigt, die hier definiert werden sollen:

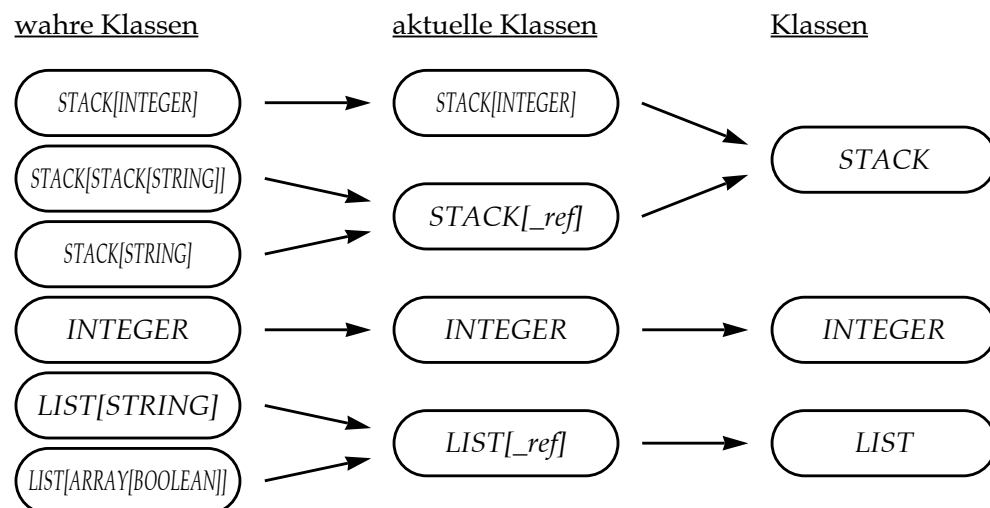
- |                        |  |
|------------------------|--|
| <i>Klasse</i>          | Mit Klasse soll eine, möglicherweise generische, Klasse ohne aktuelle generische Parameter beschrieben werden. Zu jeder Klasse eines Systems gehört genau ein Eiffel-Quelltext, der diese Klasse beschreibt. Innerhalb des hier entwickelten Compilers wird für jede Klasse bei der Übersetzung jeweils genau ein Objekt der Klassen <i>PARSE_CLASS</i> und <i>CLASS_INTERFACE</i> erzeugt.  |
| <i>wahre Klasse</i>    | Eine wahre Klasse beschreibt eine Klasse mit aktuellen generischen Parametern, falls es eine generische Klasse ist. Die aktuellen generischen Parameter selbst können beliebige andere wahre Klassen sein. Aus jeder generischen Klasse eines übersetzten Eiffel-Systems können beliebig viele wahre Klassen entstehen. Jeder nicht generischen Klasse entspricht genau eine wahre Klasse. Wahre Klassen werden innerhalb des Compilers durch jeweils ein Objekt der Klasse <i>TRUE_CLASS</i> repräsentiert. |
| <i>aktuelle Klasse</i> | Wie die wahren Klassen, so besitzen auch die aktuellen Klassen aktuelle generische Parameter. Die aktuellen generischen Parameter sind   |

entweder expandierte aktuelle Klassen oder die pseudo-Klasse *\_ref*. Alle wahren Klassen, die sich nur dadurch unterscheiden, dass ihre aktuellen generischen Parameter unterschiedliche Referenzklassen sind, werden durch eine einzige aktuelle Klasse repräsentiert. Jeder nicht generischen Klasse entspricht genau eine aktuellen Klasse. Aktuelle Klassen werden innerhalb des Compilers jeweils durch ein Objekt der Klasse *ACTUAL\_CLASS* repräsentiert.

Die Validität wird für jede generische Klasse nur einmal geprüft, sie hängt nicht von den aktuellen generischen Parametern ab. Bei der Codeerzeugung für die generischen Klassen wäre es am leichtesten, für alle in einem System vorkommenden wahren Klassen getrennt Code zu erzeugen. Für jede wahre Klasse stehen die Typen der aktuellen generischen Parameter fest, es stehen also auch die Positionen und Typen aller Attribute fest. Zugriffe auf *Current* können innerhalb der Routinen der generischen Klasse direkt erfolgen.

Diese Implementierung führt jedoch dazu, dass erheblich mehr Code erzeugt wird: Als Beispiel soll der hier entwickelte Compiler selbst dienen: Die Anzahl der Klassen des Compilers ist weniger als die Hälfte der Zahl seiner wahren Klassen.

In Eiffel-Systemen sind die häufigsten Typen und damit gewöhnlich auch die am häufigsten vorkommenden aktuellen generischen Parameter Referenzen. Referenzen haben stets dieselbe Größe. Daher können die Objekte aller wahren Klassen, deren aktuelle generische Parameter sich nur darin unterscheiden, dass sie unterschiedliche Referenztypen sind, gleich aufgebaut sein. Dies führte zu der Idee, nur für die Routinen unterschiedlicher aktueller Klassen mehrfach Code zu erzeugen. Die Beziehung zwischen einigen unterschiedlichen Klassen, wahren Klassen und aktuellen Klassen wird in *Bild 12* veranschaulicht.



**Bild 12:** „Entspricht“-Beziehung zwischen wahren Klassen, aktuellen Klassen und Klassen

Auch wenn der Programmcode für verschiedene wahre Klassen zusammengefasst wird, wenn sie derselben aktuellen Klasse entsprechen, müssen sich die Objekte dieser Klassen in ihren Typdeskriptoren unterscheiden. Die Featuredeskriptoren können dagegen gleich sein. Der Grund für die unterschiedlichen Typdeskriptoren liegt an den für

das *Assignment\_attempt* nötigen Typinformationen, die auch in den Typdeskriptoren gespeichert werden müssen. So muss bei den Attributen

```
s1: STACK[ANY];  
s2: STACK[STRING];
```

für die Ausführung des *Assignment\_attempts*

```
s2 ?= s1;
```

der aktuelle generische Parameter von *s1* geprüft werden, es reicht nicht die Information, dass dieser eine Referenz ist.

Es existiert also nicht nur für jede Klasse, sondern für jede wahre Klasse ein eigener Typdeskriptor. Der oben beschriebene Algorithmus zur Färbung des Klassengraphen muss daher den Graphen der wahren Klassen des Systems färben.

## 6.4 Parameter und lokale Variablen

### 6.4.1 Parameterübergabe

Mit Parametern sind hier alle an eine Routine übergebenen Werte gemeint, einschließlich des implizit übergebenen Wertes für *Current*.

Parameter haben in Eiffel Wertesemantik. Unstrukturierte Parameter, wie Zeiger oder *INTEGER*-Werte, können also am effizientesten direkt als Werte übergeben werden. Zusammengesetzte Parameter, also expandierte Objekte, können nicht einfach in Registern übergeben werden. Diese Objekte sollen vom Aufrufer in eine temporäre Variable kopiert werden, deren Adresse an die aufgerufene Routine übergeben wird.

Der als *Current* implizit übergebene Parameter hat immer Referenzsemantik, für ihn wird also ein Zeiger auf das Haldenobjekt oder die Adresse des expandierten Objektes übergeben.

Alle Parameter sind damit entweder Referenzen, elementare Typen oder als Referenzparameter übergebene temporäre Variablen. Die Parameter können also leicht in Prozessorregistern übergeben werden.

Auf der SPARC-Architektur stehen sechs freie *out*-Register für die Übergabe von unstrukturierten Parametern zur Verfügung. Zusätzlich könnte ein Teil der globalen Register zur Parameterübergabe verwendet werden. Alle weiteren Parameter müssen über den Stapel im Aktivierungsblock der aufrufenden Routine übergeben werden. Um die Einbindung von C-Routinen zu erleichtern, soll jedoch die in [SPARC93] festgelegte Aufrufkonvention berücksichtigt werden, die sich auf die sechs *out*-Register beschränkt.

### 6.4.2 Funktionsergebnisse

Das Resultat einer Funktion soll, wenn es sich um eine Referenz oder einen Standardtyp handelt, direkt in einem Register zurückgeliefert werden. Nur expandierte Ergebnistypen werden am einfachsten als implizite Referenzparameter implementiert: Es wird die Adresse einer temporären lokalen Variable übergeben, in die das Ergebnis geschrieben wird.

### 6.4.3 Lokale Variablen

Für die lokalen und temporären Variablen sollen, falls möglich, Register benutzt werden. Ansonsten werden sie auf dem Stapel im Aktivierungsblock der aktiven Prozedur gespeichert und über feste Offsets vom Framepointer aus angesprochen. Lokale Variablen von expandierten Objekten werden auch auf dem Stapel angelegt.

## 6.5 Assignment-Attempt

Für Anweisungen des Typs *Assignment\_attempt* ist ein Test des dynamischen Typs einer Referenz nötig. So muss bei der Anweisung

```
b ?= a;
```

geprüft werden, ob *a* (vom statischen Typ *A*) ein Zeiger auf ein Objekt enthält, der dem von *b* (dieser sei *B*) entspricht (conformance).

Eine einfache Möglichkeit, diesen Test zu implementieren, ist es, in jedem Typdeskriptor einer wahren Klasse *C* eine sortierte Liste von eindeutigen Nummern *number(D)* aller wahren Klassen *D* zu speichern, die Vorgänger von *C* sind. Der Typtest kann dann mit einer binären Suche erfolgen, der Zeitbedarf ist durch  $O(\log(n))$  beschränkt, wobei *n* die Anzahl der Vorgänger des von *b* referenzierten Objektes ist.

Ein Problem ergibt sich noch, wenn die Anweisung in einer generischen Klasse erfolgt und die wahre Klasse des statischen Typs von *a* von den aktuellen generischen Parametern abhängt. Für solche Anweisungen können die Nummern der zu testenden wahren Klassen auch in den Typdeskriptoren der Klasse gespeichert werden, die diese Anweisung enthält.

## 6.6 Once-Routinen

Once-Routinen zeichnen sich dadurch aus, dass sie für jede Ausführung eines Eiffel-Systems nur ein einziges Mal ausgeführt werden dürfen. Dies kann durch globale Variablen realisiert werden: Ein boolescher Wert, der angibt, ob die Routine bereits ausgeführt wurde, und für Funktionen zusätzlich eine globale Variable für das Funktionsergebnis des ersten Aufrufs.

Once-Routinen in generischen Klassen dürfen pro Systemausführung auch höchstens einmal ausgeführt werden, selbst wenn das System mehrere von der generischen Klasse abgeleitete wahre Klassen besitzt oder die Routine für mehrere aktuelle Klassen implementiert wurde. Damit auch in diesem Fall nur ein einziger Aufruf der unterschiedlichen Implementierungen ausgeführt wird, müssen diese alle auf dieselben globalen Variablen zugreifen. Diese Variablen dürfen für jede Klasse nur einmal existieren, auch wenn dieser Klasse mehrere aktuelle Klassen entsprechen.

## Literatur

- [Chamb92] Craig Chambers: „The Design and Implementation of the SELF Compiler“, PhD Dissertation at the Department of Computer Science of Stanford University, March 1992

- [Colnet96] Dominique Colnet and Suzanne COLLIN: „SmallEiffel V-0.88“, 1994-1996, <ftp://ftp.loria.fr/pub/loria/genielog/SmallEiffel>
- [Deutsch84] L. Peter Deutsch and Allan M. Schiffman: „Efficient Implementation of the Smalltalk-80 System“, Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, January, 1984.
- [ELKS95] Nonprofit International Consortium for Eiffel (NICE): „The Eiffel Library Standard, Vintage 95“, Version 8, <ftp://ftp.eiffel.com>, June 4, 1995
- [Meyer88] Bertrand Meyer: „Object-oriented Software Construction“, Prentice Hall International (UK) Ltd, Hertfordshire, 1988
- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertfordshire, 1992
- [Meyer97] Bertrand Meyer: „Object-oriented Software Construction“, 2nd edition, Prentice Hall, Inc, New Jersey, 1997
- [SPARC93] „System V Application Binary Interface, SPARC Processor Supplement“, 3rd edition, UNIX Press, Prentice Hall, 1993
- [Strou86] Bjarne Stroustrup: „The C++ Programming Language“, Addison-Wesley, Reading, MA, 1986
- [Strou87] Bjarne Stroustrup: „Multiple Inheritance for C++“, Proceedings of the European Unix Users Group Conference, pp. 189-207, Helsinki, May, 1987
- [Wilhelm96] Reinhard Wilhelm und Dieter Maurer: „Übersetzerbau — Theorie, Konstruktion, Generierung“, Springer-Verlag, Berlin, 1996

*Die Zwischendarstellung kann  
verschiedene Formen haben.  
– Aho, Sethi, Ullman*

---

## 7. Zwischencodenerzeugung

Auch wenn die Quelltexte nach der Validity-Überprüfung komplett untersucht wurden, wird nicht direkt mit der Erzeugung des Maschinencodes begonnen, sondern zunächst ein von der konkreten Zielmaschine unabhängiger Zwischencode erzeugt. Dies hat vor allem zwei Gründe: Zum einen können auf dem Zwischencode eine Reihe an Optimierungen leichter durchgeführt werden als auf dem konkreten Maschinencode. Andererseits können so leicht Generatoren für verschiedenen Zielcode geschrieben werden. Durch die einfache Struktur des Zwischencodes lassen sich die Zielcodegeneratoren auch verhältnismäßig leicht implementieren. Der Zwischencode ist jedoch nicht völlig maschinenunabhängig: Er ist so aufgebaut, dass speziell Code für RISC-Prozessoren [Hennes96] sehr leicht zu erzeugen ist.

### 7.1 Lokale Variablen

Die Befehle des Zwischencodes arbeiten alle auf lokalen Variablen, die durch Objekte der Klasse `LOCAL_VAR` repräsentiert werden. Eine lokale Variable ist dabei eine Abstraktion für ein Prozessorregister, einen im Aktivierungsblock gespeicherten Wert, einen Prozedurparameter oder ein Funktionsergebnis. Alle im Eiffel-Quelltext beschriebenen formalen Argumente und lokalen Deklarationen werden in einer lokalen Variablen gespeichert. Zudem gibt es jedoch auch lokale Variablen für Zwischenergebnisse, die im Quelltext nicht explizit beschrieben werden.

Welche lokalen Variablen in Registern gespeichert werden und in welchen, und welche im Aktivierungsblock und wo dort, das wird bei der Zwischencodenerzeugung noch offen gelassen.

#### 7.1.1 Typen lokaler Variablen

Lokale Variablen werden beschrieben durch Ihren Typ. Der Typ legt die Größe der Variablen, die Möglichkeit, sie in einem Register zu speichern, und die auf sie anwendbaren Zwischencodbefehle fest. Folgende Typen werden definiert:

#### 7.1.1.1 Reference

Lokale Variablen des Typs *Reference* speichern nur Zeiger, die vom Garbage-Collector verfolgt werden. Der Codegenerator muss beachten, dass diese Werte immer dann ungültig werden, wenn sie sich in Registern befinden und der Garbage-Collector aktiv werden kann. Zudem müssen die Positionen aller *Reference*-Variablen im Aktivierungsblock dem Garbage-Collector mitgeteilt werden. In diesen Variablen dürfen dann keine anderen Werte als Referenzen auf Objekte gespeichert werden.

#### 7.1.1.2 Pointer

Lokale Variablen des Typs *Pointer* stehen für alle Berechnungen von Speicheradressen zur Verfügung. In ihnen können auch Referenzen auf Objekte gespeichert werden, die vom Garbage-Collector verfolgt werden. Anders als die Variablen des Typs *Reference* ist es bei denen des Typs *Pointer* Aufgabe des Zwischencodeerzeugers, sicherzustellen, dass der Garbage-Collector nicht aktiv wird, solange der Wert einer solchen Variablen die Adresse eines Haldenobjektes ist (genauer: es darf keine Routine aufgerufen werden, die durch eine Speicheranforderung die Aktivierung des Garbage-Collectors bewirken kann, während eine *Pointer*-Variable einen solchen Wert enthält).

#### 7.1.1.3 Integer, Character und Boolean

Diese Typen entsprechen den Eiffel-Standardtypen. *Integer* kann zudem, ähnlich wie *Pointer*, für die Berechnung von Adressen hergenommen werden.

#### 7.1.1.4 Real und Double

Diese Typen unterscheiden sich von den oben angegebenen Typen dadurch, dass ihre Werte, wenn möglich, in den Fließkommaregistern gespeichert werden, und nicht in einem der *general-purpose*-Register, wie sie in vielen RISC-Architekturen existieren. Die einzigen erlaubten Operationen sind Fließkommaoperationen.

#### 7.1.1.5 Expandierte Typen

Diese Typen schließlich stehen für Werte, die nicht oder nicht sinnvoll in Registern gespeichert werden können (ein Wert, der aus 3 Characters, einem Boolean und einem Integer besteht, könnte möglicherweise in einem 64-Bit Register gespeichert werden, der Zugriff auf die Teilwerte würde dann allerdings so aufwendig, dass die Speicherung im Aktivierungsblock sinnvoller ist).

Die Werte dieser Typen sind expandierte Objekte, die lokal zu einer Routine deklariert wurden.

### 7.1.2 Arten von lokalen Variablen

Wie schon oben kurz beschrieben, gibt es die folgenden Arten von lokalen Variablen. Jede lokale Variable ist von einer bestimmten Art und hat einen bestimmten Typ.

### 7.1.2.1 Argumente

Argumente sind die lokalen Variablen für *Current* und die im Quelltext beschriebenen formalen Argumente einer Routine. Argumente dürfen nicht verändert werden.

Der Codegenerator definiert, wie die Übergabe der Argumente verschiedener Typen realisiert werden soll, also welche Typen in Registern und welche über den Aktivierungsblock des Aufrufers übergeben werden sollen. Auf der SPARC-Architektur ist dabei zudem die Verwendung der Registerfenster zu berücksichtigen.

### 7.1.2.2 Lokale Werte

Dies sind die gewöhnlichen lokalen Variablen, die explizit im Eiffel-Text als lokale Variablen deklariert wurden oder die vom Compiler für temporäre Werte benutzt werden. Ihre Werte können gelesen und geschrieben werden.

Ein besonderer lokaler Wert wird für die lokale Variable *Result* innerhalb von Funktionen angelegt. Er wird wie die anderen lokalen Werte behandelt, lediglich muss die Zielcodeerzeugung am Ende der Funktion Code für die Rückgabe dieses Wertes an den Aufrufer erzeugen.

## 7.2 Zwischencodebefehle

Anweisungen und Ausdrücke einer Eiffel-Routine werden, mit Ausnahme der Kontrollanweisungen, in Zwischencodebefehle übersetzt. Für jeden Befehl wird ein Objekt einer Nachfolgerklasse der abstrakten Klasse *COMMAND* erzeugt. Die Befehle erhalten als Argumente lokale Variablen oder Konstanten. Werden die lokalen Variablen in Registern gespeichert, so entspricht ein Zwischencodebefehl meist direkt einem Maschinenbefehl der RISC-Zielarchitektur.

Im folgenden werden die verschiedenen Klassen von Zwischencodebefehlen beschrieben, in Klammern jeweils die Namen der Klassen. Für die Veranschaulichung der Befehle wird folgende Darstellung gewählt:

```
<Klasse>: <Befehl>
```

Dabei gibt *<Klasse>* den Typ des Zwischencodebefehles an und *<Befehl>* zeigt die Argumente und die Operation des Befehls. So steht der Zwischencodebefehl

```
ARITHMETIC_COMMAND:  local1 := local2 + local3
```

für ein Objekt der Klasse *ARITHMETIC\_COMMAND*, das die zwei lokalen Variablen *local2* und *local3* addiert und das Ergebnis in *local1* speichert.

### 7.2.1 Zuweisung (ASSIGN\_COMMAND)

Die gewöhnliche Zuweisung von Werten lokaler Variablen wird durch den Zwischencodebefehl

```
ASSIGN_COMMAND:      local1 := local2
```

realisiert. Dieser Befehl kann auch für die Konvertierung von Werten unterschiedlicher Typen verwendet werden. Die beiden lokalen Werte müssen entweder von demselben

Typ sein, oder beide einer der Typen *Reference*, *Pointer* oder *Integer*, oder beide numerisch (*Integer*, *Real* oder *Double*), oder eine lokale Variable vom Typ *Integer* und die andere *Boolean* oder *Character*.

### 7.2.2 Zuweisung einer Konstanten (`ASSIGN_CONST_COMMAND`)

Dieser Zwischencodebefehl weist den Wert einer Konstanten einer lokalen Variablen zu. Es existieren fünf Versionen für *Integer*-, *Character*-, *Boolean*-, *Real*- und *Double*-Konstanten:

```
ASSIGN_CONST_COMMAND: local := Integer-constant
ASSIGN_CONST_COMMAND: local := Character-constant
ASSIGN_CONST_COMMAND: local := Boolean-constant
ASSIGN_CONST_COMMAND: local := Real-constant
ASSIGN_CONST_COMMAND: local := Double-constant
```

Dabei gelten die entsprechenden Einschränkungen für den Typ der lokalen Variablen. Für *Integer*-Konstanten sind jedoch auch *Reference*, *Pointer*, *Real* und *Double* erlaubt.

Eine spezielle Version dieses Befehls existiert noch für die Zuweisung der Adresse eines Symbols:

```
ASSIGN_CONST_COMMAND: local := <Symbol name>
```

Der lokalen Variablen wird die von dem angegebenen Linker-Symbol beschriebene Adresse zugewiesen.

### 7.2.3 Speicherzugriffe (`READ_MEM_COMMAND` und `WRITE_MEM_COMMAND`)

Für das Lesen von Werten aus dem Speichern und das Schreiben in den Speicher stehen zwei Befehle mit jeweils drei Varianten bereit:

```
READ_MEM_COMMAND:    local1 := [offset + local2]
READ_MEM_COMMAND:    local1 := [local2 + local3]
READ_MEM_COMMAND:    local := <Symbol name>
WRITE_MEM_COMMAND:   [offset + local1] := local2
WRITE_MEM_COMMAND:   [local1 + local2] := local3
WRITE_MEM_COMMAND:   <Symbol name> := local
```

Dabei bezeichnet *offset* einen konstanten Offset, der zu der Speicheradresse addiert wird, die durch eine lokale Variable angegeben wird. Die eckigen Klammern bedeuten, dass hier auf den Inhalt der angegebenen Speicheradresse zugegriffen wird.

Die jeweils ersten beiden Varianten entsprechen direkt der auf den meisten RISC-Architekturen vorhandenen Basisadressierung bzw. indizierten Adressierung.

Die dritte Variante dient zum Lesen bzw. Schreiben einer globalen Variablen, deren Adresse durch das angegebene Linkersymbol angegeben wird.

## 7.2.4 Berechnungen (ARITHMETIC\_COMMAND)

Für Arithmetik mit *Integer*-Werten gibt es einen Zwischencodebefehl mit den drei folgenden Varianten:

```
ARITHMETIC_COMMAND:  local1 := local2 <binary-op> local3
ARITHMETIC_COMMAND:  local1 := local2 <binary-op> constant
ARITHMETIC_COMMAND:  local1 := <unary-op> local2
```

Dabei steht *<binary-op>* für einen der Operatoren

```
add      sub      subf     mul      div      mod      and      nand
or       nor      xor      eqv     implies  nimplies shift_left shift_right
```

und *<unary-op>* steht für einen der beiden Operatoren

```
neg      not
```

In der zweiten Variante des Befehls steht *constant* für eine *Integer*-Konstante. Diese Variante kann für das Rechnen mit reellen Zahlen nicht verwendet werden. Die berechneten Funktionen entsprechen den Namen der Operatoren, also *add* für *Integer*-, *Real*- und *Double*-Addition, usw.

Der Operator *subf* steht für *subtract from*, die Subtraktion mit vertauschten Operatoren, und ist vor allem bei der Subtraktion von einer Konstanten interessant. So kann beispielsweise  $a := 1 - b$  auf manchen Architekturen, z. B. dem PowerPC [Motorola96], direkt mit einem Befehl berechnet werden. Auf SPARC [SPARC94] ist dies nicht möglich.

Die logischen Befehle *and*, *nand* usw. sind die bitweisen Operationen und arbeiten auf *Integer*-Werten.

## 7.2.5 Adressbestimmung (LOAD\_ADR\_COMMAND)

Ein Zwischencodebefehl ermöglicht die Bestimmung der Adresse einer lokalen Variablen:

```
LOAD_ADR_COMMAND:  local1 := ADR(local2)
```

Hier muss *local1* vom Typ Pointer sein. Der Befehl erzwingt, dass *local2* nicht in einem Register gehalten wird, sondern Speicher im Aktivierungsblock zugewiesen bekommt.

## 7.2.6 Aufrufe (CALL\_COMMAND)

Dieser Befehl erlaubt den Aufruf von Routinen, seine zwei Varianten entsprechen den Aufrufen von statisch und dynamisch gebundenen Features:

```
CALL_COMMAND:      call_static (<name>, local1, local2, ... [, Result_type])
CALL_COMMAND:      call_dynamic (rout, local1, local2, ... [, Result_type])
```

Das Argument *<name>* eines statischen Aufrufs gibt das Linkersymbol der Adresse des Routinencodes an. *rout* des dynamischen Aufrufs ist eine lokale Variable vom Pointer-typ, die die Anfangsadresse der Routine enthält.

Die folgenden Argumente sind die aktuellen Argumente der Routine. Für Funktionen muss zudem der Typ des Ergebnisses angegeben werden. Bei der Erzeugung des Zwischencodebefehls für den Aufruf einer Funktion wird eine lokale Variable erzeugt, die nach dem Aufruf das Ergebnis der Funktion enthält.

## 7.3 Kontrollstrukturen

Bisher können aus den Zwischencodebefehlen lediglich Grundblöcke ohne Verzweigungen erzeugt werden. Für bedingte Anweisungen, Schleifen, die *inspect*-Anweisung und die semi-strikten booleschen Operatoren werden jedoch Anweisungen benötigt, die den Kontrollfluss steuern.

Eine Folge von Zwischencodebefehlen ist ein Grundblock (*BASIC\_BLOCK*), wenn sie nicht durch eine Kontrollflussanweisung unterbrochen ist und es keine Sprünge an einzelne Befehle dieser Folge gibt, lediglich Sprünge an den ersten Befehl sind erlaubt.

Jeder Grundblock kann keinen, einen, zwei oder mehrere Nachfolgerblöcke besitzen, in die der Kontrollfluss nach der Ausführung der Befehle des Blocks führen kann. Die Nachfolgerblöcke werden durch ein eigenes Objekt beschrieben, das ein Nachfolger der abstrakten Klasse *BLOCK\_SUCCESSORS* ist. Um die Darstellung zu erleichtern, werden in den folgenden Abbildungen jedoch die Objekte für die Klassen *BASIC\_BLOCK* und die Nachfolgerklassen von *BLOCK\_SUCCESSORS* als ein Objekt gezeichnet, obwohl in Realität *BASIC\_BLOCK* einen Zeiger auf einen *BLOCK\_SUCCESSOR* enthält.

Im folgenden werden die unterschiedlichen Nachfolger eines Grundblocks beschrieben. In Klammern wieder die Namen der entsprechenden Klassen:

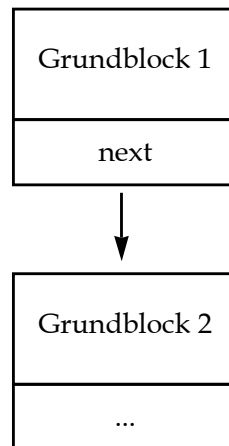
### 7.3.1 Kein Nachfolger (*NO\_SUCCESSORS*)

*NO\_SUCCESSORS* kennzeichnet den letzten Grundblock einer Routine. Der Kontrollfluss wird nach der Ausführung dieses Grundblocks an die aufrufende Routine übergeben.

### 7.3.2 Reihung (*ONE\_SUCCESSOR*)

Die einfachste Nachfolgerart für einen Grundblock ist die Reihung. Hier wird der Nachfolger durch ein Objekt der Klasse *ONE\_SUCCESSOR* beschrieben. *Bild 1* veranschaulicht dies.

Eine solche Reihung ist nur dann nötig, wenn der Kontrollfluss an die erste Anweisung des folgenden Grundblocks springen muss (z. B. durch eine Schleife), ansonsten könnten die beiden Blöcke zu einem einzigen Grundblock zusammengefügt werden.

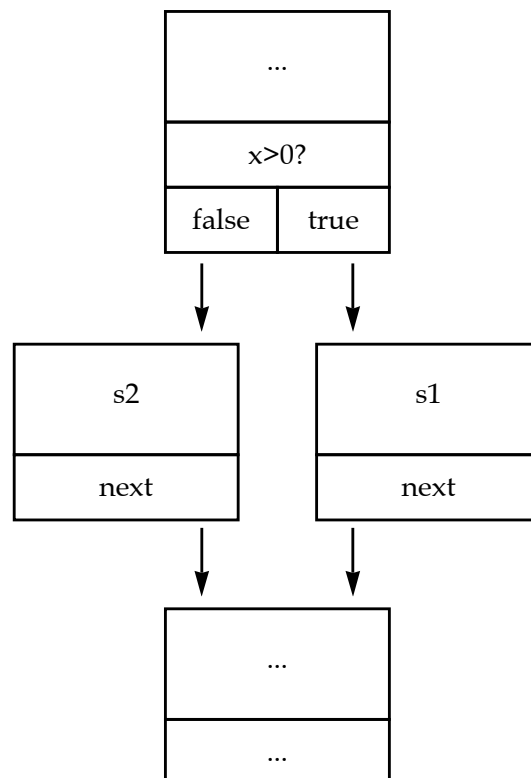


**Bild 1:** Reihung von zwei Grundblöcken durch *ONE\_SUCCESSOR*

### 7.3.3 Bedingte Anweisung (*TWO\_SUCCESSORS*)

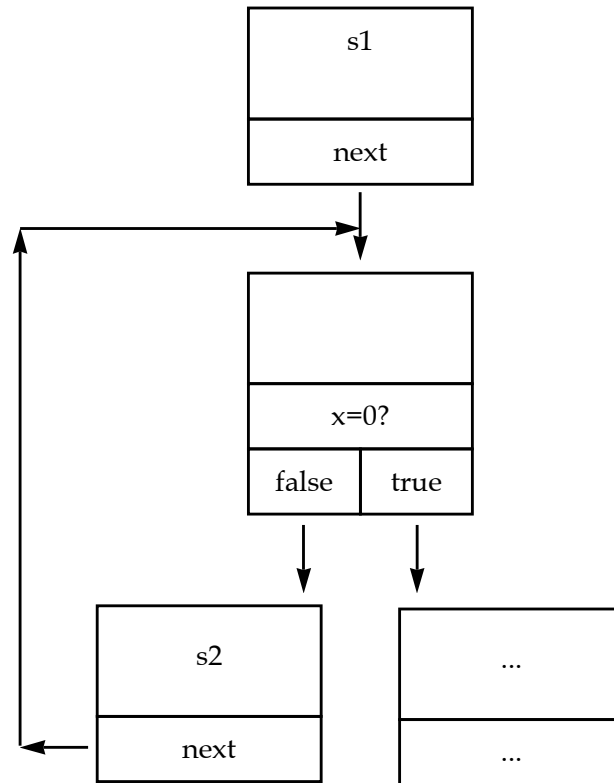
Für bedingte Anweisungen, Schleifen und die Auswertung von semi-strikten boole-schen Ausdrücken werden Grundblöcke mit zwei Folgeblöcken benötigt. Für die Ent-scheidung für einen dieser Blöcke muss eine Bedingung geprüft werden. Die Bedin-gung ist eine lokale Variable, die mit einer anderen lokalen Variablen oder einer Kon-stanten verglichen wird. Die möglichen Bedingungen sind gleich, ungleich, kleiner, kleiner gleich, größer und größer gleich. Je nach dem, ob die Bedingung erfüllt ist oder nicht, wird dann in einen der beiden Folgeblöcke verzweigt.

*Bild 2* veranschaulicht die Verwendung eines Blocks für die bedingte Anweisung *if x>0 then s1 else s2 end*.



**Bild 2:** Blöcke für die Anweisung *if x>0 then s1 else s2 end*

Die bedingten Blöcke können auch eingesetzt werden, um Zyklen von Blöcken zu erzeugen, wie dies bei Schleifen nötig ist. *Bild 3* zeigt ein Beispiel für die Schleife *from s1 until x=0 loop s2 end*. Dieses Beispiel zeigt auch, dass ein Grundblock auch aus einer leeren Folge von Zwischencodetebefehlen bestehen kann, wie hier der Block, der die Schleifenabbruchsbedingung testet.



**Bild 3:** Blöcke für die Schleife *from s1 until x=0 loop s2 end*

### 7.3.4 Multi\_branch (MULTI\_SUCCESORS)

Eine besondere Nachfolgerklasse von *BLOCK\_SUCCESSORS* steht für die Übersetzung der *Multi\_branch*-Anweisung bereit. Diese hat eine beliebige Anzahl an Folgeblöcken. Getestet wird eine lokale Variable des Typs *Integer* oder *Character*. Jeder Folgeblock ist einer Menge von Konstanten zugeordnet. Hat die lokale Variable einen der angegebenen Werte, so wird in den jeweiligen Folgeblock verzweigt. Einer der Folgeblöcke nimmt eine Sonderstellung ein: er wird angesprungen, wenn der Wert der getesteten Variablen in keiner der angegebenen Mengen der anderen Folgeblöcke vorkommt, er entspricht also dem *else*-Teil der *Multi\_branch*-Anweisung.

*Bild 4* zeigt die Struktur der Blöcke für die folgende Anweisung:

```

s0;
inspect q
when 1 then s1;
when 2,3 then s2;
else
  s3
end;
s4;
  
```

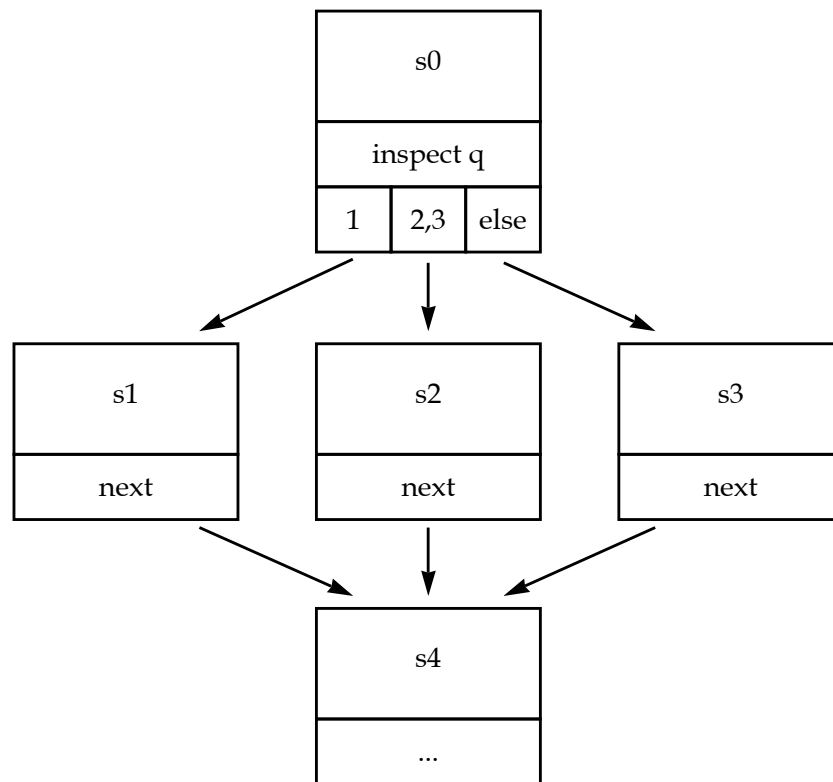


Bild 4: Block für Multi\_branch Anweisung

## 7.4 Zwischencodenerzeugung für Ausdrücke

Die Übersetzung der im abstrakten Syntaxbaum vorkommenden Ausdrücke soll hier mit den zu beachtenden Besonderheiten beschrieben werden.

### 7.4.1 Werte

Damit bei der Zwischencodenerzeugung für Ausdrücke die bereitgestellten Zwischencodebefehle auch gut ausgenutzt werden können, wird das Ergebnis eines Ausdrucks während der Zwischencodenerzeugung in einem Objekt einer Nachfolgerklasse der abstrakten Klasse *VALUE* gespeichert. Diese Objekte, im folgenden Werte genannt, existieren nur während der Zwischencodenerzeugung. Lediglich lokale Variablen sind spezielle Werte, die auch im erzeugten Zwischencode noch benutzt werden.

Alle konkreten Klassen für Werte implementieren die Routinen *need\_local* und *load\_address*, die eine lokale Variable zurückliefern, die den jeweiligen Wert bzw. die Adresse des Speicherbereichs des Wertes enthalten. Diese Routinen erzeugen den für das Laden des Wertes bzw. die Bestimmung der Adresse nötigen Zwischencode. Für boolesche Werte stehen noch weitere Routinen bereit.

Auf diese Weise wird die Zwischencodenerzeugung für Ausdrücke vereinfacht: Die von der Zwischencodenerzeugung für einen Ausdruck zurückgelieferten Werte erzeugen selbst den Code für den gewünschten Zugriff.

Die folgenden Arten von Werten werden bei der Zwischencodenerzeugung benutzt (in Klammern wieder die Namen der konkreten Klassen).

#### 7.4.1.1 Konstanten (MANIFEST\_CONSTANT\_VALUE)

Dieser Wert steht für eine Konstante. Dabei ist *MANIFEST\_CONSTANT\_VALUE* selbst eine abstrakte Klasse, von der die Klassen *BIT\_CONSTANT*, *INTEGER\_CONSTANT*, *REAL\_CONSTANT*, *CHARACTER\_CONSTANT*, *BOOLEAN\_CONSTANT* und *STRING\_CONSTANT* erben.

Diese verschiedenen Klassen sind direkt Teile des abstrakten Syntaxbaumes. Bei der Zwischencodierung müssen für ihre Werte keine neuen Objekte erzeugt werden. Die Werte entsprechen direkt den im Eiffel-Quelltext angegebenen Konstanten.

#### 7.4.1.2 Lokale Variablen (LOCAL\_VAR)

Die oben beschriebenen lokalen Variablen, die als Argumente für die Befehle des Zwischencodes benutzt werden, sind selbst auch Werte, sie sind als Nachfolger von *VALUE* definiert.

#### 7.4.1.3 Speicheradressen (OFFSET\_INDIRECT\_VALUE und INDEXED\_VALUE)

Den auf RISC-Architekturen gewöhnlich vorhandenen Adressierungsarten entsprechen diese beiden Werte. *OFFSET\_INDIRECT\_VALUE* beschreibt eine Speicheradresse, die durch einen Zeiger in einer lokalen Variablen und einem konstanten Offset bestimmt ist (in der Form  $[offset + local]$ ). *INDEXED\_VALUE* beschreibt eine Speicheradresse durch die Summe zweier lokaler Variablen (in der Form  $[local1 + local2]$ ).

Die Routine *need\_local* dieser Werte erzeugt das entsprechende *READ\_MEM\_COMMAND*, während *load\_address* die Adresse mit einer Addition (*ARITHMETIC\_COMMAND*) berechnet.

#### 7.4.1.4 Boolesche Werte (BOOLEAN\_VALUE)

Eine Sonderstellung bei der Zwischendarstellung von Ergebnissen eines Ausdrucks nehmen boolesche Werte ein. Der Grund hierfür sind die semi-strikten Operatoren *and then*, *or else* und *implies*, die effizient übersetzt werden sollen.

Ein boolescher Wert besteht aus einer Bedingung, einer *true*- und einer *false*-Liste. Die Bedingung vergleicht eine lokale Variable mit einer anderen Variablen oder einer Konstanten durch einem Vergleichsoperator. Dies entspricht exakt der Bedingung des oben beschriebenen Abschlusses für Grundblöcke *TWO\_SUCCESSORS*. Der beschriebene boolesche Wert ist genau dann *true*, wenn diese Bedingung wahr ist.

Die *true*- und *false*-Listen der booleschen Werte haben Einträge für Folgeblokeinträge in *TWO\_SUCCESSORS*-Objekten, bei denen der Nachfolgeblock noch nicht eingetragen wurde. Sobald für den Grundblock Zwischencode erzeugt wird, der ausgeführt werden soll, falls der boolesche Wert *true* ist, müssen alle in der *true*-Liste enthaltenen Einträge auf diesen Block gelenkt werden. Entsprechend muss bei der *false*-Liste vorgegangen werden, sobald der Grundblock für den Wert *false* erzeugt wird.

Für den Ausdruck

```
a or else b and then x>y
```

wird der folgende boolesche Wert erzeugt:

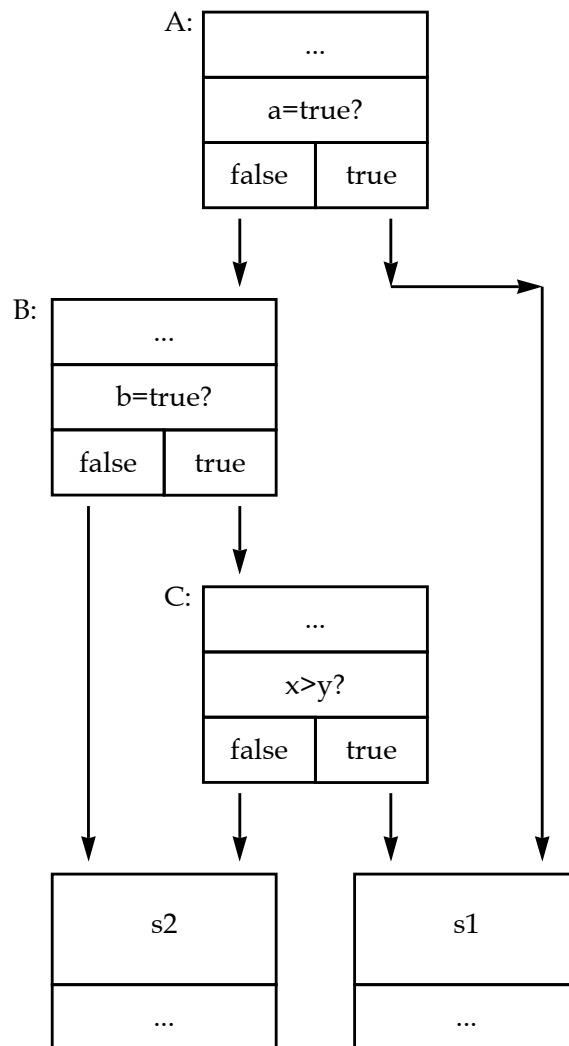
Bedingung =  $x > y$   
 true-Liste = true-Nachfolger von  $A$   
 false-Liste = false-Nachfolger von  $B$

Dabei sind  $A$  und  $B$  die in *Bild 5* dargestellten Blöcke, die bei der Übersetzung des Ausdrucks in Zwischencode erzeugt wurden. Die beiden Listen haben jeweils nur einen Eintrag.

*Bild 5* zeigt die gesamte Struktur der Grundblöcke, wie sie bei der Übersetzung der Anweisung

```
if a or else b and then x>0 then
  s1
else
  s2
end
```

entstehen würde.



**Bild 5:** Blöcke für die Anweisung *if a or else b and then x>0 then s1 else s2 end*

## 7.4.2 Aufrufe

### 7.4.2.1 Unqualifizierter Routinenaufruf

Wie in Kapitel 6 beschrieben, werden unqualifizierte Aufrufe mit *Current* als Zielobjekt stets statisch gebunden. Für einen Aufruf der Form  $f(a_1, a_2, \dots)$ , wobei  $a_1, a_2, \dots$  für beliebige Ausdrücke stehen, wird daher ein Aufruf

```
CALL_COMMAND:      call_static (<name>, Current, p1, p2, ...)
```

erzeugt. Dabei ist *<name>* entweder der Name der aufgerufenen Routine für expandierbare Objekte, wenn *Current* expandiert ist, oder derjenige für Referenzobjekte, wenn auch *Current* ein Referenzobjekt ist (für Klassen, die in einem System für expandierte Objekte und für Referenzobjekte benutzt werden, wird für beide Varianten getrennt Code erzeugt).

Als aktuelle Parameter werden die mit *need\_local* aus den Werten der Ausdrücke  $a_i$  bestimmten lokalen Variablen  $p_i$  übergeben.

### 7.4.2.2 Qualifizierter Routinenaufruf

Nun soll ein Aufruf der Form  $a.f(a_1, a_2, \dots)$  betrachtet werden. Dabei stehen das Zielobjekt  $a$  und die Parameter  $a_1, a_2, \dots$  für beliebige Ausdrücke.

Ist  $a$  expandiert, so kann ähnlich wie eben vorgegangen werden. Dann wird ein Aufruf der Form

```
CALL_COMMAND:      call_static (<name>, new_current, p1, p2, ...)
```

erzeugt. *<name>* gibt hier wiederum den Namen der Routine für expandierte Objekte an. *new\_current* ist eine neue lokale Variable vom Pointertyp. Ihr wird die mit *load\_address* bestimmte Adresse von  $a$  zugewiesen.

Handelt es sich bei  $a$  jedoch um einen Referenztyp, so muss die aufgerufene Routine dynamisch bestimmt werden. Dazu muss zunächst der Zeiger auf das Zielobjekt in eine lokale Variable geladen werden, was mit *need\_local*, angewendet auf den Wert des Ausdrucks  $a$ , geschieht. Das Resultat sei die lokale Variable  $p$ . Nun kann, wie im Kapitel über die Struktur des Zielcodes beschrieben, mit drei Speicherzugriffen die Adresse der aufzurufenden Routine aus dem Featuredeskriptor gelesen werden:

```
READ_MEM_COMMAND:  temp1 := [typedescriptor_offset + p]
READ_MEM_COMMAND:  temp2 := [color_of_static_type + temp1]
READ_MEM_COMMAND:  rout := [feature_number + temp2]
```

Die Werte *typedescriptor\_offset*, *color\_of\_static\_type* und *feature\_number* sind Konstanten. *typedescriptor\_offset* gibt den Offset des Typdeskriptorzeigers in einem Haldenobjekt an und kann direkt eingesetzt werden. *color\_of\_static\_type* ist die Färbung des statischen Typs des Zielobjektes. Dieser Wert ist auch konstant, kann allerdings erst bei der Erzeugung des Eiffel-Systems bestimmt werden, hier wird daher eine Referenz auf ein entsprechendes Linkersymbol erzeugt, damit der Linker die Konstante in den Code einträgt. *feature\_number* ist schließlich die Nummer der aufgerufenen Routine und eine bekannte Konstante.

Der eigentliche Aufruf geschieht nun mit dem Zwischencodebefehl

```
CALL_COMMAND:      call_dynamic (rout,p,p1,p2,...)
```

Dabei sind die  $p_i$  wie oben die mit *need\_local* in lokale Variablen geladenen Werte der Parameter  $a_i$ .

#### 7.4.2.3 Unqualifizierter Zugriff auf Attribute

Für einen unqualifizierten Zugriff auf ein Attribut von *Current*, also beispielsweise den Ausdruck *attr*, muss kein Code erzeugt werden. Es wird lediglich ein *OFFSET\_INDIRECT\_VALUE* zurückgeliefert mit der lokalen Variablen für *Current* und als Offset die Position des Attributes *attr* im aktuellen Objekt.

#### 7.4.2.4 Qualifizierter Zugriff auf Attribute

Der qualifizierte Zugriff auf ein Attribut *a.attr* ist auch einfach, wenn *a* expandiert ist. Dann steht die Position des Attributes statisch fest. Ist der Wert von *a* bereits ein *OFFSET\_INDIRECT\_VALUE*, so kann der Offset einfach um die Position von *attr* erhöht werden und dieser neue Wert ist der des Ausdrucks *a.attr*.

Ansonsten muss die Adresse von *a* mit *load\_address* aus dem Wert von *a* bestimmt werden, und aus dem Ergebnis und der Position von *attr* ein neuer Wert *OFFSET\_INDIRECT\_VALUE* erzeugt werden.

Ist *a* jedoch ein Referenzobjekt, so muss über den Typdeskriptor auf das Attribut zugegriffen werden, die Offsets der Attribute können in unterschiedlichen dynamischen Typen unterschiedlich sein. *a* wird dazu zuerst mit *need\_local* in die lokale Variable *p* geladen. Nun kann mit den folgenden drei Zwischencodebefehlen der Offset von *attr* bestimmt werden:

```
READ_MEM_COMMAND:  temp1 := [typedescriptor_offset + p]
READ_MEM_COMMAND:  temp2 := [color_of_static_type + temp1]
READ_MEM_COMMAND:  offset := [feature_number + temp2]
```

Die Konstanten *typedescriptor\_offset* und *color\_of\_static\_type* sind dieselben wie beim dynamischen Aufruf, während *feature\_number* nun natürlich die Nummer des Features *attr* angeben muss.

Als Ergebnis für den Ausdruck *a.attr* wird dann der *INDEXED\_VALUE* zurückgeliefert, der sich aus der Adresse des Objektes *a* und dem Offset von *attr* ergibt, also  $[p + offset]$ .

## Literatur

- [Hennessey96] John L. Hennessey and David A. Patterson: „Computer Architecture — A Quantitative Approach“, 2nd edition, Morgan Kaufmann Publishers, San Francisco, 1996
- [Motorola96] „PowerPC Microprocessor Family: The Programming Environments“, Motorola Inc., 1996
- [SPARC94] The SPARC Architecture Manual, Version 9, David L. Weaver / Tom Germond (Editors), SPARC International Inc., Prentice Hall, 1994

---

## 8. Zielcodeerzeugung

Der zielmaschinenabhängige Teil der Übersetzung, die Erzeugung der Maschinenbefehle für die Zielmaschine aus dem Zwischencode, soll hier nun beschrieben werden.

Die Zielcodeerzeugung beinhaltet zunächst die routinenweise Übertragung der einzelnen Befehle in die entsprechenden Maschinenbefehle, aber auch die Vergabe von Registern für Zwischenwerte und lokale Variablen und verschiedene Optimierungen wie Kopie-Propagierung und Instruction-Scheduling. Beim Design des Codeerzeugers wurde ein konsequent objektorientierter Ansatz gewählt. Die Zwischencodbefehle sind jeweils eigenständige Objekte, die sich selbst in verschiedenen Übersetzungsphasen in die Zielcodebefehle übersetzen und die über ihre Features Informationen für die nötigen Datenflussanalysen liefern.

Schließlich wird der erzeugte Code aller Routinen einer aktuellen Klasse (*ACTUAL\_CLASS*) zusammen mit den Featuredeskriptoren für alle Sichten auf diese Klasse in einer Objektdatei gespeichert.

### 8.1 Objektorientiertes Design

Die bei der Zielcodeerzeugung für die verschiedenen Zwischencodbefehle und Grundblocknachfolger durchzuführenden Aktionen werden nicht von globalen Routinen erledigt, sondern verteilt in Routinen der einzelnen Zwischencodbefehle und Grundblocknachfolger realisiert. Dafür implementieren die konkreten Zwischencodbefehle abstrakte Routinen der Klasse *COMMAND*, wie es für die jeweiligen Befehle nötig ist, und entsprechend implementieren die konkreten Grundblocknachfolger abstrakte Routinen der Klasse *BLOCK\_SUCCESSORS*.

Die in den verschiedenen Phasen der Zielcodeerzeugung aufgerufenen Routinen der Zwischencodbefehle sind *expand*, *get\_alive*, *get\_conflict\_matrix*, *remove\_assigns\_to\_dead* und *expand2*. Entsprechend implementieren Grundblocknachfolger die Routinen *expand*, *get\_alive*, *get\_conflict\_matrix* und *expand2*.

Auf diese Weise ist die Zielcodeerzeugung lokal zu den Zwischencodebefehlen und diese sind so leicht um weitere Befehle erweiterbar. Es müssen die abstrakten Routinen aus *COMMAND* nur entsprechend für die neuen Befehle implementiert werden, die Hauptroutinen des Codeerzeugers selbst sind von einer Erweiterung nicht betroffen.

Während der Zielcodeerzeugung werden in den Zwischencode neue Befehle eingefügt. Dies können auch Zwischencodebefehle sein, die nur auf einer bestimmten Zielarchitektur bestehen, wie beispielsweise *SETHI\_COMMAND* für SPARC. Diese Befehle des *Back-ends* müssen nur diejenigen der genannten abstrakten Routinen implementieren, die in den Phasen nach dem Einfügen in den Zwischencode noch aufgerufen werden.

## 8.2 Phasen der Zielcodeerzeugung

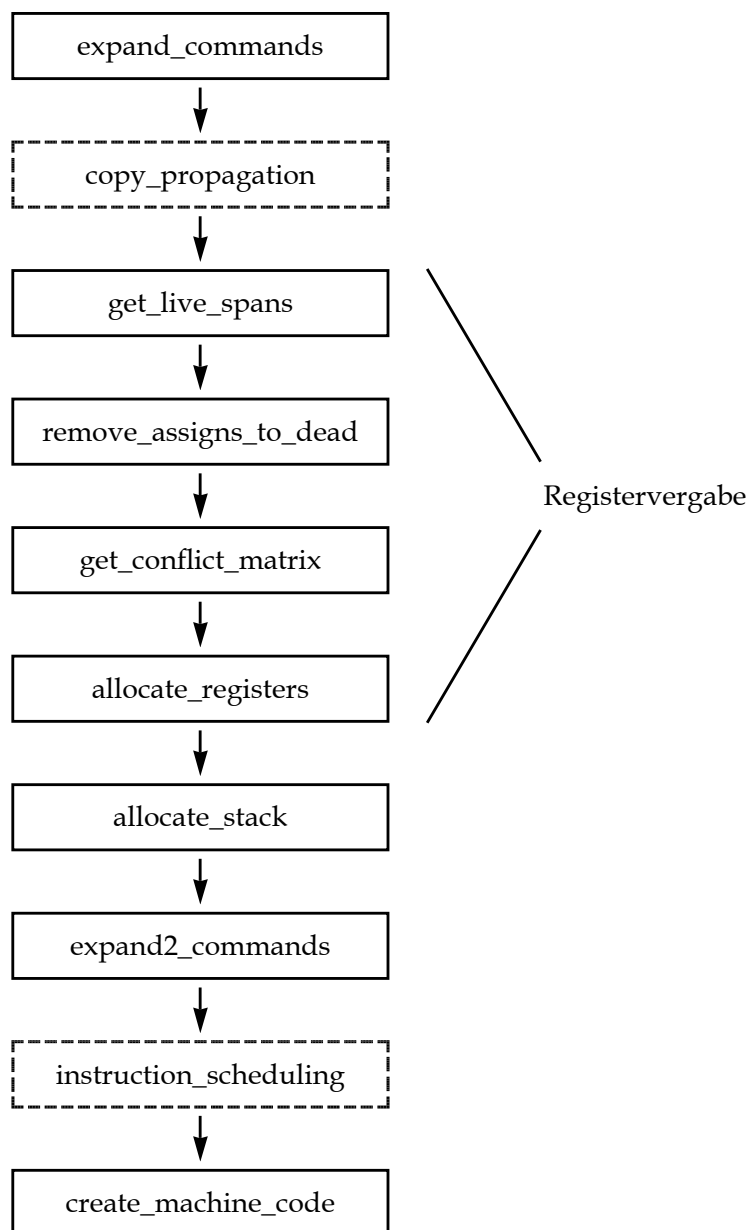


Bild 1: Die Phasen der Zielcodeerzeugung

Die Zielcodeerzeugung für jede zu übersetzende Routine gliedert sich in mehrere, zeitlich aufeinanderfolgende Phasen. In jeder Phase wird der Zwischencode weiter dem Code der Zielmaschine angepasst, oder es werden für die folgenden Phasen benötigte Informationen gesammelt. In den Phasen werden Zwischencodbefehle eingefügt, verändert oder auch gelöscht, bis nach der letzten Phase jeder verbleibende Zwischencodbefehl genau einem Maschinenbefehl der Zielarchitektur entspricht.

Bei einigen der Phasen werden zusätzlich Zwischencodbefehle für die *BLOCK\_SUCCESSORS* erzeugt, die jeden Grundblock abschließen. Der Zielcode einer Routine ist somit schließlich die Aneinanderreihung der Maschinenbefehle der Grundblöcke.

*Bild 1* zeigt die Phasen der Zielcodeerzeugung. Gestrichelt dargestellt sind dabei diejenigen Phasen, die bei der Erstellung dieser Arbeit noch nicht implementiert waren.

Für die Aktivierung der unterschiedlichen Phasen der Zielcodeerzeugung ist die Klasse *ROUTINE\_CODE* zuständig, die wiederum die den Phasen entsprechenden Routinen der Grundblöcke (Klasse *BASIC\_BLOCK*) aufruft.

Im folgenden werden die einzelnen Phasen beschrieben. Die verwendeten Beispiele betreffen den konkreten Codeerzeuger für SUN/SPARC [SPARC94].

### 8.2.1 Erste Expandierungsphase (*expand\_commands*)

In dieser Phase werden all diejenigen Zwischencodbefehle verändert oder ersetzt, für die schon vor der Registervergabe feststeht, dass sie nicht in einem einzigen Befehl der Zielarchitektur realisiert werden können. Der Grund hierfür ist meist das Fehlen des entsprechenden Befehls in der Zielarchitektur.

In dieser Phase werden beispielsweise verwendete Konstanten (etwa in Arithmetikbefehlen), die zu groß sind, um direkt im Maschinenbefehl gespeichert zu werden, durch temporäre Variablen ersetzt, die durch einen neu eingefügten Befehl zunächst geladen werden.

Für SPARC müssen zudem die Arithmetikbefehle *mul*, *div* und *rem* durch Aufrufe von Bibliotheksroutinen realisiert werden. Die entsprechenden Zwischencodbefehle werden also vollständig durch Funktionsaufrufe ersetzt.

Einige Optimierungen können in dieser Phase auch bereits erledigt werden, beispielsweise werden Multiplikationen mit konstanten Zweierpotenzen durch arithmetische Verschiebungen ersetzt.

Für den Zwischencodbefehl *CALL\_COMMAND* werden hier Befehle für die Übergabe der Argumente und die Rückgabe des Ergebnisses erzeugt, bei SPARC also nach dem Application Binary Interface [SPARC93] das Kopieren der Werte in die entsprechenden Register oder auf den Stapel und nach dem Aufruf das Auslesen des Ergebnisses aus dem jeweiligen Register.

Die *expand*-Phase betrifft auch die *BLOCK\_SUCCESSORS*-Objekte der Grundblöcke. Auch hier werden die Konstanten, die beispielsweise für die Prüfung einer Verzweigungsbedingung benötigt werden, durch zusätzliche Befehle in temporäre Variablen geladen, falls dies auf der Zielarchitektur nötig wird.

## 8.2.2 Kopie-Propagierung (copy\_propagation)

In dieser Phase sollen innerhalb eines Grundblocks unnötigerweise mehrfach in lokale Variablen geladene Konstanten optimiert werden. Zudem ist denkbar, dass mehrfach berechnete Zwischenergebnisse erkannt werden und der Code entsprechend optimiert wird (*Common subexpression elimination*).

Unnötige Zuweisungen zwischen lokalen Variablen ( $a := b$ ) sollen von dieser Phase jedoch nicht verändert werden, da dadurch weitere Optimierungen behindert werden könnten. Bei der Registervergabe wird in einem solchen Fall versucht, beiden Variablen dasselbe Register zuzuordnen und somit den Befehl unnötig zu machen.

Diese Phase ist derzeit noch nicht genauer spezifiziert und nicht implementiert.

## 8.2.3 Registervergabe

Für die Vergabe der Prozessorregister an die lokalen Variablen sind mehrere Phasen der Codeerzeugung nötig. Es werden zunächst die Lebenszeiten jeder lokalen Variablen bestimmt. Mit dieser Information kann dann eine Konfliktmatrix erstellt werden, mit deren Hilfe die Register den Variablen zugeordnet werden können.

### 8.2.3.1 Bestimmung der Lebenszeiten (get\_live\_spans)

Für die Bestimmung der Lebenszeiten ist eine Datenflussanalyse nötig, wie sie in [Aho86] oder [Plöder96] beschrieben wird. Zur Bestimmung der Lebenszeiten wird dabei iterativ jeder Grundblock rückwärts durchlaufen und, ausgehend von der Vereinigung der aktiven Variablen aller Nachfolgeböcke, die Menge der zu Beginn des Blocks aktiven Variablen bestimmt.

Jeder Zwischencodebefehl implementiert hierfür eine Routine *get\_alive*, die als Argument die Menge der nach diesem Befehl aktiven Variablen bekommt und daraus die vor diesem Befehl aktiven Variablen bestimmt, indem überschriebene Variablen aus der Menge entfernt und gelesene entsprechend eingefügt werden.

Die Bestimmung der Lebenszeiten ist unabhängig vom Zielcode, die Routinen werden daher schon im *Middle-end* und nicht erst im maschinenabhängigen *Back-end* des Compilers implementiert.

Eine Speicherung aller zwischen zwei Maschinenbefehlen aktiven Variablen erscheint als zu speicheraufwendig ( $O(n \cdot m)$  bei  $n$  lokalen Variablen in einer Routine mit  $m$  Zwischencodebefehlen). Stattdessen werden pro Grundblock nur die zu Beginn des Grundblocks existierenden aktiven Variablen gespeichert (damit ergibt sich der Speicheraufwand zu  $O(n \cdot p)$  wobei  $p$  die Anzahl der Grundblöcke pro Routine angibt). Bei Bedarf muss daher die Menge der zwischen zwei Befehlen aktiven Variablen neu berechnet werden.

Die Bestimmung der Lebenszeiten wird iterativ solange wiederholt, bis sich ein stabiler Zustand ergibt. Die Iteration terminiert, da es nur eine endliche Menge an lokalen Variablen gibt und die Mengen der aktiven Variablen sich nicht verkleinern können. Bei einer konkreten Übersetzung werden nur wenige Iterationen benötigt.

Als Beispiel für die Implementierung der Lebenszeitbestimmung hier die entsprechende Routine für den Zwischencodebefehl *ARITHMETIC\_COMMAND*:

```
get_alive (alive: SET) is
  do
    alive.exclude(dst.number)
    alive.include(src1.number);
    if src2 /= Void then
      alive.include(src2.number);
    end;
  end; -- get_alive
```

### 8.2.3.2 Zuweisungen an tote Variablen entfernen (*remove\_assigns\_to\_dead*)

Die Information über die Lebenszeit der Variablen kann gleich für eine Optimierung genutzt werden: Zuweisungen an tote (nicht aktive) Variablen, also solche Variablen, die vor der nächsten Überschreibung nicht mehr gelesen werden, können komplett entfallen. Diese Optimierung entfernt vor allem unnötige Variableninitialisierungen am Anfang einer Routine, die den lokalen Variablen ihre Default-Werte zuweisen.

Als Folge dieser Optimierung kann sich auch die Lebenszeit anderer Variablen verkürzen (wenn beispielsweise die Zuweisung  $a := b$  entfernt wird, und  $b$  sonst nirgends gelesen wird). Die dadurch zusätzlich möglichen Optimierungen sollen von diesem Compiler aber zunächst nicht berücksichtigt werden.

### 8.2.3.3 Bestimmung der Konfliktmatrix (*get\_conflict\_matrix*)

Nach der Bestimmung der Lebensbereiche der lokalen Variablen kann nun leicht die Konfliktmatrix bestimmt werden, also für jede Variable die Menge derjenigen Variablen, die zur gleichen Zeit aktiv sind.

Zur Speicherung der Konfliktmatrix wird ein zweidimensionales Feld von *BOOLEANS* benutzt. Dies führt zu dem zunächst hoch erscheinenden Speicheraufwand von  $O(n^2)$ , bei  $n$  lokalen Werten. Auch bei sehr aufwendigen Routinen ist  $n$  jedoch selten größer als 300 und der Aufwand noch erträglich. Sollte dies jedoch problematisch werden, so kann die Anzahl der betrachteten lokalen Variablen dadurch stark vermindert werden, dass für kurzlebige Zwischenergebnisse (und dies sind die meisten temporären Werte) eine spezielle Registervergabe vorgeschaltet wird oder eine bestimmte Anzahl der Prozessorregister so reserviert werden, dass diese Werte in der Konfliktmatrix nicht mehr berücksichtigt werden müssen.

Nach [Plöder96] ist es für das Erkennen eines Konfliktes ausreichend, die Zuweisungen an lokale Variablen zu betrachten. Ein Konflikt zwischen zwei Variablen  $a$  und  $b$  existiert genau dann, wenn eine Zuweisung an  $a$  existiert, während  $b$  aktiv ist, oder umgekehrt.

Die Eintragungen in die Konfliktmatrix werden von den Implementierungen der Routine *get\_conflict\_matrix* der einzelnen Zwischenbefehle vorgenommen.

Zusätzlich werden noch einige für eine gute Registervergabe nötige Informationen von *get\_conflict\_matrix* bestimmt:

Für jede Variable wird ein Zähler (*use\_count*) für jede Verwendung um ein Gewicht *weight* erhöht. Dabei ist das Gewicht innerhalb von Schleifen höher und innerhalb von

bedingten Anweisungen niedriger. Die Befehle desselben Grundblocks haben dasselbe Gewicht, das *get\_conflict\_matrix* als zusätzlicher Parameter übergeben wird. Variablen mit einem hohen Gewicht (*use\_count*) sollen bevorzugt bei der Registervergabe behandelt werden.

Zudem bestimmt *get\_conflict\_matrix* diejenigen Paare von lokalen Variablen, die bevorzugt dasselbe Register zugeordnet bekommen sollen. Dies ist bei einer Zuweisung  $a := b$  der Fall, die unnötig wird, sobald  $a$  und  $b$  in demselben Register gehalten werden.

Schließlich werden von *CALL\_COMMAND* alle während eines Routinenaufrufs aktiven Variablen als nicht-flüchtig markiert, da sie nicht in flüchtigen Registern gespeichert werden dürfen, sondern einen Aufruf unverändert überstehen müssen (flüchtige Register sind diejenigen Register, deren Inhalt bei einem Routinenaufruf verloren gehen kann, so wie in der *ABI* [SPARC93] definiert).

Als Beispiel für eine Implementierung von *get\_conflict\_matrix* wird hier der Code des Zwischencodebefehls *ASSIGN\_COMMAND* gezeigt:

```

get_conflict_matrix (code: ROUTINE_CODE;
                    alive: SET;
                    weight: INTEGER) is
do
    code.add_conflict(dst, alive);

    src.add_preferred_synonym(dst);
    dst.add_preferred_synonym(src);

    src.inc_use_count(weight);
    dst.inc_use_count(weight);

    get_alive(alive);
end; -- get_conflict_matrix

```

Zunächst wird hier der Konflikt zwischen dem Ziel der Zuweisung und allen derzeit aktiven Variablen mit *code.add\_conflict* registriert. Schließlich werden die Quell- und die Zielvariable gegenseitig als ein bevorzugtes Synonym markiert (*add\_preferred\_synonym*), dadurch bekommen sie bevorzugt dasselbe Register zugeordnet, was diese Zuweisung unnötig machen würde.

Das Gewicht der beiden beteiligten Variablen wird noch erhöht, bevor mit der schon oben beschriebenen Routine *get\_alive* die Menge der aktiven Variablen vor diesem Befehl bestimmt wird, die dann für den Aufruf von *get\_conflict\_matrix* des vorausgehenden Befehls verwendet wird (wie die Bestimmung der Lebenszeiten werden die Befehle eines Grundblocks bei der Bestimmung der Konfliktmatrix rückwärts durchgegangen).

#### 8.2.3.4 Registerallokation

Die Registerallokation ist ein Färbungsproblem im Konfliktgraphen der lokalen Variablen. Da die optimale Graphfärbung NP-vollständig ist, wird nicht versucht, eine optimale Lösung zu finden, jedoch sollte dennoch gewöhnlich eine gute Ausnutzung der Register erreicht werden.

Nach der Bestimmung der Konfliktmatrix, der nichtflüchtigen Variablen und der Gewichtungen (*use\_count*) der Variablen kann ohne weitere Betrachtung der einzelnen Befehle und Grundblöcke des Zwischencodes die Registervergabe für die lokalen Variablen erfolgen.

Dazu werden die lokalen Variablen zunächst nach ihren Gewichten sortiert – die wichtigsten lokalen Variablen sollen als erstes behandelt werden und somit die größten Chancen besitzen, ein Register zugeordnet zu bekommen.

Die Routine *allocate\_register\_for* versucht dann, jeder lokalen Variablen *l* ein möglichst gutes Register zuzuordnen. Es kommen alle verfügbaren Prozessorregister in Frage, die noch keiner anderen lokalen Variablen zugeordnet wurden, die mit *l* im Konflikt steht. Für nicht-flüchtige Variablen dürfen zudem nur nicht-flüchtige Register verwendet werden. Für Fließkommavariablen dürfen nur Fließkommaregister (oder Registerpaare für *DOUBLE*), für alle anderen Variablen nur *General-Purpose*-Register verwendet werden.

Als nächstes wird nun geprüft, ob eines der möglichen Register für *l* schon einer anderen Variablen zugeordnet wurde, die möglichst in demselben Register wie *l* gespeichert werden sollte (*preferred\_synonym*, um Zuweisungen einzusparen). Ist dies der Fall, so wird das betroffene Register benutzt.

Ansonsten wird eines der möglichen Register benutzt, vorzugsweise ein flüchtiges Register, da dieses für nicht-flüchtige Variablen nicht verwendet werden kann und daher weniger wertvoll ist. Es wäre für das *Instruction-Scheduling* (s. u.) sinnvoll, wenn nahe beieinanderliegende Befehle mit unterschiedlichen Registern arbeiten würden, um die Anzahl an Abhängigkeiten zwischen den Instruktionen zu verringern. Es wäre also sinnvoll, bei mehreren zur Wahl stehenden Registern möglichst eines zu wählen, das in der Umgebung der Befehle, die die Variable *l* benutzen, am wenigsten verwendet wird. Hierauf wird allerdings bisher keine Rücksicht genommen.

Der hier beschriebene Registerallokator kann in der SPARC-Implementierung bei der Übersetzung des Quelltextes des Compilers selbst allen lokalen und temporären Variablen, die in Registern gehalten werden können, auch ein Register zuordnen.

#### 8.2.4 Stapelallokation

All denjenigen lokalen Variablen, denen kein Register zugeordnet werden konnte, oder deren Speicheradresse benötigt wird, müssen nun einen Platz auf dem Stapel zugeordnet bekommen.

Der auf dem Stapel benötigte Speicher kann mit Hilfe der Konfliktmatrix minimiert werden. Da jedoch nur wenige Variablen auf dem Stapel gespeichert werden müssen, wurde hierauf zunächst keine Rücksicht genommen. Jeder Variablen wird ein eigener Bereich auf dem Stapel zugeordnet, unter Berücksichtigung der Alignment-Restriktionen der SPARC-Architektur.

#### 8.2.5 Zweite Expandierungsphase (*expand2\_commands*)

Aufgabe der zweiten Expandierungsphase ist es, aus den Befehlen des Zwischencodes eine Folge von Befehlen zu erzeugen, die alle direkt in jeweils genau einen Maschinenbefehl übersetzt werden können.

Dabei müssen vor allem für alle lokalen Variablen, denen kein Register zugeordnet werden konnte, zusätzliche Befehle zum Laden bzw. Speichern der Variablen eingefügt werden. Um diese zusätzlichen Befehle zu ermöglichen, wurden im *SPARC-Backend*

zwei *General-Purpose*- und zwei *Floating-Point*-Doppelregister reserviert, die bei der Registervergabe nicht zur Verfügung standen.

Neben den *COMMANDs* betrifft diese Expandierungsphase auch die *BLOCK\_SUCCES-SORS*: Für sie werden die nötigen Vergleichs- und Verzweigungsbefehle erzeugt und an das Ende des jeweiligen Grundblocks angehängt. Nach dieser Expandierungsphase spielen somit die *BLOCK\_SUCCES-SOR* keine Rolle mehr, die Grundblöcke bestehen nur noch aus einer Liste von Befehlen.

### 8.2.6 Instruction Scheduling (`instruction_scheduling`)

Da nun jeder Befehl des Zwischencodes genau einem Maschinenbefehl entspricht, kann die Reihenfolge der Befehle für einen konkreten Prozessor optimiert werden. Hierfür sollte ähnlich objektorientiert vorgegangen werden, wie bei der bisherigen Codeerzeugung: Die Zwischencodebefehle sind um Routinen zu erweitern, mit denen *set-use*-Abhängigkeiten zwischen ihnen erkannt werden können, um einen Abhängigkeitsgraphen zu erzeugen. Befehle sind so zu vertauschen, dass voneinander abhängige Befehle so weit auseinanderliegen, dass der abhängige Befehl erst nach der Latenz des ersten Befehls ausgeführt wird, während jedoch Befehle mit *set-set* und *use-set* Abhängigkeiten nicht vertauscht werden.

Eine Beschreibung des *set-use*-Algorithmus findet sich in [Wilhelm92]. Eine sehr brauchbare Beschreibung einer möglichen Implementierung des Algorithmus für die PowerPC-Prozessoren befindet sich in [IBM96]. Dort sind auch Tabellen mit den Latenzen der Maschinenbefehle der verschiedenen PowerPC Implementierungen angegeben.

Die Delay-Slots der Verzweigungs- und Call-Befehle der SPARC-Architektur können, wie in [Temp193] gezeigt, mit einem einfachen Algorithmus in den meisten Fällen sinnvoll ausgefüllt werden. Dies führt allerdings, wie in der gleichen Quelle angegeben, zu nur geringen Geschwindigkeitsvorteilen („*However, the average speed improvement and code length reduction is only about five percent.*“), so dass es sinnvoller erscheint, gleich einen leistungsfähigeren Algorithmus für das *Instruction Scheduling* zu implementieren („*Good instruction scheduling is critical.*“, [SPARC94]). Dafür war aber leider im Rahmen dieser Arbeit nicht ausreichend Zeit.

### 8.2.7 Maschinencodeerzeugung (`create_machine_code`)

Nachdem nun jeder Zwischencodebefehl genau einem Maschinenbefehl entspricht, kann direkt der Code für die Grundblöcke erzeugt werden. Die Grundblöcke werden dabei in der Reihenfolge, in der sie erzeugt wurden, in Maschinencode übersetzt (eine Umsortierung der Grundblöcke zum Einsparen von Verzweigungen erscheint mir nur dann sinnvoll, wenn Profilerinformationen über die am häufigsten ausgeführten Verzweigungen vorliegen würden).

Die Zwischencodebefehle erzeugen ihren eigenen Code selbst beim Aufruf ihrer Routine `create_machine_code`. Um die Maschinencodeerzeugung zu vereinfachen und Fehler zu vermeiden, stellt die Klasse `MACHINE_CODE` für die SPARC-Codeerzeugung Routinen zum Zusammenbasteln der Maschinenbefehle bereit. Diese Routinen setzen die Registernummern, Konstanten und verschiedene Flags an die richtigen Bitpositionen der Maschinenbefehle.

Lediglich für Vorwärtsverzweigungen kann nicht sofort Code erzeugt werden, diese Verzweigungen werden in einer speziellen (*Fixup*-) Liste gespeichert und nach der Codeerzeugung für die Routine an die korrekte Zieladresse angepasst.

### 8.3 Textliche Trennung der maschinenabhängigen Teile

Um eine leichte Portierung des Compilers auf unterschiedliche RISC-Architekturen zu ermöglichen und eine gleichzeitige Pflege aller Portierungen zu erlauben, ist es sehr wünschenswert, dass die maschinenabhängigen Teile des Compilers in getrennten Quelltexten zusammengefasst sind, und maschinenunabhängige Teile davon unberührt bleiben.

Dies erscheint auf den ersten Blick nicht ganz einfach, da die zunächst maschinenunabhängigen Befehle (*COMMAND* und *BLOCK\_SUCCESSORS*) des Zwischencodes sowohl Routinen enthalten, die von der Zielarchitektur unabhängig sind, als auch solche, die für jede Architektur unterschiedlich implementiert werden müssen.

So sind beispielsweise für den Zwischencodebefehl *ARITHMETIC\_COMMAND* die für die Berechnung der Konfliktmatrix nötigen Routinen *get\_alive* und *get\_conflict\_matrix* zielcodeunabhängig, die Expandierungsroutinen *expand* und *expand2* hängen jedoch stark von der Zielmaschine ab.

Um dieses Problem zu lösen, wurden die Quelltexte vieler Klassen zweigeteilt in eine abstrakte Klasse mit dem Präfix „*MIDDLE\_*“, die maschinenabhängige Routinen nicht implementiert. Die konkreten, maschinenabhängigen Klassen erben von den entsprechenden abstrakten Klassen und implementieren nur die maschinenabhängigen Routinen.

So implementiert die abstrakte Klasse *MIDDLE\_ARITHMETIC\_COMMAND* die von *COMMAND* geerbten abstrakten Routinen *get\_alive* und *get\_conflict\_matrix*. Die konkrete Klasse *ARITHMETIC\_COMMAND* erbt von ihr und implementiert die verbleibenden abstrakten Routinen *expand*, *expand2* und *create\_machine\_code*.

### 8.4 Erzeugung der Objektdatei

Bevor der für die Routinen einer Klasse erzeugte Maschinencode gespeichert werden kann, wird noch Code für die Initialisierung der Klasse erzeugt. Dieser Code alloziert die *STRING*-Objekte für alle in der Klasse verwendeten konstanten Zeichenketten.

Zudem sollen mit den Routinen der Klasse auch die Featuredeskriptoren für alle Vorgänger dieser Klasse erzeugt und gespeichert werden, da diese unabhängig von dem zu erzeugenden Eiffel-System bestimmt werden können. Dazu werden die bei der Validity-Überprüfung bestimmten Vorgängerlisten dieser Klasse benutzt.

Der Typdeskriptor kann noch nicht erzeugt werden, da die Positionen der Einträge von dem zu erzeugenden Eiffel-System abhängen, genauer von den bei der Färbung der Klassen des Systems den Vorgängerklassen dieser Klasse zugeordneten Farben.

Damit sind die in der Objektdatei gespeicherten Informationen nur von der aktuellen Klasse, ihren Vorgängerklassen und aller von dieser Klasse verwendeten Klassen abhängig. Eine spätere Version des Compilers kann dies erkennen und braucht bei unver-

änderten Vorgänger- und verwendeten Klassen diese Objektdatei nicht neu zu erzeugen, es kann dann sogar die Zwischencodeerzeugung und Validity-Überprüfung für diese Klasse entfallen.

Für die SUN/SPARC Implementierung wird eine Objektdatei im „*Executable and Linking Format*“ (ELF) erzeugt, wie es in [SUN93] beschrieben wird. Die für Symbole verwendeten Namen sind im Compiler-Quelltext in der Datei *DATATYPE\_SIZES.e* beschrieben.

## Literatur

- [Aho86] Alfred V. Aho, R. Sethi, J.D. Ullman: „COMPILERS: Principles, Techniques and Tools“, Bell Telephone Laboratories, 1986
- [IBM96] International Business Machines Corporation, Steve Hoxey, Faraydon Karim, Bill Hay, Hank Warren: „The PowerPC Compiler Writer’s Guide“, IBM Microelectronics Division, New York, 1996.
- [Plöder96] Prof. Dr. Erhard Plödereder: Skript zur Vorlesung „Compilerbau I“, Sommersemester 1996, Institut für Informatik, Universität Stuttgart
- [SPARC93] System V Application Binary Interface, SPARC Processor Supplement, 3rd edition, UNIX Press, Prentice Hall, 1993
- [SPARC94] The SPARC Architecture Manual, Version 9, David L. Weaver / Tom Germond (Editors), SPARC International Inc., Prentice Hall, 1994
- [SUN93] SunOS 5.3 Linker and Libraries Manual, SunSoft, Mountain View, 1993, Answerbook
- [Templ93] J. Templ: „SparcOberonImp.Text“, Sparc Oberon, ETH Zürich, 1993, <ftp://ftp.inf.ethz.ch>
- [Wilhelm92] Reinhard Wilhelm, Dieter Maurer: „Übersetzerbau“, Springer-Verlag, Berlin, 1992

---

## 9. Systemerzeugung

Die für die aktuellen Klassen eines Eiffel-Systems erzeugten Objektdateien, wie sie im vorangegangenen Kapitel beschrieben wurden, können unverändert für die Erzeugung unterschiedlicher Systeme benutzt werden. Alle Informationen, die für jedes Eiffel-System unterschiedlich sein können, werden in eine gesonderte Objektdatei gespeichert. Diese Datei enthält den für die Systemerzeugung (*root-creation*) nötigen Initialisierungscode, definiert die Färbungszahlen der wahren Klassen des Systems und enthält die von der Färbung abhängigen Typdeskriptoren. Diese Informationen werden vom SUN/SPARC *Back-end* des Compilers in der Objektdatei „*main.o*“ gespeichert.

### 9.1 Systemstart (*root-creation*)

Für den Start des Eiffel-Systems wird eine Maschinenroutine „*main*“ erzeugt. Diese Routine wird nach dem Linken des Systems zu einem Programm bei dessen Ausführung aufgerufen.

Sie muss zunächst die Initialisierungsroutinen aller am System beteiligten aktuellen Klassen aufrufen. Danach wird ein Objekt der Wurzelklasse alloziert und in die bei der Übersetzung angegebene *root-creation*-Prozedur der Wurzelklasse gesprungen und damit das Eiffel-System ausgeführt.

### 9.2 Globale Informationen

Das Ergebnis einer einmal ausgeführten *once*-Funktion und die boolesche Information, ob diese Funktion bereits ausgeführt wurde, muss für das System global gespeichert werden. Dies stellt sicher, dass *once*-Routinen auch in generischen Klassen nur einmal ausgeführt werden. Generische Klassen können durch unterschiedliche aktuelle generische Parameter zur mehrfachen Erzeugung von Code für die gleiche *once*-Routine in verschiedenen aktuellen Klassen führen. Es darf von diesen Routinen jedoch nur eine ausgeführt werden.

Aus diesem Grund werden die Ergebnisse von *once*-Funktionen und ein boolesches Flag, das anzeigt, ob die Funktion bereits ausgeführt wurde, global in Variablen von *main.o* gespeichert.

### 9.3 Typdeskriptoren

Vor der Erzeugung der Typdeskriptoren muss eine Färbung des Graphen der wahren Klassen des Systems erzeugt werden, wie im Kapitel über die Codestruktur beschrieben. Die aktuelle Implementierung des Compilers nummeriert die Klassen dazu schlicht durch, eine zukünftige Version muss hier eine bessere Färbung vornehmen, um bei der Erzeugung großer Systeme nicht unverhältnismäßig viel Speicher zu verschwenden.

Die Färbung des Graphen wird mit einem absoluten Symbol [SUN93] je wahrer Klasse in die Objektdatei „*main.o*“ geschrieben.

Nach der Färbung werden dann die Typdeskriptoren erzeugt, für jede wahre Klasse ein Deskriptor. Sie bestehen vor allem aus einem Feld von Zeigern auf die Featuredeskriptoren aller Vorgängerklassen. Diese Zeiger werden jeweils an der Position in das Feld eingetragen, die durch die Farbe der Vorgängerklasse angegeben wird.

Zusätzlich werden noch eine Reihe an Informationen über die jeweilige wahre Klasse im Typdeskriptor gespeichert: Die Größe der Objekte in Bytes, der Name der Klasse, evtl. eine Liste der aktuellen generischen Parameter, eine Liste der Attribute, die Farbe und eine eindeutige Nummer der Klasse. Der detaillierte Aufbau der Typdeskriptoren in der SUN/SPARC Implementierung kann der Datei *DATATYPE\_SIZES.e* des Compilerquelltextes entnommen werden.

### Literatur

- [SUN93] SunOS 5.3 Linker and Libraries Manual, SunSoft, Mountain View, 1993, Answerbook

*A run-time system supports execution of Eiffel systems, in particular efficient memory allocation for creation operations.*  
– Bertrand Meyer

---

## 10. Laufzeitsystem

Das Laufzeitsystem der aktuellen Version des Compilers für SUN/SPARC ist noch sehr einfach. Dies liegt vor allem an der fehlenden Implementierung des Garbage-Collectors, dem ein eigenes Kapitel gewidmet ist. Die Aufgabe des Laufzeitsystems besteht vor allem darin, eine zum Ablauf des Programms nötige Umgebung bereitzustellen. Dies umfasst die Speicherverwaltung und die Bereitstellung grundlegender Routinen, wie beispielsweise für Arithmetik. Aber auch die Erzeugung hilfreicher Meldungen im Falle eines Laufzeitfehlers ist Aufgabe des Laufzeitsystems.

Der Compiler erzeugt direkt Aufrufe der Routinen des Laufzeitsystems. Die Existenz dieser Routinen ist transparent für den Benutzer. Dieser braucht sie nicht zu kennen und benutzt sie nur implizit durch die Verwendung der Eiffel-Sprachkonstrukte.

Die Routinen des Eiffel-Laufzeitsystems werden in C in der Datei „*eiffel\_lib.c*“ implementiert. Die entsprechende Objektdatei „*eiffel\_lib.o*“ muss beim Linken jedes Eiffel-Systems mit eingebunden werden.

### 10.1 Allozierung von Speicher für Objekte

Die aktuelle Implementierung des Compilers enthält keinen Garbage-Collector. Eine mögliche Implementierung des Collectors wird später in dieser Arbeit detailliert beschrieben. Vorerst werden Objekte durch den Aufruf der C-Routine *eiffel\_new* aus „*eiffel\_lib.c*“ implementiert. Diese Routine erhält einen Zeiger auf den Typdeskriptor des neu zu allozierenden Objektes als Parameter. Der Typdeskriptor enthält unter anderem die Objektgröße in Bytes, so dass mittels *malloc* ein Block der entsprechenden Größe angefordert werden kann. *eiffel\_new* trägt den Zeiger auf den Typdeskriptor in das neu allozierte Objekt ein und gibt dann einen Zeiger auf das erste Attribut des Objektes zurück.

Ein neues Objekt wird auf der Halde entweder durch eine explizite Allokation der Form *!!r* oder *clone(r)*, oder implizit durch eine Zuweisung *r := x* (für *x* von expandiertem Typ und *r* von einem Referenztyp) erzeugt.

Im Fall der expliziten Allokation *!!r* und der Zuweisung *r := x* ist der Typ des neuen Objektes statisch bestimmbar, solange der Typ nicht von einem aktuellen generischen Parameter abhängt. In diesem Fall kann der Compiler direkt über ein Symbol der Objektdatei auf den Typdeskriptor zugreifen und einen Zeiger auf diesen an *eiffel\_new* übergeben. Eine Abhängigkeit von einem generischen Parameter entsteht z. B. in

```
class S[G]
  feature
    r: ARRAY[G];

    make is
      do
        !!r.make(1,10)
      end; -- make
  end -- S
```

Da für unterschiedliche generische Referenzparameter nicht mehrfach Code erzeugt wird, kann der Typ des von *!!r* allozierten Objekts erst zur Laufzeit bestimmt werden. Für generische Klassen enthält der Typdeskriptor daher eine Liste von Zeigern auf Typdeskriptoren für alle zu erzeugenden Objekte, die vom aktuellen generischen Parameter abhängen. In diesem Fall hätte der Typdeskriptor der wahren Klasse *S[A]* in dieser Liste einen Eintrag für *ARRAY[A]*, während der Deskriptor von *S[B[C]]* an der gleichen Stelle einen Verweis auf den Typdeskriptor von *ARRAY[B[C]]* bekäme.

Für die Allokation *!!r* kann nun Code erzeugt werden, der aus dem Typdeskriptor von *Current* den Zeiger auf den Typdeskriptor des neuen Objekts liest und mit *eiffel\_new* ein entsprechendes Objekt erzeugt.

Noch etwas anders muss bei *clone(r)* vorgegangen werden. Auch hier kann der Typ des neuen Objektes erst zur Laufzeit bestimmt werden; der Zeiger auf den Typdeskriptor ist jedoch identisch mit dem des Objektes, dessen Referenz *r* enthält. Es wird also direkt der Typdeskriptor des von *r* referenzierten Objektes an *eiffel\_new* übergeben.

## 10.2 Typüberprüfungen

Die einzige Anweisung, für die während der Laufzeit der Typ eines Referenzobjektes geprüft werden muss, ist das *Assignment\_attempt*, beispielsweise

```
r1 ?= r2;
```

Wie bei der Allokation *!!r* steht der statische Typ von *r1* zur Compilationszeit fest oder hängt von den aktuellen generischen Parametern ab. Ein Zeiger auf den Typdeskriptor dieses Typs kann also wie oben beschrieben bestimmt werden.

Damit die Zuweisung von *r2* an *r1* ausgeführt werden kann, muss der Typ von *r1* ein Vorgänger des Typs von *r2* sein. Um dies zu prüfen, bekommt jede wahre Klasse bei der Systemerzeugung eine eindeutige Nummer zugeordnet. Zudem wird in jedem Typdeskriptor eine sortierte Liste der Nummern aller Vorgängerklassen gespeichert.

Die Routine *eiffel\_conforms\_to\_number* aus „*eiffel\_lib.c*“ implementiert diesen Test: Sie erhält als Parameter die Nummer der wahren Klassen der Variablen auf der linken Seite des *Assignment\_Attempts* und als zweiten Parameter das Objekt der rechten Seite. Die Routine durchsucht die Liste der Vorgängerklassen nach der geforderten Nummer und gibt bei Misserfolg *Void*, ansonsten den Zeiger auf das zuzuweisende Objekt zurück.

Der Compiler braucht also für  $r1 \neq r2$ ; lediglich einen Aufruf von *eiffel\_conforms\_to\_number* zu erzeugen, und das Resultat an *r1* zuzuweisen.

### 10.3 Integerarithmetik

Multiplikation, Division und Divisionsrest von *INTEGER*-Werten können auf dem SPARC Prozessor [SPARC94] nicht direkt durch einen Maschinenbefehl implementiert werden. Stattdessen werden die auch von den auf der SUN verwendeten C-Compilern benutzten Bibliotheksroutinen „*.mul*“, „*.div*“ und „*.rem*“ aufgerufen.

### Literatur

- [SPARC94] The SPARC Architecture Manual, Version 9, David L. Weaver / tom Germond (Editors), SPARC International Inc., Prentice Hall, 1994

To favor the interoperability between implementations of Eiffel, it is necessary, along with a precise definition of the language, to have a well-defined set of libraries covering needs that are likely to arise in most applications. This library is known in Eiffel as the Kernel Library.  
 – NICE: The Eiffel Library Standard

## 11. Standardklassen

Einige der Routinen der Eiffel-Standardklassen [ELS95] können nicht direkt in Eiffel realisiert werden. Sie müssen entweder mit Hilfe von C- oder Assembler-Routinen implementiert werden, oder der Compiler muss Aufrufe dieser Routinen gesondert behandeln. Die Teile der Standardklassen, die keine besondere Behandlung durch den Compiler benötigen, werden hier nicht behandelt. Interessierte werden auf die Quelltexte dieser Klassen verwiesen.

### 11.1 Expandierte Standardklassen

Zu den expandierten Standardklassen gehören die Klassen *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE* und *POINTER*. Der Compiler fängt qualifizierte Aufrufe mit einem expandierten Zielobjekt (*Target*) der folgenden Routinen ab (ein Aufruf mit einem explizit angegebenem Zielobjekt heißt qualifiziert, Aufrufe von Präfix- und Infixoperatoren sind immer qualifiziert):

<u>Klasse</u>	<u>Routinen</u>
<i>BOOLEAN</i>	prefix "not", infix "and then", infix "or else", infix "and", infix "or", infix "implies"
<i>CHARACTER</i>	infix "<", infix "<=", infix ">", infix ">=", code, to_integer
<i>INTEGER</i>	prefix "+", prefix "-", infix "+", infix "-", infix "*", infix "/", infix "\\ \\", infix "<", infix "<=", infix ">", infix ">=", to_character, to_pointer
<i>REAL</i>	prefix "+", prefix "-", infix "+", infix "-", infix "*", infix "/", infix "<", infix "<=", infix ">", infix ">=", truncated_to_integer, to_double
<i>DOUBLE</i>	prefix "+", prefix "-", infix "+", infix "-", infix "*", infix "/", infix "<", infix "<=", infix ">", infix ">=", truncated_to_integer, to_real
<i>POINTER</i>	to_integer

Dabei wird bei der Zwischencodeerzeugung statt des Routinenaufrufs direkt der entsprechende Arithmetikbefehl oder der für den Vergleich oder die Typkonversion nötige Zwischencode erzeugt.

Die Infix-Operationen der Klasse *BOOLEAN* fallen durch ihre *semi-strictness* [Meyer92] etwas aus der Reihe: es müssen für ihre Berechnung entsprechende Verzweigungen erzeugt werden.

Damit diese Operationen auch dann noch korrekt funktionieren, wenn sie in Klassen verwendet werden, die von den Standardklassen erben, werden sie auch direkt in den Eiffel-Quelltexten der Standardklassen in Eiffel implementiert. So enthält der Quelltext der Klasse *INTEGER* die Routine

```
infix "+" (other: INTEGER): INTEGER is
do
    Result := item + other.item;
end; -- infix "+"
```

Für den Ausdruck *item + other.item* erzeugt der Compiler direkt Code und keinen Routinenaufruf, da das Attribut *item* vom Typ *INTEGER* ist. Der Aufruf des Features *infix "+"* einer Klasse, die von *INTEGER* erbt, wird vom Compiler nicht gesondert behandelt, sondern führt zu einem Aufruf dieser, in Eiffel implementierten Routine (wenn diese nicht redefiniert wurde). Es wird hier also auch korrekt die Summe berechnet. Entsprechend kann auch von den anderen Standardklassen geerbt werden.

## 11.2 Die Klasse GENERAL

Die Routinen *clone*, *standard\_clone*, *standard\_copy* und *standard\_is\_equal* der Klasse *GENERAL* können nicht direkt in Eiffel implementiert werden. Zudem sollte das Feature *Void* für eine bessere Effizienz direkt vom Compiler behandelt werden.

Diese Features werden alle gewöhnlich nur in unqualifizierten Aufrufen benutzt. Sie sind alle in *GENERAL* als *frozen* deklariert, können also in Nachfolgerklassen nicht redefiniert werden, sehr wohl können sie jedoch umbenannt werden. Um die eventuell umbenannten unqualifizierten Aufrufe dieser Features zu erkennen, fängt der Compiler alle unqualifizierten Aufrufe ab, deren Feature als *Origin* die Klasse *GENERAL* und als *Seed* einen der Bezeichner *clone*, *standard\_clone*, *standard\_copy*, *standard\_is\_equal* oder *Void* hat (die Definition von *Origin* und *Seed* ist in [Meyer92] gegeben).

Anstelle des Featureaufrufs erzeugt der Compiler dann direkt den entsprechenden Code, oder liefert im Falle von *Void* direkt den Wert des Nullzeigers.

In gewöhnlichen Eiffel-Programmen ist nicht zu erwarten, dass qualifizierte Aufrufe dieser Routinen vorkommen, da sie nicht vom Zielobjekt abhängen. Dennoch müssen auch qualifizierte Aufrufe korrekt übersetzt werden. Diese Aufrufe werden vom Compiler nicht gesondert behandelt, hierfür wurden die Routinen im Quelltext der Klasse *GENERAL* mit Hilfe eines unqualifizierten Aufrufs implementiert. Als Beispiel hierfür soll der Text für *clone* genügen:

```

frozen clone (other: GENERAL): like other is
do
    Result := clone(other);
ensure
    equal: equal(Result, other);
end; -- clone

```

Dies führt nicht zu einer endlosen Rekursion, da der unqualifizierte Aufruf *clone(other)* direkt vom Compiler implementiert wird.

## 11.3 Die Klasse ARRAY

### 11.3.1 Zugriff auf Elemente des Feldes

Die Routinen *put*, *infix "@"* und *item* der Standardklasse *ARRAY* können nicht direkt in Eiffel implementiert werden. Da hier eine hohe Effizienz wichtig ist, sollte für sie direkt vom Compiler Code erzeugt werden.

Die Implementierung der Routinen wird dadurch erschwert, dass für Ausführung dieser Routinen auf Attribute des *ARRAY*-Zielobjektes zugegriffen werden muss. Dieses Objekt kann auch ein (möglicherweise mehrfacher) Nachfolger der Klasse *ARRAY* sein, wie beispielsweise in

```

class MULTI_ARRAY

inherit
    ARRAY[INTEGER]
        rename
            put as put1,
            infix "@" as infix "@1",
            item as item1,
            storage as storage1,
            ...
        end;
    ARRAY[BOOLEAN]
        rename
            put as put2,
            infix "@" as infix "@2",
            item as item2,
            storage as storage2,
            ...
        end;
    ...

end; -- MULTI_ARRAY

```

Das von *put*, *infix "@"* und *item* benutzte Attribut *storage*, das einen Zeiger auf den Speicherbereich der Feldelemente enthält, kann also nicht an einer festen Position in den Objekten aller Nachfolgeklassen von *ARRAY* stehen, da es in Objekten einer Klasse wie *MULTI\_ARRAY* sogar doppelt vorkommen muss.

Damit auch in komplizierten Fällen wie diesem Feldzugriffe korrekt übersetzt werden, erzeugt der Compiler den Code für die Zugriffsroutinen direkt in der Klasse *ARRAY*, ihre Implementierung im Quelltext von *ARRAY* ist leer. Dieser Code wird für Nachfolgeklassen von *ARRAY* bei Bedarf dupliziert, so dass immer auf das korrekte *storage*-Attribut zugegriffen wird.

Aufrufe der Routinen *put*, *infix "@"* und *item* werden vom Compiler nicht abgefangen, es wird ein normaler, dynamisch gebundener Aufruf erzeugt. Dies ist etwas unbefriedigend, es wäre wünschenswert, wenn Feldzugriffe vom Compiler optimiert werden und nicht zu einem Aufruf für jeden Zugriff führen.

Eine zukünftige Version des Compilers kann die Aufrufe einiger oder aller Feldzugriffe durch direkten Code ersetzen: In einem System, in dem keine Klasse von *ARRAY* erbt, ist dies stets möglich. Ansonsten kann mit einer Datenflussanalyse bestimmt werden, welche Feldzugriffe keinesfalls auf eine Nachfolgerklasse von *ARRAY* zugreifen. Diese Aufrufe können durch den direkten Zugriff ersetzt werden.

### 11.3.2 Verwaltung des Speichers für die Feldelemente

Die Allokation des für die Feldelemente nötigen Speichers kann im Quelltext der Klasse *ARRAY* mit Hilfe der C-Bibliotheksroutinen *malloc*, *realloc*, *memset* und *memcpy* in Eiffel implementiert werden. Hierzu muss nur eine Möglichkeit bereitgestellt werden, mit der die Speichergröße des aktuellen generischen Parameters von *ARRAY[G]* bestimmt werden kann.

Hierfür wird in *ARRAY* ein Feature *element\_size* definiert. Unqualifizierte Aufrufe von *element\_size* (genauer: eines Features mit *ARRAY* als *Origin* und dem *Seed element\_size*) werden vom Compiler direkt implementiert, ähnlich wie dies bei den oben beschriebenen Routinen der Klasse *GENERAL* geschieht. Die Aufrufe werden durch die entsprechende Konstante des aktuellen generischen Parameters ersetzt.

## 11.4 Die Klasse *STRING*

Durch die feste Größe der Elemente (*CHARACTER*), aus denen Zeichenketten bestehen, können alle Features der Standardklasse *STRING* direkt in Eiffel mit Hilfe von wenigen C-Bibliotheksroutinen implementiert werden. Eine Unterstützung des Compilers ist nicht nötig, es ist jedoch wünschenswert, dass eine zukünftige Version des Compilers Zugriffe auf die einzelnen Zeichen einer Zeichenkette optimiert. Allerdings treten hier die gleichen Probleme wie beim Zugriff auf die Elemente eines *ARRAYs* auf, es darf also nur in Systemen optimiert werden, in denen keine Klasse von *STRING* erbt, oder es ist eine aufwendige Datenflussanalyse notwendig.

## Literatur

- [ELS95] Nonprofit International Consortium for Eiffel (NICE): „The Eiffel Library Standard, Vintage 95“, Version 8, <ftp://ftp.eiffel.com>, June 4, 1995
- [Meyer92] Bertrand Meyer: „Eiffel: The Language“, Prentice Hall International (UK) Ltd, Hertfordshire, 1992

*Its advantage is that replication of management algorithms and premature partitioning of resources are avoided. The disadvantage is that management algorithms are fixed once and forever and remain the same for all applications. The success of a centralized resource management therefore depends crucially on its flexibility and its efficient implementation.*

*– Niklaus Wirth*

---

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

Ziel soll es sein, einen effizienten Garbage-Collector für die Implementierung von Sprachen wie Eiffel zu beschreiben. Eiffel ist von Anfang an als Sprache mit einem Garbage-Collector konzipiert worden, was die Implementierung des Collectors erleichtert.

### 12.1 Grundüberlegungen

Als Grundlage dienen die in [Dijkstra78] und [Wilson93] beschriebenen Garbage-Collection-Algorithmen. Ein paralleler Garbage-Collector, der nicht durch spezielle Hardware unterstützt wird, benötigt die Kooperation des Mutators, also desjenigen Prozesses, der Speicher anfordert und den Speichergraphen verändert. Dabei muss der Mutator gewöhnlich bei Zeigerzuweisungen der Form

```
a := b
```

eine atomische Anweisung der Form

```
if b /= Void then
  Markiere b
end;
a := b;
```

erzeugen. Diese kann insbesondere auf modernen RISC-Prozessoren nicht einfach als atomische Anweisung implementiert werden. Wird für den parallelen Garbage-Collector-Prozess ein Betriebssystemprozess verwendet, so muss über einen zusätzlichen Schutzmechanismus diese Anweisungsfolge geschützt werden (sonst könnte der Collector die Markierung des Objektes *b* löschen und damit die Zuweisung illegal machen, s. u.). Dies geschieht z. B. folgendermaßen:

```
if b /= Void then
  Protect b
  Markiere b
end
a := b
if b /= Void then
  UnProtect b
end
```

Zeigerzuweisungen gehören zu den häufigsten in Eiffel-Programmen vorkommenden Operationen. Sie kommen nicht nur explizit in Zuweisungsanweisungen vor, sondern auch bei der Parameterübergabe beim Routinenaufruf, bei der Rückgabe des Funktionsresultats und überall dort, wo der Compiler Zwischenergebnisse, die Zeiger sind, in temporären Variablen speichert.

Es liegt uns also viel daran, den Aufwand für Zeigerzuweisungen so gering wie möglich zu halten.

Der hier vorgestellte Garbage-Collector soll daher nicht als eigenständiger Prozess ablaufen, wie diese z. B. unter Unix bereitgestellt werden. Stattdessen sollen Collector und Mutator als Coroutinen zusammenarbeiten, insbesondere soll der Mutator häufig genug einen Wechsel zum Collector erlauben.

### 12.2 Algorithmus

Der in [Dijkstra78] verwendete Tricolor-Algorithmus soll verwendet werden. Danach sind alle allozierten Objekte immer mit einer der Farben *weiß*, *grau* oder *schwarz* gefärbt. Die Objekte befinden sich also beispielsweise in einer der drei Listen *Weiß*, *Grau* oder *Schwarz*. Die Farben haben dabei folgende Bedeutung:

<i>Weiß</i>	Dieses Objekt ist nicht erreichbar von den bisher vom Collector untersuchten Objekten. Am Ende eines Collector-Zyklus sind weiße Objekte die gefundenen unerreichbaren Objekte, deren Speicher freigegeben werden kann.
<i>Grau</i>	Dies ist ein als erreichbar erkanntes Objekt, das vom Collector noch untersucht werden muss, der Collector muss also noch bestimmen, welche Objekte von diesem Objekt aus erreichbar sind.
<i>Schwarz</i>	Dies ist ein erreichbares Objekt, das vom Collector bereits untersucht wurde.

Um die Korrektheit des Collectors sicherzustellen, müssen der Collector und der Mutator stets das Bestehen der folgenden Invarianten garantieren:

<i>INV</i>	Kein schwarzes Objekt enthält Referenzen auf ein weißes Objekt.
------------	---

#### 12.2.1 Der Collectorzyklus

Der Collector läuft als eine Endlosschleife parallel zum Mutator, in der immer wieder folgendes geschieht:

1. Alle Objekte werden weiß markiert. Da keine schwarzen Objekte existieren, ist *INV* erfüllt.
2. Das Wurzel-Objekt oder die Wurzel-Objekte werden grau markiert.
3. Solange graue Objekte existieren, nehme ein graues Objekt *g* und
  - a) Für alle Objekte *r*, auf die *g* Referenzen enthält, markiere *r* grau, falls *r* weiß ist.
  - b) Markiere *g* schwarz. *INV* bleibt erfüllt, da eben alle von *g* erreichbaren weißen Objekte grau markiert wurden.
4. Nehme die weißen Objekte in die Liste der freien Objekte.

Im Schritt 4 ist sichergestellt, dass die weißen Objekte unerreichbar sind, da nach der Schleife in Schritt 3 nur noch schwarze und weiße Objekte existieren, die Wurzelobjekte in Schritt 2 grau markiert wurden und daher nun schwarz sind und nach *INV* keine schwarzen Objekte Referenzen auf weiße enthalten dürfen.

Die Terminierung des Algorithmus ist sichergestellt, da in Schritt 3 in jedem Durchgang ein Objekt schwarz markiert wird, diese Markierung nie entfernt wird, und es nur endlich viele Objekte gibt.

### 12.2.2 Aufgaben des Mutators

Es muss nun noch sichergestellt werden, dass *INV* durch den Mutator nicht verletzt wird.

#### 12.2.2.1 Zeigerzuweisungen

Die wichtigste Situation, die hierbei zu beachten ist, ist die Zuweisung eines Zeigers an ein Attribut eines Objekts, in Eiffel beispielsweise durch eine Anweisung

```
a := b
```

wobei *a* und *b* Zeiger sind und *a* ein Attribut von *Current* ist. Falls *Current* schwarz und *b* weiß markiert ist, wird *INV* verletzt. Es muss daher für die Zuweisung Code der Form

```
if (Current ist schwarz) and then
  (b /= Void) and then
  (b ist weiß)
then
  markiere b grau
end
a := b
```

erzeugt werden.

Andere Situationen, in denen die Einhaltung von *INV* berücksichtigt werden muss, sind das Allokieren neuer Objekte und die Veränderung der Liste der Wurzelobjekte, speziell das Anlegen und Zerstören von Aktivierungsblöcken von Routinen.

#### 12.2.2.2 Erzeugen von Objekten

Bei der Erzeugung eines Objekts durch eine Anweisung wie

```
!!a ,
```

bei der der neue Zeiger einem Attribut von *Current* zugewiesen wird, kann *INV* einfach dadurch erfüllt werden, dass neu erzeugte Objekte gleich schwarz markiert werden.

Dies führt aber auf jeden Fall dazu, dass neu erzeugte Objekte erst mit dem nächsten Garbage-Collector-Zyklus freigegeben werden können. Besser wäre eine Markierung in Abhängigkeit vom aktuellen Zustand von *Current*, also beispielsweise:

```
!!a
if Current ist schwarz then
  markiere a schwarz
else
  markiere a weiß
end
```

### 12.2.2.3 Globale Variablen

Wird einer globalen Variablen ein Zeiger auf ein Objekt zugewiesen, so entsteht ein neues Wurzelobjekt. Hat der Collector die Wurzelobjekte bereits untersucht, so muss dieses Objekt auch markiert werden. In Eiffel gibt es keine expliziten globalen Variablen, implizit werden diese jedoch bei *once*-Routinen verwendet, z. B. in

```
beispiel : C is
once
...
  Result := a;
...
end; -- Beispiel
```

wird Code der Form

```
if Beispiel_FirstCall then
...
  Result := a;
...
  Beispiel_Result := Result;
  Beispiel_FirstCall := false;
else
  Result := Beispiel_Result;
end;
```

erzeugt. Dabei sind *Beispiel\_Result* und *Beispiel\_FirstCall* globale Variablen. Der an *Beispiel\_Result* zugewiesene Wert muss in die *Grau*-Liste aufgenommen werden, falls er weiß ist, da der Collector die globalen Variablen schon betrachtet haben könnte. Für die Zuweisung an *Beispiel\_Result* muss also folgendermaßen vorgegangen werden:

```
if (Result /= Void) and then
  (Result ist weiß)
then
  markiere Result grau
end
Beispiel_Result := Result;
```

Die in den Aktivierungsblöcken gespeicherten lokalen (und temporären) Variablen können genauso behandelt werden. Für eine bessere Effizienz sollen lokale Variablen jedoch wie folgt gesondert behandelt werden:

### 12.2.3 Lokale Variablen

Lokale Variablen sind die in den Aktivierungsblöcken von Routinen gespeicherten Variablen. Da Zuweisungen an lokale Variablen der häufigste Zuweisungstyp sind, möch-

ten wir hier gerne auf den zusätzlichen Markierungscode verzichten. Wenn der Collector lokale Variablen ebenfalls gesondert behandelt, ist dies auch möglich.

[Wilson93] schlägt vor, die auf dem Stapel gespeicherten Objekte erst so spät wie möglich vom Collector zu betrachten. So können lokal erzeugte, kurzlebige Objekte möglicherweise noch in demselben Collectorzyklus freigegeben werden, in dem sie erzeugt wurden.

In Eiffel können keine Aliasse für lokale Variablen erzeugt werden, es gibt keine Möglichkeit, die Adresse einer lokalen Variablen zu bestimmen (selbst der Adressoperator "\$" für *external calls* darf nicht auf lokale Variablen angewendet werden).

Dadurch ist sichergestellt, dass lokale Variablen in Aktivierungsblöcken, deren Routinen nicht aktiv sind, sich nicht ändern. Der Collector kann also bei der Untersuchung der äußersten (ältesten) Aktivierungsblöcke beginnen. Gibt er die Kontrolle an den Mutator, so können sich bis zur nächsten Aktivierung des Collectors lediglich die Variablen derjenigen Aktivierungsblöcke ändern, deren Routinen aktiv waren.

Der Collector setzt daher eine globale Variable *ExaminedAB* auf die Adresse des jüngsten untersuchten Aktivierungsblocks. Bekommt der Collector die Kontrolle wieder, so erwartet er, dass der Aktivierungsblock, auf den *ExaminedAB* zeigt, und alle älteren Aktivierungsblöcke bereits untersucht wurden und sich nicht verändert haben.

Sobald im Mutator die Routine, deren Aktivierungsblock *ExaminedAB* ist, aktiv wird, muss *ExaminedAB* auf den Aktivierungsblock der dynamisch umschließenden Routine gesetzt werden.

### 12.2.4 Implementierung

Eine mögliche Implementierung des hier beschriebenen Algorithmus in Pseudo-Eiffel-Code wird ausführlich in *Anhang A* gezeigt und erklärt.

## 12.3 Aktivierung des Collectors und Zeitaufteilung

Ein Problem bei parallelen Garbage-Collectoren ist die Zeitaufteilung zwischen Collector- und Mutator-Prozess. Dabei sollte keine Rechenzeit durch den Collector verschwendet werden, andererseits darf der Collector nicht zuwenig Rechenzeit bekommen. Die Situation, in der das System wegen Speichermangels für einen kompletten Collectorzyklus angehalten werden muss, soll vermieden werden.

In 12.3.2 wird ein Verfahren vorgestellt, bei dem dem Collector nur wenig Rechenzeit zugeteilt wird, aber dennoch sichergestellt ist, dass die Aktivierungszeit des Collectors beschränkt und klein ist, solange der Mutator nur einen bestimmten Anteil des Gesamtspeichers als erreichbare Objekte speichert.

Folgende Begriffe werden im folgenden häufig benötigt:

*Allozierte Objekte*      Alle Objekte, für die Speicher alloziert wurde und die vom Collector noch nicht als unerreichbar erkannt wurden.

<i>Erreichbare Objekte</i>	Alle allozierten Objekte, die entweder Wurzelobjekte sind oder auf die andere erreichbare Objekten Referenzen besitzen. Dies ist also der derzeit vom Mutator benötigte Speicher.
<i>Unerreichbare Objekte</i>	Diejenigen Objekte, die nicht erreichbar sind.
<i>Freie Objekte</i>	Diejenigen unerreichbaren Objekte, die vom Collector als unerreichbar erkannt wurden und deren Speicher daher freigegeben wurde. Dies ist gerade die Komplementmenge der allozierten Objekte.

Für den Wechsel zwischen Mutator- und Collectorprozess sind vor allem zwei Lösungen denkbar: Der Mutator könnte einen Wechsel zum Collector an vielen „Sollbruchstellen“ im Programm ermöglichen, und damit den Collector quasi als einen parallelen Prozess laufen lassen, oder die Übergabe der Kontrolle an den Collector könnte lediglich beim Allozieren neuer Objekte geschehen.

### 12.3.1 Wechsel zwischen Mutator und Collector an Sollbruchstellen

Ich möchte zunächst die Variante betrachten, bei der der Mutator in Zeitintervallen, die eine bestimmte Länge nicht überschreiten dürfen, den Wechsel zum Collectorprozess ermöglicht. Dazu soll der Compiler im erzeugten Code ausreichend viele Anweisungen der Form

```
if Switch then
  To_Collector();
end
```

erzeugen. Insbesondere müssen diese Anweisungen innerhalb von Schleifen, deren Durchlaufzahl nicht statisch bekannt ist, und in rekursiven Routinen vorkommen. Der boolesche Wert *Switch* soll vom Laufzeitsystem nach einer gewissen Zeit gesetzt werden, z. B. durch eine zeitabhängig ausgelöste Exception.

Entsprechend muss der Collector regelmäßig, beispielsweise nach jedem Untersuchen eines Objektes, einen Wechsel zurück zum Mutator ermöglichen, damit dieser nicht merklich verzögert wird.

Nun stellt sich die Frage, wie der Prozessverwalter im Laufzeitsystem die Rechenzeit zwischen Collector und Mutator sinnvoll aufteilen soll. Am einfachsten wäre ein festes Zeitverhältnis, beispielsweise 9:1, wobei der Mutator 90% und der Collector 10% der Rechenzeit bekommen. In den meisten Fällen wäre dies jedoch unbefriedigend, da der Collector meist mehr Zeit verbraucht, als nötig, andererseits kann es jedoch auch zu Situationen kommen, in denen der Speichergraph so schnell wächst, dass der Collector nicht ausreichend Zeit hat und schließlich das System wegen Speichermangels für einen kompletten Collectorzyklus anhalten muss.

Es sind flexiblere Modelle denkbar, in denen beispielsweise dem Collector abhängig von Menge an belegtem und insgesamt verfügbarem Speicher Rechenzeit zugeteilt wird. Aber auch hier ist es schwierig, eine sinnvolle Lösung zu finden: Viel angeforderter Speicher bedeutet nicht automatisch, dass viel Garbage vorhanden ist, der vom Collector eingesammelt werden kann.

### 12.3.2 Wechsel zum Collector nur beim Allozieren

Die einfachsten, nicht parallelen Garbage-Collector aktivieren den Collector nur beim Allozieren von Objekten, nämlich dann, wenn nicht ausreichend Speicher für die geforderte Allokation vorhanden ist. Dann führt der Collector einen kompletten Zyklus durch. Dies bedeutet, dass das Allozieren sehr lange dauern kann und damit das Programm keine kurze Antwortzeit garantieren kann.

Der Collector soll daher nicht erst bei Speichermangel panisch einen kompletten Collectorzyklus unternehmen, sondern diesen Zyklus möglichst sinnvoll über mehrere Allokationen verteilen und sicherstellen, dass der Zyklus beendet ist, bevor der gesamte Speicher aufgebraucht ist.

Die meiste Zeit verbringt der Collector mit der Markierungsschleife (Schritt 3 in dem in Abschnitt 12.2 beschriebenen Algorithmus). Ein handliches Maß für die Arbeit des Collectors ist die Anzahl an Durchläufen dieser Schleife, d.h. die Anzahl der untersuchten Objekte. Bei jeder Allokation eines neuen Objektes soll nun der Collector eine bestimmte Zahl  $u$  an Objekten untersuchen. Das Problem ist nun, einen sinnvollen Wert für  $u$  zu finden.

#### 12.3.2.1 Konstantes $u$

Die einfachste Lösung ist wiederum, einen konstanten Wert für  $u$  zu wählen, beispielsweise 3 oder 10. Dies hat jedoch ähnliche Nachteile wie die oben beschriebene Aufteilung der Rechenzeit mit einem festen Verhältnis: Ist nur wenig Speicher alloziert und viel frei, so würde der Collector unnötig viele Collectorzyklen unternehmen, während bei wenig freiem Speicher ein Zyklus möglicherweise nicht beendet werden kann, bevor der Speicher ausgeht.

#### 12.3.2.2 Variables $u(n)$

Besser wäre es also,  $u$  abhängig zu machen von der Menge an alloziertem Speicher. Der Einfachheit halber nehme ich zunächst einmal an, dass alle Objekte dieselbe Größe haben. Insgesamt stehe Speicher für  $m$  Objekte zur Verfügung, von denen derzeit  $n$  alloziert und  $m-n$  frei sind.

$m$	maximale Anzahl allozierter Objekte (= Speichergröße)
$n$	Anzahl allozierter Objekte
$f=m-n$	Anzahl freier Objekte

Es soll nun  $u(n)$  in Abhängigkeit von  $n$  angegeben werden. Ist noch kein Objekt alloziert, also  $n=0$ , so kann und soll auch der Collector keines untersuchen, also

$$u(0) = 0 \quad (1)$$

Ansonsten soll bei jeder Allokation mindestens ein Objekt untersucht werden, damit der Collector nicht langsamer arbeitet, als der Mutator Objekte alloziert:

$$u(n \geq 1) \geq 1 \quad (2)$$

Ist kein freier Speicher mehr vorhanden, so ist ein kompletter Collectorzyklus nötig, damit sichergestellt ist, dass vorhandene unerreichbare Objekte freigegeben werden:

$$u(m) = m \quad (3)$$

Zudem soll nur wenig Zeit für den Collector aufgewendet werden, solange noch viel freier Speicher vorhanden ist, also beispielsweise für 50% freien Speicher

$$u(m/2) = \textit{klein} \quad (4)$$

Um einen Ausdruck für  $u(n)$  zu finden, betrachten wir die Extremsituation, in der nur noch wenige freie Objekte vorhanden sind. Den Extremfall, dass kein freier Speicher mehr vorhanden ist und bei der nächsten Allokation ein kompletter Collectorzyklus nötig ist, hatten wir oben schon mit (3):  $u(m)=m$ . Ist dagegen noch ein freies Objekt vorhanden, so muss innerhalb der nächsten zwei Allokationen ein Zyklus beendet werden, es wäre also sinnvoll, bei dieser Allokationen die Hälfte der Objekte zu untersuchen. Entsprechend muss bei zwei freien Objekten innerhalb der nächsten drei Allokationen ein Zyklus beendet werden, es soll also ein Drittel der Objekte untersucht werden. So können wir  $u(n)$  berechnen zu

$$\begin{aligned} u(m) &= m \\ u(m-1) &= m/2 \\ u(m-2) &= m/3 \\ u(m-3) &= m/4 \end{aligned}$$

allgemein bei  $f$  freien Objekten

$$u(m-f) = m/(f+1)$$

Mit  $n=m-f$  lässt sich  $u(n)$  ausdrücken als

$$u(n) = m/(m-n+1) \quad (5)$$

Bei Abrundung auf die nächste ganze Zahl erfüllt dieses  $u(n)$  auch die oben angegebenen Forderungen (1) und (2):  $u(0)=0$  und  $u(n \geq 1) \geq 1$ . (3) ist auch erfüllt und für (4) ergibt sich, wenn noch die Hälfte des Speichers frei ist:

$$\begin{aligned} u(n=m/2) &= m/(m/2+1) \\ &= 2 * (m / (m+2)) \\ &< 2 \end{aligned}$$

Es werden also nur wenige Objekte untersucht, in diesem Fall ungefähr zwei, solange noch viel Speicher frei ist.

Interessant zu untersuchen ist, wie groß der Aufwand für das Allozieren gestiegen ist. Dazu nehme ich an, der Mutator alloziert nacheinander den gesamten Speicher, erzeugt jedoch keine unerreichbaren Objekte, die der Collector einsammelt. Hierfür kann der Zeitaufwand für den Collector berechnet werden zu

$$\begin{aligned} &\sum_{n=0..m-1} u(n) \\ &= \sum_{n=0..m-1} m/(m-n+1) \\ &= \sum_{n=2..m+1} m/n \\ &= m (-1 + \sum_{n=1..m+1} 1/n) \\ &\leq m (-1 + \log(m+1)) \\ &\in O(m \cdot \log(m)) \end{aligned}$$

Steht also beispielsweise Speicher für 1 000 000 Objekte zur Verfügung, so untersucht der Collector beim Allokieren des gesamten Speichers weniger als etwa 20 000 000 Objekte.

Ziel des parallelen Collectors ist es, große Verzögerungen durch den Collector zu vermeiden. Es ist für Anwendungen daher wichtig zu wissen, wie lange sie maximal beim Allokieren eines Objektes angehalten werden. Dies können wir jetzt ausrechnen, wenn wir voraussetzen, dass das Programm einen maximalen Anteil von  $x\%$  des gesamten Speichers als erreichbare Objekte alloziert hat, es wird also maximal  $x\%$  des Speichers benötigt.

Als Beispiel nehmen wir an, der gesamte Speicher wäre ausreichend für 1 000 000 Objekte und 75% der Objekte, also 750 000, sind maximal erreichbar. Irgendwann erreicht das Programm nun die Situation, in der 750 000 allozierte Objekte existieren. Wir wollen nun bestimmen, wann spätestens der nächste Collectorzyklus beendet ist. Schlimmstenfalls hat dieser Zyklus gerade begonnen. Dann müssen 156 042 Objekte alloziert werden, bis der nächste Zyklus beendet ist. Danach ist nämlich die Summe der  $u(n)$  (also  $u(750\,000) + u(750\,001) + \dots + u(750\,000 + 156\,041)$ ) größer als  $750\,000 + 156\,042 = 906\,042$ . Möglicherweise wurde in diesem Zyklus jedoch kein einziges unerreichbares Objekt gefunden, falls vor dem Zyklus alle allozierten Objekte erreichbar waren und auch keines der während des Zyklus allozierten Objekte als frei erkannt werden konnte. In diesem Fall haben wir also 906 042 (90,6%) allozierte Objekte. Wir wissen jedoch, dass nie mehr als 75% der Objekte erreichbar sind.

Während des nun folgenden Collectorzyklus werden also mindestens  $906\,042 - 750\,000 = 156\,042$  unerreichbare Objekte freigegeben. Analog können wir auch hier berechnen, wann dieser Zyklus beendet ist: Nachdem weitere 59 227 Objekte alloziert wurden. Zum Ende dieses Zyklus haben wir  $906\,042 + 59\,227 = 965\,269$  (96,5%) allozierte Objekte, von denen nun mindestens 156 042 freigegeben werden. Es bleiben also weniger Objekte alloziert als vor diesem Zyklus alloziert waren. Beim dritten und allen weiteren Zyklen wird damit die Zahl 965 269 (96,5%) an allozierten Objekten nicht überschritten.

Für diese Höchstzahl an allozierten Objekten können wir  $u(n=965\,269)=28$  berechnen, es wird also bei einer Allokation der Collector niemals länger aktiviert, als für die Untersuchung von 28 Objekten nötig ist. Entsprechend können für gegebene Mengen an maximal erreichbaren Objekten die maximale Zahl an allozierten Objekten und damit die maximale Zahl an Objekten, die bei einer Allokation untersucht werden, berechnet werden. Dies ist für verschiedene Mengen an maximal erreichbaren Objekten in Prozent des verfügbaren Speichers in *Tabelle 1* angegeben.

Zusätzlich zu der maximalen Menge an allozierten Objekten und dem Maximalwert für  $u(n)$  ist in der Tabelle auch ein maximaler Mittelwert für  $u(n)$  gegeben. Dies ist der Mittelwert über alle  $u(n)$  in dem Collectorzyklus, in dem die maximale Anzahl an allozierten Objekten erreicht wird. Die in der Tabelle gegebenen Werte ändern sich für andere Werte für  $m$  kaum.

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

Maximaler Anteil erreichbarer Objekte am Gesamtspeicher	Maximaler Anteil der allozierten Objekte am Gesamtspeicher	Maximale Anzahl an Objekten, die bei einer Allokation untersucht werden	Maximaler Mittelwert von $u(n)$
5,0%	82,9%	5	2,9
10,0%	85,0%	6	3,3
15,0%	86,8%	7	3,9
20,0%	88,1%	8	4,4
25,0%	89,1%	9	4,8
30,0%	90,0%	10	5,3
35,0%	90,9%	10	5,9
40,0%	91,5%	11	6,3
45,0%	92,1%	12	6,9
50,0%	92,7%	13	7,5
55,0%	93,6%	15	8,7
60,0%	94,4%	17	10,0
65,0%	95,1%	20	11,4
70,0%	95,8%	24	13,5
75,0%	96,5%	28	16,2
80,0%	97,2%	36	20,6
85,0%	97,9%	48	27,9
90,0%	98,6%	73	42,2
95,0%	99,3%	147	85,2
96,0%	99,5%	184	106,7
97,0%	99,6%	245	142,5
98,0%	99,7%	368	214,2
99,0%	99,9%	738	429,3
99,5%	99,9%	1 477	859,8
100,0%	100,0%	1 000 000	1 000 000,0

**Tabelle 1:**

Aufwand für den Collector bei unterschiedlichen Mengen an maximal erreichbaren Objekten. Werte für  $m = 1\,000\,000$ .

Soll sichergestellt sein, dass eine Allokation nicht um mehr Zeit verlangsamt wird, als das Untersuchen von beispielsweise 50 Objekten benötigt, so muss lediglich sichergestellt werden, dass niemals mehr als 85% des gesamten Speichers alloziert und erreichbar ist.

Aber auch in die andere Richtung kann man mit dieser Tabelle Rückschlüsse ziehen: Sind beispielsweise 99,5% aller Objekte alloziert, so waren mindestens 96% aller Objekte auch erreichbar, die Menge an Garbage, also an allozierten und nicht erreichbaren Objekten, ist in diesem Fall maximal 3,5% des Gesamtspeichers. Sind also nur noch wenige freie Objekte vorhanden, so ist auch die Anzahl an unerreichbaren Objekten klein.

### 12.3.2.3 Objekte unterschiedlicher Größe

Das oben gesagte lässt sich wie folgt auf Systeme mit Objekten unterschiedlicher Größe übertragen. In einem solchen System sind folgende Werte leicht bestimmbar:

- $m$  zur Verfügung stehender Speicher in Bytes
- $n$  derzeit allozierter Speicher in Bytes
- $n'$  Anzahl allozierter Objekte
- $s=n/n'$  Durchschnittliche Größe der allozierten Objekte.

Nun kann  $u(n)$  ähnlich wie oben definiert werden zu

$$u(n) = m/(m-n+1)$$

Dieser Wert sind die Anzahl an Bytes, die für jedes neu allozierte Byte an Speicher untersucht werden müssen. Um die Anzahl an Objekten zu erhalten, muss der Wert noch durch die durchschnittliche Größe der Objekte geteilt werden:

$$u(n) = m/(m-n+1) / s$$

Ist das zu allozierende Objekt  $size$  Bytes groß, so müssen

$$u(n, size) = m/(m-n+1) * size / s$$

Objekte untersucht werden.

Solange Fragmentierung nicht zu einem Problem wird, ist das oben gesagte übertragbar auf diesen Fall mit Objekten unterschiedlicher Größe. Es ist jedoch zu beachten, dass die Anzahl an untersuchten Objekten nun auch von der Größe des neu allozierten Objektes abhängt.

### 12.3.2.4 Sweep-Phase des Collectors

Bisher wurde bei der Zeit, die dem Collector zugeteilt wird, nur die Markierungsphase betrachtet. Dies wird problematisch, wenn die Übernahme der weißen Objekte beim Sweep nicht in konstanter Zeit geschehen kann. Gewöhnlich wird hier nochmals lineare Zeit in der Anzahl allozierter Objekte benötigt.

Nehmen wir an, dass ein Schritt der Sweep-Phase 10% der Zeit benötigt, die das Markieren eines Objekts in der Markierungsphase dauert. Dann müssen dem Collector diese 10% zusätzlich bereitgestellt werden. Am einfachsten führen wir dazu ein verbessertes  $u(n)$  ein:

$$u'(n) = 110\% * u(n).$$

Befindet sich der Collector in der Markierungsphase, so werden hier 10% mehr Objekte untersucht. In der Sweep-Phase soll der Collector  $10 * u'(n)$  Schleifendurchläufe bearbeiten. An den vorher gemachten Überlegungen ändert dies jedoch nichts, lediglich wird dem Collector ein konstanter Faktor an zusätzlicher Rechenzeit zuteil.

Entsprechend kann auch vorgegangen werden, wenn die Anzahl der Wurzelobjekte nicht konstant ist.

Eine Implementierung des Wechselmechanismus zwischen Collector und Mutator in Pseudo-Eiffel-Code ist in *Anhang B* gegeben.

## 12.4 Größenklassen

Für die freien Objekte sollen Listen verwaltet werden, in denen jeweils Objekte derselben Größe gespeichert sind. Die Anzahl der möglichen Größen muss beschränkt werden, um nicht unnötig viele Listen speichern zu müssen. [Wilson93] erlaubt als Objektgrößen nur Zweierpotenzen. Dies hat den Vorteil, dass bei einer Allokation größere Ob-

jekte sehr leicht in kleinere zerlegt werden können, falls die Liste der kleineren Objekte leer ist.

Allerdings wird durch die Beschränkung auf Zweierpotenzen möglicherweise bis zu 50% des Speichers verschwendet, beispielsweise würden bei der Allokation von 1000 Objekten der Größe 1028 Byte nicht etwas mehr als 1MB, sondern ganze 2MB an Speicher alloziert. Kommen alle Größen gleichverteilt vor, so bleibt durchschnittlich 25% des Speichers ungenutzt.

Die Schwierigkeit besteht nun darin, eine Folge von Größen zu finden, bei der im Durchschnitts- und im schlechtesten Fall weniger Speicher verschwendet wird, bei der größere Objekte aber immer noch leicht in kleinere zerlegt werden könne.

Eine Möglichkeit besteht darin, die Größe der Objekte nicht jeweils zu verdoppeln, sondern abwechselnd mit  $\frac{3}{2}$  und  $\frac{4}{3}$  zu multiplizieren. Dann haben wir Objekte der Größen

n	1	2	3	4	5	6	7	8	9
size(n)	1	2	3	4	6	8	12	16	24...
size(n+1)/size(n)	-	$\frac{3}{2}$	$\frac{4}{3}$	$\frac{3}{2}$	$\frac{4}{3}$	$\frac{3}{2}$	$\frac{4}{3}$	$\frac{3}{2}$	

Es gilt

$$\begin{aligned} \text{size}(n) &= n, && \text{falls } n \leq 2 \\ \text{size}(n) &= \frac{3}{2} * \text{size}(n-1), && \text{falls } n \text{ MOD } 2 = 1 \\ \text{size}(n) &= \frac{4}{3} * \text{size}(n-1), && \text{falls } n \text{ MOD } 2 = 0 \end{aligned}$$

Objekte können immer noch leicht in kleinere Objekte zerlegt werden, da die Größe jedes Objektes durch Addition von wenigen kleineren Objekten erhalten werden kann. Hier wird nur noch maximal ein Drittel des Speichers verschwendet, im Durchschnitt weniger als ein Sechstel.

Dieses Verfahren lässt sich noch beliebig durch kleinere Schritte verbessern, so ergibt sich bei einer Periode 4 anstelle von 2 bei der Vergrößerung der Objekte:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
size(n)	1	2	3	4	5	6	7	8	10	12	14	16	20	24	28	
size(n+1)/size(n)					$\frac{5}{4}$	$\frac{6}{5}$	$\frac{7}{6}$	$\frac{8}{7}$	$\frac{5}{4}$	$\frac{6}{5}$	$\frac{7}{6}$	$\frac{8}{7}$	$\frac{5}{4}$	$\frac{6}{5}$	$\frac{7}{6}$	$\frac{8}{7}$

Es gilt entsprechend

$$\begin{aligned} \text{size}(n) &= n, && \text{falls } n \leq 4 \\ \text{size}(n) &= \frac{5}{4} * \text{size}(n-1), && \text{falls } n \text{ MOD } 4 = 1 \\ \text{size}(n) &= \frac{6}{5} * \text{size}(n-1), && \text{falls } n \text{ MOD } 4 = 2 \\ \text{size}(n) &= \frac{7}{6} * \text{size}(n-1), && \text{falls } n \text{ MOD } 4 = 3 \\ \text{size}(n) &= \frac{8}{7} * \text{size}(n-1), && \text{falls } n \text{ MOD } 4 = 0 \end{aligned}$$

Hier wird nur noch maximal ein Fünftel des Speichers verschwendet, im Durchschnitt weniger als 10%.

Entsprechend bei einer Periode von 8 wird nicht mehr als ein Neuntel verschwendet und durchschnittlich nur 5%.

Allerdings können bei diesem Verfahren beim Zerlegen von Objekten in kleinere leicht Objekte erzeugt werden, die so klein sind, dass sie später nicht mehr alloziert werden können, sondern erst nach einer Kompaktierung (s. u.) wieder zur Verfügung stehen. Um diesen Effekt so gering wie möglich zu halten, soll freier Speicher zunächst als ein großer Block gehalten werden. Bei einer Allokation wird nun zunächst geprüft, ob ein freies Objekt der gewünschten Größe vorhanden ist. Ist dies nicht der Fall, so wird versucht, vom freien Block ein Stück der gewünschten Größe zu nehmen. Erst wenn dies auch scheitert, wird ein größeres freies Objekt genommen und zerkleinert. Eine Implementierung von *Allocate* in Pseudocode wird in *Anhang C* gezeigt.

Um abzuschätzen, wieviel Speicher durch Zerstückelung maximal verloren geht, nehmen wir an, dass alle Objekte, die kleiner sind als das gerade allozierte, zu klein für weitere Allokationen sind. Das gerade allozierte Objekt habe die Größe *size*.

Im Fall der Periode 2 entsteht durch die Zerlegung schlimmstenfalls ein Objekt der Größe  $size/2$ , es entsteht also maximal 33% Fragmentierung, die vom Kompaktierer aufgeräumt werden muss.

Entsprechend bei der Periode 4 kann schlimmstenfalls ein Objekt der Größe  $3/4 * size$  entstehen, maximal werden also  $3/7$  oder 42.9% des Speichers fragmentiert.

Diese Fragmentierung ist im schlimmsten Fall schmerzlich groß, im Durchschnittsfall (wenn die kleineren Speicherblöcke für andere Objekte noch verwendet werden) ist jedoch zu erwarten, dass sie weit geringer ausfällt und damit ein weniger ernstes Problem ist. Zudem ist dieser Speicher nicht völlig verloren, wie der bei der Beschränkung auf Zweierpotenzen als Objektgrößen verloren gehende: nach der nächsten Kompaktierung steht er wieder voll zur Verfügung.

### 12.5 Kompaktierung

Das Anfordern und Freigeben von Speicherblöcken unterschiedlicher Größe verursacht gewöhnlich Fragmentierung. Dadurch werden die zusammenhängenden freien Speicherbereiche mit der Zeit immer kleiner. Aufgabe des Kompaktierers ist es, den allozierten Speicher so zu verschieben, dass der gesamte freie Speicher nur noch einen zusammenhängenden Block ergibt.

Eine einfache Implementierung eines parallelen Kompaktierers verlangt, dass keine Referenzen direkt sind, sondern Referenzen auf Objekte sind stets Indizes in einem globalen Feld von Referenzen. Jedem allozierten Objekt wird ein Eintrag in diesem Feld zugeordnet, der die Referenz auf dieses Objekt enthält. Alle Zugriffe finden nur über Indizierung im Referenzenfeld und dann Dereferenzierung statt. Dadurch kann das Objekt leicht verschoben werden, ein Angleichen der Referenz in Referenzenfeld ist ausreichend, damit der Mutator weiterarbeiten kann, ohne das Verschieben überhaupt zu bemerken.

Parallele Garbage-Collectoren unterstützten meist keine Kompaktierung, da ein paralleler Kompaktierer einen erheblichen Mehraufwand beim Mutator verursacht und eine Kompaktierung nur selten benötigt wird. Andererseits kann durch Fragmentierung eine Programmausführung wegen Speichermangels fehlschlagen, selbst wenn insgesamt ausreichend Speicher vorhanden wäre. Dieses Problem existiert nicht nur bei der Speicherverwaltung mit Garbage-Collector, sondern kann beispielsweise auch bei gewöhnlichen C-Programmen auftreten.

Ein solcher Programmabbruch kann durch eine Kompaktierung des Speichers verhindert werden. Dadurch kann es zwar passieren, dass die Programmausführung kurz angehalten wird, dies erscheint jedoch vertretbar, wenn die einzige Alternative ein Abbruch des Programms wegen Speichermangels ist.

Der hier beschriebene Kompaktierungsalgorithmus kommt ohne zusätzliche Kooperation des Mutators aus. Es müssen lediglich alle Speicherstellen, die Referenzen enthalten, bekannt sein. Dies ist für die Untersuchung durch den Garbage-Collector bereits der Fall.

Der Algorithmus hat drei Phasen:

1. Die allozierten Objekte werden in der Reihenfolge ihrer Speicheradresse durchlaufen. Dabei wird für jedes Objekt eine neue Adresse bestimmt, an die das Objekt geschoben werden soll.
2. Alle Referenzen, die von Objekten ausgehen und alle Zeiger auf Wurzelobjekte werden an die neuen, in den Objekten gespeicherten Positionen angepasst.
3. Die Objekte werden nochmals durchlaufen und dabei an ihre neuen Positionen verschoben.

Aussehen des Speichers vor der Kompaktierung:

```

      A      B      C      D
|---|x|--|xxx|xxx|--|xx|----|

```

und danach:

```

      A      B      C      D
|x|xxx|xxx|xx|-----|

```

Eine detailliertere Implementierung des Kompaktierers in Pseudocode ist in *Anhang D* gegeben.

## 12.Anhang A: Implementierung des Collectors

Detailliertere Implementierung des Collection-Algorithmus:

Bei der hier beschriebenen Implementierung wird von einem 32-Bit-Zielsystem ausgegangen. Für andere Wortgrößen sind an wenigen Stellen Änderungen nötig.

### 12.A.1 Objektstruktur

Die einfachste Implementierung für die drei Listen *weiß*, *grau* und *schwarz* sind doppelt verkettete Listen. Ein Entfernen aus einer Liste und Einordnen in eine andere Liste ist dann recht effizient machbar, für einen effizienten Test auf Zugehörigkeit zu einer bestimmten Liste muss jedoch im Objekt noch zusätzlich die Listenzugehörigkeit gespeichert werden.

Bei der hier beschriebenen Implementierung werden lediglich die grauen Objekte in einer einfach verketteten Liste gespeichert, die Zugehörigkeit zur weißen oder schwarzen Menge wird durch besondere Werte des Zeigers für die Verkettung implementiert. So

wird statt drei zusätzlichen Einträgen (für die doppelte Verkettung und die Farbmarkierung) pro Objekt nur eines benötigt.

Hier in Pseudo-Eiffel-Code der Aufbau eines Objekts:

```

Class Object
  next : Object;      -- Link in MemList, oder in einer der freelists

  color : Object;     -- Link in Grau-Liste, IstWeiß, IstSchwarz oder IstFrei
  IstWeiß is 0;       -- color = IstWeiß (= 0) für weiße Objekte.
  IstSchwarz is -1;   -- color = IstSchwarz (= 0FFFFFFFH) für schwarze Objekte.
  GrauEnde is 1;     -- Endemarkierung in Grau-Liste
  IstFrei is 2;      -- Gesetzt für Objekte in FreeLists und für den freien Block

  newAdr: Object;    -- neue Adresse beim Kompaktieren

  type : TypeDescriptor; -- Zeiger auf Typ dieses Objekts

  size : INTEGER;    -- Größe des freien Objekts

  data : ...

  IstGru : BOOLEAN is
  require
    color /= IstFrei    -- Farbe ist undefiniert für freie Objekte
  do
    Result := (color /= IstWeiß) and
              (color /= IstSchwarz);
  end; -- IstGru

invariant
  color = IstFrei IMPLIES type undefined
  color /= IstFrei IMPLIES size undefined
  newAdr undefined OR color undefined -- während Kompaktierung wird color gelöscht.
end;

```

Dabei ist *color* entweder einer der Werte *IstWeiß*, *IstSchwarz* oder *IstFrei* oder der Zeiger in der Liste der grauen Objekte, oder der Wert *GrauEnde* für das letzte Element in der Liste der grauen Elemente. Natürlich muss sichergestellt sein, dass keines der Objekte an einer der Adressen -1, 0, 1 oder 2 liegt, damit diese Werte nicht mit Zeigern in der *Grau*-Liste verwechselt werden können.

Über *next* sind alle allozierten Objekte und alle freien Objekte gleicher Größe miteinander verbunden.

Das Feld *type* zeigt auf einen Deskriptor, in dem der Typ des Objekts beschrieben wird, insbesondere die Größe und die relativen Adressen von Referenzen in diesem Objekt.

Die Invariante drückt aus, dass einer der Einträge *type* und *size* immer undefiniert ist, so dass für *type* und *size* zusammen nur eine Speicherzelle pro Objekt nötig ist. Entsprechend ist immer entweder *newAdr* oder *color* undefiniert, diese beiden Werte können also zudem in einem einzigen Feld gespeichert werden. Zusammen mit dem Eintrag *next* werden also pro Objekt drei Einträge für zusätzliche Informationen gebraucht, auf 32-Bit-Rechnern ein zusätzlicher Aufwand von 12 Bytes pro Objekt.

### 12.A.2 Der Collector-Zyklus

Der Algorithmus des Collectors in Pseudo-Code sieht nun wie folgt aus:

```

Roots: SET OF Object; -- Wurzelobjekte = globale Variablen

MemList: Object;      -- Liste aller allozierten Objekte, verbunden über OBJECT.next, Ende=Void
Gru: Object;         -- Liste aller grauen Objekte, verbunden über OBJECT.color, Ende=GruEnde

StackStart: ActivationBlock; -- der äußerste Aktivierungsblock mit lokalen Variablen
ExaminedAB: ActivationBlock; -- jüngster untersuchter Aktivierungsblock

```

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

---

```
Collect is          -- Der Collectoralgorithmus als unendlicher Zyklus
local
  m,g,r,o,prev: Object;
do
  -- 1. Alle Objekte werden weiß markiert:
  -- dies geschieht vor der Schleife, da am Ende der Schleife dieser Zustand automatisch wieder-
  -- hergestellt wird

  Grau := GrauEnde;    -- Grau-Liste soll leer sein

  from
    m := MemList;
  until m = Void loop
    m.color := IstWeiß;
    m := m.next;
  end;

  from
  until false loop    -- GC Hauptschleife

    -- 2. Die Wurzel-Objekte werden grau markiert:

    MarkRootRefs;
    ExaminedAB := Void; -- bisher kein Aktivierungsblock untersucht.

    -- 3. Solange es graue Objekte gibt, diese untersuchen

    from
    until Grau = GrauEnde loop

      from
      until Grau = GrauEnde loop
        g := Grau;
        Grau := Grau.next;

        -- 3.a Für alle Objekte r, auf die g Referenzen enthält, markiere r grau, falls es weiß ist.
        MarkObjectRefs(g);

        -- 3.b markiere g schwarz.
        g.color := IstSchwarz;

        ToMutator(markValue); -- Context-Switch zu Mutator ermöglichen
      end;

      -- lokale Variablen markieren:

      -- Die Variable ExaminedAB zeigt stets den jüngsten vom Collector untersuchten AB an oder ist Void,
      -- falls es einen solchen nicht gibt. Der Mutator muss ExaminedAB auf den AB der umschließenden
      -- Routine ändern, wenn die zu diesem Aktivierungsblock gehörende Routine aktiv wird und ihre
      -- lokalen Variablen verändern kann (siehe Anhang A.4).

      from
      if ExaminedAB = Void THEN
        ExaminedAB := StackStart
        MarkLocalRefs(ExaminedAB);
      end;
    until Grau /= GrauEnde or (ExaminedAB+ab.type.size=StackTop) loop
      ExaminedAB := ExaminedAB + ExaminedAB.type.size;
      MarkLocalRefs(ExaminedAB);
    end;
  end;

  -- 4. Nehme die weißen Objekte in die Liste des freien Speichers.

  from
    m := MemList;
    prev := Void;    -- prev zeigt auf das letzte, nicht freigegebene Objekt.
  until m = Void loop
    o := m;
    m := m.next;
    if o.color = IstWeiß then
      if prev=Void then    -- o aus MemList entfernen
        MemList := m;
      else
        prev.next := m;
      end;
      FreeMem(o);    -- o freigeben
    else
      o.color := IstWeiß; -- o weiß markieren und in MemList belassen
      prev := o;
    end;
    ToMutator(sweepValue); -- Context-Switch zu Mutator ermöglichen
  end;

end; -- GC Hauptschleife

end; -- Collect
```

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

---

```
MarkGrey (o: Object) is      -- markiere o grau, falls o.color=IstWeiß
ensure
  old o.color=IstWeiß implies o.IstGrau;
do
  if o.color=IstWeiß then
    o.color := Grau;
    Grau := o;
  end;
end MarkGrey;

MarkObjectRefs (g: Object) is -- markiere von g referenzierte Objekte grau
require
  g /= Void
local
  r: Object;
do
  for all r : g zeigt auf r do
    MarkGrey(r);
  end;
ensure
  for all r : g zeigt auf r : r.color /= IstWeiß
end; -- MarkRefs

MarkRootRefs is           -- markiere alle Wurzelobjekte grau
local
  m: Object;
do
  for all m : m in Roots do
    MarkGrey(m);
  end;
ensure
  for all m : m in Roots : m.IstGrau;
end; -- MarkRootRefs

MarkLocalRefs(ab: ActivationBlock) is -- markiere alle von ab referenzierten Objekte
local
  r: Object;
do
  for all r : ab zeigt auf r do
    MarkGrey(r);
  end;
end; -- MarkLocalRefs
```

### 12.A.3 Zeigerzuweisungen

Nun gilt es, den Mutator noch möglichst effizient zu implementieren, hier ist die meiste Mühe gefordert, da der Mutator häufig Zeigerzuweisungen ausführen muss, so dass unnötig aufwendiger Code hierfür den größten Effekt auf die Ausführungszeit des Programms hat. Als Beispiel wird wieder die Zuweisung

```
a := b
```

verwendet, die dem Attribut *a* des Objekts *Current* den Zeiger *b* zuweist. Der zu erzeugende Code sieht in der einfachsten Form wie folgt aus:

```
if (Current.color = IstSchwarz) and then
  (b /= Void) and then
  (b.color = IstWeiß)
then
  b.color := Grau;
  Grau := b;
end;
a := b;
```

Dies ist ein recht aufwendiges Stück Code, das es zu vereinfachen gilt. Wenn wir für den Wert *Void* nicht 0 verwenden, sondern ein ausgezeichnetes Objekt, das stets schwarz ist, kann der Test  $b \neq \text{Void}$  vermieden werden. Es ergibt sich:

```

if (Current.color = IstSchwarz) and then
  (b.color = IstWeiß)
then
  b.color := Grau;
  Grau := b;
end;
a := b;

```

Die Zahlenwerte für *IstWeiß* und *IstSchwarz* wurden absichtlich zu 0 und -1 gewählt, die Bedingung der *if*-Anweisung kann dadurch wie folgt umgeschrieben werden:

```

if (~Current.color | b.color) = 0 then
  b.color := Grau;
  Grau := b;
end;
a := b;

```

dabei ist " $|$ " die bitweise oder-Verknüpfung und " $\sim$ " das Bit-Komplement. Die Äquivalenz zur alten Form lässt sich leicht zeigen:  $(\sim x | y) = 0 \Leftrightarrow (\sim x=0) \& (y=0) \Leftrightarrow (x=-1) \& (y=0) \Leftrightarrow (x=\text{IstSchwarz}) \& (y=\text{IstWeiß})$ .

Für die Implementierung soll hier als Beispiel entsprechender PowerPC-Assemblercode [Motorola96] gezeigt werden. Es wird dabei angenommen, dass sich in den folgenden Registern die beschriebenen Werte befinden:

<i>rGrau</i>	Zeiger auf erstes Element der Grauliste
<i>rB</i>	Zeiger auf das Objekt, das <i>b</i> enthält
<i>rCurrent</i>	Current
<i>rCurColor</i>	Current.color

Der optimierte Code sieht wie folgt aus:

```

lw      r1,offset_b(rB)
lw      r2,color(r1)
orc.    r3,r2,rCurColor
bneqz   label
stw     rGrau,color(r1)
mr      rGrau,r1
label:  stw     r1,offset_a(rCurrent)

```

Verglichen mit dem Code, der ohne Garbage-Collector erzeugt werden müsste:

```

lw      r1,offset_b(rB)
stw     r1,offset_a(rCurrent)

```

haben wir 7 statt 2 Anweisungen und 4 statt 2 Speicherzugriffe. Gehen wir davon aus, dass mit Wahrscheinlichkeit 0,5  $\text{Current.color}=\text{IstSchwarz}$  und die Wahrscheinlichkeit für  $b.\text{color}=\text{IstWeiß}$  davon unabhängig und auch 0,5 ist, so müssen durchschnittlich 5,5 Anweisungen mit 3,25 Speicherzugriffen zusätzlich ausgeführt werden.

## 12.A.4 Aktivierungsblöcke

Die Aktivierungsblöcke haben die Struktur

```
Class ActivationBlock
  dyn : ActivationBlock; -- dynamischer Link zum umschließenden Aktivierungsblock
  type : ABTypeDescriptor; -- beschreibt diesen AB, insbesondere wo er Referenzen enthält.
  data : ... -- die lokalen Variablen;
end; -- class ActivationBlock
```

Beim Aufruf einer Routine wird der neue Aktivierungsblock folgendermaßen erzeugt:

```
oldAB := AB;
AB := StackPointer;
StackPointer := StackPointer + type.size;
AB.dyn := oldAB;
AB.type := type_of_new_activation_block;
```

Beim Verlassen einer Routine wird der Aktivierungsblock entsprechend abgebaut. Zudem muss jedoch noch sichergestellt werden, dass die vom Collector gesetzte Variable *ExaminedAB* auf den nächstälteren Aktivierungsblock gesetzt wird, falls die zu *ExaminedAB* gehörende Routine aktiv wird:

```
StackPointer := StackPointer - type.size;
AB := AB.dyn;
IF AB=ExaminedAB THEN
  ExaminedAB := ExaminedAB.dyn;
END;
```

Der Test auf *AB=ExaminedAB* ist dabei in den allermeisten Fällen *false*. Für diese Fälle kann die für den Test benötigte Rechenzeit auf null verringert werden, indem der Collector die Rücksprungadresse der nächstjüngeren Routine umleitet auf eine andere Routine *AdjustExaminedAB*. Diese soll *ExaminedAB* anpassen, die Rücksprungadresse der nächsten Routine umleiten und dann an die alte Rücksprungadresse springen. Diese Routine sieht in etwa wie folgt aus:

```
AdjustExaminedAB is
local
  Temp: ADDRESS;
do
  ExaminedAB := ExaminedAB.dyn; -- ExaminedAB auf älteren AB setzen
  Temp := OldLink;
  OldLink := ExaminedAB.link; -- Alte Rücksprungadresse merken
  ExaminedAB.link := AdjustExaminedAB; -- Rücksprungadresse umleiten
  JUMP Temp -- an alte umgeleitete Adresse springen
end;
```

Die globale Variable *OldLink* speichert die Rücksprungadresse in die Routine des Mutators, deren Aktivierungsblock *ExaminedAB* ist. Entsprechend muss der Collector beim ändern von *ExaminedAB* folgendes machen:

```
ExaminedAB.link := OldLink;
ExaminedAB := NewExaminedAB;
OldLink := ExaminedAB.link;
ExaminedAB.link := AdjustExaminedAB;
```

## 12.Anhang B: Wechsel zwischen Mutator und Collector

Der Collector und Mutator arbeiten als Coroutinen zusammen. Die Routinen *ToCollector* und *ToMutator* sollen den Coroutinenwechsel zwischen beiden implementieren.

Pseudo-Eiffel-Code für die Verwaltung der Rechenzeitverteilung:

```

memSize: INTEGER;    -- Gesamtspeicher des Systems
allocSize: INTEGER; -- Größe des allozierten Speichers
allocObj:  INTEGER; -- Anzahl allozierter Objekte

markValue is 10;    -- Arbeitspunkte, die dem Collector für einen Markierschritt gutgeschrieben werden.
sweepValue is 1;    -- Arbeitspunkte für einen Durchlauf der Sweep-Schleife

remaining_u: INTEGER; -- noch von Collector abzuarbeitende Arbeitspunkte

u (size) : INTEGER is
-- Anzahl der Objekte die der Collector bei Allocate() untersuchen soll
do
-- ACHTUNG: Hier muss noch sichergestellt werden, dass kein Integer-Überlauf entsteht!
Result := memSize // (memSize - allocSize + 1) * size // (allocSize // allocObj);
end; -- u

ToCollector(size: INTEGER) is
-- Wechselt zum Collector bei der Allocation von size Bytes.
do
remaining_u := remaining_u + (markValue+sweepValue) * u(size);
if remaining_u>0 then
Switch(Collector) -- coroutine-context-switch to collector
end;
end; -- ToCollector

ToMutator(n: INTEGER) is
-- Der Collector hat n Arbeitspunkte abgearbeitet. Ist damit sein Soll erfüllt, soll zum
-- Mutator gewechselt werden
do
remaining_u := remaining_u - n;
if remaining_u <= 0 then
Switch(Mutator); -- coroutine-context-switch to mutator
end;
end; -- ToMutator;

```

## 12.Anhang C: Allocate und Free

Pseudo-Eiffel-Code für *Allocate* und *Free*:

```

min: is 16; -- Mindestobjektgröße

sizelist: ARRAY[INTEGER] is
<<
1 * min,      2 * min,      3 * min,      4 * min,
5 * min,      6 * min,      7 * min,      8 * min,
10 * min,     12 * min,     14 * min,     16 * min,
20 * min,     24 * min,     28 * min,     32 * min,
40 * min,     48 * min,     56 * min,     64 * min,
80 * min,     96 * min,    112 * min,    128 * min,
160 * min,    192 * min,    224 * min,    256 * min,
320 * min,    384 * min,    448 * min,    512 * min,
640 * min,    768 * min,    896 * min,    1024 * min,
1280 * min,   1536 * min,   1792 * min,   2048 * min,
2560 * min,   3072 * min,   3584 * min,   4096 * min,
5120 * min,   6144 * min,   7168 * min,   8192 * min,
10240 * min,  12288 * min,  14336 * min,  16384 * min,
20480 * min,  24576 * min,  28672 * min,  32768 * min,
40960 * min,  49152 * min,  57344 * min,  65536 * min,
81920 * min,  98304 * min,  114688 * min, 131072 * min,
163840 * min, 196608 * min, 229376 * min, 262144 * min,
327680 * min, 393216 * min, 458752 * min, 524288 * min,
655360 * min, 786432 * min, 917504 * min, 1048576 * min,
1310720 * min, 1572864 * min, 1835008 * min, 2097152 * min,
2621440 * min, 3145728 * min, 3670016 * min, 4194304 * min,
5242880 * min, 6291456 * min, 7340032 * min, 8388608 * min,
10485760 * min, 12582912 * min, 14680064 * min, 16777216 * min,
20971520 * min, 25165824 * min, 29360128 * min, 33554432 * min,
41943040 * min, 50331648 * min, 58720256 * min, 67108864 * min,
83886080 * min, 100663296 * min, 117440512 * min
>>

block: POINTER; -- Freier Speicherblock

```

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

```
freelists: ARRAY[OBJECT]; -- Liste der freien Objekte der verschiedenen Größen

memStart, memEnd: ADDRESS;

make (start,end: ADDRESS) is
-- Initialisierung
require
end>start
local
  i: INTEGER;
do
  memStart := start;
  memEnd := end;
  memSize := end-start;

  block := memStart;
  block.color := IstFrei;
  block.size := memSize;

  allocSize := 0;
  allocObj := 0;

  from -- freelists leeren
    i := freelists.lower
  until i>freelists.upper loop
    freelists.put(Void,i);
    i := i+1
  end;

  MemList := Void;
end;

Allocate (type: INTEGER) : OBJECT is
-- alloziert Speicher der Größe sizelist@type.index
require
  type /= Void;
  type.index<=size.upper;
  type.index>=size.lower
ensure
  Result /= Void;
  Result.type = type
local
  split: OBJECT;
  splitsize: INTEGER;
  try: INTEGER;
  size: INTEGER;
do
  size := sizelist@type.index;
  ToCollector(size); -- GC aktivieren
  from
  until Result /= Void loop
    if freelists@type.index /= Void then -- ist ein passendes freies Object vorhanden?
      Result := freelists@type.index;
      freelists.put(Result.next,type.index);
    elseif block.size >= size then -- oder ist noch ausreichend freier Speicher da?
      Result := block;
      block := block + size;
      block.color := IstFrei;
      block.size := Result.size - size
    else -- beides nicht, gibt es vielleicht ein größeres freies Objekt?
      from
        j := type.index + 1;
      until (j > freelists.upper) or else (freelists@j /= Void) loop
        j := j+1;
      end;
      if j<=freelists.upper then -- wurde ein größeres Objekt gefunden?
        split := freelists@j; -- dann wird es aus freelist entfernt und zerlegt
        freelist.put(split.next,j);
        Result := split; -- zunächst ein Teil für diese Allokation abgeschnitten
        split := split + size;
        from -- der Rest wird in die FreeLists der kleineren Objekte verteilt.
          splitsize := sizelist@j - size;
        until splitsize=0 loop
          j := j-1;
          if size@j>=splitsize then
            split.next := freelists@j;
            freelists.put(split,j);
            split := split + size@j;
            splitsize := splitsize - size@j;
          end;
        end;
      else
        -- Panic: Kein freier Speicher mehr vorhanden
      try := try+1;
      if memSize-allocSize>size then -- ist prinzipiell ausreichend Speicher vorhanden?
        Compact; -- dann kompaktiere, der zweite Versuch wird dann erfolgreich
      else
        if try=3 then -- nach 2 Collectorzyklen sind alle unerreichbaren Objekte frei
          raise Out_Of_Memory -- danach kann also nur noch eine Exception ausgelöst werden
        end;
      end;
    end;
  end;
end;
```

## 12. Effizienter, inkrementeller Garbage-Collector und Kompaktierer

---

```
        else
            ActivateCollector(Complete-Cycle) -- aktuellen Collectorzyklus beenden
        end;
    end;
end; -- if größeres Objekt gefunden then ... else ...
end; -- if passendes Objekt gefunden then ... elseif ... else ...
end; -- until Result /= Void loop
allocSize := allocSize + size;
allocObj := allocObj + 1;
Result.color := IstWeiß;
Result.type := type;
end; -- Allocate

Free (object: OBJECT) is
-- vom Collector zum Freigeben von Objekten aufgerufen
require
    object /= Void
local
    index: INTEGER;
do
    index := object.type.sizeIndex;
    object.next := freelists@index;
    object.color := IstFrei;
    object.size := sizelist@index;
    allocSize := allocSize - object.size;
    allocObj := allocObj - 1;
    freelists.put(object,index);
end; -- Free
```

## 12.Anhang D: Kompaktierer

Pseudo-Eiffel-Code für den Kompaktieralgorithmus:

```
Compact; -- Kompaktiere Speicher
local
    m,next: Object;
    root: RootRef;
    ref: Reference;
    adr: Object;
    i: INTEGER;
is

-- Phase 0: Vorwärtsverkettete Liste der nach Speicheradressen sortierten Objekte bestimmen
from
    adr := MemStart
    MemList := Void;
until adr=block loop
    if adr.color = IstFrei then -- freier Speicherbereich
        adr := adr + adr.size;
    else -- alloziertes Objekt
        if MemList=Void then
            MemList := adr;
        else
            m.next := adr;
        end;
        adr.next := Void;
        m := adr;
        adr := adr + sizelist@adr.type.sizeIndex;
    end;
end;

-- Phase 1: Neue Adressen bestimmen
from
    m := MemList;
    adr := MemStart;
until m=Void loop
    m.newAdr := adr;
    adr := adr + sizelist@m.type.sizeIndex;
    m := m.next;
end;

-- Phase 2: Zeiger an neue Adressen anpassen

from
    m := MemList;
until m=Void loop
    for all ref : ref is reference in m do
        m.ref := m.ref.newAdr;
    end;
    m := m.next;
end;
for all root : root is Rootobject do
    root := root.newAdr;
end;
```

```
-- Phase 3: Objekte verschieben

from
  m := MemList;
until m=Void loop
  next := m.next;
  MoveMem(m,m.newAdr,sizelist@m.type.sizeIndex); -- Aufruf: MoveMem(src,dst,size)
  m := next;
end;

-- Speicher freigeben:

block := adr;
block.color := IstFrei;
block.size := MemEnd - block;

from
  i := sizelist.lower
until i>sizelist.upper loop
  freelists.put(Void,i);
  i := i+1;
end;

-- Nun sind alle allozierten Objekte zusammengeschoben, die freelists sind leer und der gesamte
-- freie Speicher ist ein durchgängiger Bereich, dessen Adresse block enthält.

end; -- Compact
```

### Literatur:

- [Dijkstra78] Edsger W. Dijkstra, L. Lamport, A. Martin, C. Scholten and E. Steffens: „On-the-Fly Garbage Collection: An Exercise in Cooperation“, Communications of the ACM 21, 11 (November 78), pp. 966-975.
- [Motorola96] „PowerPC Microprocessor Family: The Programming Environments“, Motorola Inc., 1996
- [Wilson93] Paul R. Wilson and Mark S. Johnstone, Dept. of Computer Sciences, University of Texas at Austin, „Position paper for the 1993 ACM OOPSLA Workshop on Memory Management and Garbage Collection.“

---

## 13. Benutzeranleitung

Der für SUN/SPARC unter Solaris implementierte Compiler ist sehr einfach zu installieren und zu benutzen. Das Programm heißt *FEC* und wird von der Shell aus bedient.

### 13.1 Installation

Um FEC zu installieren, ist es ausreichend, das gesamte Verzeichnis des Compilers mit allen mitgelieferten Dateien falls nötig zu entpacken und in ein eigenes Verzeichnis zu kopieren. Dabei darf die Unterverzeichnisstruktur nicht verändert werden. Die folgenden Dateien und Unterverzeichnisse werden benötigt:

<i>fec</i>	Das ausführbare Kommando des Compilers selbst. Diese Datei muss sich bei der Übersetzung im aktuellen Verzeichnis oder in einem in <i>\$PATH</i> angegebenen Pfad befinden.
<i>fecrc</i>	Der ausführbare Compiler wie <i>fec</i> , jedoch mit Überprüfungscode für <i>Preconditions</i> kompiliert.
<i>std_lib/</i>	Dieses Unterverzeichnis enthält die Eiffel-Quelltexte der Standardklassen und den C-Quelltext <i>my_lib.c</i> .
<i>std_lib/ANY.e</i>	Eiffel-Standardklasse.
<i>std_lib/ARRAY.e</i>	Eiffel-Standardklasse.
<i>std_lib/BOOLEAN.e</i>	Eiffel-Standardklasse.
<i>std_lib/CHARACTER.e</i>	Eiffel-Standardklasse.
<i>std_lib/COMPARABLE.e</i>	Eiffel-Standardklasse.
<i>std_lib/DOUBLE.e</i>	Eiffel-Standardklasse.
<i>std_lib/GENERAL.e</i>	Eiffel-Standardklasse.
<i>std_lib/INTEGER.e</i>	Eiffel-Standardklasse.
<i>std_lib/LOW_LEVEL.e</i>	Diese Klasse erlaubt Zugriff auf einige C-Routinen. Sie wird von manchen Standardklassen benutzt.
<i>std_lib/NUMERIC.e</i>	Eiffel-Standardklasse.
<i>std_lib/POINTER.e</i>	Eiffel-Standardklasse.
<i>std_lib/REAL.e</i>	Eiffel-Standardklasse.

<i>std_lib/STRING.e</i>	Eiffel-Standardklasse.
<i>std_lib/my_lib.c</i>	C-Quelltext, in dem nötige Laufzeitroutinen implementiert sind.
<i>obj/</i>	Vom Compiler erzeugte Objektdateien werden in diesem Unterverzeichnis gespeichert. Es muss zudem <i>my_lib.o</i> enthalten.
<i>obj/my_lib.o</i>	Objektdatei von <i>my_lib.c</i> .

## 13.2 Aufruf des Compilers

Die Aufrufsyntax des Kommandos ist

```
> fec <Root Class> [<Creation>]
```

dabei gibt *<Root Class>* den Namen der Wurzelklasse eines Systems an. Das Suffix ".e" darf angegeben werden, ist jedoch nicht nötig. *<Creation>* ist optional und gibt den Namen der *Root-Creation*-Prozedur an. Wird es nicht angegeben, so benutzt *fec* die Routine *make* als *Root-Creation*-Prozedur. Die Argumente sind Case-insensitiv, die Aufrufe „*fec test\_all make*“ und „*fec TEST\_all Make*“ sind also gleichwertig.

## 13.3 Arbeitsweise des Compilers

Der Compiler sucht die Eiffel-Quelltexte im aktuellen Verzeichnis und dem Unterverzeichnis *std\_lib/*. Für jeden eingelesenen Text wird eine Zeile der Art „ - *TEST\_ALL.e*“ auf dem Bildschirm angezeigt.

Die Namen der Quelltextdateien können komplett in Großbuchstaben oder komplett in Kleinbuchstaben geschrieben sein. Sie haben das Suffix ".e", das immer klein geschrieben wird.

Die Übersetzung hat zwei Phasen, nämlich die Analyse, in der die Quelltexte eingelesen werden und ihre Gültigkeit geprüft wird, und dann die Codeerzeugung, in der Objektdateien erzeugt werden. Die Objektdateien werden im Unterverzeichnis *obj/* gespeichert.

Die Objektdateien bekommen als Namen die Namen der Klassen, deren Code sie darstellen, mit dem Suffix „.o“. Bei generischen Klassen werden zudem die aktuellen generischen Parameter im Namen in eckigen Klammern und mit Minuszeichen getrennt angegeben. Sind die aktuellen generischen Parameter Referenzen, so wird für sie nur „\_ref“ eingetragen. Objektdateien bekommen einen Präfix "r#" oder "x#" für deren explizite Referenz- bzw. Expanded-Klassen. Objektdateinamen werden vor dem Suffix auf max. 29 Zeichen gekürzt (dies führt bei langen Klassennamen zu Problemen).

Eine besondere Objektdatei ist „*obj/main.o*“. Sie enthält den zum Start eines Programms nötigen Code.

Um das Linken zu erleichtern, wird noch die Datei „*elink*“ erzeugt. Dies ist ein kleines Skript, das *gcc* aufruft, um ein ausführbares Programm aus den Objektdateien zu erzeugen. Das ausführbare Programm bekommt den Namen der Wurzelklasse in Kleinbuchstaben ohne Suffix.

## 13.4 Beispielcompilation

Das Beispielprogramme „*test\_all*“, kann wie folgt compiliert werden:

```
> fec test_all
```

Der Compiler erzeugt hierfür folgende Ausgabe:

```
FEC -- Fridi's Eiffel Compiler
- TEST_ALL.e
- std_lib/ANY.e
- std_lib/GENERAL.e
- std_lib/BOOLEAN.e
- std_lib/LOW_LEVEL.e
- std_lib/STRING.e
- std_lib/COMPARABLE.e
- HELLO.e
- SIEVE.e
- INSPECT_TEST.e
- REAL_TEST.e
- DYNAMIC_TEST.e
- GENERIC_TEST.e
- std_lib/INTEGER.e
- std_lib/NUMERIC.e
- std_lib/POINTER.e
- std_lib/CHARACTER.e
- std_lib/DOUBLE.e
- std_lib/REAL.e
- std_lib/ARRAY.e
- C.e
- A.e
- B.e
- D.e
- NUMBERED_STRING.e
- SORTABLE.e
- LIST.e
- PS_ARRAY.e
- SORTED_ARRAY.e
- BINARY_SEARCH_ARRAY.e
getting used types:
getting type sizes:
finding features to be duplicated:
compiling:
+ obj/test_all.o
+ obj/any.o
+ obj/hello.o
+ obj/sieve.o
+ obj/inspect_test.o
+ obj/real_test.o
+ obj/dynamic_test.o
+ obj/generic_test.o
+ obj/general.o
+ obj/integer.o
+ obj/array[boolean].o
+ obj/boolean.o
+ obj/character.o
+ obj/real.o
+ obj/double.o
+ obj/a.o
+ obj/b.o
+ obj/c.o
+ obj/d.o
+ obj/numbered_string.o
+ obj/list[_ref].o
```

```
+ obj/ps_array[_ref-_ref].o
+ obj/string.o
+ obj/sorted_array[_ref-_ref].o
+ obj/low_level.o
+ obj/r#integer.o
+ obj/numeric.o
+ obj/comparable.o
+ obj/pointer.o
+ obj/r#boolean.o
+ obj/r#character.o
+ obj/r#real.o
+ obj/r#double.o
+ obj/sortable[_ref].o
+ obj/array[_ref].o
+ obj/binary_search_array[_ref-_ref].o
+ obj/r#low_level.o
+ obj/r#pointer.o
+ obj/main.o
+ elink
ok.
315 internal routines
278 inherited routines
283 locals used (max)
27 locals used (average)
49108 Bytes machinecode created.
```

Als nächstes kann das Skript *elink*, das einen Aufruf „*gcc -o test\_all*“ gefolgt von den Namen aller erzeugten Objektdateien und *obj/my\_lib.o* enthält, zum Erzeugen eines ausführbaren Programmes benutzt werden:

```
> elink
```

Schließlich kann das übersetzte Beispielprogramm mit

```
> test_all
```

ausgeführt werden.

## 13.5 Standardbibliotheken

Als Dokumentation für die bisher implementierten Standardklassen sollten einfach deren Quelltexte benutzt werden. Der Compiler erwartet eine Reihe an Features der Standardklassen exakt so, wie sie dort implementiert sind. Änderungen an den Standardklassen können zu Compilerabstürzen oder falschem Code führen.

---

Ich versichere,  
dass ich diese Arbeit selbstständig verfasst und  
nur die angegebenen Quellen verwendet habe.

---

(Fridtjof Siebert)