Universität Hamburg
Fachbereich Informatik

Studienarbeit

Bootstrap einer persistenten objektorientierten Programmierumgebung

eingereicht bei:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
TU HAMBURG HARBURG

vorgelegt von:
Axel Wienberg
Hinzeweg 9
21075 Hamburg
Tel.: 040 - 76 76 284

Hamburg, den 12. August 1997

Typographie

In dieser Arbeit werden unterschiedliche Schriftarten verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist in der folgenden Tabelle wiedergegeben:

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	englischer Begriff oder Definition	*class precedence list*
Sans Serif	Firmen- oder Produktname	Tycoon-2 yacc
`Typewriter`	TL-2-Ausdrücke oder Klassennamen	`out.writeString("hi!")` `MethodManager`

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Motivation 1

1.2. Problemstellung 2

1.3. Struktur der Arbeit 3

2. Die gewählte Bootstrap-Prozedur **5**

2.1. Ausgangslage 5

2.2. Zielzustand 5

2.3. Weg 5

2.4. Aus dem Bootstrap resultierende Anforderungen 6

3. Anforderungen **9**

3.1. Die Sprache TL-2 9

3.1.1. Die Klassen-Präzedenz-Liste 10

3.1.2. Metaklassen 11

3.1.3. Objekterzeugung 12

3.1.4. Variablen 12

3.1.5. Wertausdrücke 12

3.1.6. Funktionen höherer Ordnung 14

3.1.7. Ausnahmen 14

3.1.8. Zusicherungen 15

3.1.9. Typen 15

3.2. Benutzerinteraktion 16

3.2.1. Typüberprüfung 16
3.2.2. Klassendefinitionen 17
3.2.3. DO-Kommandos 17
3.3. Die virtuelle Zielmaschine 18
3.3.1. Der Objektspeicher 19
3.3.2. Die Klassentabelle 19
3.3.3. Versenden von Nachrichten 20
3.3.4. Methoden 20
3.3.5. Bytecode 21
3.4. Erweiterungen 22

4. Architektur **23**

5. Beschreibung ausgewählter Systemkomponenten **27**
5.1. Lexikalische und Syntaktische Analyse 27
5.2. Semantische Analyse 29
5.3. Zwischencodeerzeugung 31
5.4. Codeoptimierung 32
5.5. Codeerzeugung 33
5.6. Verwaltung des Klassengraphen 33
5.6.1. Vererbung 34
5.6.2. Isoliertheit der Änderungen 34
5.6.3. Metaklassen 35
5.6.4. Kapselung der Klassenobjekte 35

6. Zusammenfassung und Ausblick **37**

A. Syntax von Tycoon-2 **39**

A.1. Tokens 39

A.2. Reservierte Schlüsselworte 41

A.3. Übersetzungseinheit 41

A.4. Klassen 42

A.5. Werte 43

A.6. Signaturen 44

A.7. Typen 44

A.8. Präzedenzen 44

B. Beispielsitzung **45**

C. Dateiauflistung **46**

C.1. builtins.h 46

C.2. tyc.h 49

C.3. tvm.doc 54

C.4. Schnittstellen der Systemkomponenten 59

Literaturverzeichnis **65**

1. Einleitung

Die Erfahrung bei der Implementation großer Software-Systeme zeigt, daß die verwendeten modularen, imperativen Programmiersprachen zu wenig Möglichkeiten für die Wiederverwendung und Erweiterung von Programmkomponenten bieten. Genau dieses Problem verspricht die objektorientierte Softwareentwicklung zu lindern. Daher liegt es nahe, für ein neues Softwaresystem eine objektorientierte Anwendungs- und Implementationssprache zu wählen.

Das in dieser Arbeit beschriebene Tycoon-2 ist aus dem am Arbeitsbereich DBIS entstandenen Tycoon-System hervorgegangen. Tycoon bietet ein persistentes Objektsystem mit einer integrierten Programmierumgebung. Für die Konstruktion großer Systeme wichtig sind insbesondere die statische Typisierung, das polymorphe Typsystem und die Modularität der verwendeten Programmiersprache TL.

Tycoon-2, der kommerzielle Nachfolger von Tycoon, erhält diese Eigenschaften und fügt konsequente Objektorientierung in der Programmiersprache TL-2 hinzu.

1.1. Motivation

Ziel der Arbeit ist eine objektorientierte Implementation der Programmierumgebung für TL-2 in TL-2, die Code für eine speziell auf TL-2 zugeschnittene virtuelle Maschine erzeugt.

Grundlage dafür bildet eine in der funktional-imperativen Programmiersprache TL des Tycoon-Systems vorliegende prototypischen Implementation der Programmierumgebung für TL-2. Die Entwicklung muß zunächst in dieser Umgebung stattfinden und sich dann mittels *Bootstrap* von ihr lösen.

Unter einem *Bootstrap* versteht man die Implementation eines Programms, das in der Lage ist, sich selbst zu übersetzen. Der genaue Ablauf wird in Kapitel 2 beschrieben. Die Programmierumgebung wird in ihrer eigenen Sprache implementiert und wird mit sich selbst übersetzt, um von folgenden Vorteilen zu profitieren:

Eigenständigkeit Die Programmierumgebung wird unabhängig von der alten Umgebung. Dies ist besonders wichtig, wenn Performanz, Benutzerfreundlichkeit oder Wartung der alten Umgebung zu wünschen übrig lassen.

Einheitlichkeit Durchgängige Verwendung einer Programmiersprache erhöht die Einheitlichkeit und daher auch Verständlichkeit des Systems.

Selbsttest Die Programmierumgebung ist ein nichttriviales Softwareprodukt und daher ein wertvoller Test für Korrektheit, Performanz und Robustheit der Programmierumgebung und der Bibliotheken der implementierten Sprache.

Geschwindigkeit Verbesserungen in der Codeerzeugung kommen direkt der Programmierumgebung zugute.

Direkte Rückmeldungen Die Autoren der Programmierumgebung sind selbst auch Anwender der Programmierumgebung, so daß Benutzererfahrungen aus erster Hand vorliegen und verwertet werden können.

Sprachentwurf Bei experimentellen Sprachen wie TL-2 kommt Erfahrung über die Tauglichkeit des Sprachentwurfs hinzu.

Bibliotheken Die Programmierumgebung kann die Möglichkeiten der Programmiersprache voll ausschöpfen. Insbesondere hat sie Zugriff auf alle Bibliotheken, auch solche, die ursprünglich für Anwendungen entwickelt wurden (z.B. zur graphischen Benutzerinteraktion)

Dadurch, daß die Programmierumgebung auf der eigenen Zielmaschine läuft, müssen die Ergebnisse der Übersetzung nicht erst auf eine andere Maschine übertragen werden, bevor sie ausgeführt werden können. Durch die sprachliche Einheitlichkeit eröffnet sich zudem die Möglichkeit, auf Bestandteile der Programmierumgebung zur Laufzeit als vollwertige Sprachobjekte zuzugreifen (*Reflektion*). Die Unterscheidung zwischen der Schnittstelle zum System und den Schnittstellen zu anderen Anwendungsteilen wird unnötig. Bibliotheken werden gemeinsam von Anwendungen und Programmierumgebung benutzt, was den Speicherbedarf senkt.

1.2. Problemstellung

Die von der prototypischen Implementation der Programmierumgebung gebotene Funktionalität soll analysiert werden. Davon ausgehend sind die Komponenten der neu zu implementierenden Programmierumgebung zu identifizieren. Eine Orientierung an der Architektur des Prototyps ist aufgrund des Paradigmenwechsels nur begrenzt sinnvoll.

Die Programmierumgebung soll mit Hilfe des Prototyps so weit implementiert werden, daß auf der neuen Maschine TL-2-Programme übersetzt und inkrementell geladen werden können. Dieser Zustand soll möglichst schnell erreicht werden, um kommerzielle Anwendungen mit der neuen Maschine schneller und speicherschonend ausführen zu können. Bis alle Entwicklungswerkzeuge auch in der neuen Programmierumgebung zur Verfügung stehen, muß die Anwendungsentwicklung weiterhin mit Hilfe des Prototyps stattfinden, was hohe Anforderungen an die Kompatiblität des neuen Systems stellt.

1.3. Struktur der Arbeit

Kapitel 2 beschreibt den Weg vom Prototypen zum eigenständigen Tycoon-2-System unter Verwendung eines *Bootstrap*.

In Kapitel 3 werden die äußeren Anforderungen an das System dargestellt: Die zu übersetzende Programmiersprache, die durch den Prototyp vorgegebene Schnittstelle zum Benutzer und die Schnittstelle zur virtuellen Maschine.

Kapitel 4 gibt einen Überblick über die Architektur des Systems, der in Kapitel 5 anhand von Übersetzer und Klassenverwaltung vertieft wird.

Diese Arbeit findet ihren Abschluß im Kapitel 6 mit einer Zusammenfassung und einer Bewertung des entstandenen Systems.

2. Die gewählte Bootstrap-Prozedur

Dieses Kapitel beschreibt den in der Arbeit gewählten Weg von der prototypischen TL-2-Entwicklungsumgebung im Tycoon-System zum eigenständigen Tycoon-2-System. Da das Tycoon-2-System schließlich in der Lage ist, sich selbst zu übersetzen, also von Tycoon unabhängig wird, ist dieser Vorgang ein *Bootstrap*.

2.1. Ausgangslage

In der Ausgangssituation verfügen wir über das Tycoon-System mit seiner Programmiersprache TL. Das Tycoon-System läuft auf einer virtuellen Maschine, der *tycoon virtual machine* TVM. Der Prototyp der TL-2-Programmierumgebung läuft im Tycoon-System und erzeugt Code für die TVM.

Der TL-2-Prototyp ist in der Sprache TL implementiert und wird von der TL-Programmierumgebung des Tycoon-Systems übersetzt. Aufgrund der großen Unterschiede zwischen TL und TL-2 wird die TL-Implementation hier nicht näher betrachtet.

2.2. Zielzustand

Im zu erreichenden Zustand läuft die Programmierumgebung für TL-2 innerhalb des Tycoon-2-Systems. Dieses verwendet die speziell auf TL-2 angepasste virtuelle Maschine TVM-2. Der erzeugte ausführbare Code ist TVM-2-Code.

Die Programmierumgebung ist in TL-2 implementiert. Es besteht die Möglichkeit, die Programmierumgebung mit sich selbst zu übersetzen und so ein neues Tycoon-2-System zu erzeugen.

2.3. Weg

Da sich sowohl Implementations- als auch Ausgabesprache der Quell- und der Ziel-Programmierumgebung deutlich unterscheiden, muß die Programmierumgebung neu geschrieben werden. Der eigentliche Bootstrap besteht dann darin, sie in eine auf der TVM-2

ausführbare Form zu bringen. Nach [Lecarme, Peyrolle-Thomas 78] handelt es sich hier um einen *half-bootstrap*, da die Entwicklung auf einer anderen Maschine als der Zielmaschine stattfindet.

Bei einem *full-bootstrap* findet die komplette Entwicklung auf der Zielmaschine statt. Dies setzt allerdings voraus, daß auf der Zielmaschine bereits eine Programmierumgebung für eine höhere Programmiersprache existiert, was bei der neu entwickelten TVM-2 nicht der Fall ist.

Mit Hilfe der von [Bratman 61] eingeführten und von [Earley, Sturgis 70] in die heutige Form gebrachten T-Diagramme läßt sich der gewählte Ablauf wie in Abbildung 2.1 darstellen. Existierende Komponenten sind schraffiert dargestellt, zu erstellende Komponenten sind grau unterlegt.

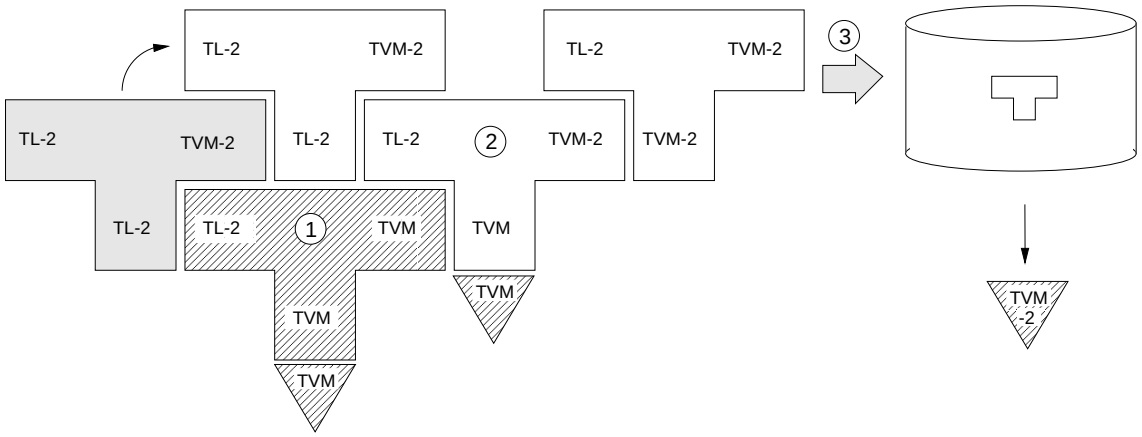


Abbildung 2.1.: Übersetzung der TL-2-Programmierumgebung

Die neue Programmierumgebung (in der Zeichnung grau unterlegt) wird im ersten Schritt mit Hilfe des Prototyps (schraffiert) in eine auf der TVM ausführbare Form gebracht. Sie wird im zweiten Schritt benutzt, um sich selbst zu übersetzen. Im dritten Schritt wird das Ergebnis aus dem Tycoon-System herausgeschrieben und von der TVM-2 als neues Tycoon-2-System gestartet. Zur Überprüfung kann dieser Prozeß wie in Abbildung 2.2 dargestellt im Tycoon-2-System wiederholt werden. Die Programmierumgebung wird im Tycoon-2-System übersetzt und herausgeschrieben. Das Ergebnis muß identisch zu dem vom Tycoon-System aus erzielten sein.

2.4. Aus dem Bootstrap resultierende Anforderungen

Der beschriebene Weg impliziert einige Anforderungen an die Programmierumgebung. Wie [Appel 94] beobachtet, kann Bootstrapping über die geschickte Handhabung von T-Diagrammen hinausgehen, wenn Übersetzer und übersetztes Programm wichtige Datenfor-

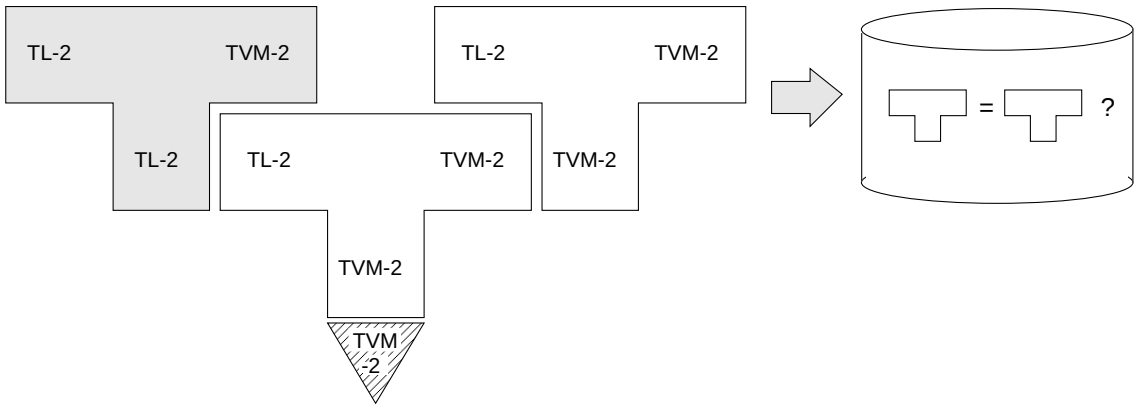


Abbildung 2.2.: Validierung der TL-2-Programmierumgebung

mate und Teile der Umgebung gemeinsam nutzen. Im Tycoon-2-System ist dies aufgrund der Inkrementalität des Übersetzers und der dynamischen Integration übersetzter Klassen sicherlich der Fall. Eine strenge Axiomatisierung wie bei Appel wurde jedoch nicht durchgeführt. Stattdessen werden veränderte Versionen der Programmierumgebung so geschrieben, daß ihr Quellcode im alten System und auf der alten virtuellen Maschine lauffähig ist und keinen übersetzten Code ausführt, so lange der Bootstrap nicht vollständig ist. Die Entscheidung, ob eine Änderung des Compilers einen neuen Bootstrap erfordert, wird ad hoc gefällt (im Zweifelsfalle experimentell).

Zunächst einmal muß der Quellcode der Programmierumgebung also auf beiden virtuellen Maschinen lauffähig sein. Daher muß der für den Bootstrap benötigte Teil der TVM-2-Schnittstelle auf der TVM emuliert werden. Die Programmierumgebung muß an diesen Stellen abfragen, auf welcher Maschine sie gerade ausgeführt wird, und ggf. das Verhalten in TL-2 nachbilden, soweit es die darüberliegenden Komponenten betrifft.

Außerdem darf die Programmierumgebung nicht davon ausgehen, daß ihre Ausführungsumgebung gleich ihrer Zielumgebung ist, auch wenn dies im normalen Betrieb auf der TVM-2 der Fall sein wird. Die von der Programmierumgebung *benutzte* Klasse `Foo` ist also i.A. nicht die gleiche wie die von ihr *verwaltete*.

Zwecks Bootstrap und Validierung muß es möglich sein, mindestens zwei Exemplare der Programmierumgebung im selben Store zu halten. Das zweite Exemplar schreibt den erzeugten Code in ein neues Systemabbild, statt ihn an die aktuelle virtuelle Maschine weiterzugeben. Daher dürfen die Komponenten der Programmierumgebung nicht als *Singleton* implementiert werden.

Eine weitere Anforderung ergibt sich aus der in Abbildung 2.1 und 2.2 als grauer Pfeil dargestellten Erzeugung eines neuen Stores. In dem auf Platte geschriebenen Systemabbild dürfen nur Objekte enthalten sein, die von der TVM-2 interpretierbar sind. Bei der Erzeugung des neuen Stores werden daher alle Verweise auf die TVM-Klassen durch Verweise auf

die verwalteten TVM-2-Klassen ersetzt. Dies ist nicht möglich, wenn der im alten System konstruierte Objektgraph Funktionsabschlüsse enthält, da deren Verhalten nicht eindeutig durch ihre Klasse beschrieben wird. Soweit in den am Systemabbild beteiligten Klassen parametrisiertes Verhalten erwünscht ist, muß dies also durch Vererbung statt durch Funktionsabschlüsse implementiert werden.

3. Anforderungen

Dieses Kapitel beschreibt die Anforderungen an die äußeren Schnittstellen des Systems. Dazu gehört das erwartete Verhalten gegenüber dem Benutzer, und auf der anderen Seite die Schnittstelle zur virtuellen Maschine.

Die Programmierumgebung nimmt eine Mittlerstellung zwischen Benutzer und virtueller Maschine ein. Für den Benutzer entsteht die Illusion, seine Kommandos würden direkt ausgeführt; die tatsächliche interne Verarbeitung bleibt verborgen. Dadurch wird es möglich, die Interna (wie für diese Arbeit geschehen) ohne Einfluß auf den Benutzer auszutauschen.

Dies entspricht der Implementations- und Plattformunabhängigkeit von Programmiersprachen, geht aber noch weiter, da nicht nur funktional das Verhalten des Programmes in Abhängigkeit vom Quelltext beschrieben wird, sondern auch der Prozeß der Programmerstellung in Abhängigkeit von Benutzeraktionen.

Im folgenden wird kurz auf die intendierte Programmsemantik von TL-2-Quelltexten eingegangen. Eine Beschreibung der Syntax von TL-2 findet sich in Anhang A. Die Interaktion zwischen Programmierer und Programmierumgebung wird in Abschnitt 3.2 dargestellt.

Nach unten interagiert die Programmierumgebung mit der virtuellen Maschine, um die Benutzerwünsche nach geeigneter Vorverarbeitung zur Wirkung zu bringen. Die Schnittstelle der virtuellen Maschine wird in Abschnitt 3.3 beschrieben.

3.1. Die Sprache TL-2

TL-2 ist eine typisierte, rein objektorientierte Sprache mit Mehrfachvererbung und Metaklassen. Ein Problem wird mit Hilfe von *Objekten* modelliert, die in *Exemplarvariablen* ihren aktuellen Zustand speichern. Jedes Objekt ist *Exemplar* einer *Klasse*, die definiert, über welche Exemplarvariablen das Objekt verfügt und wie das Objekt auf *Nachrichten* reagiert. Eine Nachricht besteht aus einem *Selektor* sowie weiteren *Argumenten*. Die Klasse ordnet einem Selektor die Implementation durch eine *Methode* zu. Empfängt ein Objekt eine Nachricht, so wird die dem Selektor entsprechende Methode mit einem Verweis auf das *Subjekt* (*self*) und mit den aktuellen Argumenten aufgerufen, und das Berechnungsergebnis wird dem *Aufrufer* zurückgegeben.

Alle Zugriffe auf den Objektzustand finden über Nachrichten statt. Für jede Exemplarvariable werden automatisch eine Lese- und eine Schreibmethode erzeugt; für den Aufrufer ist nicht ersichtlich, ob die aufgerufene Methode direkt auf eine Exemplarvariable zugreift oder ob sie vom Benutzer definiert ist. Daher wird die äußere Schnittstelle einer Klasse vollständig durch die von ihren Exemplaren verstandenen Nachrichten beschrieben.

Von außen kann nur auf die *öffentlichen* Methoden einer Klasse zugegriffen werden. Vom Subjekt aus ist auch der Zugriff auf *private* Methoden möglich. Gleiches gilt für öffentliche und private Exemplarvariablen.

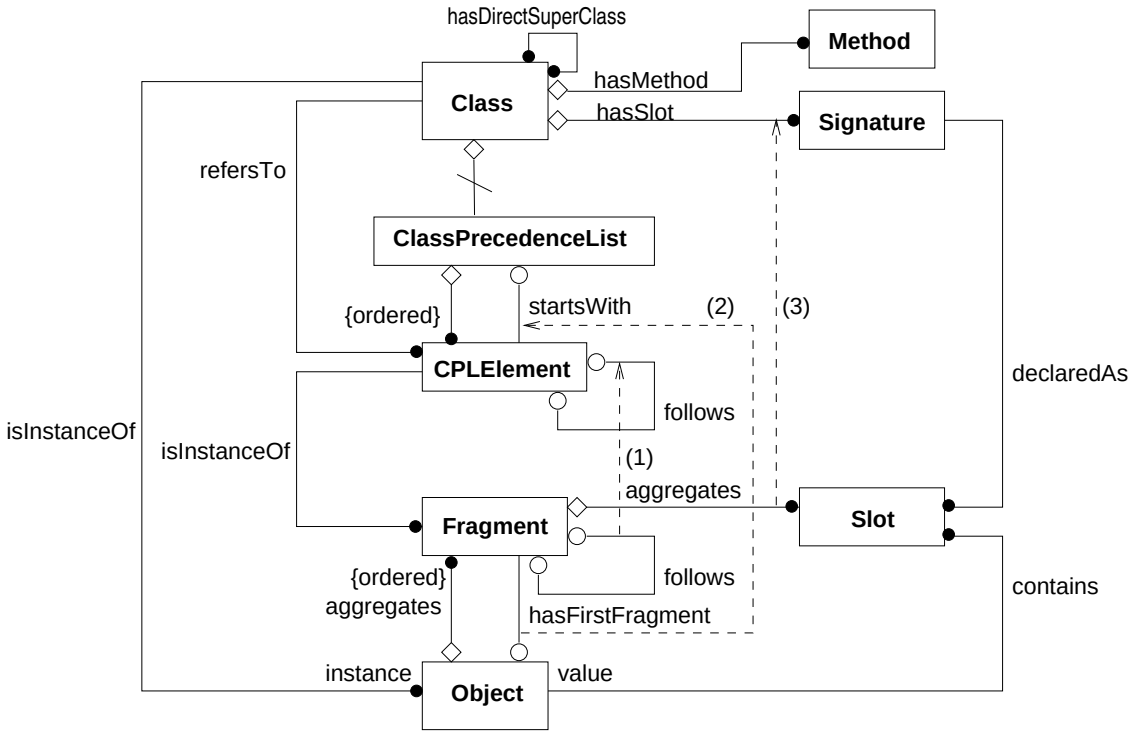
Diese Kapselung hinter einer Nachrichtenschnittstelle erlaubt eine Reimplementation ohne Einfluß auf bestehende Verwendungen. Besonders interessant ist dies im Zusammenhang mit *Vererbung*: Eine Klasse kann als Spezialisierung einer oder mehrerer *Vorfahrenklassen* definiert werden. Sie muß die vollständige Schnittstelle der Vorfahrenklassen erfüllen, kann sie aber durch eigene Methoden erweitern, und kann ererbte Methoden *redefinieren*, d.h. ihre Implementation austauschen. Auch bei Redefinition einer Methode kann die ererbte Implementation weiterverwendet werden, indem die neue Implementation die vorhergehende explizit mittels *super send* aufruft.

3.1.1. Die Klassen-Präzedenz-Liste

In Zusammenhang mit Mehrfachvererbung wird wichtig, welche von verschiedenen ererbten Implementationen einer Methode zum Zuge kommt, und welche Implementation beim *super send* angesprochen wird. Dafür verwendet TL-2 das aus Loops [Bobrow, Stefik 81] bekannte Konzept der *Klassen-Präzedenz-Liste* (*class precedence list*). Der Graph der Vorfahrenklassen wird so linearisiert, daß jede Klasse vor ihren Vorfahrenklassen erscheint und für jede Klasse die bei der Angabe der Vorfahrenklassen gegebene Reihenfolge beibehalten wird. Klassen, für deren Vorfahrengraph sich keine solche Linearisierung finden läßt, z.B. weil er einen Kreis enthält, werden vom System nicht akzeptiert.

Empfängt ein Objekt nun eine Nachricht, wird die Klassen-Präzedenz-Liste von vorn nach hinten durchgegangen, und die erste Methode mit passendem Selektor wird ausgeführt. Kommt es während der Ausführung zu einem *super send*, wird ab der letzten Fundstelle weitergesucht. Existiert keine Methode zum angegebenen Selektor, wird ein Ausnahmezustand ausgelöst (s. Abschnitt 3.1.7, Ausnahmen).

Die Klassen-Präzedenz-Liste bestimmt auch, aus welchen Exemplarvariablen Exemplare der Klasse bestehen. Wie in Abbildung 3.1 dargestellt, existiert für jedes Element der Klassen-Präzedenz-Liste genau ein Objektfragment mit allen in der jeweiligen Klasse definierten Exemplarvariablen.



(1): { Fragment1 follows Fragment2
=> Fragment1.CPLElement follows Fragment2.CPLElement }
(2): { Object hasFirstFragment Fragment
=> Object.Class.ClassPrecedenceList startsWith Fragment.CPLElement }
(3): { Fragment aggregates Slot
=> Fragment.CPLElement.Class hasSlot Slot.Signature }

Abbildung 3.1.: Vererbung in TL-2

3.1.2. Metaklassen

In TL-2 sind auch Klassen Objekte der Berechnung. Daher ist auch jede Klasse wiederum Exemplar einer Klasse, ihrer *Metaklasse*. Metaklassen zeichnen sich dadurch aus, daß sie Exemplare der besonderen Klasse `MetaClass` sind, die die Wurzel dieser Reflektion bildet (d.h. `MetaClass` ist ein Exemplar ihrer selbst).

Im Gegensatz zu Smalltalk [Goldberg, Robson 83] wird vom System keine eins-zu-eins-Beziehung zwischen Klasse und Metaklasse vorgeschrieben. Es ist problemlos möglich, mehrere benutzerdefinierte Klassen als Exemplare einer gemeinsamen Metaklasse (z.B. `SimpleConcreteClass`) zu definieren.

3.1.3. Objekterzeugung

Klassenobjekte sind dafür zuständig, ihre Exemplare zu erzeugen. Intern geschieht dies mit Hilfe der vordefinierten privaten Methode `_new`, die von einer Systemklasse ererbt wird. Metaklassen können in ihrer öffentlichen Schnittstelle verschiedenste Methoden zur Erzeugung von Objekten in verschiedenen Anfangszuständen anbieten.

3.1.4. Variablen

TL-2 verfügt über vier Arten von Variablen: die oben beschriebenen Exemplarvariablen, sowie *globale Variablen*, *Argumentvariablen* und *lokale Variablen*. Exemplarvariablen und globale Variablen werden indirekt über Nachrichten zugegriffen, während lokale und Argumentvariablen direkt gebunden werden.

Argumentvariablen (auch formale Argumente genannt) werden an die einer Nachricht beigefügten aktuellen Argumente gebunden und sind nicht änderbar. Sie sind innerhalb des gesamten Methodenkörpers sichtbar.

Lokale Variablen haben ebenfalls einen statisch definierten Sichtbarkeitsbereich, der von der Stelle der Definition bis zum Ende des umschließenden Blocks verläuft. Sie werden bei der Definition mit berechneten Werten initialisiert und können später beliebig oft neu zugewiesen werden.

Globale Variablen sind systemweit zugreifbar. Erhält ein Objekt eine Nachricht, die von keiner Vorfahrenklasse verstanden wird, wird zuletzt im globalen Variablenverzeichnis (*pool*) gesucht. Für jede vom Benutzer definierte globale Variable existieren dort eine Methode für lesenden und eine für schreibenden Zugriff. Zusätzlich sind alle im System definierten Klassen im *pool* unter ihrem Namen zugreifbar.

3.1.5. Wertausdrücke

Dieser Abschnitt gibt einen informellen Überblick über die Evaluations-Semantik von Wertausdrücken in TL-2. Grundlage sind abstrakte Syntaxbäume, wie sie in Abbildung 3.2 in OMT-Notation dargestellt werden. Die konkrete Syntax für TL-2 findet sich in Anhang A.

Beim Aufruf einer Methode wird der zugehörige Methodenkörper in einer neu geschaffenen *Umgebung* ausgewertet, die *Bindungen* für den Empfänger und die Argumente der Nachricht enthält. Gebundene Werte können durch Bezeichner-Ausdrücke (`IdeValue`) referenziert werden, und sie können durch Zuweisungs-Ausdrücke (`AssignValue`) modifiziert werden, falls es sich um *änderbare* Bindungen handelt.

`LetValue`-Ausdrücke dienen zur Erweiterung der Umgebung. Der Unterausdruck `initialValue` wird in der ursprünglichen Umgebung ausgewertet, und der resultierende

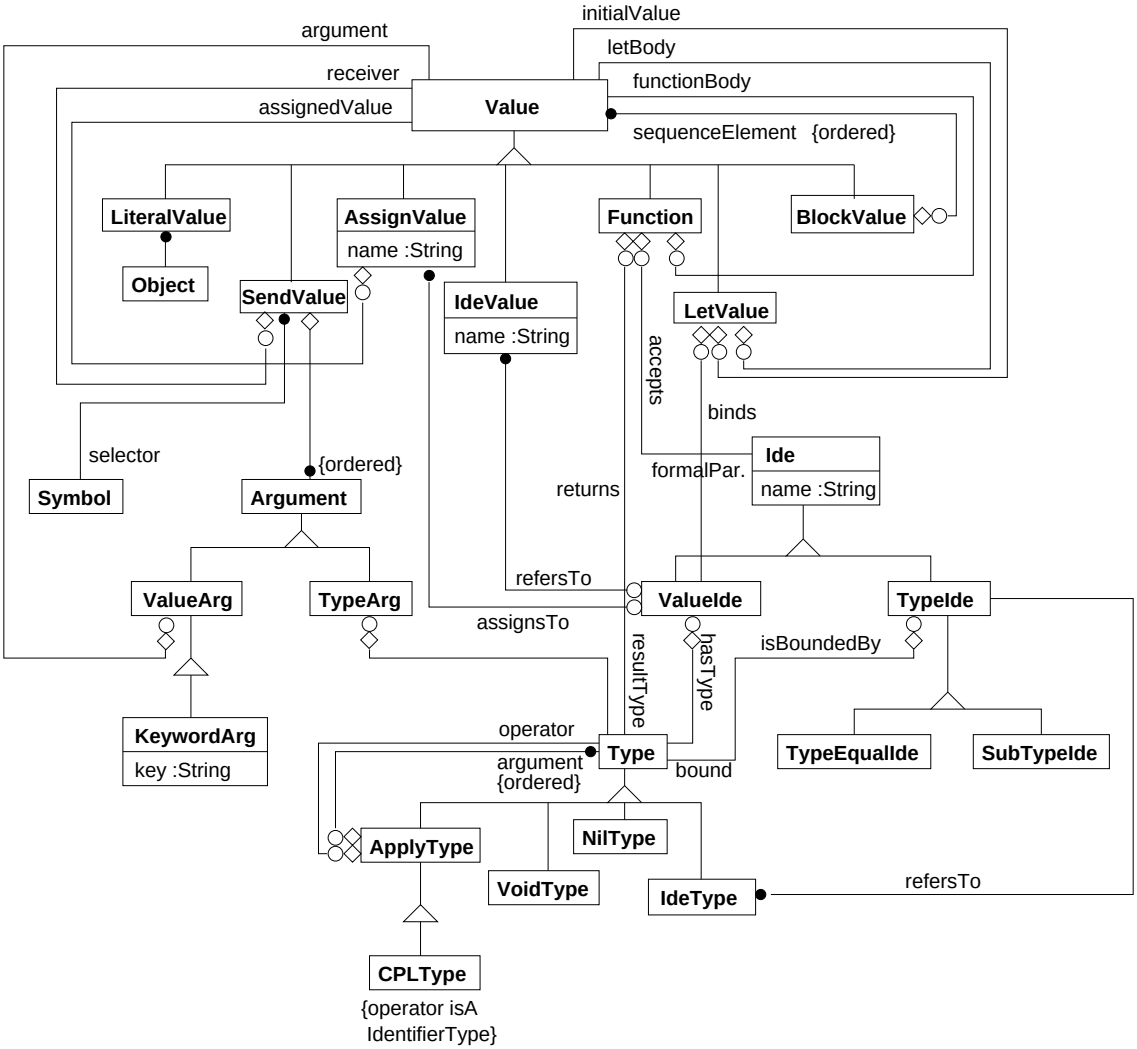


Abbildung 3.2.: Statisches Objektmodell für Wertausdrücke in TL-2

Wert für die Auswertung der Körper-Ausdrucks an den Bezeichner (`ValueSig`) gebunden. Der Körper-Ausdruck wird in der erweiterten Umgebung ausgewertet.

Die aktuelle Umgebung kann durch Funktionsausdrücke (`Function`) eingefangen werden (s. Abschnitt 3.1.6, Funktionen höherer Ordnung). Die Lebensdauer von Bindungen ist daher nicht begrenzt.

Aktionen außerhalb des Methodenkörpers werden durch `SendValue`-Ausdrücke angesprochen. Empfänger und Argumente werden von links nach rechts ausgewertet, und dem Empfänger-Objekt wird eine Nachricht aus Selektor und Argument-Objekten gesandt. Um anhand Empfänger und Selektor die aufzurufende Methode zu identifizieren, wird der in

Abschnitt 3.1.1 (Die Klassen-Präzedenz-Liste) beschriebene Mechanismus benutzt.

Nachrichten werden in TL-2 grundsätzlich synchron gesendet, d.h. der Aufrufer wartet mit der weiteren Ausführung bis zur Rückkehr der aufgerufenen Methode.

3.1.6. Funktionen höherer Ordnung

Durch Funktionsausdrücke können dynamisch Funktionsobjekte erzeugt werden, die eigenes Verhalten zeigen, ohne die Definition einer eigenen Klasse zu erfordern. Funktionsausdrücke sind überall dort möglich, wo TL-2-Ausdrücke erlaubt sind. Aus dem Körper des Funktionsausdrucks können weiter außen definierte Variablen (*freie Variable*) einschließlich des Subjekts referenziert werden, es gelten die statischen Sichtbarkeitsbereiche.

Bei der Auswertung eines Funktionsausdrucks wird die aktuelle Umgebung zusammen mit dem Funktionskörper in einem Funktionsobjekt (*closure*) eingefangen. Wie alle TL-2-Objekte haben Funktionsobjekte unbegrenzte Lebenszeit. Funktionsobjekte verstehen eine spezielle Aufrufs-Nachricht, die den Funktionskörper in der um die Argumente erweiterten eingefangenen Umgebung aufruft. Man beachte, daß eingefangene Bindungen, insbesondere änderbare, von allen betroffenen Funktionen und allen ihren Aufrufen gemeinsam benutzt werden.

Funktionen höherer Ordnung sind besonders nützlich, um Iterationen und bedingten Code kompakt formulieren zu können. Weitere Anwendung finden sie zur Parametrisierung und Verknüpfung existierender Komponenten.

3.1.7. Ausnahmen

Bei der Ausführung können Ausnahmezustände eintreten, etwa, wenn dem Objekt `nil` eine Nachricht gesendet wird, oder wenn eine Zusicherung nicht eingehalten wird (s. Abschnitt 3.1.8, Zusicherungen). Ausnahmen sind als Laufzeitobjekte zugreifbar. Von der Klasse `Exception` erben sie die Methode `raise`, mit der sie auch explizit vom Programmierer ausgelöst werden können.

Für die Verarbeitung von Ausnahmen stellt TL-2 das `try`-Konstrukt zur Verfügung. `try` besteht aus einem Körper und einer Ausnahmebehandlung. Der Körper wird ausgewertet, und wenn währenddessen keine Ausnahme auftritt, ist sein Ergebnis auch das Ergebnis des `try`-Ausdrucks. Im Falle einer Ausnahme wird die Ausnahmebehandlung mit dem aufgetretenen Ausnahmeobjekt als Argument aufgerufen, und das Ergebnis der Ausnahmebehandlung wird zum Ergebnis des `try`.

Ausnahmen, die während einer Ausnahmebehandlung auftreten, werden nicht mehr vom gleichen `try` behandelt. Sie werden wie bei normalen Ausdrücken an das nächstäußere `try` in der Aufrufkette zurückgereicht.

3.1.8. Zusicherungen

Die von Eiffel [Meyer 88] bekannte Programmierung durch Verträge wird in TL-2 durch die optionale Angabe von Vor- und Nachbedingungen unterstützt. Die Vorbedingung einer Methode gibt an, welche Leistungen vom Aufrufer zu erbringen sind, unter welchen Bedingungen die Nachricht also gesendet werden darf; jede Implementation dieser Schnittstelle verpflichtet sich, bei erfüllter Vorbedingung die Nachbedingung umzusetzen oder mit einer Ausnahme abzubrechen.

Eine weitere aus Eiffel bekannte Art der Zusicherung sind die Klasseninvarianten. Diese grenzen die vom Programmierer erwarteten und erlaubten Objektzustände ein.

Im Methodentext können Zusicherungen mit Hilfe des Schlüsselwortes `assert` aufgeführt werden. Es handelt sich hier um die Spezifikation einer konkreten Implementation, nicht wie bei bei Klasseninvarianten und Vor- und Nachbedingungen um einen Teil der Klassenschnittstelle.

Zusicherungen im Methodentext sind die einzigen, die im untersuchten System Einfluß auf die Programmausführung haben. Klasseninvarianten und Vor- und Nachbedingungen mit ihrer methodisch wichtigen Funktion der Dokumentation und Spezifikation werden als Kommentar behandelt.

3.1.9. Typen

Typen in TL-2 sind orthogonal zu den ausführbaren Bestandteilen. In einem typkorrekten Programm ist sichergestellt, daß Objekte nur Nachrichten erhalten, die sie auch verstehen. Typen dienen somit der Stabilität des Programms, haben Spezifikations- und Dokumentationsfunktion. Programme sind aber auch völlig ohne Typangaben ausführbar.[1]

Das polymorphe Typsystem von TL-2 verwendet strukturelle Subtypisierung auf Klassenschnittstellen mit kontravarianter Typisierung von Methodenargumenten. Klassen und auch einzelne Methoden können über beschränkte Typparameter verfügen, so daß u.a. generische Behälterklassen akkurat typisiert werden können [Cardelli 91].

Als Schranke kann auch rekursiv die Klasse selbst verwendet werden (*F-bounded subtype polymorphism*), wodurch ähnlich wie durch *matching* [Bruce 93] die Spezifikation binärer Operationen möglich wird.

Eine Klasse mit Typargumenten definiert einen Typoperator. Auch solche nicht-geschlossene Typen können als Schranke und als Typargument verwendet werden. Diese *Subtypisierung höherer Ordnung* wird im Tycoon-2-System von den generischen Klassenbibliotheken tatsächlich nutzbringend eingesetzt, auch wenn sie dem einfachen Anwendungsprogrammierer selten begegnen wird.

[1] Eine Ausnahme bilden externe Methoden, da hier die Schnittstellentypen für die Umwandlung der Argumente ins C-Format und zurück verwandt werden.

3.2. Benutzerinteraktion

Die Programmentwicklung findet im textuellen Dialog mit dem System statt. Nach einer Eingabeaufforderung tippt der Programmierer sein Anliegen. Dies wird syntaktisch und auf Typkorrektheit überprüft und ggf. zurückgewiesen. Ist die Anfrage wohlgeformt, wird sie ausgeführt und das System gibt (widerum textuell) seine Antwort aus.

Wie in den meisten realen Lisp- und Smalltalk-Systemen [Steele 90], [Goldberg, Robson 83] ist der Dialog keine reine *read-eval-print-loop*, da neben Wertausdrücken der Programmiersprache auch Anweisungen an die Programmierumgebung akzeptiert werden. Folgende Eingaben sind möglich:

Ausdrücke der Sprache Tycoon-2 werden direkt ausgewertet und eine textuelle Repräsentation des Ergebnis-Objekts wird ausgegeben.

Klassendefinitionen ersetzen die bisher im System unter diesem Namen bekannte Klasse oder machen eine neue Klasse bekannt. Existierende Exemplare der Klasse verwenden sofort die neue Definition.

Definition globaler Variablen erzeugt Einträge im Verzeichnis der globalen Variablen (*pool*). Auf diese Variablen kann von beliebiger Stelle von definierten Klassen und vom Benutzerdialog aus zugegriffen werden.

DO-Kommandos erlauben es, Optionen der Programmierumgebung zu setzen, den aktuellen Zustand zu inspizieren und zu ändern. Mittels `DO load` *Dateiname* kann die Eingabe aus einer Datei umgeleitet werden. Dieser Mechanismus ermöglicht es, Klassendefinitionen in Dateien entsprechenden Namens auszulagern. Skriptdateien werden benutzt, um zusammengehörige Klassendefinitionen einzulesen und ggf. zu initialisieren.

3.2.1. Typüberprüfung

Mittels `DO check` wird eine Überprüfung aller Klassen eingeleitet, deren Typsicherheit aufgrund von Änderungen seit der letzten Überprüfung nicht mehr garantiert werden kann. Da die Ausführung wie in Abschnitt 3.1.9 (Typen) gesagt auch völlig ohne Kenntnis der Typebene möglich ist, können Klassen übersetzt und ausgetestet werden, bevor man die Typüberprüfung startet. Dies senkt den Zeitaufwand des Test-Zyklus und erlaubt die schnelle Erstellung von Anwendungsprototypen (*rapid prototyping*).

Die Typüberprüfung wurde zwecks schnellen Bootstraps in dieser Arbeit nicht genauer untersucht und nicht implementiert, da sie zur Ausführung von Anwendungen nicht notwendig ist. Sie bildet das Thema einer eigenen Studienarbeit [Ernst 97].

3.2.2. Klassendefinitionen

Syntaktisch fehlerhafte Klassendefinitionen werden vom System sofort zurückgewiesen. Ist diese Hürde genommen, wird die Klassen-Präzedenz-Liste der geänderten Klasse und aller ihrer Nachfahren neu berechnet. Entsteht dabei ein inkonsistenter Vererbungsgraph (s. Abschnitt 3.1.1, die Klassen-Präzedenz-Liste), bricht der Vorgang ab. Im untersuchten System kann es dadurch zu inkonsistenten Zuständen kommen. Es wurde darauf geachtet, dieses Problem im neu erstellten System durch Umsetzung einer transaktionalen Semantik zu vermeiden.

Bei Klassen mit vielen Nachfahren (z.B. `Object`, der Wurzel der Klassenhierarchie) kann die Berechnung aller Präzedenzlisten einige Zeit dauern, weshalb die Umgebung den Programmierer über den Fortgang der Aktion auf dem Laufenden hält.

Durch die Vererbung wirken Änderungen an einer Klasse auf alle ihre Nachfahren. Dies betrifft die definierten Exemplarvariablen und die implementierten Methoden. Während das neu implementierte Verhalten unmittelbar wirksam wird, ist die Änderung von Exemplarvariablen problematisch. Die automatisch erzeugten Methoden für den Variablenzugriff referenzieren die neu berechneten Positionen, während sich die Anordnung in den existierenden Exemplaren der Klassen nicht verändert hat.

Auch wenn dieses Verhalten sogar die Systemsicherheit gefährden kann, wenn auf Speicherstellen außerhalb des für ein Objekt reservierten Bereichs zugegriffen wird, wurde es vorerst übernommen. Eine Umwandlung der Klassenextension ähnlich wie in Smalltalk [Goldberg, Robson 83] ist jedoch geplant (s. Abschnitt 3.4, Erweiterungen).

3.2.3. DO-Kommandos

DO-Kommandos wirken anders als Klassendefinitionen, Ausdrücke und Variablendefinitionen auf die Programmierumgebung selbst. Die Trennung wird notwendig, da die Programmierumgebung im untersuchten System nicht selbst als Objekt der Sprache vorliegt, also auch nicht mit Sprachmitteln manipuliert werden kann.

Im neu erstellten System soll das anders sein. Optionen etwa der Codeerzeugung und der Menge der im Benutzerdialog auszugebenden Information können direkt durch Nachrichten an die jeweiligen Systemkomponenten gesetzt werden. Durch die reflektive Natur des neuen Systems wäre eine Beschränkung auf Ausdrücke vollkommen ausreichend, auch Variablen- und Klassendefinitionen sind wie in Smalltalk über Nachrichten möglich. Der Benutzer muß dann nur eine einzige Sprache lernen, um mit dem System kommunizieren zu können.

Eine rein textuelle Benutzerführung verträgt sich aber ohnehin schlecht mit dem heutigen Wissen über Mensch-Maschine-Kommunikation. Es ist zu erwarten, daß das neu erstellte System bald über eine graphische Interaktionskomponente verfügt, deren Implementation dann die von den Systemkomponenten gebotene Nachrichtenschnittstelle verwendet. Eine

genauere Untersuchung und Implementation der DO-Kommandos ist daher auch aus Zeitgründen nicht ratsam.

Als einziges DO-Kommando ist `DO load` *Dateiname* unbedingt erforderlich, um Skriptdateien des alten Systems verwenden zu können. `DO load` fügt den Inhalt der angegebenen Datei an der aktuellen Position in die Eingabe ein. Dadurch können Klassendefinitionen in Dateien entsprechenden Namens ausgelagert und durch Eingabe eines kurzen Befehls neu eingelesen werden. Die eingefügte Datei kann weitere `DO load`-Anweisungen enthalten, so daß zusammengehörige Klassendefinitionen durch Laden einer übergeordneten Skriptdatei gemeinsam eingelesen und initialisiert werden können. Auf niedriger Abstraktionsstufe erlaubt dies die Unterteilung in Pakete.

3.3. Die virtuelle Zielmaschine

Die virtuelle Zielmaschine bietet eine sehr breite Schnittstelle zum in Tycoon-2 geschriebenen Teil des Systems. Dadurch entsteht eine relativ enge Kopplung an die Implementation, andererseits vermeidet man ein die Performanz beeinträchtigendes Nadelöhr (*performance-bottleneck*).

Die Kommunikation mit der Zielmaschine findet statt über

gemeinsame Datenstrukturen: Der Maschine wird ein im Objektspeicher liegendes Wurzel-Objekt bekannt gemacht, von dem aus alle von der Maschine benötigten Strukturen erreichbar sind. Das Speicherformat der benötigten Strukturen wird von der Maschine vorgegeben. Auf die gemeinsamen Datenstrukturen greift die Maschine nur lesend zu, alle ändernden Zugriffe werden in TL-2 implementiert.

elementare Methoden: Als `builtin` markierte, elementare Methoden werden direkt von der virtuellen Maschine ausgeführt. Dadurch steht die volle Funktionalität von Nachrichten für die Kommunikation mit der Maschine zur Verfügung: Die Maschine wird über ein Ereignis informiert, empfängt zusätzliche Nutzdaten, kann eine Aktion ausführen und liefert ein Berechnungsergebnis zurück.

Die Maschine kann die Bearbeitung an eine bei der Definition der Methode angegebene TL-2-Implementation weiterreichen. So können für die Behandlung von Fehlern oder z.B. unerwarteten Argumentklassen die vollen TL-2-Sprachmittel eingesetzt werden.

feste Tabellen: Zur Übersetzungszeit der virtuellen Maschine wird automatisch eine Beschreibung der Zielmaschine generiert, mit der die Programmierumgebung konfiguriert wird.

Die gemeinsamen Datenstrukturen dienen dazu, die Maschine jederzeit über den aktuellen Zustand der Tycoon-2-Seite zu informieren. Dazu gehört die Tabelle aller im System definierten Klassen inklusive der von ihren Exemplaren verstandenen Methoden (die vollständige

Beschreibung findet sich in Abschnitt C.2 und C.3). Änderungen, z.B. die Redefinition einer Methode, werden direkt wirksam. Der Maschine muß nur mitgeteilt werden, daß eine Änderung stattgefunden hat, damit interne Caches u.ä. mitgeführt werden können.

Elementare Methoden dienen dem Informationsfluß in beide Richtungen. Sie werden benutzt, um eine Schnittstelle zum System (Anbindung externer Dienste durch *dynamic link libraries*) und zum persistenten Objektspeicher herzustellen (persistentes commit; Erzeugung, Manipulation und Untersuchung von Objekten verschiedenen Formats) und um essentielle Spracheigenschaften zu implementieren (Vergleich der Objektidentität; Klasse eines Objekts erfragen; Auslösung von Ausnahmen und Aufruf von Funktionsobjekten). Eine vollständige Liste findet sich in Anhang C.1.

Die statischen Informationen über die Maschine (verfügbare Bytecodes, elementare Methoden und vorgegebene Klassennummern) werden der Programmierumgebung mittels fester Tabellen mitgeteilt. Die Programmierumgebung wird bei der Erzeugung mit diesen Tabellen parametrisiert und braucht nicht mit späteren Änderungen zu rechnen. Dadurch kann sie diese Informationen aufbereiten (indizieren) und so im Betrieb schnelleren Zugriff gewähren.

Maschine und Programmierumgebung operieren auf einem gemeinsamen Datenmodell, so daß der Implementationsaufwand flexibel zwischen dem niedersprachlichen, in C implementierten Teil und dem hochsprachlichen Teil verteilt werden kann. Im Einzelfall können so Performanzbetrachtungen berücksichtigt werden.

3.3.1. Der Objektspeicher

Die Maschine definiert einige elementare Methoden, um Objekte verschiedenen Formats anzulegen. Das Subjekt dieser Methoden ist immer die Klasse, von der ein Exemplar erzeugt werden soll. Den Klassencode und die Speichergröße des zu erzeugenden Objekts erfährt die Maschine aus den Exemplarvariablen des Klassenobjekts.

Da der Objektspeicher über eine automatische Speicherbereinigung verfügt, ist keine explizite Freigabe nötig.

Der aktuelle Zustand des Objektspeichers inklusive aller laufenden Berechnungen läßt sich durch einfachen Aufruf einer elementaren Methode persistent sichern.

3.3.2. Die Klassentabelle

Die Maschine verwendet fortlaufende Nummern, um die im System vorhandenen Klassen zu identifizieren. Durch Aufruf einer elementaren Methode wird eine Position in der vom Wurzelobjekt erreichbaren Klassentabelle reserviert. An dieser Position erwartet die Maschine dann einen Verweis auf die identifizierte Klasse.

Exemplare der Klasse enthalten nur noch die Nummer ihrer Klasse, was gegenüber einem Objektverweis Speicherplatz spart. Außerdem kann innerhalb der Maschine direkt mit

kleinen, vordefinierten Konstanten für bestimmte Standardklassen gearbeitet werden. Ein Nachteil der Numerierung besteht darin, daß die Klassentabelle explizit verwaltet werden muß, anstatt diese Arbeit dem persistenten Objektspeicher mit seiner Speicherbereinigung zu überlassen.

3.3.3. Versenden von Nachrichten

Wenn einem Objekt eine Nachricht geschickt werden soll, wird zunächst die Nummer seiner Klasse festgestellt. In der über das Wurzelobjekt erreichbaren Klassentabelle wird die zu dieser Nummer gehörige Klasse nachgeschlagen. Wie in Abschnitt 3.1.1 (Die Klassen-Präzedenz-Liste) beschrieben, müssen nun der Präzedenzliste entsprechend die Methodenverzeichnisse der Vorfahrenklassen durchgegangen werden. Dazu enthält jedes Klassenobjekt eine fertig berechnete Liste von Methodenverzeichnissen, die als Hashtabellen eine Abbildung von Selektor auf Methode definieren.

Damit dieser recht aufwendige Vorgang nicht bei jeder Nachricht stattfinden muß, verwaltet die Maschine einen zentralen Methodencache. Die Tycoon-Seite ist dafür verantwortlich, die Maschine durch Aufruf einer elementaren Methode zu informieren, wenn sich die Methoden einer Klasse ändern, da dadurch der Inhalt des Cache ungültig wird.

3.3.4. Methoden

Jede Methode trägt ihren Selektor und die Anzahl erwarteter Argumente, um beim Aufruf korrekt identifiziert zu werden. Darüber hinaus kennt die Maschine mehrere Unterklassen von `Method`. Methoden zum Zugriff auf Exemplarvariablen speichern die Position der zugegriffenen Variable im Objekt; die Methoden für globale Variablen kennen die Speicherzelle, die den aktuellen Variablenwert enthält. Weiterhin existieren Methoden zum Zugriff auf C-Strukturen und zum Aufruf von C-Funktionen. Uns interessieren hier besonders die in TL-2 geschriebenen Methoden (`CompiledMethod`).

Laut Anhang C.2 erwartet die Maschine in einem Exemplar von `CompiledMethod` folgende Exemplarvariablen:

`tyc_Symbol pSelector;` Name der Methode

`Bool fIsPrivate;` Ist diese Methode privat? Wenn ja, kann sie nur von Methoden des selben Objekts und seines Klassenobjekts aus zugegriffen werden.

`int nArgs;` Anzahl Wertargumente der Methode

`tyc_Class * pClass;` Klasse, in der die Methode definiert wurde. Wird für *super send* benötigt.

`Byte * pbCode;` Bytecode für diese Methode. Wird im nächsten Abschnitt näher erläutert.

`tyc_Array * pLiterals;` Literale, auf die der Bytecode Bezug nimmt.

`tyc_SelectorID idSelector;` numerischer Identifikator für den Selektor und die Argumentzahl dieser Methode. Die Selektor-Nummer ist ein Index in der im Wurzel-Objekt verzeichneten Selektor-Tabelle. Die Selektor-Nummern werden beim Aufruf verwendet, was die explizite Angabe eines Selektor-Objekts im Literalvektor erspart, die Codierung im Message-Cache erleichtert und den Vergleich beschleunigt.

`int cwStackPeak;` Maximaler Stackbedarf dieser Methode. Die Maschine braucht nur beim Aufruf einer Methode unter Verwendung dieser Exemplarvariable auf Stacküberlauf zu prüfen, nicht bei jedem einzelnen *push*.

`tyc_CatchFrame * asHandlerTable;` Eine Tabelle, die die in der Methode definierten Ausnahmebehandlungen (`try`) beschreibt. Jeder `CatchFrame` enthält den Bytecodebereich, für den die Ausnahmebehandlung gilt, die Bytecodeposition, an die im Falle einer Ausnahme zu verzweigen ist, und die Stackhöhe beim Eingang in die Ausnahmebehandlung. Die Ausnahmebehandlungen sind nach Bytecodepositionen sortiert. Bei verschachtelten Ausnahmebehandlungen wird die weiter innen liegende zuerst angegeben. Dadurch kann die Maschine die Ausnahmebehandlungen von vorn nach hinten durchgehen und die erste passende anspringen.

Der Übersetzer muß Werte für alle diese Exemplarvariablen aus dem Quelltext extrahieren.

3.3.5. Bytecode

Die virtuelle Maschine versteht einen stapelorientierten Bytecode mit Sprüngen und Befehlen zum Senden von Nachrichten. Der zu sendende Methodenselektor und die Anzahl übergebener Argumente werden wie oben beschrieben in einer Selektor-Nummer kodiert. Für kleine, feste Argumentzahlen existieren spezielle Bytecodes, damit die Maschine nicht bei jedem Nachrichten-Befehl die Tabelle konsultieren muß.

Die Sprungbefehle gibt es jeweils in zwei Formen für kurze und lange Sprünge (8 oder 16 Bit Distanz zum Sprungziel). Da der Großteil der Methoden kurz ist (im Durchschnitt nur 35 Bytes), lohnt sich die kompaktere und schneller ausführbare Kodierung kurzer Sprünge.

Literale werden im allgemeinen im Literalvektor des Methodenobjekts gespeichert und per Index zugegriffen. Um Codedichte und Ausführungsgeschwindigkeit zu erhöhen, gibt es spezielle Bytecodes für häufig vorkommende Literale und kleine Ganzzahlen.

Funktionsabschlüsse (s. Abschnitt 3.1.6, Funktionen höherer Ordnung) mit flacher Repräsentation der freien Variablen werden durch einen Befehl zur Erzeugung eines Funktionsobjekts und einen Befehl zum Zugriff auf freie Variablen unterstützt. Bei der Erzeugung wird die

Position des `CompiledFun`-Objekts im Literalvektor und die Anzahl der freien Variablen angegeben.

Wird auf eine änderbare Variable aus Funktionsabschlüssen heraus zugegriffen, muß ihr aktueller Wert in einem Objekt außerhalb des Stacks gespeichert werden, da die Lebensdauer der Variable nicht mehr der Stackdisziplin gehorcht. Zur Speicherung werden Exemplare der Klasse `Cell` verwendet, für deren Erzeugung und Zugriff ebenfalls Bytecodes existieren.

Die bei der Entwicklung verwendete Spezifikation der TVM-2 findet sich in Anhang C.3.

3.4. Erweiterungen

Beim Entwurf ist auch auf zu erwartende zukünftige Anforderungen zu achten, damit das System in den fraglichen Punkten erweiterbar bleibt.

In Abschnitt 3.2.3 (DO-Kommandos) wurde bereits die Entwicklung einer graphischen Oberfläche angesprochen, durch deren Einsatz die Bedeutung des textuellen Dialogs sinkt. Die Schnittstelle zur Programmierumgebung soll als Nachrichtenschnittstelle implementiert werden, um so gegenüber verschiedenen Interaktionskomponenten offen zu bleiben und um für deren Implementation die volle Ausdrucksmächtigkeit von TL-2 einsetzen zu können.

Da auch mehrere gleichzeitige Dialoge mit dem selben System denkbar sind, ist auf Reentranz der beteiligten Komponenten zu achten.

Die Schnittstelle soll breit genug sein, um die nachträgliche Implementation weiterer Programmierwerkzeuge zu gestatten. Vorgesehen ist u.a. ein Werkzeug zum Abgleich der Quelltexte zwischen Objektspeicher und Platte sowie Werkzeuge, mit deren Hilfe dem Benutzer graphisch aufbereitete Versionen des Programms präsentiert werden können.

Die Sprache TL-2 befindet sich noch in der Entwicklung. Insbesondere die Oberflächen-Syntax wird sich noch ändern. Um flexibel auf Änderungen reagieren zu können, muß das Wissen über die konkrete Syntax lokalisiert und die Erstellung des Parsers automatisiert werden. Die Verwendung eines Parsergenerators und abstrakter Syntaxbäume als stabile interne Repräsentation bieten sich an.

TL-2 enthält außer Klassen noch keine Konzepte zur Strukturierung großer Anwendungen. Bei einer zukünftigen Unterteilung in mehrere Namens- und Zugriffsräume werden zumindest verschiedene Variablenverzeichnisse statt eines globalen *pools* nötig werden.

Wie in Abschnitt 3.2.2 (Klassendefinitionen) erläutert, ist nach Änderung der in einer Klasse definierten Exemplarvariablen eine Umwandlung der im Objektspeicher vorhandenen Exemplare dieser Klasse vorgesehen. Hierzu muß Information über die alte und neue Anordnung der Exemplarvariablen verfügbar sein.

4. Architektur

Dieses Kapitel gibt einen Überblick über die Architektur der entwickelten TL-2-Programmierumgebung. Eine genauere Beschreibung ausgewählter Komponenten folgt in Kapitel 5.

Die Komponenten des Systems sind aus den in Abschnitt 3.2.3 (DO-Kommandos) angegebenen Gründen als Objekte mit einer definierten Nachrichtenschnittstelle implementiert. Der Zugriff auf andere von einer Komponente benötigte Komponenten erfolgt über explizite Verweise im Objekt. Gegenüber einer Lösung mittels globaler Variablen oder *Singletons* [Gamma et al. 95] bietet dies den Vorteil klar ersichtlicher Abhängigkeiten und erlaubt es überhaupt erst, während des Bootstraps eine zweite Ausprägung der Programmierumgebung im Objektspeicher zu haben (s. Abschnitt 2.4, Aus dem Bootstrap resultierende Anforderungen). Der Quelltext der Schnittstellen findet sich in Anhang C.4.

Die Dialogebene regelt die Kommunikation mit dem Benutzer (s. Abschnitt 3.2, Benutzerinteraktion). Die Anfrage des Benutzers wird syntaktisch analysiert und einer der vier möglichen Eingabearten zugeordnet. Zur weiteren Verarbeitung wird sie dann an die zuständige Systemkomponente weitergegeben.

Die Komponenten für die syntaktische Analyse (Scanner, Parser) werden automatisch generiert. Als Spezifikation dienen reguläre Ausdrücke bzw. grammatikalische Regeln mit semantischen Aktionen in Form von *higher-order functions*. Man beachte, daß Scanner und Parser als Objekte im laufenden System erzeugt werden, ohne den von lex [Lesk, Schmidt 75] und yacc [Johnson 75] bekannten Umweg über Programmquelltext, und so eine dynamische und voll typisierte Schnittstelle bieten können. Darin ähneln sie dem in [Schröder, Matthes 92] beschriebenen *Tycoon Compiler Toolkit*.

Im Falle von Ausdrücken und Klassendefinitionen wird der bei der syntaktischen Analyse entstandene abstrakte Syntaxbaum zunächst vom Übersetzer bearbeitet (s. Abb. 4.2). Dieser fügt von der virtuellen Maschine benötigte Information (insbesondere den Bytecode) in den Syntaxbaum ein. Man beachte, daß das letztendlich von der Maschine bearbeitete Objekt tatsächlich dasselbe wie das während der Syntaxanalyse entstandene ist: Die Objektidentität wird im Verlauf der Berechnung beibehalten. Dadurch steht während der Ausführung und für die Testhilfe alle dem System bekannte Information (Quelltextposition, Typen und Methodenkörper) unmittelbar zur Verfügung.

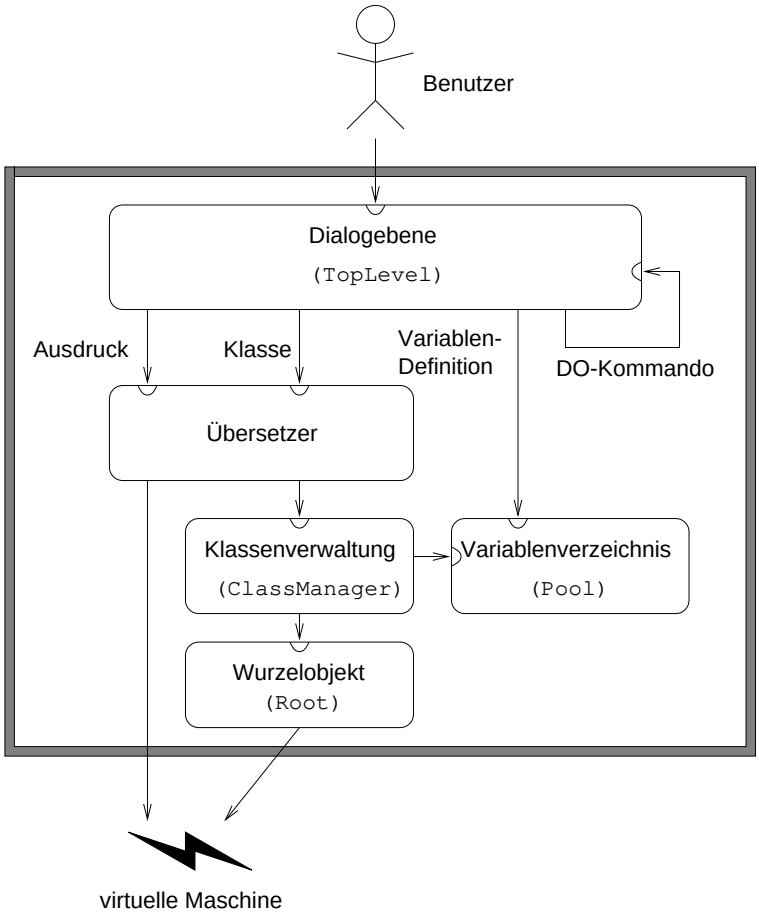


Abbildung 4.1.: System-Überblicks-Diagramm der TL-2-Programmierumgebung

Konsequenterweise findet die Berechnung dieser Attribute innerhalb der Objekte des Syntaxbaums statt: Die Implementation der semantischen Analyse und der Codeerzeugung ist größtenteils auf die Methoden der Ausdrucksklassen verteilt.

Ein Optimierungslauf ist nicht nötig, da dieser ähnlich wie in Self [Hölzle 94] für die Programmierumgebung transparent innerhalb der virtuellen Maschine stattfinden soll.

Der abstrakte Syntaxbaum für Klassendefinitionen hat ein Exemplar der Klasse `ClassBuilder` als Wurzel. Die Übersetzung wird angestoßen, indem diesem Objekt eine `build`-Nachricht geschickt wird. Die Implementation dieser Methode findet sich in Abbildung 4.3.

Der `ClassBuilder` ergänzt zunächst optionale Teile der Klassendefinition. Sodann werden die Sichtbarkeitsbereiche für alle in der Klassendefinition selbst benutzten Typen aufgelöst. Dazu gehören die Typparameter der Klasse, der `Self`-Typ, der Metaklassen-Typ und die Typen aller definierten Exemplarvariablen.

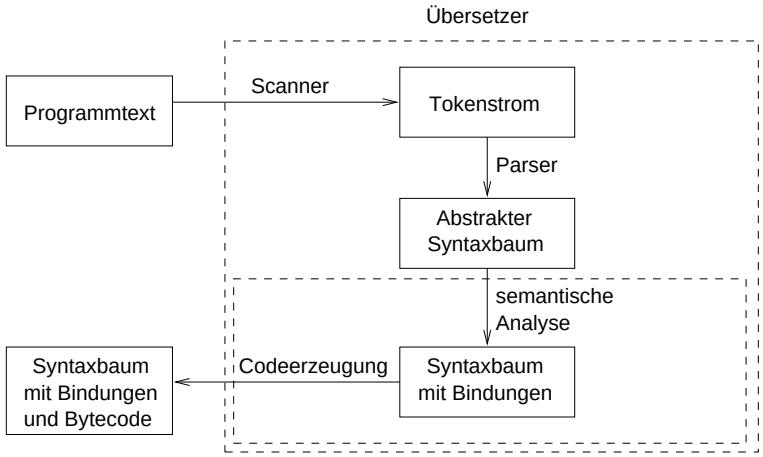


Abbildung 4.2.: Struktur des Übersetzers

```
build(compiler :Compiler, classManager :ClassManager)
{
  (* add omitted Self type bound and metaclass declaration *)
  normalize,
  (* establish scoping regions for types in class definition.
     Compute an environment including the class domain, the Self type,
     and the receiver. *)
  let scoper = scopeClass,
  (* check for doubly defined selectors in slots and methods *)
  checkDuplicateSelectors,
  (* for all methods... *)
  methodDictionary.do(fun(m :Method) {
    (* establish scoping regions *)
    m.scopeMethod(scoper),
    (* and generate code *)
    m.compileMethod(compiler)
  }),
  (* pass complete class description to the class manager *)
  classManager.set(self)
}
```

Abbildung 4.3.: Die Implementation von ClassBuilder::build (vereinfacht)

Nach einer Überprüfung auf eindeutige Selektoren wird für alle Methoden die Semantische Analyse und die Codeerzeugung durchgeführt, indem den Methoden-Objekten die entsprechenden Nachrichten gesendet werden.

Die vollständig attributierte Klassenbeschreibung wird schließlich der Klassenverwaltung übergeben. Die Umsetzung dieser Klassenbeschreibung bildet aus Stabilitätsgründen die einzige Möglichkeit, auf den Zustand der Klassenverwaltung Einfluß zu nehmen. Ist es der Klassenverwaltung nicht möglich, den beschriebenen Zustand zu erreichen, setzt sie die Klasse einfach in den vorher geretteten Zustand zurück. Der Zustandsübergang wird also

atomar und konsistenzerhaltend durchgeführt. Durch die Persistenz des Objektspeichers ist die Änderung dauerhaft, und wird die Klassenverwaltung als Monitor implementiert, erfüllt sie auch das Kriterium der Isoliertheit. Insgesamt bietet die Klassenverwaltung somit transaktionale Semantik.

Die Klassenverwaltung ist für die Beziehungen zwischen Klassen verantwortlich, in der Hauptsache für die Konsistenz aller Klassen-Präzedenz-Listen. Neben der Vererbung wird auch die Metaklassen-Beziehung hier behandelt: Soll eine Klassenbeschreibung mit bis dato unbekanntem Namen umgesetzt werden, wird das neue Klassenobjekt als Exemplar seiner Metaklasse erzeugt. Ändert sich später die Metaklassendeklaration, muß ein neues Klassenobjekt angelegt werden. Eine Klasse, die als Metaklasse verwendet wird, muß von `Class` erben. Darauf muß auch bei späteren Änderungen der Metaklasse geachtet werden.

Neu erzeugte Klassenobjekte erhalten über das Wurzelobjekt eine Nummer und werden in die dortige Klassentabelle eingetragen (s. Abschnitt 3.3.2, Die Klassentabelle). Damit sind sie der virtuellen Maschine bekannt und können eingesetzt werden.

5. Beschreibung ausgewählter Systemkomponenten

Die Kernfunktionalität der Programmierumgebung wird von dem Übersetzer und der Klassenverwaltung bereitgestellt. Eine Umstrukturierung der Dialogebene und des Variablenverzeichnisses ist abzusehen, wird die Kernkomponenten aber kaum beeinflussen.

Nach [Aho et al. 88] besteht ein optimierender Übersetzer konzeptuell aus den in Abb.5.1 gezeigten Phasen.

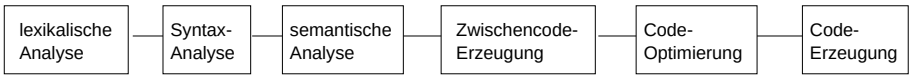


Abbildung 5.1.: Phasen eines optimierenden Compilers

In den folgenden Abschnitten wird die Umsetzung dieser Phasen im implementierten System beschrieben. Im letzten Abschnitt wenden wir uns der Klassenverwaltung zu.

5.1. Lexikalische und Syntaktische Analyse

In der aktuellen Implementation des Systems findet die lexikalische und syntaktische Analyse bereits auf der Dialogebene statt, da dort ohnehin eine Unterscheidung nach verscheidenen Eingabearten stattfinden muß. Die Analyse der TL-2-Texte gehört jedoch konzeptuell zum Übersetzer.

Lexikalische und syntaktische Analyse unterscheiden sich durch die Komplexität der erkannten Struktur. Die in der ersten Phase untersuchte Struktur läßt sich durch einen endlichen Automaten beschreiben, während für die zweite Phase eine mächtigere, in der Analyse aufwendigere kontextfreie Grammatik verwendet wird.

In der lexikalischen Analyse wird der Programmtext vom *Scanner* linear durchgegangen und in eine Folge von *Tokens* umgewandelt, die jeweils einem Terminalsymbol der Grammatik zugeordnet sind und noch weitere Information tragen können. Das gemeinsame Vokabular an Terminalsymbolen wird von einer separaten Komponente verwaltet.

So wird zum Beispiel die Eingabe

```
let i :Int = 10, 5*i;
```

interpretiert als

<let> <ident "i"*> <colon> <ident* "Int"*> <equals> <int* 10*> <comma>*
<int 5*> <star> <ident* "i"*> <eof>*

Anhand der Terminalsymbole entscheidet der *Parser* in der syntaktischen Analysephase, aus welchen Produktionen der Grammatik der eingegebene Satz hergeleitet wurde. Als Eingabe des eingesetzten LALR(1)-Parsergenerators dient eine *syntaxgesteuerte, S-attributierte Definition* [Aho et al. 88]: Jeder Produktion der Grammatik ist eine semantische Aktion zugeordnet, die aus den synthetisierten Attributen der Terminale und Nonterminale ihrer rechten Seite das Attribut ihrer linken Seite ableitet. In unserem Fall handelt es sich bei den synthetisierten Attributen meist um abstrakte Syntaxbäume. Unter Verwendung der Klassen aus Abb. 3.2 entsteht aus dem obigen Beispiel der folgende konkrete Syntaxbaum:

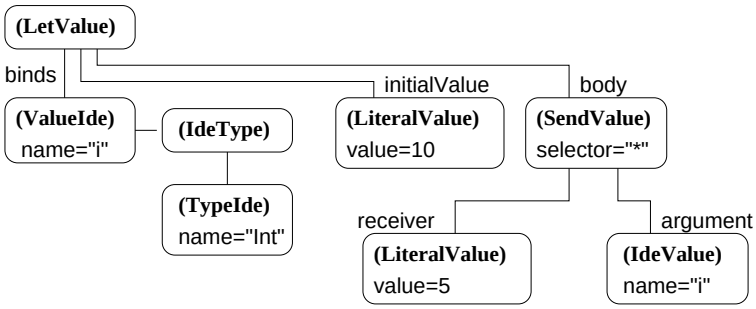


Abbildung 5.2.: Ein konkreter Syntaxbaum. Dynamisches Objektmodell zu Abb. 3.2

Zur Spezifikation der lexikalischen Struktur dienen reguläre Ausdrücke mit semantischen Aktionen, wie beim Parser in Form von Funktionen höherer Ordnung. Aus den regulären Ausdrücken wird, wie in [Hopcroft, Ullman 79] beschrieben, automatisch ein deterministischer endlicher Automat erzeugt, der wiederholt die längste als Token interpretierbare Zeichenfolge von der Eingabe liest und die zum verwendeten regulären Ausdruck gehörige semantische Aktion ausführt. In der semantischen Aktion kann der erkannte Quelltext weiter interpretiert werden, und alle für spätere Phasen relevante Information wird in Form eines Tokens zurückgegeben.

Sowohl Parser- als auch Scannergenerator sind Objekte des Tycoon-2-Systems. Die Spezifikation und Generierung von Scanner und Parser findet mit normalen TL-2-Sprachmitteln, d.h. insbesondere voll typisiert statt. Da auch die resultierenden Scanner und Parser Objekte sind, stellt es im Gegensatz zu auf Quelltextebene operierenden Werkzeugen wie lex und yacc kein Problem dar, mehrere Scanner gleichzeitig in einer Anwendung zu verwenden und sogar dynamisch nach Erfordernissen der Anwendung zu generieren.

Dies ist auch wichtig für die geforderte Reentranz dieser Systemkomponenten (s. Abschnitt 3.4). Im ISO/OSI-Architekturmodell für offene Rechnernetze [Kerner 89] wird das Problem der Reentranz durch eine Trennung in *Instanz* und *Vorkommnis* einer Komponente gelöst: Die Instanz ist eine „ablaufbereite Prozeßschablone“, von der bei Bedarf eine Vorkommnis genannte Kopie gestartet werden kann. Dies kann in TL-2 direkt umgesetzt werden.

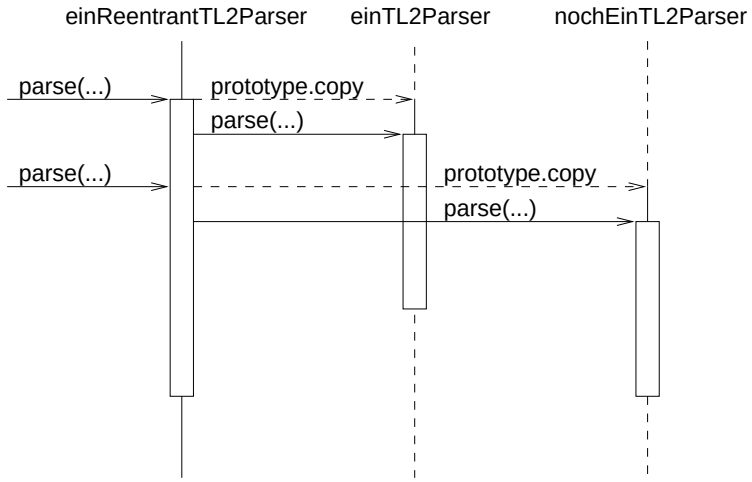


Abbildung 5.3.: Reentranz der Parser-Komponente

Wenn ein Objekt der Klasse `ReentrantTL2Parser` eine `parse`-Nachricht erhält, delegiert es sie an ein unter Verwendung des in [Gamma et al. 95] beschriebenen Prototyp-Musters neu erzeugtes, nicht-reentrantes `TL2Parser`-Objekt. Auf diese Weise können mehrere Vorkommnisse des Parsers gleichzeitig aktiv sein, ohne sich zu stören. Derselbe Mechanismus wird für den Scanner verwendet.

5.2. Semantische Analyse

Wesentliche Aufgabe der semantischen Analyse ist die Berechnung von Sichtbarkeitsbereichen (s. Abschnitt 3.1.4). Im abstrakten Syntaxbaum (AST) werden referenzierende Auftreten von Bezeichnern durch einen Objektverweis auf die Definition des Bezeichners ergänzt. Nicht lokal gebundene Bezeichner-Referenzen werden erkannt und in `Send`-Knoten umgewandelt, da in der TL-2-Syntax argumentlose Nachrichten an das Subjekt die gleiche Form wie Bezeichern-Referenzen haben und vom Parser zunächst als solche in den Syntaxbaum aufgenommen werden.

Die Implementation ist auf die AST-Klassen verteilt und folgt dem *Interpreter*-Muster von [Gamma et al. 95]. Der *Kontext*, in diesem Fall die aktuellen Bindungen, wird von der Wurzel

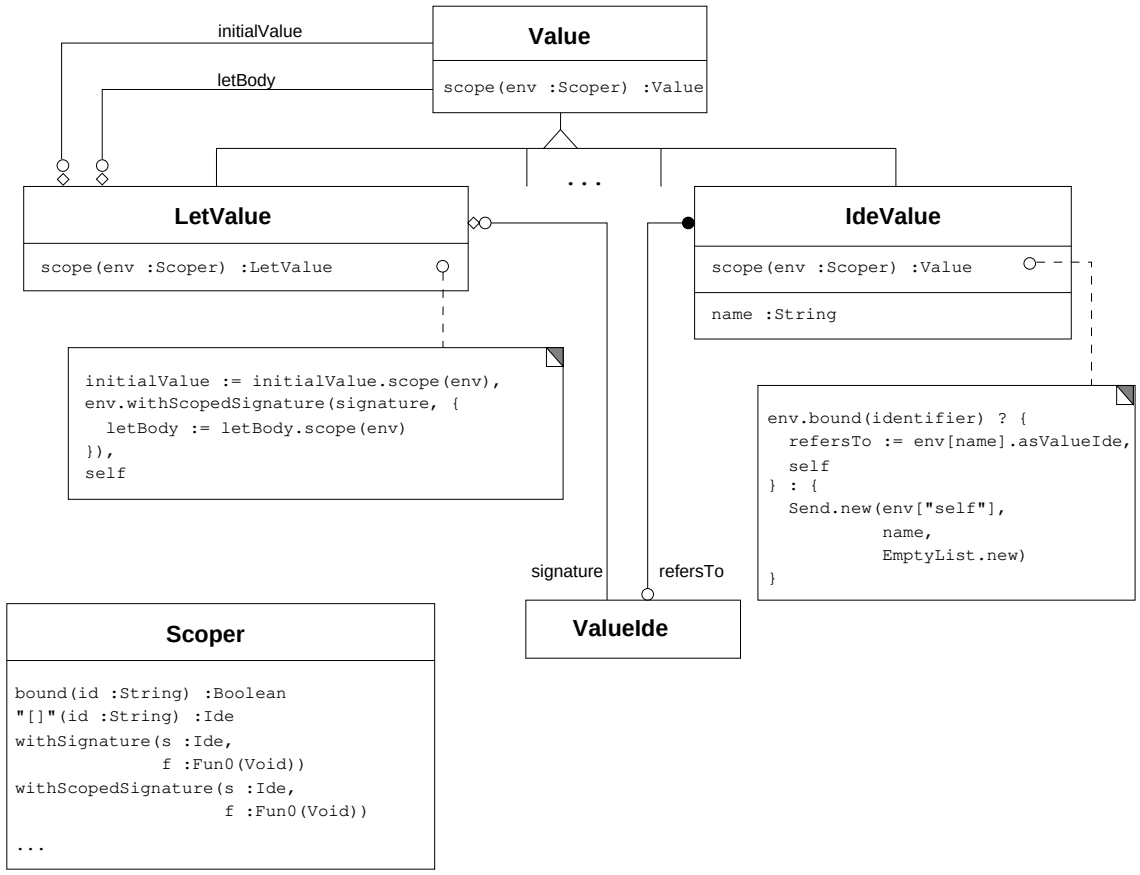


Abbildung 5.4.: Klassendiagramm für die semantische Analyse (Ausschnitt)

zu den Blättern durchgereicht, und der umgewandelte Syntaxbaum wird von den Blättern zurück zur Wurzel berechnet.

Die Umwandlung geschieht destruktiv, d.h. nicht-terminale Knoten des Syntaxbaums ersetzen ihre Unter-Bäume durch die umgewandelte Form, statt funktional eine modifizierte Kopie ihrer selbst zu erzeugen.

Die Klasse `Scoper`, die die Rolle des Kontextes einnimmt, erhält ihren Namen von der Hauptaufgabe dieser Phase. Neben der Abfrage der aktuellen Bindungen enthält sie auch Methoden, um korrekt geschachtelte Sichtbarkeitsbereiche zu erstellen. Dies geschieht mit Hilfe von Funktionen höherer Ordnung: Der Methode `withSignature` wird eine Bindung und eine parameterlose Funktion übergeben, während deren Ausführung die Bindung gültig sein soll. Die Bindung wird durchgeführt, die Funktion ausgewertet, und die Bindung wird wieder entfernt. Der Klient braucht nicht mehr selbst auf korrekte Paarung der Aufrufe zu achten. Dieses Muster wird u.a. auch in Scheme [Clinger, Rees 91] verwendet, um die Aufrufe zum Öffnen und Schliessen einer Datei zu paaren, so daß sich die verschachtelte

Programmstruktur der Problemstruktur anpaßt.

Abb. 5.4 zeigt die Implementation am Beispiel der Klassen `LetValue` und `IdeValue`.

Der `Scoper` protokolliert jede aufgelöste Referenz, so daß bei der Abarbeitung von Funktions-Ausdrücken bekannt ist, welche statisch weiter aussen definierten Wertbezeichner bei der Ausführung der Funktion benötigt werden. Bei der Berechnung der Sichtbarkeitsbereiche werden also auch die freien Bezeichner berechnet [Tennent 81]. Diese Information wird später zur Codeerzeugung für Funktionsabschlüsse und zum Zugriff auf die globalen Variablen eines Funktionsabschlusses benutzt.

5.3. Zwischencodeerzeugung

Traditionelle Aufgabe der Zwischencodeerzeugung ist es, den abstrakten Syntaxbaum in eine lineare, maschinencode-ähnliche Form zu bringen, die jedoch möglichst wenig Abhängigkeiten von der Zielarchitektur durchscheinen läßt. Die Isolation von der Zielarchitektur ist softwaretechnisch als Kapselung sinnvoll und erleichtert die Portierung auf eine neue Maschine. In unserem Fall fiel es sehr leicht, den Übersetzer an Entwicklungen (z.B. die Einführung der *special sends*) auf der Seite der virtuellen Maschine anzupassen.

Der Zwischencode wird durch Nachrichten an ein Exemplar der Klasse `MethodManager` ausgegeben, `MethodManager` ist also ein *Erbauer* [Gamma et al. 95] für Zwischencode. Das Format des Zwischencodes wird durch die verfügbaren Methoden der Klasse `MethodManager` definiert. Neben symbolischen Sprungzielen verwaltet der `MethodManager` auch die Zugriffspfade auf die aktuell gebundenen Variablen, wodurch Klienten völlig vom fehlerträchtigen Umgang mit numerischen Indices entlastet werden.

Der `Methodmanager` wird ähnlich wie der `Scoper` rekursiv durch den abstrakten Syntaxbaum nach unten weitergereicht. Er nimmt damit die Rolle des *Kontextes* im *Interpreter*-Muster an.

Der Übersetzer unterscheidet vier verschiedene Arten von Fortsetzungen (continuations) und erzeugt darauf angepaßten Code. In der Klasse `Value` werden vier verschiedene Methoden zur Codeerzeugung definiert, die als Argumente den `MethodManager` und je nach Fortsetzung weitere Informationen erwarten. Diese Methoden sind:

genCode Ausdruck auswerten und Ergebnis auf dem Stack ablegen.

genCodeIgnore Ausdruck auswerten, um Seiteneffekte zu erzielen, und Ergebnis verwerfen.

genCodeReturn Ausdruck auswerten und Ergebnis zum Aufrufer der Funktion oder Methode zurückgeben.

genCodeBranch(dest :Label, branchIf :Bool) Ausdruck auswerten, und Sprung nach ⟨dest⟩ durchführen, wenn Ergebnis ⟨branchIf⟩, andernfalls folgende Befehle ausführen.

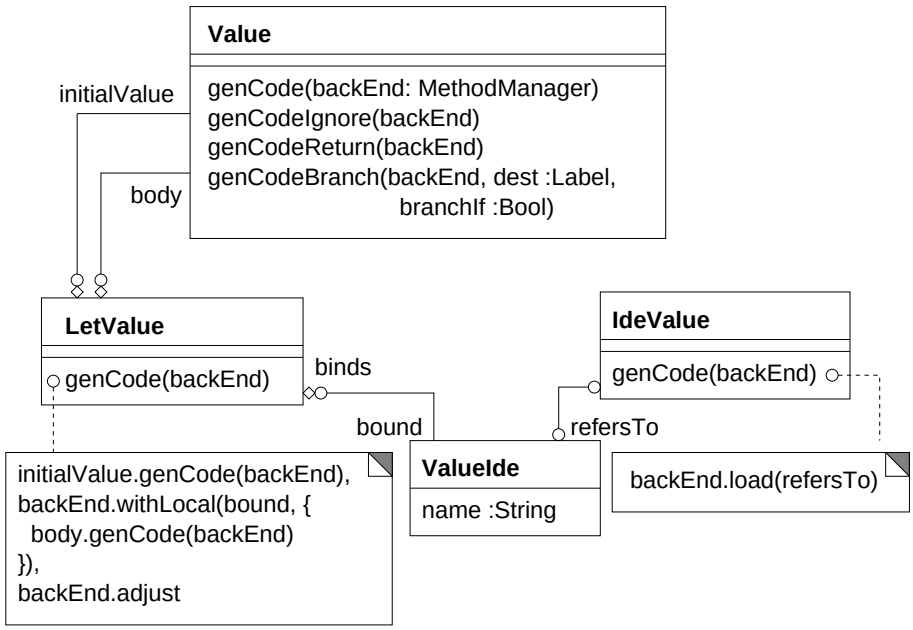


Abbildung 5.5.: Zwischencode-Erzeugung

`genCodeIgnore`, `genCodeReturn` und `genCodeBranch` sind in der Klasse `Value` defaultmäßig mit Hilfe von `genCode` implementiert. In Nachfahrenklassen werden sie redefiniert, um effizienteren Code zu erzeugen:

Für eine `ignore`-Fortsetzung sind Konstanten oder Funktionsausdrücke nutzlos, es wird also eine Warnung ausgegeben und kein Code erzeugt.

Für eine `return`-Fortsetzung brauchen lokale Variablen nicht explizit vom Stapel entfernt zu werden, da nach dem `return` ohnehin das gesamte Aktivierungssegment entfernt wird.

Die Auszeichnung von `branch`-Fortsetzungen dient dazu, Sprungmarken zu propagieren (*label assigment*, [Wulf et al. 73]) und boolesche Ausdrücke kompakter zu übersetzen (*boolean short-circuiting*).

Nachfahrenklassen müssen darauf achten, die richtige Fortsetzung an ihre Unterausdrücke weiterzugeben, d.h. die korrekte Methode zur Codeerzeugung aufzurufen.

5.4. Codeoptimierung

Die Codeerzeugung versucht, ohne aufwendige Optimierungen brauchbaren Code zu erzeugen. Es wird kein Versuch unternommen, über Methodengrenzen hinweg zu optimieren.

Einzelne Tycoon-2-Methoden sind durch den hohen Grad von Wiederverwendung typischerweise sehr kurz. Aus dem traditionellen Compilerbau bekannte Optimierungen lohnen sich

daher erst, wenn mehrere Methoden gemeinsam optimiert werden. Durch diese Art der Optimierung entstehen jedoch Abhängigkeiten zwischen Klassen. Nach kleinen Änderungen an einer Basisklasse müßten evtl. große Teile des Systems neu übersetzt werden, was der Reaktivität abträglich ist.

Es ist geplant, auf einer tieferen Systemebene weitergehende Optimierungen (Typabschätzung, customization, siehe [Hölzle 94]) bei der Übersetzung in plattformspezifischen Maschinencode vorzunehmen. Der in Tycoon-2 erzeugte Bytecode aber ist vollständig generisch und hängt allein vom Quelltext der jeweiligen Methode ab.

5.5. Codeerzeugung

Zur Erzeugung des eigentlichen Bytecodes benutzt der `MethodManager` einen `CompiledCodeWriter`, der das von der Maschine verlangte Format hinter einer Nachrichtenschnittstelle kapselt. Hier handelt es sich um eine sehr klare Ausprägung des Erbauer-Musters, da es für die durch die Klasse `ByteCodeWriter` definierte Schnittstelle verschiedene Implementationen gibt. Neben dem `CompiledCodeWriter` existiert noch eine Implementation, welche statt einer maschinenlesbaren eine menschenlesbare Ausgabe produziert. Diese Klasse `Disassembler` wurde während der Entwicklung zur Überprüfung des MethodManagers benutzt. Im laufenden System findet sie Verwendung für die Testhilfe.

Das von der Maschine erwartete Format ist in Abschnitt 3.3.4 beschrieben.

Die Schnittstelle der Klasse `ByteCodeWriter` bietet eine Methode für jeden von der Maschine definierten Bytecode. Es findet keine weitere Optimierung durch Auswahl von Kurzformen statt. Die Argumente einer Methode entsprechen direkt den Argumenten des zugehörigen Bytecodes.

Einzige Ausnahme bilden die Sprungbefehle. Hier wird zunächst mit einem symbolischen Verweis auf das Sprungziel gearbeitet, der erst am Ende der Codeerzeugung durch einen numerischen Offset ersetzt wird. Erst dann findet auch die Entscheidung über Kurz- oder Langform des Sprungbefehls statt.

Die Codeerzeugung nimmt zunächst an, daß die Kurzform ausreichen wird. Stellt sich heraus, daß doch die Langform nötig wird, wird der Sprungbefehl umgewandelt. Nun kann aber die Länge eines Sprungs auch die Sprungdistanz von anderen Befehlen beeinflussen. Im allgemeinen sind daher mehrere Iterationen nötig. Das Verfahren terminiert, da die Länge der Befehle nur vergrößert, nie aber verkleinert wird, und das Ergebnis ist dann garantiert optimal.

5.6. Verwaltung des Klassengraphen

Die Klassenverwaltung ist für die Beziehungen zwischen Klassen verantwortlich, in der Hauptsache für die Konsistenz aller Klassen-Präzedenz-Listen. Wird eine neue Klasse über-

setzt, muß die Klassenverwaltung alle betroffenen Klassen identifizieren und aktualisieren. Treten dabei Fehler auf, muß der Ausgangszustand wiederhergestellt werden.

Die Verantwortung für die Konsistenz der von der Maschine verwendeten Klassenobjekte ist zwischen den Klassenobjekten und der Klassenverwaltung verteilt. Es sind zwar alle von der Maschine zugegriffenen Exemplarvariablen im Klassenobjekt selbst gespeichert. Die Variablen, auf deren Berechnung auch der Zustand anderer Klassen Einfluß hat, gehören aber konzeptuell zur Klassenverwaltung. Im Klassenobjekt ist nur der Teil der Berechnung implementiert, der allein mit dem Quellcode dieser Klasse arbeitet.

5.6.1. Vererbung

Wie in Abschnitt 3.1.1 beschrieben, berechnet sich die Klassen-Präzedenz-Liste transitiv aus den direkten Vorfahren einer Klasse. Aus der Klassen-Präzedenz-Liste und den in ihren Elementen definierten Methoden und Exemplarvariablen berechnet sich wiederum die Liste der Exemplarvariablen und die Liste der beim Nachrichtenempfang zu durchsuchenden Methodenverzeichnisse. Zum Verständnis dieses Vorgangs mag Abbildung 3.1 nützlich sein: Die in einer Klasse definierten Slots und Methoden erscheinen in einem *Fragment* der konkreten Objekte. Ein Exemplar einer Klasse enthält ein solches Fragment für jedes Element der Klassen-Präzedenz-Liste seiner Klasse. Die einzelnen Klassenobjekte sind damit für die Beschreibung ihrer Fragmente zuständig, der Klassenverwaltung obliegt die Beschreibung der vollständigen Exemplare.

Wird in einer Klasse eine neue Exemplarvariable oder Methode definiert, ohne daß sich die Vererbung ändert, kann die Neuberechnung der Klassen-Präzedenz-Listen übergangen werden. Das Format und die Methodenverzeichnisse der Nachfahrenklassen können ausgehend von der unveränderten Klassen-Präzedenz-Liste erstellt werden.

5.6.2. Isoliertheit der Änderungen

In der aktuellen Implementation verletzt die Klassenverwaltung das Kriterium der *Isoliertheit* von Transaktionen: Wenn vor Abschluß der Veränderung einer Klasse Nachrichten an Exemplare der betroffenen Klassen geschickt werden, können unvorhersehbare Effekte eintreten. Dieser Effekt läßt sich durch die Implementation in Tycoon nicht ganz vermeiden, er läßt sich jedoch einschränken, indem im relevanten Abschnitt nur wenige Systemklassen verwendet werden. Zur Aktivierung der geänderten Definition sind drei Zuweisungen an die maschinenrelevanten Exemplarvariablen nötig: `instanceSize`, `methodDicationaries`, `slotMap`. Danach muß die Maschine aufgefordert werden, ihren Cache für die betreffende Klasse zu löschen. Die Implementation im `ClassManager` sieht wie folgt aus:

```
c.instanceSize := instanceSize,
c.methodDictionaries := methodDictionaries,
c.slotMap := slotMap,
```

```
flushCache(c)
```

Ein Problem tritt z.B. dann auf, wenn die Namen der Exemplarvariablen von `Class` geändert werden: Nach der Zuweisung der neuen Methodenverzeichnisse wird die Nachricht `"slotMap:="` vom Klassenobjekt nicht mehr verstanden. Das Problem wäre vermeidbar, indem man obigen Code in eine atomare primitive Operation der virtuellen Maschine zusammenfaßt. Dies empfiehlt sich auch im Zusammenhang mit Nebenläufigkeit und Schemaevolution.

5.6.3. Metaklassen

Neben der Vererbung behandelt die Klassenverwaltung auch die Metaklassen-Beziehung: Soll eine Klassenbeschreibung mit bis dato unbekanntem Namen umgesetzt werden, wird das neue Klassenobjekt als Exemplar seiner Metaklasse erzeugt. Ändert sich später die Metaklassendeklaration, muß ein neues Klassenobjekt angelegt werden, da kein Objekt seine Klasse ändern kann, auch nicht Klassenobjekte. Alle Klassenvariablen (Exemplarvariablen des Klassenobjekts) gehen damit verloren.

Eine Klasse, die als Metaklasse verwendet wird, muß von `Class` erben, damit die von der Klassenverwaltung benötigte Funktionalität zur Verfügung steht. Darauf muß bei der Überprüfung der Metaklassendeklaration geachtet werden.

5.6.4. Kapselung der Klassenobjekte

Die Klassenverwaltung setzt erfolgreich eine transaktionale Semantik um. Eine echte Kapselung der Klassenobjekte ist jedoch nicht gelungen, auch wenn dies aus Sicherheitsgründen wünschenswert ist. Ähnliche Probleme werden immer dann auftreten, wenn eine Graphenstruktur nach außen hin gekapselt werden soll, insbesondere, wenn der Benutzer wie hier Nachfahrenklassen für die Knoten des Graphen definieren kann und einen Rückverweis von der Benutzerinformation zum Knoten des Graphen benötigt.

Um Exemplare einer Klasse erzeugen zu können, muß der Klienten-Code dem Klassenobjekt Nachrichten zur Objekterzeugung schicken können. Zu diesem Zweck ist jedes Klassenobjekt im globalen *pool* unter seinem Namen verfügbar. Dadurch wird jedoch auch die für die Aufgaben der Klassenverwaltung nötige breite Schnittstelle öffentlich verfügbar, so daß fehlerhafter oder bösartiger Code z.B. die im Klassenobjekt gespeicherte Klassen-Präzedenz-Liste verändern kann. Die Klassenverwaltung kann so genaugenommen keine Zusicherungen über deren Konsistenz machen.

Selbst wenn es gelänge, eine Typisierung zu finden, die der Klassenverwaltung Zugriff erlaubt, ihn anderen Systemteilen aber untersagt, hätte man immer noch keinen Schutz vor ohne Typprüfung ausgeführtem Code. Schutz vor ungeprüftem Code bietet nur die dynamische Trennung von öffentlichen und privaten Methoden.

Ein sichereres Szenario könnte wie folgt aussehen: Klassenobjekte werden zweigeteilt in einen System- und einen Benutzerteil. Global zugreifbar ist nur der Benutzerteil, der einen privaten Verweis auf den Systemteil enthält. Nachrichten zur Objekterzeugung werden vom Benutzerteil an den Systemteil weitergereicht. Ein Nachteil besteht in den von der zusätzliche Indirektion verursachten Performanzeinbußen.

Auch dieses Szenario ist noch nicht ganz sicher, da benutzerdefinierte Metaklassen als Nachfahrenklassen von `Class` freien Zugriff auch auf private Methoden haben: TL-2 bietet keinen dem `private protected` oder `private` von Java vergleichbaren Sichtbarkeitsbereich, der Methoden vor Nachfahrenklassen verbirgt.

Der Verweis auf den Systemteil darf daher nicht die Gestalt eines echten Objektverweises haben. Gerät nämlich der Objektverweis nach außen zur benutzerdefinierten Metaklasse, so ist die Kapselung wieder aufgebrochen. Einziger Ausweg ist also eine zweite Indirektion durch eine *handle* in Form eines weiteren Objekts oder einer Ganzzahl.

Im Sinne eines produktiven, performanten und verständlichen Systems wurde trotz der angegebenen Angriffspunkte die einfachste Implementation gewählt. Lediglich auf der Typebene wird von der Klassenverwaltung eine eingeschränkte Sicht auf die verwalteten Klassenobjekte geboten.

6. Zusammenfassung und Ausblick

In dieser Arbeit wurde die objektorientierte Implementation eines inkrementellen Übersetzers für die objektorientierte Programmiersprache Tycoon-2 beschrieben. Die Programmierumgebung, im Gegensatz zum Begriff Übersetzer verstanden als Gesamtheit der Werkzeuge zur Manipulation der Programme, ist in ihrer eigenen Eingabesprache geschrieben, und erlaubt übersetztem Code reflektiven Zugriff auf ihre Komponenten. Erzeugter Code wird der virtuellen Zielmaschine über gemeinsam genutzte Datenstrukturen im persistenten Objektspeicher zur Verfügung gestellt.

Die Programmierumgebung wurde durch einen Bootstrap ausgehend von einem Prototyp im Tycoon-1-System erstellt, indem ein eigenständiges, auf der neuen virtuellen Maschine lauffähiges Systemabbild erzeugt wurde.

Zur Beschreibung des Entwurfs diente die OMT-Notation. Die in [Gamma et al. 95] beschriebenen Entwurfsmuster konnten erfolgreich eingesetzt werden, und wurden weiter bereichert durch die Kombination mit *higher-order functions*.

Die Programmierumgebung ist funktionsfähig und wird kommerziell eingesetzt. Die in dieser Arbeit nicht weiter betrachtete Typüberprüfung steht kurz vor der Vollendung. Ein echter Bootstrap, d.h. die Erzeugung eines neuen Systemabbilds von der neuen Maschine aus, steht ebenfalls noch aus. Bis diese Teile fertiggestellt sind, wird noch parallel im Prototyp und im neuen System entwickelt, was die hohe Kompatiblität der Systeme verdeutlicht.

Zukünftige Ziele sind die weitergehende Untersuchung der Beziehung zwischen *higher-order functions* und Klassen, insbesondere im Hinblick auf Schemaevolution und Programmentwicklung im persistenten System. Auf der praktischen Seite steht die Entwicklung einer grafischen, intranet-basierten Interaktionskomponente für die Programmierumgebung mit Hilfe des Tycoon Web Server Framework an.

Bedanken möchte ich mich bei meinem Betreuer, Florian Matthes, der immer die richtigen Fragen hatte, und bei Andreas Gawecki von der Firma Higher Order, ohne den die praktische Realisierung nicht möglich gewesen wäre, sowie bei Marc Weikard, der die andere Seite der Brücke baute.

A. Syntax von Tycoon-2

Die Definition von syntaktischen und lexikalischen Elementen benutzt folgende Notation: *Id* bezeichnet ein nicht-terminales Symbol, A und B bezeichnen syntaktische Ausdrücke.

$Id_1, \ldots, Id_n ::= A;$	die nicht-terminalen Symbole Id_i sind definiert als A ($n \geq 1$)
Id	ein nicht-terminales Symbol
if	ein terminales Symbol
"x"	das Zeichen *x* ("" ist die leere Zeichenkette, """ sind Anführungsstriche)
(A)	bedeutet A
$A\ B$	bedeutet A gefolgt von B (stärkste Bindung)
$A \mid B$	bedeutet A oder B
[A]	bedeutet ("" | A)
{ A }	bedeutet ("" | A { A })

A.1. Tokens

Der Quelltext eines Tycoon-2-Programms besteht aus einer Zeichenfolge, die in eine Folge von *tokens* der Kategorien *int, real, char, string, identifier* sowie *punctuation* umgewandelt wird.

Die verfügbaren Formatierungszeichen bilden eine implementationsabhängige Untermenge der nicht-druckbaren Zeichen und enthalten mindestens die Zeichen Leerstelle, Tabulator, Zeilenrücklauf, Zeilenvorschub.

Kommentare sind Folgen beliebiger druckbarer oder Formatierungszeichen, die in *(* *)* eingeschlossen sind. Kommentare können geschachtelt werden.

Um ein *token* zu lesen, werden alle Formatierungszeichen übersprungen und es wird die längste Zeichenfolge akzeptiert, die ein gültiges *token* bildet.

int::=
    digit { *digit* }
real::=
    int "." *digit* { *digit* } |
    int ["." *digit* { *digit* }] ("E" | "e") ["+" | "-"] *int*
char::=
    "'" (*digit* | *alpha* | *special* | *escape* | *delimiter* | *space*) "'"
string::=
    "" { *digit* | *alpha* | *special* | *escape* | *delimiter* | *space* } ""
identifier::=
    alpha { *digit* | *alpha* }
punctuation::=
    "<" | "<=" | ">" | ">=" | "=" | "==" | "!=" | "!==" |
    "|" | "||" | "&" | "&&" | "=>" | "=>=>" | "?" | "!" |
    "+" | "*" | "−" | "/" | "%" | "<<" | ">>" |
    "<:" | ":" | *delimiter*
delimiter::=
    "(" | ")" | "{" | "}" | "[" | "]" | "." | "," | ";"
digit::=
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
alpha::=
    "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
space ::=
    " "
special::=
    "@" | "#" | "$" | "%" | "&" | "*" | "_" | "+" | "−" | "=" |
    "|" | "\" | "`" | ":" | "<" | ">" | "/" | "^" | "?" | "!"
escape::=
    "\" ("n" | "t" | "r" | "f" | "\" | "'" | "" | *digit digit digit*)

Außer der Leerstelle können Formatierungszeichen in Zeichenketten und Zeichen nur durch Escape-Sequenzen dargestellt werden. Escape-Sequenzen werden wie folgt interpretiert:

`\n`	Zeilenumbruch
`\t`	Tabulator
`\r`	Wagenrücklauf
`\f`	Seitenvorschub
`\'`	'
`\"`	"
`\\`	\
`\nnn`	Das Zeichen mit dem Code *nnn* (drei Dezimalziffern, die eine natürliche Zahl im Bereich [0,255] darstellen)

Die obigen Definitionen erlauben eine Implementation durch einen Scanner mit nur einem Zeichen *lookahead*.

A.2. Reservierte Schlüsselworte

Die folgenden Bezeichner sind reservierte Schlüsselworte und können in Tycoon-2-Programmen nicht als benutzerdefinierte Bezeichner verwendet werden.

DO define
class metaclass invariant public private methods
deferred extern builtin require ensure
Self Void Let
fun let var assert self super nil true false

A.3. Übersetzungseinheit

Ausgehend von den im vorigen Abschnitt definierten *tokens* und Schlüsselworten beschreiben die folgenden Produktionen und Präzendenzregeln eine eindeutige LALR(1)-Grammatik für Tycoon-2. *Unit* ist die Übersetzungseinheit der Sprache.

Unit::=
    (*Class* | *DoCommand* | *PoolDefinition* | *Sequence*) ";"
DoCommand::=
    DO *ident* { *string* | *ident* | *true* | *false* }
PoolDefinition::=
    define *ValueSignatures*

A.4. Klassen

Class::=
    class *ident* [*Domain*]
    [**super** *Types*]
    [*SelfConstraint*]
    [**metaclass** *Type*]
    [**invariant** *Value*]
    [**public** *Definitions*]
    [**private** *Definitions*]
Domain::=
    "(" *TypeSignature* { "," *TypeSignature* } ")"
SelfConstraint::=
    Self "<:" *Type* |
    Self "=" *Type*
Definitions::=
    Slots [**methods** { *Method* }]
Slots::=
    [*Slot* { "," *Slot* }]
Slot::=
    ValueSignature
Method::=
    Selector ["(" *Signatures* ")"] [":" *Type*]
    [**require** *Value*]
    [**ensure** *Value*]
    [*MethodBody*]
MethodBody::=
    Body |
    deferred |
    extern [*Language* [*ExternalName*]] |
    builtin [*Body*]
Language, ExternalName::=
    string
Selector::=
    ident | *string*

A.5. Werte

Value::=
    nil | **true** | **false** | **self** |
    Number | *string* | *char* |
    ident |
    "!" *Value* |
    Value Binop Value |
    Value "?" *Value* |
    Value "?" *Value* ":" *Value* |
    (*Value* | **super**) "." *Selector* ["(" *Arguments* ")"] |
    (*Value* | **super**) "." *Selector* ":=" *Value* |
    Selector "(" *Arguments* ")" |
    ident ":=" *Value* |
    Value "[" *Arguments* "]" |
    Value "[" *Arguments* "]" ":=" *Value* |
    "(" *Value* ")" |
    [**fun** "(" *Signatures* ")" [":" *Type*]] *Body* |
    assert *Value*
Number::=
    ["+" | "−"] (*int* | *real*)
Body::=
    "{" *Sequence* "}"
Sequence::=
    [*SequenceElement* { "," *SequenceElement* }]
SequenceElement::=
    Let *ident* "=" *Type* |
    let [**var**] *ident* [":" *Type*] "=" *Value* |
    Value
Arguments::=
    [*Argument* { "," *Argument* }]
Argument::=
    Value | ":" *Type* |
    let *ident* [":" *Type*] "=" *Value*
binop ::=
    "*" | "/" | "%" | ">>" | "<<" | "+" | "−" |
    "<" | "<=" | ">" | ">=" | "=" | "==" | "!=" | "!==" |
    "&" | "&&" | "|" | "||" | "=>" | "=>=>"

A.6. Signaturen

Signatures::=
[*Signature* { "," *Signature* }]
Signature::=
TypeSignature | *ValueSignature*
TypeSignature::=
[*ident*] "<:" *Type* |
[*ident*] "=" *Type*
ValueSignature::=
[*ident*] ":" *Type*

A.7. Typen

Type::=
ident | **Void** | **Self** |
Type "(" *Types* ")"
Types::=
[*Type* { "," *Type* }]

A.8. Präzedenzen

Die folgende Tabelle gibt Präzedenz und Assoziativität der in Wertausdrücken auftretenden Operatoren an. Die Zeilen sind nach absteigender Präzedenz geordnet. Operatoren in einer Zeile haben die gleiche Präzedenz.

Operator	Assoziativität
. [ ]	links
!	rechts
* / << >> %	links
+ −	links
< <= > >=	links
= == ! = ! ==	links
& &&	links
| ||	links
=> =>=>	links
:	keine
?	rechts
:=	links
assert	keine

B. Beispielsitzung

```
System re-started.
true
|> "hello" + " " + "world";
"hello world"
|> true;
true
|> tycoon.stdout.writeln("hello" + " " + "world"), true;
hello world
true
|> class Test
super Object
metaclass SimpleConcreteClass(Test)
public methods

sayHello
{ tycoon.stdout.writeln("hello") }
;

[Building class Test
]
|> Test.new.sayHello, true;
hello
true
|> define t :Test;
|> t := Test.new;
a Test
|> t.sayHello, true;
hello
true
|> class Test
super Object
metaclass SimpleConcreteClass(Test)
public methods

printOn(out :Output)
{ out.writeln("hello") }
;

[Building class Test
]
|> t;
hello
|> DO exit;
```

C. Dateiauflistung

C.1. builtins.h

```
/*
  Copyright (c) 1996 Higher-Order GmbH, Hamburg. All rights reserved.

  builtins.h 1.15 97/01/31 Marc Weikard

  Tycoon Virtual Machine Builtins

  format:  TVM_BUILTIN(<classname>, <selector>, <c name>, <arity>)
*/


  /* */
  TVM_BUILTIN(Builtin,"",fail,-1)

  /* */
  TVM_BUILTIN(Int,"+",add,1)
  TVM_BUILTIN(Int,"-",sub,1)
  TVM_BUILTIN(Int,"*",mult,1)
  TVM_BUILTIN(Int,"/",div,1)
  TVM_BUILTIN(Int,"%",mod,1)
  TVM_BUILTIN(Int,"==",equal,1)
  TVM_BUILTIN(Int,"<",less,1)
  TVM_BUILTIN(Int,"<=",lessOrEqual,1)
  TVM_BUILTIN(Int,">",greater,1)
  TVM_BUILTIN(Int,">=",greaterOrEqual,1)
  TVM_BUILTIN(Int,"|",or,1)
  TVM_BUILTIN(Int,"&",and,1)
  TVM_BUILTIN(Int,"xor",xor,1)
  TVM_BUILTIN(Int,"not",not,0)
  TVM_BUILTIN(Int,">>",shiftRight,1)
  TVM_BUILTIN(Int,"<<",shiftLeft,1)
  TVM_BUILTIN(Int,"asChar",asChar,0)

  /* */
  TVM_BUILTIN(Object,"==",equal,1)
  TVM_BUILTIN(Object,"class",class,0)
  TVM_BUILTIN(Object,"shallowCopy",shallowCopy,0)
  TVM_BUILTIN(Object,"_typeCast",_typeCast,1)
  TVM_BUILTIN(Object,"_hash",_hash,0)
  TVM_BUILTIN(Object,"_setHash",_setHash,1)
  TVM_BUILTIN(Object,"__basicSize",_basicSize,0)
  TVM_BUILTIN(Object,"__basicAt",_basicAt,1)
  TVM_BUILTIN(Object,"__basicAtPut",_basicAtPut,2)
```

```
/* */
TVM_BUILTIN(String,"[]",at,1)
TVM_BUILTIN(String,"size",size,0)
TVM_BUILTIN(String,"_set",_set,2)
TVM_BUILTIN(String,"_replace",_replace,4)

/* */
TVM_BUILTIN(MutableString,"[]:=",atPut,2)

/* */
TVM_BUILTIN(Array,"[]",at,1)
TVM_BUILTIN(Array,"size",size,0)
TVM_BUILTIN(Array,"_set",_set,2)
TVM_BUILTIN(Array,"_replace",_replace,4)

/* */
TVM_BUILTIN(ByteArray,"[]",at,1)
TVM_BUILTIN(ByteArray,"[]:=", atPut,2)
TVM_BUILTIN(ByteArray,"size",size,0)
TVM_BUILTIN(ByteArrayClass,"_new1",_new1,1)

/* */
TVM_BUILTIN(ShortArray,"[]",at,1)
TVM_BUILTIN(ShortArray,"[]:=", atPut,2)
TVM_BUILTIN(ShortArray,"size",size,0)
TVM_BUILTIN(ShortArrayClass,"_new1",_new1,1)

/* */
TVM_BUILTIN(IntArray,"[]",at,1)
TVM_BUILTIN(IntArray,"[]:=", atPut,2)
TVM_BUILTIN(IntArray,"size",size,0)
TVM_BUILTIN(IntArrayClass,"_new1",_new1,1)

/* */
TVM_BUILTIN(LongArray,"[]",at,1)
TVM_BUILTIN(LongArray,"[]:=", atPut,2)
TVM_BUILTIN(LongArray,"size",size,0)
TVM_BUILTIN(LongArrayClass,"_new1",_new1,1)

/* */
TVM_BUILTIN(MutableArray,"[]:=",atPut,2)

/* */
TVM_BUILTIN(Char,"asInt",asInt,0)

/* */
TVM_BUILTIN(DLL,"_open",_open,1)
TVM_BUILTIN(DLL,"_close",_close,1)

/* */
TVM_BUILTIN(Exception,"_raise",_raise,0)

/* */
TVM_BUILTIN(Fun0,"[]",apply,0)
TVM_BUILTIN(Fun1,"[]",apply,1)
TVM_BUILTIN(Fun2,"[]",apply,2)
TVM_BUILTIN(Fun3,"[]",apply,3)
TVM_BUILTIN(Fun4,"[]",apply,4)
TVM_BUILTIN(Fun5,"[]",apply,5)
TVM_BUILTIN(Fun6,"[]",apply,6)
```

```
TVM_BUILTIN(Fun7,"[]",apply,7)
TVM_BUILTIN(Fun8,"[]",apply,8)
TVM_BUILTIN(Fun9,"[]",apply,9)
TVM_BUILTIN(Fun10,"[]",apply,10)

/* */
TVM_BUILTIN(Tycoon,"errno",errno,0)
TVM_BUILTIN(Tycoon,"backTrace",backTrace,0)
TVM_BUILTIN(Tycoon,"_rollback",_rollback,0)
TVM_BUILTIN(Tycoon,"_commit",_commit,0)
TVM_BUILTIN(Tycoon,"builtinArgv",builtinArgv,0)
TVM_BUILTIN(Tycoon,"_platformCode",_platformCode,0)

/* */
TVM_BUILTIN(ConcreteClass,"_new",_new,0)
TVM_BUILTIN(AbstractArrayClass,"_new1",_new1,1)
TVM_BUILTIN(AbstractStringClass,"_new1",_new1,1)

/* */
TVM_BUILTIN(ClassManager,"flushAll",flushAll,0)
TVM_BUILTIN(ClassManager,"flushClass",flushClass,1)
TVM_BUILTIN(ClassManager,"flushSingle",flushSelector,2)

/* */
TVM_BUILTIN(AbstractScanner,"_builtinMove",move,4)

/* */
TVM_BUILTIN(Root,"newClassId",newClassId,2)

/* */
TVM_BUILTIN(CStructClass,"_new",_new,0)
```

C.2. tyc.h

```
/*
  Copyright (c) 1996 Higher-Order GmbH, Hamburg. All rights reserved.

  tyc.h 1.16 97/01/31 Andreas Gawecki, Marc Weikard

  Tycoon Objects used by the machine

*/

#ifndef TYC_H
#define TYC_H

#include "hox.h"
#include "rtdll.h"
#include "tsp.h"

/* Boxed values: */

typedef struct tyc_Bool {
  Bool value;
} tyc_Bool;

typedef struct tyc_Char {
  Char value;
} tyc_Char;

/* only for large integers: */
typedef struct tyc_Int {
  Int value;
} tyc_Int;

typedef struct tyc_Long {
  Long value;
} tyc_Long;

typedef struct tyc_Real {
  Real value;
} tyc_Real;

extern tsp_OID * tyc_boxInt(Int value);
extern tsp_OID * tyc_boxLong(Long value);
extern tsp_OID * tyc_boxReal(Real value);

#define tyc_boxChar(value) \
  ((tsp_OID)(tyc_pRoot->apCharTable[(value) & 0xff]))
#define tyc_boxBool(value) \
  ((tsp_OID)((value) ? tyc_pRoot->pTrue : tyc_pRoot->pFalse))

#define tyc_IS_TRUE(x)  ((x) == tyc_pRoot->pTrue)
#define tyc_IS_FALSE(x) ((x) == tyc_pRoot->pFalse)
#define tyc_IS_NIL(x)   ((x) == NULL)

#define tyc_TRUE  (tyc_pRoot->pTrue)
#define tyc_FALSE (tyc_pRoot->pFalse)
#define tyc_NIL   NULL

#define tyc_CLASS(id)     (tyc_pRoot->apClassTable[(id)])
#define tyc_SELECTOR(id)  (tyc_pRoot->apSelectorTable[(id)])
```

```
#define tyc_ARGUMENTS(id) (tyc_pRoot->apArgTable[(id)])


/* ClassId codes used by the virtual machine: */

#define TYC_CLASSID(id, l, s) tyc_ClassId_ ## id,
typedef enum {

  #include "classids.h"

  tyc_ClassId_nPredefined
} tyc_PredefinedClassIds;
#undef TYC_CLASSID

typedef Word tyc_ClassId;


/* Tycoon object format characters
   for CStructs and ExternalMethods */
typedef enum {
  tyc_Type_Bool =          'B',
  tyc_Type_Char =          'C',
  tyc_Type_Int =           'I',
  tyc_Type_Long =          'L',
  tyc_Type_Real =          'R',
  tyc_Type_Object =        'O',
  tyc_Type_String =        'S',    /* ExternalMethod argument or return type only */
  tyc_Type_MutableString = 'M',    /* ExternalMethod argument or return type only */
  tyc_Type_Void =          'V'     /* ExternalMethod return type only */
} tyc_Type;


/* Tycoon objects: */

typedef HWord tyc_SelectorID;
typedef HWord tyc_ClassID;

typedef char *    tyc_Symbol;
typedef tsp_OID * tyc_Array;
typedef Byte *    tyc_ByteArray;
typedef Short *   tyc_ShortArray;
typedef Int *     tyc_IntArray;
typedef void      tyc_Nil;


typedef struct tyc_List {
  tsp_OID pHead;
  struct tyc_List * pTail;
} tyc_List;

typedef struct tyc_Class {
  char * pszName;                  /* class name */
  tsp_OID wInstanceId;             /* instance id (tagged tyc_ClassID) */
  tsp_OID wInstanceSize;           /* instance size in slots (tagged int) */
  tyc_List * pMethodDictionaries;  /* list of method dictionaries */
  tsp_OID _slotMap;                /* reserved */
} tyc_Class;

typedef struct tyc_MethodDictionary {
  tsp_OID _elementCount;           /* tagged int */
```

```
  tyc_Symbol * apszKeys;
  struct tyc_Method ** apElements;
  tyc_Class * pClass;
} tyc_MethodDictionary;

typedef struct tyc_DLL {
  tsp_OID hDLL;                     /* must be an int */
  char * pszPath;
} tyc_DLL;

typedef struct tyc_CatchFrame {     /* ip relative to beginning of bytecode */
  short ipFrom;                     /* start of exception frame */
  short ipTo;                       /* end of exception frame */
  short ip;                         /* start of handler */
  short cwLocals;                   /* locals (words) on stack */
} tyc_CatchFrame;                   /* internal representation as ShortArray */

typedef struct tyc_Method {
  tyc_Symbol pSelector;             /* selector */
  Bool fIsPrivate;
  int nArgs;                        /* number of value arguments */
} tyc_Method;

typedef struct tyc_CompiledMethod {
  tyc_Method method;

  tyc_Class * pClass;
  Byte * pbCode;
  tyc_Array * pLiterals;
  tyc_SelectorID idSelector;
  int cwStackPeak;                  /* stack peak */
  tyc_CatchFrame * asHandlerTable;  /* array, may be null (no handlers) */
  void * pNativeCode;               /* currently null */
} tyc_CompiledMethod;

typedef struct tyc_BuiltinMethod {
  tyc_CompiledMethod compiledMethod;

  int iNumber;
} tyc_BuiltinMethod;

typedef struct tyc_CompiledFun {
  tyc_CompiledMethod compiledMethod;

} tyc_CompiledFun;

typedef struct tyc_ExternalMethod {
  tyc_Method method;

  char * pszLanguage;
  char * pszLabel;
  char * pszTycoonArgs;         /* uses tyc_Type */
  char * pszCArgs;              /* uses tsp_Field */
} tyc_ExternalMethod;

typedef struct tyc_slotMethod {
  tyc_Method method;

  int iOffset;
} tyc_SlotMethod;
```

```
typedef struct tyc_SlotAccessMethod {
  tyc_SlotMethod slotMethod;

} tyc_SlotAccessMethod;

typedef struct tyc_SlotUpdateMethod {
  tyc_SlotMethod slotMethod;

} tyc_SlotUpdateMethod;

typedef struct tyc_PoolMethod {
  tyc_Method method;

  tsp_OID * pCell;
} tyc_PoolMethod;

typedef struct tyc_PoolAccessMethod {
  tyc_PoolMethod poolMethod;

} tyc_PoolAccessMethod;

typedef struct tyc_PoolUpdateMethod {
  tyc_PoolMethod poolMethod;

} tyc_PoolUpdateMethod;

typedef struct tyc_CSlotAccessMethod {
  tyc_SlotAccessMethod slotAccessMethod;

  char cTycoonType;
} tyc_CSlotAccessMethod;

typedef struct tyc_CSlotUpdateMethod {
  tyc_SlotUpdateMethod slotUpdateMethod;

  char cTycoonType;
} tyc_CSlotUpdateMethod;

typedef struct tyc_Fun {            /* only one struct for all FunNs */
  tyc_CompiledFun * pCompiledFun;
  tsp_OID awGlobals[1];             /* actually 0..n */
} tyc_Fun;


typedef tsp_OID * tyc_Stack;

typedef struct tyc_Thread {
  void * hOsThread;
  tyc_Stack pStack;
  tyc_Stack sp;
  tyc_StackFrame * fp;
  Byte * ip;
} tyc_Thread;

/* exceptions generated by the machine: */

typedef struct tyc_TypeError {
  tsp_OID pObject;
  tyc_Class * pClass;
```

```
} tyc_TypeError;

typedef struct tyc_DoesNotUnderstandError {
  tsp_OID pReceiver;
  tyc_Symbol pSelector;
} tyc_DoesNotUnderstandError;

typedef struct tyc_MustBeBooleanError {
  tsp_OID pObject;
} tyc_MustBeBooleanError;

typedef struct tyc_DLLOpenError {
  tsp_OID pDLL;
} tyc_DLLOpenError;

typedef struct tyc_DLLCallError {
  tsp_OID pDLL;
  char * pszEntry;
} tyc_DLLCallError;

typedef struct tyc_DivisionByZeroError {
  tsp_OID pObject;
} tyc_DivisionByZeroError;

typedef struct tyc_IndexOutOfBoundsError {
  tsp_OID pObject;
  tsp_OID pIndex;
} tyc_IndexOutOfBoundsError;


typedef struct tyc_Root {
  tyc_Thread * pThread;         /* actually tyc_Fun0 during bootstrap */
  tyc_Class ** apClassTable;    /* mapping class id <-> tyc_Class object */
  tyc_Symbol * apSelectorTable; /* mapping selector id <-> tyc_Symbol object */
  Byte       * apArgTable;      /* mapping selector id <-> # of arguments */
  tyc_Bool * pTrue;
  tyc_Bool * pFalse;
  tyc_Nil  * pNil;              /* NULL */
  tyc_Char ** apCharTable;      /* mapping ascii <-> tyc_Char object */
} tyc_Root;


extern tyc_Root * tyc_pRoot;
  /* the root of the persistent store */

#endif
```

C.3. tvm.doc

```
Copyright (c) 1996 Higher-Order GmbH, Hamburg. All rights reserved.

File: tvm.doc
Authors: Mark Weikard, Andreas Gawecki, Axel Wienberg
Purpose: describes the tycoon virtual machine

Root object
-----------

threads :List(Thread)          - circular list of threads
classTable :Array(Object)      - mapping class ids to class objects
selectorTable :Array(Object)   - mapping name to selector ids
argumentTable :Array(Int)      - mapping arity to selector ids
true  :Object
false :Object
charTable :Array(Object)
...


Registers
---------

  ip     instruction pointer
  fp     frame pointer
  sp     stack top pointer


Stack frame layout
------------------

stack grows downwards  (towards zero)

[high address]
     [previous frames]
     ...
     arg0 (= self/receiver)
     arg1
     ...
     argn
     saved ip                          first control word
     saved fp
     receiver
     CompiledCode                      last control word
fp:  ByteCode
     local1
     ...
sp:  localn
     [first free word]

Note: The bytecode is independent of the number of control words since there
are special instructions to access local variables.

Compiled Code Object
--------------------

CompiledCode objects represent both compiled methods and compiled functions.

Slots:
```

```
class_ :Class_                the class the method is compiled in
byteCode :ByteArray
literals :Array(Object)
selectorID :Int
stackPeak :Int
arguments :Int                without receiver
exceptionTable :ShortArray    may be nil
nativeCode :ByteArray         may be nil


exceptionTable: quadrupels fromIP..toIP -> handlerIP,sp

Try/catch is implemented as follows.  For each 'try' that appears in the
source, the exceptionTable contains the ip range of the corresponding try body,
an ip to jump to if an exception occurs, and the corresponding stack depth (in
words) relative to fp (must be positive).

The list of exception handlers is sorted by ip. If one range is inside another,
then outermost range (the one that encompasses the other) appears last in the
list.  Therefore, it is possible to search forward, and the first one that
matches is the innermost one.


Instruction set
---------------
Instructions are encoded as one byte "opcode" followed by argument bytes.
<selector-id>s consist of two bytes, <n> is a one-byte-value (->note 4).
values > 1 byte are encoded in little endian. Jump offsets are relative
to the next instruction.

send <selector-id>
        <selector-id> encodes the selector name and number of arguments.
        Arguments are expected to be pushed from left to right, receiver first.
        Performs a method lookup on the receiver (argument lowest on the stack).
        - If no method is found, a DoesNotUnderstand exception is raised.
        - If an OrdinaryMethod is found, the stackPeak is examined,
          extending the stack if necessary,
          and a new stack frame is created and initialized. Execution continues
          at the beginning of the method's byteCode.
        - If a BuiltinMethod is found, the builtin code for the method's number
          gets a chance to handle the call.  If it can't, the TL2 code is
          executed as in an OrdinaryMethod.
        - SlotAccessMethods push their receiver's nth slot on the stack.
        - SlotUpdateMethods set their receiver's nth slot to their argument,
          and return the argument.
        - CStructAccessMethods and CStructUpdateMethods are similar, but
          consider the architecture-specific offset

send0 <selector-id>
send1      "
send2      "
send3      "
send4      "
send5      "
        like send <selector-id> with fixed number of arguments

sendSuper <selector-id>
        same as send, except that method lookup skips the method dictionaries
        of the receiver's CPL up to and including the method dictionary which
```

```
        points to the CompiledCode.class, i.e. the class in which the current
        CompiledCode object has been defined in.

global <n>
        push the nth global variable.
        The current CompiledCode object must be a CompiledFun
        created by closure <nb_globals>, where 1 <= n <= nb_globals.

  pseudocode:
        push(fp->receiver[n])

literal <n>
        push the nth element of the literal array (zero based)

arg <n>
        push an argument. n must obey 0 <= n < nb_args.
        self is argument number n, last argument is 0.
  pseudocode:
        push(fp[nb_control_Words + n])

(-> note 1)

local <n>
        push the nth local variable, 1 <= n <= nb_locals.
  pseudocode:
        push(fp[-n])

char <n>
        push oid for nth character object
  pseudocode:
        push(pRoot->charTable[n])

byte <n>
        push signed byte extended to tagged int

short <n>
        push signed short extended to tagged int

true, false, nil, minus_one, zero, one, two
        push frequently used constants

storeLocal <n>
        store (do not pop) the topmost stack element in the nth local variable,
        1 <= n <= nb_locals.
  pseudocode:
        let val = top(),
        fp[-n] := val

cellNew
        create a new cell for mutable bindings (array(tsp_OID) with 1 element)
             sp: value
  pseudocode:
        *sp = tsp_newCell(*sp)

cellLoad
        load value from allocated cell
             sp: cell oid
  pseudocode:
        *sp = (*sp)[0]
```

```
cellStore
        store value into cell
                 value
             sp: cell oid
  pseudocode:
        (*sp)[0] = *(sp + 1)
        pop()

return
        Pops the current frame and arguments from the stack,
        and leaves the topmost stack element where the arguments were.
        The number of arguments is accessible through the compiledCode pointer.
  pseudocode:
        let retval = top(),
        sp = ((Word*)(fp + 1)) + fp->compiledCode->nArgs;
        ip = fp->parent.ip;
        fp = fp->parent.fp;
        *sp = (retval)

ifTrue <offset 8 bit> || <offset 16 bit>
        If the topmost stack element is not a boolean, the exception
        MustBeBoolean is raised.
        Otherwise, the element is popped and the instruction pointer is
        adjusted by offset, a one (or two, for a long jump version)
        byte two's complement value, if the element is true

  pseudocode:
        let val = pop(),
        val == true ? {
          ip += offset
        }

ifFalse <offset 8 bit> || <offset 16 bit>
        cf. ifTrue

jump <offset 8 bit> || <offset 16 bit>
        unconditional jump
  pseudocode:
        ip += pos

pop
        pop one element off the stack
  pseudocode:
        pop()

drop <n>
        pop n elements off the stack
  pseudocode:
        sp := sp + n

adjust <n>
        pop n element under the top element off the stack.
  pseudocope:
        v = *sp
        sp := sp + n
        *sp = v

closure <n>
        Create a closure. Stack contents before the instruction:
              globaln
```

```
                ...
                global1
          sp: CompiledFun
        These are replaced by the newly created function object.

sendTail <selector-id> <n>
        Discards the current frame and arguments, leaving the n topmost stack elements
        where the arguments in this stack frame were, and sends <selector-id>.
        Similar effect like "send-tail <selector-id> ; return <n>", but avoids
        harmful stack growth.

send"+"/1
send"=="/1
...
        Several instructions are allocated as sends with fixed selector.
        This makes the code more compact (one byte instead of three).
        In addition, the interpreter knows about builtins (like Int."+"/1),
        and in these cases can skip the method lookup (-> note 3)
```

C.4. Schnittstellen der Systemkomponenten

```
(* File: MutableClassState.tc
 Author: Axel Wienberg
Purpose: parameter block for ClassManager::set
*)
class MutableClassState
super ClassState
metaclass SimpleConcreteClass(MutableClassState)
public

pos :SourcePos,
sourceTime :Long,
fingerPrint :Long,

name :String,

selfTypeSig :TypeIde,
    (* Self ide: constraints on Self, i.e. =,<: type *)

domain :List(TypeIde),

supers :List(CPLType),
    (* CPLTypes of the direct superclasess in declaration order. *)

metaClassDeclaration :CPLType,

publicSlots :List(ValueIde),

privateSlots :List(ValueIde),

methodDictionary :MethodDictionary(Method)

methods

setState(state :ClassState)
  require state.name == name
;

(* File: ClassBuilder.tc
 Author: Axel Wienberg
Purpose: constructing classes; root of the abstract syntax tree for
         class definitions.
Usage:

   The client (TWLGram) creates a ClassBuilder object, specifying some
   information, and then sets private and public slots and methods.
   (methods have to be assigned to {public,private}Methods, not into the
   methodDictionary!)

   Later, classBuilder.build creates a proper class from this
   information and installs the class with the system.
*)
class ClassBuilder
super MutableClassState
metaclass ClassBuilderClass
public

privateMethods: List(Method),
```

```
publicMethods: List(Method)

methods

build(compiler :Compiler, classManager :ClassManager, errorLog :ErrorLog)
;

(* File: ClassManager.tc
 Author: Axel Wienberg
   Date:
Purpose: manage the systems inheritance and metaclass graph.
*)
class ClassManager
super TL2
metaclass ClassManagerClass
invariant
    (* unique key (primary index) on Class::name *)
     classTable.keys.all(fun(s:String) {
       classTable[s].name = s
     })
   & classTable.all(fun(c:Class_) {
       c.typeIde == signatures[c.name]
     })
   & (* cpl computed correctly *)   (* ### order *)
     classTable.all(fun(c:Class_) {
       directSuperClasses(c).all(fun(c1 :Class_) {
         superclasses(c).includesAll(superclasses(c1))
       })
     })
   & (* graph is acyclic *)
     classTable.all(fun(c:Class_) {
       !superclasses(c).includes(c)
     })
   & (* all classes are scoped correctly *)
     true  (* not expressible *)

public methods

(* update the system so <state> represents the current
   state of class <state.name>.  Issue errors on <errorLog>.
   If errors occur, the state remains unchanged. *)
set(state :ClassState, errorLog :ErrorLog)

(* get the current definition of class <name>.
   we only need to access the public part. *)
get(name :String) :ClassPublic

(* read-only view on the class table. *)
classTable :KeyedContainer(String, ClassPublic)

(* throw away all abstract syntax trees (save memory) *)
finalize

superclasses(clazz :ClassPublic) :Reader(ClassPublic)
  require classTable.includes(clazz)

subclasses(clazz :ClassPublic) :Reader(ClassPublic)
  require classTable.includes(clazz)

"metaclass"(clazz :ClassState) :ClassPublic
```

```
directSuperClasses(c :ClassState) :List(ClassPublic)
;

(* File: VMInterface.tc
 Author: Axel Wienberg
   Date:
Purpose: static information about the virtual machine
*)
class VMInterface
super TL2
metaclass VMInterfaceClass
public

opcodes :Dictionary(String, OpcodeDescriptor),

(* literal object --> opcode name *)
specialLiterals :Dictionary(Object, String),

builtins :Array(BuiltinDescriptor),

(* Selector --> opcode name *)
specialSends :Dictionary(Selector, String)
;

(* File: Pool.tc
 Author: Axel Wienberg
   Date: ??-Jun-1996
Purpose: global variable pool
*)
class Pool
super TL2
metaclass SingletonClass(Pool)
public methods

(* the MethodDictionary holding all pool methods.
   used by the class manager. *)
poolMethods :MethodDictionary(PoolMethod)

poolCells :KeyedContainer(Symbol, Cell(Object))

(* add a cell for <v>, if there is none with that name yet,
   and pool methods for accessing and updating <v>. *)
defineVariable(v :ValueIde)

(* add a cell with initial content <clazz> and an access
   method for clazz's name.
   make sure there is no update method. *)
defineClass(clazz :ClassPublic)

(* remove a pool variable, and its access and update method.
   <name> must have been previously created using definePoolVariable.
   Don't call this for classes! *)
undefinePoolVariable(name :String)
;

(* File: TL2Terminals.tc
 Author: Axel Wienberg
   Date: ??-Jun-1996
Purpose: an object storing all terminal symbols, decoupling the parser from
```

```
        the scanner.
        The symbols are entered and read by TL2Gram and read by
        ReentrantTL2Scanner.
*)
class TL2Terminals
super TL2
metaclass SimpleConcreteClass(TL2Terminals)
public

ident :ValueTerminal(Positioned(String)),
string :ValueTerminal(Positioned(String)),
int :ValueTerminal(Positioned(Int)),
char :ValueTerminal(Positioned(Char)),
real :ValueTerminal(Positioned(Real)),

symbols :Dictionary(String, ValueTerminal(SourcePos)),
keywords :Dictionary(String, ValueTerminal(SourcePos))

methods

(* return the Terminal representing keyword <name>.
   <name> must match idRegExp.
   Multiple calls with the same <name> argument return the same
   SimpleTerminal object.
 *)
keyword(name :String) :ValueTerminal(SourcePos)

(* return the Terminal representing the simple Symbol <name>.
   <name> may not match any other regexp (esp. idRegExp!)
   Multiple calls with the same <name> return the same SimpleTerminal object.
 *)
symbol(name :String) :ValueTerminal(SourcePos)
;

(* File: AbstractParser.tc
 Author: Axel Wienberg
   Date:
Purpose: interface for parsers, whatever the implementation
*)
class AbstractParser(Derived <: Object)
super Parse
metaclass AbstractClass
public methods

(* compute some value from the token stream <in>.
   In case of errors, call <error> with the offending token and return nil. *)
parse(in :Reader(Token(Object)),
      error :Fun1(Token(Object), Void)) :Derived
  deferred
;

(* File: ReentrantTL2Scanner.tc
 Author: Axel Wienberg
   Date:
Purpose: create independent TL2Scanners for each input source
*)
class ReentrantTL2Scanner
super TL2
metaclass ReentrantTL2ScannerClass
public methods
```

```
(* read input from <in>.  Produced tokens give <where> as source path.
   errors are issued to <errorLog>. *)
scan(where :String,
     in :Reader(Char),
     errorLog :ErrorLog) :Reader(Token(Object))
;

class ReentrantTL2ScannerClass
super ConcreteClass(ReentrantTL2Scanner)
metaclass MetaClass
public methods

new(twlTerminals :TWLTerminals) :ReentrantTL2Scanner
;


(* File: Compiler.tc
 Author: Axel Wienberg
   Date:
Purpose: compilation and evaluation of TL-2 expressions.
         mainly used to evaluate function expressions, resulting
         in compiled functions.
*)
class Compiler
super TL2
metaclass CompilerClass
public

errorLog :ErrorLog,
(* ### the error log should be specific to each activation of the compiler!
       no guarantee what happens when doing multiple compilations in parallel now. *)

methods

(* evaluate <expression> in an environment consisting only of "self" with
   <inClass>'s type, bound to <self_>.  Issue errors on <errorLog>. returns
   result of evaluation. *)
simpleEval(self_ :Object, inClass :ClassPublic, expression :Value, errorLog :ErrorLog) :Object

(* evaluate <expression> in <environment>.  For the needs of super send,
   this code is assumed to reside in class <inClass>.  Issue errors on
   <errorLog>. <environment> has to contain at least a binding for "self".
   returns result of evaluation. *)
eval( environment :Dictionary(ValueIde, Object),
      inClass :ClassPublic,
      expression :Value,
      errorLog :ErrorLog) :Object

(* for the MethodManagers *)
getSelectorId(selector :Selector) :Int

vmInterface :VMInterface
;

class CompilerClass
super ConcreteClass(Compiler)
metaclass MetaClass
public methods
```

```
(* create a new compiler using the specified components *)
new(vmInterface :VMInterface, root :Root) :Compiler
;


(* File: TopLevel.tc
 Author: Axel Wienberg
   Date:
Purpose: user interface to the TL-2 system
*)
class TopLevel
super TL2ParseUnitVisitor(Void), TL2
metaclass TopLevelClass
public

printByteCode :Bool,
printTycoonSource :Bool

methods

(* execute contents of <path> *)
file(path :String)

(* execute contents of <str> *)
string(str :String)

(* execute <stdin> until the next eof *)
stdin
;


class TopLevelClass
super ConcreteClass(TopLevel)
metaclass MetaClass
public methods

(* create a new toplevel using the specified components *)
new( scanner :ReentrantTL2Scanner,
     parser :AbstractParser(TL2ParseUnit),
     compiler :Compiler,
     classManager :ClassManager,
     pool :Pool) :TopLevel
;
```

Literaturverzeichnis

Aho et al. 88: A. V. Aho, R. Sethi, und J. D. Ullman. *Compilerbau.* Addison-Wesley, 1988.

Appel 94: A. Appel. „Axiomatic Bootstrapping: A guide for compiler hackers". Technischer Bericht CS-TR-451-94, Princeton University, 1994.

Bobrow, Stefik 81: D. G. Bobrow und M. Stefik. „The Loops Manual". Technischer Bericht, Xerox Palo Alto Research Center, 1981.

Bratman 61: H. Bratman. „An alternate form of the 'Uncol diagram'". *Communications of the ACM*, 4(3):142, 1961.

Bruce 93: K. B. Bruce. „Safe Type Checking in a statically typed object-oriented language". In: *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

Cardelli 91: L. Cardelli. „F-sub, the System". Technischer Bericht, DEC SRC, 1991.

Clinger, Rees 91: W. Clinger und J. Rees. „Revised^4 Report on the Algorithmic Language Scheme". *LISP Pointers*, 4(3):1–55, 1991.

Earley, Sturgis 70: J. Earley und H. Sturgis. „A formalism for translator interaction". *Communications of the ACM*, 13:607–617, 1970.

Ernst 97: M. Ernst. „Typprüfung in TL-2". Studienarbeit, Universität Hamburg, 1997.

Gamma et al. 95: E. Gamma, R. Helm, R. Johnson, und J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

Goldberg, Robson 83: A. Goldberg und D. Robson. *Smalltalk-80: The language and its implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

Hölzle 94: U. Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* Dissertation, Stanford University, 1994.

Hopcroft, Ullman 79: J. E. Hopcroft und J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

Johnson 75: S. C. Johnson. „Yacc: Yet Another Compiler-Compiler“. Technischer Bericht, Bell Laboratories, 1975.

Kerner 89: H. Kerner, Hrsg. *Rechnernetze nach ISO–OSI, CCITT*. ??, 1989.

Lecarme, Peyrolle-Thomas 78: O. Lecarme und M.-C. Peyrolle-Thomas. „Self-compiling Compilers: An Appraisal of their Implementation and Portability“. *IEEE Software — Practice and Experience*, 8:149–170, 1978.

Lesk, Schmidt 75: M. E. Lesk und E. Schmidt. „Lex – a Lexical Analyzer Generator“. Technischer Bericht, Bell Laboratories, 1975.

Meyer 88: B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

Schröder, Matthes 92: G. Schröder und F. Matthes. „Using the Tycoon Compiler Toolkit“. Technischer Bericht, Universität Hamburg, 1992.

Steele 90: G. L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 2. Auflage, 1990.

Tennent 81: R. D. Tennent. *Principles of Programming Languages*. Prentice Hall International, 1981.

Wulf et al. 73: W. A. Wulf, R. K. Johnsson, C. B. Weinstock, und S. O. Hobbs. „The Design of an Optimizing Compiler“. Technischer Bericht AFOSR-TR-74-0096, Carnegie Mellon University / Air Force Office of Scientific Research, 1973.