

PÜ1

Projektübungsblatt: Interpreter = Zerteiler + Besucher

Our belief in the correctness of a translator can never be founded on succesful [!!] tests alone, but is ultimately derived from the clean and systematic structure of the translator and from nothing else.

Edsger W. Dijkstra
Some meditations on Advanced Programming
Proceedings IFIP Congress 1962

I can still recommend single-pass top-down recursive descent both as an implementation method and as a design principle for a programming language.

Charles Antony Richard Hoare
The Emperor's Old Clothes
ACM Turing Award Lecture 1980

Within the Euler, Algol-W and PL360 projects, much consideration was given to the development of table-driven, bottom-up syntax analysis techniques. Later I switched back to the simple recursive-descent, top-down method, which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen.

Niklaus Wirth
From Programming Language Design to Computer Construction
ACM Turing Award Lecture 1984

PÜ1.1

Vorbemerkungen

Dieses Projekt greift das Thema „Ausdrücke berechnen – Interpreter mit rekursivem Abstieg“ von *Informatik 2 Praktikum, Übungsblatt 4, Sommersemester 2009* auf, zu dem die Ideenspender

CP-Referenzlösungen I2ArithmeticExpressionsRD*, I2ArithmeticExpressionsDB*

in *Component Pascal* mit *BlackBox* in den Informatik-Hug-Subsystemen existieren, erreichbar unter

http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_1+2
ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrmaterial/Informatik_1+2

Eine Implementation in *Eiffel* mit den Klassen

Eiffel-Referenzlösung *EXPRESSION_CALCULATOR*, *EXPRESSION_INTERPRETER**

steht bereit unter

http://13/Eiffel/EiffelStudio/simple_examples
ftp://13/Eiffel/EiffelStudio/simple_examples

Geeignete Implementationssprachen: Alle.

Aspekte: prozeduraler, modularer, objektorientierter Entwurf, Trennung von Aspekten, Kommandozeilenanwendung, Zerteilermethode rekursiver Abstieg, Modul, Klasse, Kunde-Lieferant-Beziehung, Schnittstelle, Ausnahmebehandlung, Korrektheit, Robustheit, Gebrauchstauglichkeit, Änderbarkeit, Erweiterbarkeit, Vererbung, Polymorphie, dynamische Objektstruktur Baum, Entwurfsmuster Besucher

PÜ1.2

Lernziele und Aufgabenbeschreibung

Lernziele

Zu den Lernzielen gehört, dass Sie

- exemplarisch einige Standardtechniken für bekannte Problemklassen kennen und anwenden lernen,
- beim Anwenden dieser Techniken planvolles, zielgerichtetes Vorgehen üben,
- über das Migrieren einer Problemlösung von zwei Implementationssprachen in eine dritte Sprache Erfahrung sammeln.

Zuerst studieren Sie Grundlagen der Zerteilermethode rekursiver Abstieg (s. PÜ1.3), der Übersetzerarchitektur, und der objektorientierten Entwurfsmuster.

Funktion

Mit diesem informatischen Basiswissen entwickeln Sie inkrementell Varianten von Interpretierern, die arithmetische und relationale Ausdrücke auf verschiedene Arten interpretieren können; die funktionalen Anforderungen sind in PÜ1.4 und PÜ1.5 detailliert beschrieben. Das Programm konstruieren Sie so, dass

Softwarequalität

- Ein-/Ausgabe und funktionale Aspekte voneinander getrennt sind,
- es leicht änderbar und erweiterbar, d.h. leicht an zusätzliche Anforderungen anpassbar ist, und
- für den Benutzer zwar schlicht, aber robust und gebrauchstauglich ist.

Dies alles erreichen Sie nicht mit Bastelei, sondern indem Sie systematisch vorgehen. Dazu stellen Sie einen eigenen Plan mit Teilaufgaben auf, der z.B. folgende Schritte in einer geeigneten Reihenfolge umfasst:

Vorgehensweise

- Standardmethode rekursiver Abstieg kennen lernen, recherchieren.
- Referenzlösungen studieren.
- Prozeduraler, strukturierter Entwurf im Kleinen: Zu EBNF-Regeln für Ausdrücke die Zerteilerprozeduren mit rekursivem Abstieg in Pseudocode entwerfen.
- Fehlerbehandlung in Zerteiler einbauen.
- Zerteiler zu Interpretierer erweitern.
- Modularer oder objektorientierter Entwurf im Großen: Welche Module oder Klassen gibt es? Funktion und Ein-/Ausgabe trennen. Saubere Schnittstellen definieren.
- Pläne und Entwürfe dem Betreuer zeigen und abnehmen lassen.
- Implementationssprache kennen lernen.
- Pseudocode in Implementationssprache umformen.
- Entwicklungsumgebung kennen lernen, Probeprojekt zum Laufen bringen.
- Programm gut dokumentieren.
- Programm gut testen.
- Ausbaustufen planen.

Selbstverständlich gelten die [MB.6 Minimalanforderungen für das erste Gespräch eines Projekts](#) und die [MB.7 Minimalanforderungen für das letzte Gespräch eines Projekts](#) in [MB Merkblatt](#) auch hier.

PÜ1.3

Rekursiver Abstieg

Motivation

In Abschlussarbeiten mit Softwarethemen sind oft Übersetzer für kleine Sprachen zu entwickeln. Daher sollten mki-Studenten zumindest die einfache **Top-Down-Zerteilung** (*top-down parsing*) mit **rekursivem Abstieg** (*recursive descent*) kennen. Die Methode wurde Anfang der 1960er Jahre von mehreren Softwareentwicklern unabhängig voneinander beim Bau von Kompilern für *Algol 60* entwickelt. Sie ist knapp in der Sekundärquelle Hug: *Module, Klassen, Verträge*, S. 270–274 beschrieben, ausführlich in Übersetzerbau-Literatur, z.B. in der Primärquelle Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley (1996), ganz knapp und für PÜ1.4 vereinfacht hier:

Zerteiler mit
rekursivem Abstieg

Gegeben sind die EBNF-Regeln einer Sprache L , **gesucht** ist ein Zerteiler (*parser*), ggf. ein Interpretierer (*interpreter*) für L . **Lösung:** Definiere zu jedem Nichtterminal X eine **Zerteilerprozedur** $\text{Read}X$, die X analysiert. Implementiere jede Prozedur nach diesem Schema:

- next Prozedur, die versucht, das nächste Zeichen von der Eingabe zu lesen und es ggf. symbol zuweist
- symbol Variable, die das zuletzt gelesene Zeichen enthält, falls die Eingabe nicht erschöpft ist
- first (E) Menge der Startsymbole des EBNF-Ausdrucks E
- error Prozedur, die den Ablauf mit einer Syntaxfehlermeldung abbricht

Tabelle PÜ1.1

Umformungsregeln von
der EBNF zum
Zerteiler-Algorithmus

EBNF-Ausdruck E	Zerteiler-Anweisung $S(E)$
"x"	IF symbol = "x" THEN next ELSE error END
Nonterminal	ReadNonterminal
(Expression)	$S(\text{Expression})$
[Expression]	IF symbol IN first (Expression) THEN $S(\text{Expression})$ END
{ Expression }	WHILE symbol IN first (Expression) DO $S(\text{Expression})$ END
Factor ₀ Factor ₁ .. Factor _n	$S(\text{Factor}_0); S(\text{Factor}_1); \dots S(\text{Factor}_n)$
Term ₀ Term ₁ .. Term _n	IF symbol IN first (Term ₀) THEN $S(\text{Term}_0)$ ELSIF symbol IN first (Term ₁) THEN $S(\text{Term}_1)$.. ELSIF symbol IN first (Term _n) THEN $S(\text{Term}_n)$ ELSE error END

Einfacher Lexer

next und symbol bilden eine einfache Lexer-Schnittstelle nach dem Aktion-Abfrage-Schema. Ein Lexer (*scanner*) analysiert normalerweise lexikalische Elemente, die Symbole heißen. Die Aufgabe in PÜ1.4 ist so vereinfacht, dass sie nicht zwischen lexikalischer und syntaktischer Ebene unterscheidet und der Zerteiler lexikalische *und* syntaktische Elemente analysiert, sodass der Lexer nur Zeichen lesen muss. Bei einer allgemeineren, hier nicht nötigen Lexer-Schnittstelle zeigt die Abfrage symbol nicht das letzte Zeichen an, sondern das zuletzt gelesene lexikalische Element in codierter Form.

Der Lexer erhält den Eingabetext (hier den Ausdruck) als Zeichenkette. Er liest den Eingabetext nur einmal sequenziell zeichenweise von links nach rechts.

Fehlerbehandlung
Zerteiler

Nachdem der Zerteiler syntaktisch korrekte Eingaben korrekt behandelt, soll er bei fehlerhafter (auch fehlender) Eingabe an der frühest möglichen Stelle eine für den Benutzer verständliche Fehlermeldung liefern und die Analyse beenden. Das Entwurfsschema sieht die Prozedur error als Fehlerausgang vor. Für fehlende Eingabe wird die Lexer-Schnittstelle um eine boolesche Abfrage `endOfInput` erweitert. Der Zerteiler darf symbol nur lesen, wenn `NOT endOfInput` gilt, d.h. `NOT endOfInput` wird Vorbedingung für Lesezugriffe auf symbol.

Interpretierer	Die Implementation einer Zerteilerprozedur bildet das algorithmische Gerüst, in das Anweisungen zur Interpretation des analysierten Nichtterminals einzufügen sind. Jede Zerteilerprozedur erhält einen Ausgabeparameter für das Interpretationsergebnis des von ihr zerteilten Nichtterminals (ReadNonTerminal (OUT nonTerminal : REAL)).
Lösungsvariante	Statt Prozeduren können Funktionen die Interpretationsergebnisse als Funktionsergebnisse zurückgeben. Zerteilerfunktionen heißen wie ihre Nichtterminale, d.h. das Read im Namen entfällt, weil ein Funktionsname keine Tätigkeit, sondern das Ergebnis bezeichnen soll (NonTerminal () : REAL). Die Lösung mit Funktionen ist je nach Standpunkt elegant oder wegen der unvermeidbaren Nebeneffekte scheußlich.
Fehlerbehandlung Interpretierer	Nachdem der Interpretierer syntaktisch und semantisch korrekte Eingaben korrekt behandelt, soll er bei fehlerhafter Eingabe frühest möglich eine verständliche Fehlermeldung liefern und die Interpretation beenden. Dazu wird einfach die Fehlerbehandlung des Zerteilers erweitert.

PÜ1.4

Ausdrücke berechnen – Interpretierer mit rekursivem Abstieg

Problem

Ein Programm soll arithmetische Ausdrücke der Art

$$-(a + b) * c - d / a$$

berechnen. Die Ausdrücke können bis zu $n = 4$ verschiedene reelle Variablen und die vier Grundrechenarten enthalten. Die Werte der Variablen lassen sich getrennt vom Ausdruck eingeben. Die Syntax der arithmetischen Ausdrücke ist durch diese EBNF-Regeln definiert:

```

Expression = [ AddOperator ] Term { AddOperator Term }.
AddOperator = "+" | "-".
Term = Factor { MulOperator Factor }.
MulOperator = "*" | "/".
Factor = Ident | "(" Expression ")".
Ident = "a" | "b" | "c" | "d".

```

Entwurf in Pseudocode	(1) Entwerfen Sie einen Zerteiler für diese arithmetischen Ausdrücke nach der Methode des rekursiven Abstiegs und passen Sie ihn an interpretative Verarbeitung an! Schon im Entwurf trennen Sie Ein-/Ausgabe und funktionale Aspekte . Lösungen, bei denen Funktion und E/A ineinander verschmiert sind, werden abgelehnt.
Studium der CP-Referenzlösungen	(2) Studieren Sie die in <i>Component Pascal</i> formulierten Aufgabenlösungen mit den Dialogmodulen I2ArithmeticExpressionsDB[1 2] und den Funktionsmodulen I2ArithmeticExpressionsRD*! Die Ein-/Ausgabe erfolgt hier mit einem Dialog, wobei der Ausdruck und die Werte der Variablen in getrennten Feldern eingegeben werden und ein Melder (<i>notifier</i>) zu den Eingabefeldern den Interpretierer aufruft. Der Interpretierer stellt den funktionalen Teil dar, bestehend aus bei ◆ I2ArithmeticExpressionsRD1 nur einer Prozedur mit Ein- und Ausgabeparametern für rein prozedurale Kommunikation, ◆ I2ArithmeticExpressionsRD2* einer Prozedur mit Eingabeparametern und zwei Abfragen für die Kommunikation nach dem Aktion-Abfrage-Schema.
Studium der Eiffel-Referenzlösung	(3) Studieren Sie die in <i>Eiffel</i> formulierte Aufgabenlösung mit der Ein-/Ausgabeklasse <i>EXPRESSION_CALCULATOR</i> und den Funktionsklassen <i>EXPRESSION_INTERPRETER</i> *! Die Eingabe erfolgt hier von der Kommandozeile, der Ausdruck und die Werte der Variablen werden nacheinander hingeschrieben. Die Ausgabe geht auf die Konsole. Der Interpretierer stellt den funktionalen Teil dar, bestehend aus einer Prozedur mit Eingabeparametern und zwei Abfragen für die Kommunikation nach dem Aktion-Abfrage-Schema. Vererbung wird eingesetzt, um zwischen der Lösung des einfachen Problems und der Erweiterung (9) zu differenzieren und gemeinsame Teile heraus zu faktorisieren.

- | | |
|---|---|
| Funktionaler Teil | <p>(4) Entwickeln Sie eine Aufgabenlösung! Beachten Sie für den funktionalen Teil:</p> <ul style="list-style-type: none"> ◆ Kennt die Implementationssprache keine geschachtelten Prozeduren, so entschachteln Sie die Struktur (siehe Skript/Krimskrams)! ◆ Kennt die Sprache keine Ausgabeparameter, so arbeiten Sie mit geeigneten Ersatzmechanismen (siehe Skript/Krimskrams)! <p>Da <i>Eiffel</i> weder geschachtelte Routinen noch Ausgabeparameter kennt, nutzt die <i>Eiffel</i>-Lösung ungeschachtelte Zerteilerfunktionen.</p> |
| Ein-/Ausgabeteil | <p>(5) Den Ein-/Ausgabeteil realisieren Sie mit einer kommandozeilenorientierten Ein-/Ausgabe etwa dieser Form:</p> <pre>>calculate "-(a+b)*c-d/a" 1 2 3 4 -13 ></pre> <p>Nach dem Kommandonamen folgen der Ausdruck ohne Leerzeichen und die Werte der Variablen; das Leerzeichen ist Trennzeichen. Es empfiehlt sich, den Ausdruck mit " " zu klammern, weil manche Kommandosprachen manche Operatorzeichen (z.B. "!", "<", ">") als Steuerzeichen verwenden.</p> <p>Die Ein-/Ausgabe gestalten Sie robust mit gebrauchstauglichen Meldungen bei Fehleingaben. Dazu gehört, dass der Benutzer bei einem syntaktisch fehlerhaften Ausdruck Informationen zur Fehlerstelle erhält. Eine einfache Lösung dazu gibt das akzeptierte Anfangsstück der Eingabe zur Fehlermeldung aus:</p> <pre>>calculate "-(a+b)*\$hallo!-?-bug=Fehlerhaft" 2 3 4 5 Parsed input = -(a+b)* = Input error: number or identifier or '(' expected! ></pre> <p>Lösungen ohne angemessene Fehlerbehandlung werden abgelehnt. Kein Verbrechen, aber Unsinn ist eine interaktive Anwendung mit Eingabe von der Tastatur.</p> |
| Schnittstelle des funktionalen Teils zum E/A-Teil | <p>(6) Offenbar handelt es sich um ein zum EVA-Prinzip passendes Problem:</p> <p style="padding-left: 40px;">Eingabe – Verarbeitung – Ausgabe.</p> <p>Der Benutzer gibt Eingabedaten ein (hier fünf: einen Ausdruck und vier Zahlen), das Programm verarbeitet sie zu Ausgabedaten (hier zwei: die berechnete Zahl oder eine Fehlermeldung, und das akzeptierte Anfangsstück des Ausdrucks), der Benutzer liest die Ausgabedaten ab. Im Kern ist die Lösung prozedural, aber zwecks Trennung von Ein-/Ausgabe und Funktion formulieren Sie sie modular oder objektorientiert: Die Eingabe und die Ausgabe fassen Sie zu einem Ein-/Ausgabeteil zusammen, die Verarbeitung stecken Sie in einen funktionalen Teil (oder mehrere Teile). Die Teile können Module oder Klassen sein.</p> <p>Wie ist die Schnittstelle des funktionalen Teils zu gestalten? Möglichst einfach! Sie soll mit möglichst wenigen möglichst leicht verständlichen und einfach zu benutzenden Diensten dem E/A-Teil alles Wichtige bieten, und sie muss gut dokumentiert sein, denn nur so können andere Programmierer sie verwenden. Der funktionale Teil darf nicht die beiden Ergebnisse (Zahl oder Fehlermeldung, akzeptiertes Anfangsstück) zu einer Zeichenkette zusammenfügen, denn das würde dem E/A-Teil (z.B. ein GUI-Teil) die Freiheit rauben, diese Ergebnisse unterschiedlich auszugeben.</p> <ul style="list-style-type: none"> ◆ Parameterlose Dienste sind einfacher als Dienste mit Parametern. ◆ Zwei parameterlose Abfragen einfachen Typs ($q_1, q_2 : T$) sind einfacher als eine, die statisch eine offene Reihung mit dynamisch stets zwei Elementen zurückgibt ($q : \text{ARRAY}[T]$ post result.index_set = {1, 2} end). ◆ Eine parametrisierte Aktion und zwei schlichte Abfragen ($a(p : P); q_1, q_2 : T$) sind einfacher als eine parametrisierte Abfrage (also eine Funktion) der Form $q(p : P) : \text{ARRAY}[T]$ post result.index_set = {1, 2} end. |

- ◆ Nebeneffektfreie Funktionen sind einfacher als nebeneffektbehaftete Funktionen.
- ◆ Dienste, die keine Ausnahmen werfen, sind einfacher als Dienste, die Ausnahmen werfen.

Deshalb: **Die Schnittstelle des funktionalen Teils bietet keine Funktionen mit Nebeneffekten und keine Dienste, die Ausnahmen werfen.** Ausnahmen dürfen nicht aus dem funktionalen Teil heraus fliegen, sondern müssen innerhalb des funktionalen Teils gefangen und in eine Abfrage umgesetzt werden.

Für C*-Implementationen: Ist der funktionale Teil als Klasse mit nichtstatischen Daten realisiert, braucht diese einen Konstruktor? Konstruktoren sollen die Daten von Objekten initialisieren, die dann von mehreren Diensten bearbeitet werden. Wenn aber wie hier ein EVA-Problem vorliegt, genügt zur Verarbeitung ein einziger Dienst, der die Daten des vorigen Aufrufs verwirft und neue Eingabedaten verarbeitet. Das Objekt braucht keinen Zustand, der Aufrufe des Verarbeitungsdienstes überdauern muss. Der Einsatz eines Konstruktors zur Übergabe der Eingabedaten hat hier mindestens zwei Nachteile:

- ⊗ Für jede neuen Eingabedaten muss ein neues Objekt erzeugt werden, weil Konstruktoren nur einmal beim Objekterzeugen aufrufbar sind. Jedes Objekt kann nur *eine* Verarbeitung durchführen und verschleißt dabei zu Müll: Plastikbecher ex und hopp? Nein, Ressourcen schonen und wiederverwenden!
- ⊗ Der E/A-Teil muss zuerst den Konstruktor, dann den Verarbeitungsdienst aufrufen. Zusammengehörige Arbeit ist verteilt. Die Schnittstelle ist komplizierter als nötig.

Deshalb: **Konstruktoren nur sinnvoll einsetzen!** Auf Unnötiges verzichten.

Fehlerbehandlung

- (7) Die unhandliche prozedurale Fehlerbehandlung der *Component-Pascal*-Lösung I2ArithmeticExpressionsRD1 können Sie durch Anwenden des Ausnahmebehandlungskonstrukts der Implementationssprache ersetzen. Als Ausgangslösungen dafür eignen sich I2ArithmeticExpressionsRD2* besser. Die Fehlerbehandlung müssen Sie an das Ausnahmebehandlungskonzept der Implementationssprache anpassen. Das Beispielprogramm *EXCEPTION_HANDLING_EXAMPLE.e* zeigt Ihnen, wie Ausnahmebehandlung in *Eiffel* eingesetzt wird. Welche Elemente der Ausnahmebehandlungskonstrukte von *Eiffel* und *C#* sich entsprechen, steht hier:

Tabelle PÜ1.2
Ausnahmebehandlung
in Eiffel und C#

<i>Eiffel</i>	<i>C#</i>
<i>raise</i> ("fatal error!")	<code>throw new System.Exception ("fatal error!");</code>
Routinenrumpf	try-Block
rescue -Abschnitt	<code>catch (System.Exception e)</code> -Block
<i>is_developer_exception</i> differenziert nach Ausnahmeart.	Nicht notwendig, aber durch Exception-Klassen bei catch modellierbar.
<i>developer_exception_name</i> bezeichnet, was <i>raise</i> als Parameter erhalten hat.	<code>e.Message</code> bezeichnet, was <code>throw new ...</code> als Parameter erhalten hat.
retry führt zum wiederholten Aufruf der Routine.	Gibt es nicht.
Die Variable <i>done</i> : <i>BOOLEAN</i> steuert den Ablauf beim retry -Aufruf.	Entfällt.
inherit <i>EXCEPTIONS</i>	Entfällt.

Erweiterbarkeit

- (8) Überlegen Sie eine Lösung zu diesem Problem: Sind bisher *n* verschiedene Variablen in Ausdrücken erlaubt, so sind jetzt *n+1* Variablen erlaubt. An wie vielen und welchen Stellen müssen Sie 1. die Spezifikation, 2. die Implementation

	ändern? Wie viele und welche Änderungsstellen sind trotz „weiser Voraussicht“ unvermeidlich?
Erweiterungen	(9) Erweitern Sie wie in <i>Informatik 2 Praktikum, Übungsblatt 4, Sommersemester 2009</i> die Funktionalität, indem Sie
Zahlen	(a) in den arithmetischen Ausdrücken auch ganz- und gleitpunktzahlige literale Konstanten zulassen, z.B. $(a + b) * c - 1.23E-4 / 5 * (d - 6.7)$
Relationen	(b) auch relationale Ausdrücke zulassen, z.B. $(a + b) * c < d - a.$
	Ergänzen Sie dazu jeweils die Syntax der Ausdrücke durch EBNF-Regeln, den Entwurf und die Implementation des Interpretierers! Der Interpretierer muss die Zahlen aus den Zeichen selbst konstruieren. Hier eine Bibliotheksroutine zum Konvertieren von Zeichenketten in Zahlen zu benutzen ist verboten , da es dem didaktischen Zweck der Aufgabe (s. Überschrift) widerspräche. Die <i>Eiffel</i> -Lösung nutzt für die Erweiterungen die objektorientierten Konzepte Vererbung und Polymorphie und das Entwurfsmuster Schablonenmethode.
Testen	(10) Sammeln Sie Testfälle, die möglichst viele Fehler des Prüflings und Eingabefehler erkennen! Stellen Sie die Testfälle in einer Batch-Datei zusammen oder programmieren Sie eine Testklasse zum funktionalen Teil Ihrer Lösung mit den gesammelten Testfällen! Ungetestete Lösungen werden abgelehnt.

PÜ1.5

Ausdrücke bearbeiten – Syntaxbaum, Entwurfsmuster Besucher

Nun soll ein Interpretierer Ausdrücke nicht nur berechnen, sondern auch anders bearbeiten können, hier z.B. von der Infixnotation mit Klammern in eine klammerlose Präfixnotation umformen, etwa so:

```
>transform "(a+b)*c-d/a"
-*+abc/da
>
```

Da Operatoren, die sowohl ein- als auch zweistellig auftreten ("+", "-"), bei Präfixnotation zu Mehrdeutigkeiten führen, lassen Sie die einstelligen Operatoren entweder weg, oder Sie unterscheiden sie notationell von den zweistelligen Operatoren.

Die Zerteilung eines Ausdrucks verläuft wie zu seiner Berechnung, nur die Interpretation ist anders. Daher wollen wir den Zerteiler wiederverwenden. Die einfache Berechnungs-Interpretation von PÜ1.4 ist aber in die Zerteilerprozeduren „hineingeschmiert“, wodurch dieser Zerteiler nicht wiederverwendbar ist. (Fehlende Trennung der Aspekte „Zerteilung“ und „Interpretation“ hat zu mangelnder Softwarequalität geführt.) Statt der prozeduralen Struktur brauchen wir daher eine modulare Struktur mit einem wiederverwendbaren Zerteiler und austauschbaren Interpretiererteilen, die wir **Codeerzeuger** nennen. Es folgen Entwurfsaspekte.

PÜ1.5.1

Architektur

(1) Der Interpretierer gliedert sich in eine **Zwei-Phasen-Architektur**:

Bild PÜ1.1

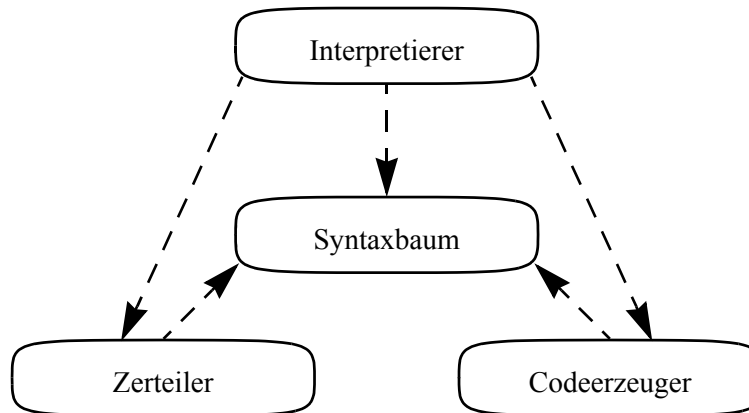
Modulare Architektur

Frontend

Eingabe, Fehlerausgabe

Backend

Ausgabe



- ◆ Ein **Frontend** übernimmt die **Zerteilungsphase**, indem es den Quelltext (hier den Ausdruck) liest und zerteilt und dabei einen Syntaxbaum generiert.
- ◆ Ein **Backend** übernimmt die **Codeerzeugungsphase**, indem es aus dem Syntaxbaum den Zieltext generiert („Code erzeugt“).

Der Syntaxbaum dient der Kommunikation zwischen Zerteiler und Codeerzeuger. Das vom Zerteilen des Quelltexts getrennte Erzeugen des Zieltexts ermöglicht es, Codeerzeuger auszutauschen und den Interpretierer um beliebige Codeerzeuger zu erweitern.

(2) **Objektorientierter Entwurf**: Der Interpretierer realisiert den Syntaxbaum als dynamische Objektstruktur mit bekannten Entwurfsmustern.

PÜ1.5.2

Entwurfsmuster

Tabelle PÜ1.3 zeigt im Überblick, wo sich welche Entwurfsmuster aus dem Standardwerk Gamma, Helm, Johnson, Vlissides: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley (1995) verwenden lassen. Informieren Sie sich dazu selbstständig!

Tabelle PÜ1.3

Entwurfsmuster und ihre Verwendung

Entwurfsmuster	Verwendung bei
Fabrikmethoden	ggf. Knoten des Syntaxbaums, ggf. Codeerzeuger
Singleton	Zerteiler, Besucherobjekte, Interpretierer
Kompositum	Knoten des Syntaxbaums
Fassade	Interpretierer
Besucher	Codeerzeuger: Baumausgabe, Berechnung, Präfixumformung

Das Entwurfsmuster **Besucher** (*visitor*) ist hier das methodisch wichtigste. Als Besucher sind zu einer Codeerzeuger-Schnittstellenklasse exemplarisch drei Implementationsklassen zu definieren:

- (1) eine für die Ausgabe des Syntaxbaums zu Test- und Diagnosezwecken,
- (2) eine für die Berechnung von Ausdrücken, und
- (3) eine für die Umformung von Infixausdrücken in Präfixnotation.

Der Interpretierer mit der Berechnungsvariante soll dasselbe leisten wie der einfache Interpretierer von PÜ1.4 (von einstelligen Operatoren abgesehen).

PÜ1.5.3

Syntaxbaum

Aufgabe des Syntaxbaums ist, die syntaktische Struktur des Quelltexts zu speichern. Diese Information liefert der Zerteiler. Den Syntaxbaum entwerfen Sie als Modul `SyntaxTree`, das eine Klassenstruktur mit Vererbung, Komposition und Aggregation enthält.

- (1) Der Syntaxbaum entsteht beim Lesen des Quelltexts. Der Zerteiler steigt top-down rekursiv zur untersten syntaktischen Einheit ab. Die Knoten des Syntaxbaums zu den untersten syntaktischen Einheiten werden zuerst erzeugt und beim Wiederaufstieg zu den größeren Einheiten verknüpft. Der Syntaxbaum wird also von den Blättern aufsteigend zur Wurzel, also von unten nach oben aufgebaut. Das Verfahren ähnelt dem Interpretierer, der Ausdrücke von innen nach außen auswertet.
- (2) Die Knotenklassen des Syntaxbaums entsprechen den syntaktischen Einheiten. Die abstrakte Basisklasse aller Knoten heißt `Node`. Zu jedem substantziellen Nichtterminal `S` der EBNF-Definition gibt es eine direkte oder indirekte Erweiterungsklasse `S` von `Node`. Da ein Expression ein Term, dieser ein Factor sein kann, folgt beispielsweise die Erweiterungskette:

Factor erweitert Term erweitert Expression erweitert Node

- (3) Die Klassenstruktur enthält mehrfach das Entwurfsmuster **Kompositum**. Um die Bestandteile der Nichtterminal-Knotenklasse `S` zu bestimmen, sind in der EBNF-Regel zu `S` ggf. auftretende Wiederholungen durch Rekursion zu ersetzen. Beispielsweise führt ein linksassoziativer Operator zu Linksrekursion:

Term = Factor { MulOperator Factor }.

Term = [Term MulOperator] Factor.

Die linksrekursive Regel eignet sich freilich nicht für rekursiven Abstieg, aber im Syntaxbaum besteht ein Term-Knoten aus einem optionalen Term mit `MulOperator` und einem obligatorischen Factor. Vereinfachung: Auf Term-Knoten, die nur einen Factor haben, verzichten; dann hat jeder Term-Knoten einen weiteren Term, einen `MulOperator` und einen Factor.

- (4) Zu jeder konkreten Knotenklasse `S` einschließlich der Wurzelklasse `Expression` gibt es eine **Fabrikfunktion** `NewS(...): S`. Jede Fabrikfunktion hat Eingabeparameter, die den Bestandteilen des von ihr erzeugten Knotens entsprechen. (Im ersten Lösungsansatz genügt eine entsprechende Erzeugungsprozedur oder ein Konstruktor.)
- (5) Der Zerteiler, der die syntaktische Struktur des Quelltexts analysiert, ruft bei jeder erkannten Einheit die entsprechende Fabrikfunktion auf.
- (6) Das Modul `SyntaxTree` definiert eine abstrakte Klasse `Visitor` als Teil des Entwurfsmusters **Besucher**. Ihr Zweck ist, verschiedene Codeerzeuger als Erweiterungsklassen von `Visitor` definieren zu können. Für `Visitor` gilt:
 - ◆ Zu (fast) jedem Nichtterminal `S` definiert `Visitor` eine meist abstrakte Aktion `VisitS(s: S)`.
 - ◆ `Node` hat eine Aktion `Accept(v: Visitor)`.
 - ◆ Eine konkrete Erweiterungsklasse `S` von `Node` implementiert `Accept(v: Visitor)` in der Regel durch `v.VisitS(this)`.

PÜ1.5.4

Codeerzeuger

Jeder Codeerzeuger generiert aus dem Syntaxbaum seinen Zieltext. Codeerzeuger sind konkrete Besucherklassen nach dem Entwurfsmuster **Besucher**.

- (1) Für jeden Zieltext `L` gibt es eine konkrete Erweiterungsklasse `LCodeGenerator` von `SyntaxTree.Visitor`, die die geerbten Besuchsaktionen implementiert.

- (2) Der erste Zieltext zeigt den Inhalt und die Struktur des Syntaxbaums zu Test- und Diagnosezwecken etwa so:

```
>show "(a+b)*c-d/a"
-
  *
    +
      a
      b
    c
  /
    d
    a
>
```

- (3) Der zweite Zieltext sind die reellen Zahlen für die Ergebnisse der Berechnung der Ausdrücke.
- (4) Der dritte Zieltext gehört zu der durch EBNF-Regeln zu definierenden Sprache der Präfixausdrücke, in die die Infixausdrücke umgeformt werden.
- (5) Codeerzeuger sind **Singletons**. Erzeugt werden sie durch eine **Fabrikfunktion** `NewCodeGenerator(...): SyntaxTree.Visitor`, die garantiert, dass von jeder Codeerzeugerklass nur ein Objekt existiert.

PÜ1.5.5

Interpretierer

Aufgabe der Komponente Interpreter ist die Steuerung der Interpretation, hier der zwei Phasen Zerteilung und Codeerzeugung.

- (1) Der Interpretierer ist ein **Singleton**.
- (2) Interpreter bietet dieselbe Benutzungsoberfläche wie der einfache Interpretierer von PÜ1.4, aber mit der zusätzlichen Möglichkeit, zwischen Ausgabe des Syntaxbaums, Berechnung und Präfixumformung zu wählen.
- (3) Interpreter entspricht dem Entwurfsmuster **Fassade**: Es bietet eine Schnittstelle, hinter der alle anderen Komponenten des Interpretierers – seine Module, Klassen und Strukturen – verborgen sind.
- (4) Interpreter arbeitet so:
 - ◆ Rufe den Zerteiler auf, erhalte einen Syntaxbaum, `syntaxTree` genannt.
 - ◆ Rufe eine Fabrikfunktion auf, die ein Singleton des gewählten Codeerzeugers liefert.
 - ◆ Rufe `syntaxTree.Accept`, übergebe dabei das Codeerzeuger-Objekt als Besucher.

PÜ1.5.6

Beispiel

Die genannten Methoden und Entwurfsmuster verwendet der *Cleo*-Compiler in *Component Pascal* mit *BlackBox* im Informatik-Hug-Subsystem Cleo, erreichbar unter

http://informatik.karlheinz-hug.de/Lehrmaterial/Informatik_1+2/BlackBox-Informatik-Hug-09-10-WS/Cleo

ftp://studinf.reutlingen-university.de/MKI/Hug/Lehrmaterial/Informatik_1+2/BlackBox-Informatik-Hug-09-10-WS/Cleo

Ein beschreibendes Dokument ist zu finden unter

<http://informatik.karlheinz-hug.de/lars/01/Projektbericht.pdf>

PÜ1.6**Minimalanforderungen für das erste Projektgespräch**

Das in *MB Merkblatt, MB.6 Minimalanforderungen für das erste Gespräch eines Projekts* Verlangte ist hier konkretisiert und Grundlage für das Testat. Die Fragen verweisen auf Beachtenswertes.

- (1) Liste offener Fragen zum Problem
 - PÜ1.3: Zu rekursivem Abstieg selbst recherchiert
 - PÜ1.4(2): *CP*-Lösungen studiert, Strukturdiagramm dazu gezeichnet
 - PÜ1.4(3): *Eiffel*-Lösung studiert, Strukturdiagramm dazu gezeichnet
- (2) Entwurfsskizzen:
 - (a) EBNF-Regeln zu Erweiterungen definiert:
 - PÜ1.4(9)(a): literale Zahlen
 - PÜ1.4(9)(b): Relationen
 - (b) Zu EBNF-Regeln Zerteilerprozeduren in Pseudocode konstruiert:
 - PÜ1.4(1): arithmetische Ausdrücke
 - PÜ1.4(9)(a): literale Zahlen
 - PÜ1.4(9)(b): Relationen
 - (c) In *Eiffel*-Lösung Zerteilerfunktionen farbig markiert:
 - Zerteiler gelb
 - Fehlerbehandlung rot
 - Interpretierer grün
 - (d) PÜ1.4(1): Fehlerbehandlung in Zerteiler eingebaut
 - (e) PÜ1.4(1): Zerteiler zu Interpretierer erweitert
 - (f) PÜ1.4(1): Fehlerbehandlung des Interpretierers eingebaut
 - (g) PÜ1.4(6): Schnittstelle des funktionalen Teils entworfen und spezifiziert:
 - nebeneffektfrei, ausnahmefrei, konstruktorfrei, getrennte Ergebnisse: Zahl oder Fehlermeldung, akzeptierter Anfang der Eingabe
 - Schema? prozedural mit Ausgabeparameter, funktional, Aktion-Abfrage
- (3) Liste benötigter Sprachkonstrukte
 - Wie die Sprache kennen gelernt?
 - Sekundärliteratur, Google, Skript, Sprachbeschreibung
 - PÜ1.2: Schnupperprojekt zum Laufen gebracht?
 - nein, ja
 - Wie viele kleine Beispiele zum Laufen gebracht?
 - 0 1 2 3 4
- (4) Vorgehensplan
 - Wie ist die Vorgehensweise?
 - unselbstständig, plagiiierend, selbstständig
 - unsystematisch, systematisch
 - nichtinkrementell, inkrementell

Prozeduraler,
strukturierter Entwurf
im Kleinen

Modularer,
objektorientierter
Entwurf im Großen

PÜ1.7**Minimalanforderungen für das letzte Projektgespräch**

Die minimale Bearbeitungsstufe ist PÜ1.4 komplett. Das in *MB Merkblatt, MB.7 Minimalanforderungen für das letzte Gespräch eines Projekts* Verlangte ist hier konkretisiert und Grundlage für das Testat:

- (1) Korrektheit
Testfälle in Batch-Datei zusammengestellt
Alle korrekten Eingaben korrekt interpretiert
- (2) Benutzbarkeit:
Robustheit: Alle unbrauchbaren Eingaben korrekt abgelehnt
Akzeptierter Anfang einer Eingabe Teil einer Fehlermeldung
Fehlermeldung präzise und freundlich
- (3) Testbarkeit
- (4) Zerlegung in mindestens zwei Module: E/A, Interpretierer
- (5) Trennung von E/A und Funktion
- (6) Datenabstraktion
- (7) Nebeneffektfreie Schnittstellen
- (8) Ausnahmefreie Schnittstellen
- (9) Dokumentationssprache Englisch
- (10) Kopfkommentare in Quelldateien
- (11) Sinnvolle Namen
- (12) Dokumentierte Schnittstellen
EBNF-Regel bei jeder Zerteilerfunktion
- (13) Minimaler Funktionsumfang:
 - (a) PÜ1.4(1): arithmetische Ausdrücke
 - (b) PÜ1.4(9)(a): literale Zahlen
Zahlen aus Zeichen konstruiert (ohne Bibliotheksfunktion)
 - (c) PÜ1.4(9)(b): Relationen
 - (d) PÜ1.4(8): Erweiterung auf n+1 Variablen überlegt

PÜ1.8

Datum,
Unterschrift des
Betreuers

Testate

Name des Studenten:

Abgenommene Leistungen

Offene Aufgaben