

**Konrad Zuses Plankalkül – Seine Genese und eine
moderne Implementierung**

Raúl Rojas, Cüneyt Göktekin, Gerald Friedland,
Mike Krüger, Ludmila Scharf

Freie Universität Berlin
Institut für Informatik
Takustr 9
14195 Berlin

Denis Kuniß, Olaf Langmack

Transformal
Gitschiner Straße 91, 2. Hof
D-10969 Berlin
www.transformal.com

Korrespondenz und Korrekturen an:
Prof. Dr. Raul Rojas
Freie Universität Berlin
FB Mathematik und Informatik
Takustr. 9
14195 Berlin

Zusammenfassung. Konrad Zuse hat in den Jahren 1942 bis 1945 verschiedene Notizen über einen „Kalkül“ angelegt, die letztendlich zum Entwurf einer Programmiersprache, dem „Plankalkül“, geführt haben. Plankalkül war die erste höhere Programmiersprache der Welt, wurde aber bis vor kurzem nicht implementiert.

In diesem Beitrag wird die Genese des Plankalküls rekapituliert. Für Zuse war die Prädikatenlogik zunächst eine reine Beschreibungssprache. Schrittweise hat er jedoch eine mögliche Computerimplementierung konzipiert. Dafür wurde die ursprüngliche Notation mit imperativen Konstrukten ergänzt. Zuse hat dann zwischen der „impliziten“ (prädikatenlogischen) und „expliziten“ (imperativen) Form eines Programms unterschieden.

Der Beitrag beschreibt auch die erste von uns erstellte Implementierung des Plankalküls. Mehr als 55 Jahre nach seiner Konzeption sind die ersten Plankalkül-Programme im Februar 2000 mit einem modernen Rechner ausgeführt worden.

Schlüsselworte: Konrad Zuse, Plankalkül, Geschichte des Computers.

Abstract. Konrad Zuse wrote down from 1942 to 1945 several short pieces in his notebooks about a "calculus" which he would gradually transform into a fully fledged programming language, the "Plankalkül" (calculus of programs). Plankalkül was the first high-level programming language in the world, but remained unimplemented for many decades.

In this article we review the genesis of Plankalkül. Zuse first used the predicate calculus as a notational instrument, but after some time he conceived a computer implementation. He supplemented the original notation with imperative elements so that programs could be written in "implicit" (predicate calculus form) or in "explicit" (imperative) form.

This article describes also our implementation of Plankalkül, the first ever made. Plankalkül programs first ran in a modern computer in February 2000, i.e. 55 years after the language was defined by Konrad Zuse,

Keywords: Konrad Zuse, Plankalkül, History of Computing.

Ich habe daher die Hoffnung, daß mein Plankalkül nach einem zwölfjährigen Dornröschenschlaf doch noch zum Leben erweckt werden wird ... es sollte mich freuen, wenn sich das eventuell in Zusammenarbeit mit Hochschulinstituten ermöglichen ließe.

Konrad Zuse, 1957

1 Die Logistik – 1938

Die Geschichte von Konrad Zuses Rechenmaschinen ist von verschiedenen Autoren ausführlich erforscht und dargestellt worden [7,8]. Zuses Manuskript mit dem Entwurf einer Programmiersprache, dem Plankalkül (PK), ist in den siebziger Jahren von der GMD veröffentlicht worden [12], hat aber nicht zu vergleichbar vielen Untersuchungen wie seine Rechenmaschinen geführt. Nur Friedrich L. Bauer, Hans Wössner [2] und Wolfgang Giloi [3] haben detailliertere Artikel über den Plankalkül geschrieben. J. Hohmann hat in seiner 1979 veröffentlichten Dissertation den Plankalkül kritisch unter die Lupe genommen und mit anderen imperativen Programmiersprachen (ALGOL, PL1, usw.) verglichen [6].

Wir wissen, daß Konrad Zuse in Bezug auf Rechenautomaten weitgehend ein Autodidakt war [13]. Den logischen Entwurf seiner ersten Maschine, die Z1, hat er selbständig und ohne besondere Kenntnisse der damals bereits vorhandenen mechanischen Rechenautomaten erstellt. Für die Beschreibung der Schaltungen hat Zuse eine eigene graphische Darstellung erfunden und die logische Beschreibung hat er "Bedingungskombinatorik" genannt. Erst später hat er entdeckt, daß diese nur ein anderer Name für die Aussagenlogik war. Diese Erkenntnis hat ihn zum Studium der Logik getrieben, um seinen Überlegungen eine strengere mathematische Form zu geben.

Im Mai 1939 schrieb Zuse in seinem Notizbuch:

"Seit etwa einem halben Jahr allmähliches Einführen in die formale Logik. Viele meiner früheren Gedanken habe ich dort wieder gefunden. (Bedingungskombinatorik = Aussagenlogik; Lehre von den Intervallen = Gebietenkalkül). Ich plane jetzt die Aufsetzung des 'Plankalküls'. Hierzu sind eine Reihe von Begriffen zu klären."

Dies ist die erste Erwähnung des Plankalküls in Zuses Unterlagen. Die Z1 ist zu diesem Zeitpunkt vollendet, obwohl wie Zuse an derselbe Stelle wortkarg schreibt, "Rechenwerk fertig, aber funktioniert schlecht." Das Zitat oben zeigt, daß Zuse 1939 bereits erkannt hatte, daß es möglich wäre, ein "Kalkül" für Programme zu schaffen, ähnlich wie aus der Prädikatenlogik ein Klassenkalkül (für Mengen) ableitbar ist. Besonders interessant ist, daß Zuse bereits Schach als den "Benchmark" identifiziert hatte mit dem die Möglichkeiten des Plankalküls gemessen werden könnten. Später wird Zuse das erste Schachprogramm der Welt schreiben (in den

Jahren 1942 bis 1945), obwohl in der Literatur das von Claude Shannon 1950 geschriebene Schachprogramm als weltweit Erstes angeführt wird. Auch Alan Turing hat sich mit Schach beschäftigt, aber erst 1951 einen Artikel über seine Überlegungen zu diesem Thema geschrieben.

Es war also für Zuse zu diesem frühen Zeitpunkt klar, daß die allgemeinste Form des Rechnens von der Form " $F(V) \Rightarrow R$ " ist, d.h. die Auswertung einer beliebigen Funktion F eines variablen Argumentes V , um ein Resultat R zu produzieren. Parallel zur Arbeit an seinen Rechenmaschinen, beschloß Zuse eine Dissertation über die Mechanisierung des Rechners zu schreiben. Der Entwurf dafür blieb aber bis zum Ende des Krieges unvollständig und erst nach der Flucht aus Berlin und in der Abgeschiedenheit des bayerischen Dorfes Hinterstein fand Zuse die Zeit, um die endgültige Beschreibung des Plankalküls zu schreiben. Zuses Entwurf von 1945 endet mit den Prozeduren für das Schachprogramm von dem in den Notizen von 1939 die Rede war [12].

Der Plankalkül (PK) kann mit Hilfe moderner Terminologie wie folgt umrissen werden:

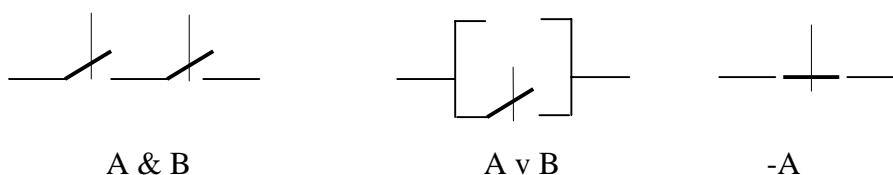
- PK ist eine höhere imperative Programmiersprache.
- Programme werden als Funktionen definiert.
- Funktionen (Programme) können aufgerufen werden, sind aber nicht rekursiv.
- Argumente von Funktionen können nicht verändert werden (call by value).
- Es gibt nur lokale Variablen.
- Es gibt einen einzigen primitiven Datentyp: das Bit.
- Zusammengesetzte Typen sind Arrays und Tupel.
- Jede Variable wird mit ihrem Typ angegeben.
- Es gibt bedingte Befehle (IF-THEN-Befehl).
- Es gibt eine WHILE- und eine FOR-Schleife.
- Es gibt keinen GOTO-Befehl.

Der einzige exotische Aspekt von Plankalkül ist sein gemischtes ein- und zweidimensionales Layout. Variablen werden in vier Zeilen geschrieben anstatt Klammern für die Indizes und Datentyp zu verwenden.

Der Plankalkül sollte die Programmiersprache für eine neuartige Rechenmaschine werden, eine die nicht mit bloßen Zahlen, sondern mit allgemeinen logischen Ausdrücken operieren konnte: die *logistische Maschine*. Diese war in der Tat minimal: sie sollte nur zwei Register mit jeweils einem Bit enthalten und die Wortlänge des Speichers sollte auch nur ein Bit betragen. Die logistische Maschine wurde konzipiert, um die unterste Verarbeitungsebene zu implementieren, d.h. die logischen Bit-Operationen AND, OR und NOT. Im Grunde genommen ähnelt die logistische Maschine eher eine Turing-Maschine als einem modernen Rechner. Hier wird aber die Tragweite des Plankalküls offengelegt: was auf der Programmierenebene ein komfortabler Kalkül für den Programmierer ist, soll in Elementarschritte zerlegt werden. Dies ist in der Prädikatenlogik möglich – es ist dann auch mit einer Rechenmaschine durchführbar.

2 Plankalkül als Notation

Wie oben bereits erwähnt, hat Konrad Zuse eine eigene graphische Notation für kombinatorische Schaltungen entwickelt, mit der er seine mechanischen und elektromechanischen Geräte beschrieben hat. Er hat später entdeckt, daß er mit seiner Notation eine \exists äquivalente Beschreibung der Aussagenlogik geschaffen hat. In seiner geplanten Dissertation mit dem umständlichen Titel "Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen" zeigt Zuse, wie die verschiedenen aussagenlogischen Ausdrücke mit verallgemeinerten Relais implementiert werden können [11]. Für die Konjunktion, Disjunktion und Negation reichen die in dem Bild gezeigten Schaltungen:



Jedoch hat Zuse mit der Prädikatenlogik vor allem eine Sprache gefunden, mit der beliebig geartete Rechnungen beschrieben werden könnten. Die All- und Existenzquantoren und die Relationen-Notation hat er als mächtiges Werkzeug betrachtet, womit jede Art von gedanklichem Konstrukt effektiv dargestellt werden könnte. Diese Erkenntnis war wichtig, weil ihm ab dann klar war, daß seine Geräte Z1 und Z3 nur Spezialfälle der mächtigeren "logistischen Maschine" waren.

In der Notation von David Hilbert haben die All- und Existenzaussagen folgende Form:

$(\forall x)$	$R(x)$	"Für alle x gilt $R(x)$."
$(\exists x)$	$R(x)$	"Es gibt ein x , für die $R(x)$ gilt."

Auch Mengen können aus Existenzaussagen aufgebaut werden. Die so generierten Mengen sind dann die Grundelemente eines Klassenkalküls:

\hat{x}	$R(x)$	"Die Menge aller x , für die $R(x)$ zutrifft."
x'	$R(x)$	"Das Element x , für das $R(x)$ zutrifft."

Genau diese Art der Notation bildet den Ausgangspunkt für Zuses Überlegungen. Dazu kommt noch der Relationen-Kalkül, da arithmetische Operationen wie $1+2$ in funktionaler Form, z.B. "Addition(1,2)", geschrieben werden können.

3 Plankalkül als Programmiersprache

Zuse hat von der Prädikatenlogik die funktionale Notation für den Plankalkül übernommen. Abstrakt gesehen ist jedes Plankalkül-Programm eine Funktion $R_m(m)$

eine Zahl) mit n variablen Argumenten V_0, V_1, \dots, V_n . Diese Funktion kann in anderen Programmen aufgerufen werden, indem man $R_m(A_1, A_2, \dots, A_n)$ schreibt, wobei A_1, \dots, A_n die Argumente des Aufrufs darstellen. Das Ergebnis kann einen beliebigen Datentyp haben. Um die Definition eines solchen Programms als Funktion verstehen zu können, müssen wir jedoch vorher etwas über die zulässigen Datentypen sagen.

3.1 Variablen und Datentypen

Daten für Programme werden in Variablen gespeichert und manipuliert. Plankalkül unterscheidet zwischen drei Arten von Variablen:

- V-Variablen von der Form V_0, V_1 , usw., die nur gelesen werden können.
- Z-Variablen von der Form Z_0, Z_1 , usw., die gelesen und überschrieben werden können.
- R-Variablen von der Form R_0, R_1 , usw., die nur überschrieben werden können.

Diese Aufteilung, nicht üblich in heutigen Programmiersprachen, hat eine wohldefinierte semantische Bedeutung: V-Variablen sind Eingabevariablen für Funktionen und dürfen daher nicht verändert werden (sie sind read-only). R-Variablen stellen die Ergebnisse der Funktion dar und sollten deswegen nicht gelesen, sondern nur geschrieben werden (write-only). Z-Variablen werden für Zwischenergebnisse verwendet und können daher sowohl gelesen als auch geschrieben werden. Alle Variablen sind "Behälter" für beliebig geartete Daten.

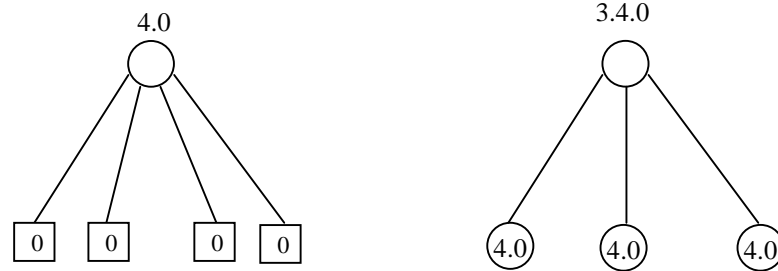
Daten haben einen bestimmten Datentyp. Im Plankalkül ist der elementarste davon ein Bit. Zusammengesetzte Datentypen sind Felder und Tupel von anderen Datentypen. Folgende Datentypen können verwendet werden:

- Ein Bit, als "0" dargestellt.
- n Bits, wobei n eine ganze Zahl ist, als " $n.0$ " dargestellt.
- Tupel von Typen, wie z.B. "(3.0, 4.0)", d.h. ein Tupel von zwei Elementen, die jeweils 3 bzw. 4 Bits enthalten.
- m -mal jeder andere Typ (ein Array von anderen Typen)

Plankalkül-Datenstrukturen können als Bäume interpretiert werden. Ein Blatt vom Baum ist immer ein einziges Bit. Unterbäume sind Zwischenstrukturen.

Das Bild links zeigt die Struktur des Typs "4.0". Sie entspricht einem Baum mit vier Blättern. Das Bild rechts zeigt die Struktur 3.4.0. Sie besteht aus einer Wurzel, die auf drei Unterbäume der Struktur "4.0" zeigt.

Variablen - Bäume



Der einzige Unterschied zwischen Feldern und Tupeln ist, daß bei Arrays alle Kinder der Wurzel die gleiche Struktur haben. Bei Tupeln hat jedes Kind eine womöglich unterschiedliche Struktur.

Die Baumstruktur aller Plankalkül-Datenstrukturen vereinfacht auch die Implementierung im Computer, da nur solche Bäume unterstützt werden brauchen.

Zuse hat eine tabellarische Darstellungsform für die Variablen gewählt, bei der jede Variable in vier Zeilen beschrieben wird. Das Beispiel unten zeigt wie die Variable Z1 vom Typ "5.0" geschrieben wird. Der Subindex der Variablen wird in der V-Zeile geschrieben, der Typ in der S-Zeile (Strukturzeile). Die Annotation der Zeilenamen braucht nicht angegeben werden und dient, wenn vorhanden, einer klaren Darstellung.

	Z
V	1
K	
S	5.0

Die K-Zeile (Komponentenzeile) wird verwendet, um auf die Komponenten von Variablen zuzugreifen, wie im unteren Beispiel, in dem die Komponente 0 der Variablen Z1 angesprochen wird.

	Z
V	1
K	0
S	0

Die Komponenten der Variablen werden ab 0 durchnummeriert. Die Strukturzeile gibt immer die Struktur der ausgewählten Komponente an (im obigen Beispiel ein Bit).

Die indexierte Adressierung von Variablen ist möglich. Im unteren Beispiel ist die Komponente der Variable 1, die wir ansprechen möchten, in Variable Z2 gespeichert.

chert. Die Verbindungslinie bedeutet, daß wir auf die Komponente von Z1, deren Nummer in Z2 gespeichert ist, zugreifen möchten.

V	Z	Z
K	1	2
S	0	8.0

3.2 Plankalkül-Anweisungen

Obwohl Zuse im Entwurf des Plankalküls mehrere sehr spezielle Operatoren und Anweisungen definierte, kann der größte Teil seiner Beispielprogramme mit einer kleinen Untermenge des Sprachumfangs geschrieben werden. Die wichtigsten imperativen Plankalkül-Anweisungen sind eigentlich nur drei:

- a) Zuweisung
- b) Bedingte Ausführung (IF ... THEN)
- c) Programmschleifen (WHILE und FOR Schleifen)

Eine Folge von Befehlen kann in einem Block zusammengefaßt werden. Dafür wird die Befehlsfolge innerhalb eckigen Klammern geschrieben. Es gibt auch vordefinierte Systembefehle. Der wichtigste davon ist "FIN n ", wobei n eine Zahl ist. Mit FIN kann der aktuelle Block (bis auf n Ebenen nach oben) unterbrochen werden.

Außerdem kann ein Programm als Funktion aufgerufen werden (was wir heute einen CALL nennen). Deswegen werden bei jedem Programm in einem Header die Eingabe- und Ausgabe-Variablen deklariert.

3.3 Die Zuweisung

Arithmetische und logische Terme, sowie Tupel von solchen Termen, können in Variablen kopiert werden. Das Zuweisungssymbol ist " \Rightarrow ".

Ein Term ist eine Konstante, eine Variable, das Resultat eines Funktionsaufrufs oder das Resultat einer Operation. Es gibt numerische und logische Konstanten. Numerische Konstanten sind Dezimalzahlen. Logische Konstanten sind "+" (für Wahr) und "-" (für Falsch), sowie Bitfolgen wie z.B. "+ + - - +".

Die arithmetischen Operationen sind die vier üblichen, d.h. Addition, Subtraktion, Multiplikation und Division. Die logischen Operationen sind die Konjunktion, Disjunktion, Negation, Äquivalenz und XOR.

Im unteren Beispiel wird die Komponente 0 von der Variable V0 mit der Komponente 2 derselben Variable addiert und das Resultat wird in der Variable Z2 gespeichert. Der Typ der V0-Komponenten ist ein Byte, sowie auch der Typ von Z2.

V	+	V	⇒	Z
0		0		2
0		2		
8.0		8.0		8.0

Dieses Beispiel zeigt bereits eines der großen Probleme des Plankalküls: der Programmierer muß jedesmal den Typ seiner Variablen angeben und die Konsistenz der befohlenen Operationen durchdenken. Bevor wir weitere Plankalkül-Befehle besprechen, sollten wir vorher den Programmrumpf und den "Randauszug" kennenlernen.

3.4 Variablendeklarationen und der Randauszug

Ein Beispiel kann die allgemeine Struktur von Plankalkül-Programmen illustrieren:

P3	R	(V, V)	⇒	(R, R)
	0	1		0 1
	8.0	8.0		4.0 4.0

(Sequenz von Befehlen)
END

Hier wird Programm Nummer 3 definiert. Das Resultat der Anwendung, mit den Variablen V0 und V1 als Argumente, ist ein Tupel, der die Variablen R0 und R1 enthält. Der Typ aller Eingabevariablen ist 8.0, der Typ der Ausgabevariablen ist 4.0. Die Anzahl der Eingaben und Ausgabevariablen kann im Randauszug beliebig sein und die Variablen werden steigend durchnummeriert, von Null angefangen.

Diese einführende Information ist der "Randauszug", der die Schnittstellen von der Funktion definiert. Nach dem Randauszug kommt eine Folge von Befehlen, die mit dem Schlüsselwort END abgeschlossen wird. END war in der ursprünglichen Definition des Plankalküls nicht vorhanden – wir haben sie eingebaut, um einen textuellen Abschluß der Funktionsdefinition zu haben. Die oben definierte Funktion kann in einem anderen Programm als R3 mit zwei Argumenten aufgerufen werden.

In der Definition kann statt "R" ein Bezeichner für den Funktionsnamen geschrieben werden und diese kann für den Aufruf der Funktion verwendet werden.

Es ist anzumerken, daß die Komponentenzeile in dem Randauszug nicht verwendet wird. Eingabe- und Ausgabevariablen werden als ganzes definiert. Deswegen ist eigentlich jede spätere Angabe der Struktur von diesen Variablen in den Befehlszeilen redundant. Solche Angaben könnten bei den V und R-Variablen gespart

werden und Zuse tut dies oft im Entwurf des PK, vor allem wenn aus dem Kontext klar ist, welche Art von Variablen verwendet wird.

Schwieriger ist es jedoch mit Z-Variablen umzugehen. Diese können innerhalb des Programmumpfes verwendet werden, ohne vorher deklariert worden zu sein. Hier muß der Programmierer besonders aufmerksam sein.

3.5 Bedingte Kommandos

Die bedingte Ausführung wird in PK mit einem Pfeil dargestellt. Das Symbol \rightarrow trennt einen logischen Term von einem Befehl. Der Befehl rechts vom Pfeil wird nur ausgeführt, wenn der logische Term links wahr ist.

Gegeben sei folgendes Programmfragment:

Z	^	Z	→	V	+	V	⇒	Z
0		1		0		0		3
				0		2		
0		0		8.0		8.0		8.0

Diese Anweisung bedeutet, daß nur, wenn die Konjunktion der beiden in Z0 und Z1 gespeicherten Bits wahr ist, die darauf folgende Addition und Zuweisung in die Variable Z3 ausgeführt wird. Der Pfeil bindet stärker als die Zuweisung und die arithmetisch-logischen Operatoren stärker als der Pfeil und das Zuweisungssymbol, so daß es keine Mehrdeutigkeit gibt. Trotzdem kann man mit Klammern die Syntax deutlicher machen, wie unten gezeigt:

(Z	^	Z)	→	(V	+	V)	⇒	Z
0		1		0		0		3
				0		2		
0		0		8.0		8.0		8.0

Klammern werden nur in der obersten Zeile geschrieben.

Jeder Befehl wird in eine Zeile geschrieben. Befehle können zu einem Befehlsblock zusammengefaßt werden, indem sie in eckige Klammern gesetzt werden, siehe folgendes Beispiel, in dem ein Additions- und ein Multiplikationsbefehl in einem Block enthalten sind:

[Z	+	2	⇒	Z]
	0				2	
	8.0				8.0	
	Z	×	Z	:	Z	
	2		1		3	
	8.0		8.0		8.0	
]						

Bedingungen können mit Hilfe der Vergleichsoperatoren "=", ">", und "<" überprüft werden. Diese Operatoren testen jeweils, ob das linke Argument gleich, kleiner oder größer als das rechte Argument ist. Tupel oder Arrays können auf Gleichheit getestet werden, aber nur Strukturen der Form n.0 können mit den anderen beiden Operatoren verglichen werden. Das folgende Programmfragment speichert die größte von zwei Zahlen in die Variable Z3.

$$\begin{array}{l} Z \\ 1 \end{array} \Rightarrow \begin{array}{l} Z \\ 3 \end{array}$$

$$8.0 \qquad \qquad 8.0$$

$$\begin{array}{l} Z \\ 1 \end{array} < \begin{array}{l} Z \\ 2 \end{array} \rightarrow \begin{array}{l} Z \\ 2 \end{array} \Rightarrow \begin{array}{l} Z \\ 3 \end{array}$$

$$8.0 \qquad \qquad 8.0 \qquad \qquad 8.0 \qquad \qquad 8.0$$

3.6 Schleifen

Plankalkül kennt Schleifen, aber nicht die GOTO-Anweisung. Eine WHILE-Schleife wird in der Form

$$W [\text{Block}]$$

geschrieben, wobei "[Block]" einen Befehlsblock darstellt. Im allgemeinen hat der Block die Form:

$$W \left(\begin{array}{l} C1 \rightarrow S1 \\ C2 \rightarrow S2 \\ \dots \\ Cn \rightarrow Sn \end{array} \right)$$

Der Block wird wiederholt ausgeführt, so lange zumindest eine der Bedingungen C1, C2, usw. wahr ist. Wenn in einem Durchlauf alle getesteten Bedingungen falsch sind, wird die Schleife abgebrochen. Die Befehle S1, S2, usw. werden nur ausgeführt wenn ihre jeweiligen Bedingungen wahr sind.

Schleifen können geschachtelt werden, wenn sie durchnumeriert werden. Zwei geschachtelte Schleifen haben die Form:

$$\begin{array}{l} W [\dots W [\text{Block}] \dots] \\ 0 \qquad 1 \end{array}$$

Die äußere Schleife ist W[0], die innere W[1].

Der Konstrukt $W0(n)$ gleicht der FOR-Schleife in anderen Programmiersprachen. Der darauffolgende Block wird $(n+1)$ -mal wiederholt. Wenn wir einen Iterationsindex benötigen, muß das Konstrukt $W1(n)$ benutzt werden. Der darauf folgende Block wird ebenfalls $(n+1)$ -mal wiederholt, aber eine Index-Variablen mit dem Namen "i" läuft nun von 0 bis n , entsprechend einen Schritt pro Iteration. Bei geschachtelten Schleifen heißen die Indexvariablen "i0", "i1" usw., je nachdem zu welcher Schleife sie gehören. Diese Schleifen-Variablen haben einen generischen numerischen Typ (z.B. 32 Bit). Das Beispiel unten zeigt zwei geschachtelte $W1$ -Schleifen und ihre Index-Variablen:

$$W1 \left(\begin{array}{c} \dots i \dots W1 \left(\dots i \dots \\ \dots 0 \dots 1 \left(\dots 1 \dots \right) \end{array} \right) \right)$$

Die Schleifen-Variablen dürfen nur innerhalb ihre jeweiligen Blöcke verwendet werden.

Es gibt im Plankalkül eine vordefinierte Funktion "N", die eine Struktur als Argument nimmt und die Anzahl ihrer Elemente (auf der obersten Ebene) wiedergibt. Wenn die Variable $V0$ z.B. den Typ "12.8.0" besitzt, so ergibt $N(V0)$ die Zahl 12.

4 Die Implementierung

Eine Implementierung des Plankalküls erfordert eine der Linearisierung der Sprache, sowie Transformationen der prädikatenlogischen Ausdrücke.

4.1 Linearisierung der Sprache

Plankalkül-Programme können mit einem konventionellen Editor geschrieben werden, wenn eine passende Linearisierung der tabellarischen Form definiert wird. Wir werden schreiben Variablen folgendermaßen:

$V0[1 :5.0]$	Variable $V0$, Komponente 1, vom Typ 5.0
$Z1[5.3 :9.0]$	Variable $Z0$, Komponente 3 der Komponente 5, vom Typ 9.0
$Z2[:12.3.0]$	Variable $Z2$ vom Typ 12.3.0

Spezielle Symbole des Plankalküls werden mit ASCII-Zeichen ersetzt. Konjunktion, Disjunktion und Negation werden mit den Zeichen "&", "|" und "!" dargestellt. Zuweisung wird mit "=>" und bedingte Befehle mit "->" dargestellt.

Der bedingte Befehl am Anfang des Abschnitts 3.5 kann deswegen umgeschrieben werden als:

$$Z0:0 \ \& \ Z1:0 \ \rightarrow \ V0[0 :8.0] + V1[2 :8.0] \Rightarrow Z3[:8.0]$$

Der Rest der Beispiele in diesem Aufsatz zeigt, wie PK-Programme in linearisierter Form geschrieben werden. Folgendes Programmfragment berechnet die Summe aller 16 Komponenten einer Variable V0 der Struktur "16.8.0".

$$\begin{aligned} 0 \Rightarrow Z1[:8.0] \\ W1(16) [Z1[:8.0] + V0[i :8.0] \Rightarrow Z1[:8.0]] \end{aligned}$$

Es ist anzumerken, daß die Typen der Konstante 16 und der Index-Variable "i" nicht angegeben werden. Sie besitzen beide einen generischen numerischen Typ (32.0 in unserer Implementierung).

4.2 Transformation der prädikatenlogische Ausdrücke

Wir haben oben bereits erläutert, wie wichtig für Zuse die prädikatenlogische Notation am Anfang der Entstehung vom Plankalkül gewesen ist. Es war ihm klar, daß für eine Computerimplementierung die deklarativen logischen Ausdrücke in lauffähige imperative Programme transformiert werden müssen. In diesem Abschnitt zeigen wir die von Zuse verwendeten Transformationen. Wir benutzen sowohl Pseudocode als auch linearisierte PK-Programme.

Der Ausdruck "Für alle x gilt $R(x)$ ", d.h., $(x) R(x)$ muß konkretisiert werden, indem die universelle Menge für x ausdrücklich benannt wird. In Plankalkül wird also geschrieben:

$$(x) (x \text{ in } V0 \ \& \ R(x))$$

, d.h. für alle x im Array V0 gilt $R(x)$. In Pseudocode entspricht dies der Berechnung

$$\begin{aligned} \text{flag} &:= \text{true}; \\ \text{FOR } i &= 0 \text{ TO } N(V0)-1 \\ &\quad \text{flag} := \text{flag AND } R(V0(i)) \end{aligned}$$

, wobei die logische Variable "flag" am Ende das Resultat enthält. Im PK könnten wir der Variablen Z1 das Resultat folgendermaßen zuweisen:

$$\begin{aligned} + \Rightarrow Z1[:0] \\ W1(N(V0)-1) [Z1[:0] \ \& \ R(V0[i:32.0]) \Rightarrow Z1[:0]] \end{aligned}$$

Die Existenzaussage "Für jedes x in $V0$ existiert ein x , so daß $R(x)$ gilt." wird so geschrieben:

$$(\exists x) (x \text{ in } V0 \ \& \ R(x))$$

und wird in PK folgendermaßen umgesetzt:

$$\begin{array}{l} - \Rightarrow Z1[:0] \\ W1 (N(V0)-1) [R(V0[i:32.0]) \rightarrow [+ \Rightarrow Z1[:0] \\ \text{FIN 2} \quad]] \end{array}$$

Falls ein Element gefunden wird, das die gewünschte Relation erfüllt, wird die Schleife mit "FIN 2" abgebrochen (2 wegen der zwei geschachtelten Blöcke).

Es ist nicht schwer zu sehen, wie die beiden Ausdrücke

$$\begin{array}{ll} \hat{x} R(x) & \text{"Diejenigen } x \text{ in } V0, \text{ die } R \text{ erfüllen."} \\ \text{und} & \\ x' R(x) & \text{"Dasjenige } x \text{ aus } V0, \text{ das } R \text{ erfüllt."} \end{array}$$

in Plankalkül-Programme umgesetzt werden können. Beim ersten Ausdruck muß noch überprüft werden, ob ein Element, das R erfüllt, nicht bereits vorgekommen ist, so daß es nicht doppelt in die Menge aufgenommen wird. Falls Elemente auch wiederholt vorkommen dürfen, wird statt \hat{x} , $\hat{\hat{x}}$ geschrieben.

All dies bedeutet, daß ein PK-Compiler die prädikatenlogischen Ausdrücke vorher einer Quellcode-Transformation unterwerfen sollte. Deswegen brauchen sie auch nicht im eigentlichen Sprachumfang enthalten zu sein. Wir betrachten sie als deklarative "Makroausdrücke" der Sprache.

4.3 Die Grammatik

Im Appendix findet der interessierte Leser eine Zusammenfassung der Grammatik für eine berechnungsuniverselle Untermenge des Plankalküls. Diese Untermenge haben wir Plankalkül 2000 genannt. Mit dieser Untermenge können Programme wie Insertion-Sort, ein bekannter Sortieralgorithmus, geschrieben werden. Zuse hat den Code in seinem Plankalkül-Manuskript niedergeschrieben und wir geben ihn hier in linearisierter Form wieder.

Insertion Sort (Zuse 1972)

P327 Sort (V0[:m.s]) => R0[:m.s]

V0[:m.s] => Z0[:m.s]

```

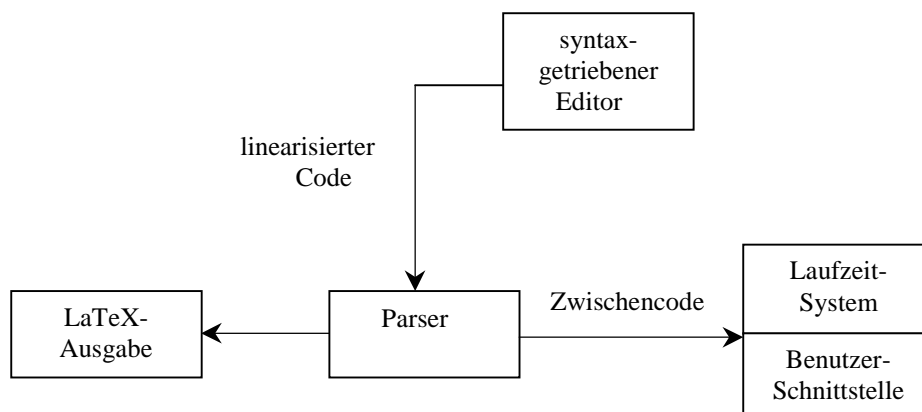
W1(m-1) [ Z0[i+1:s] => Z1[:s]
          i => Z2[:32.0]
          W [ Z2 > -1 -> [
                    Z1[:s] < Z0[Z2[:32.0]:s] ->
                    [ Z0[Z2[:32.0]:s] => Z0[Z2[:32.0]+1:s]
                      Z2[:32.0] -1 => Z2[:32.0] ]
                    ]
          ]
          ]
          Z2[:32.0] = -1 -> Z1[:s] => Z0[:s]
          ]
Z0[:m.s] => R0[:m.s]

```

5 Struktur des Laufzeitsystems

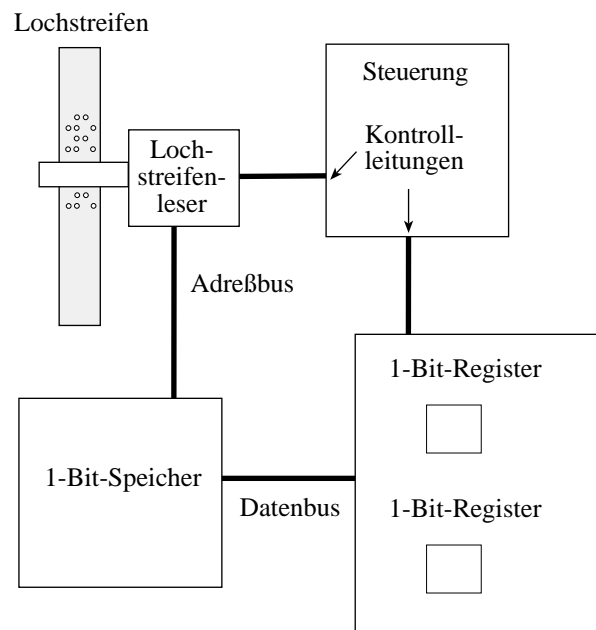
Wir haben Plankalkül 2000 in Java und C implementiert. Die Abbildung unten zeigt die Struktur des Systems. Ein syntaxgesteuerter Editor erstellt Plankalkül-Programme, wobei die vierzeilige Notation unterstützt wird. Der Editor ist kontextsensitiv und erlaubt es dem Programmierer nur syntaktisch korrekte Ausdrücke anzugeben.

Die mit dem Editor erstellten Programme werden in linearisierter Form an einem Parser weitergegeben, der einen Zwischencode generiert, der direkt vom Laufzeitsystem interpretiert wird. Der Parser generiert auch LaTeX-Darstellungen, mit denen die Programme direkt in eine druckreife Form gebracht werden können.



6 Die logistische Maschine

Die Abbildung unten zeigt das Diagramm der logistischen Maschine. Sie besteht aus einem Lochstreifenleser, der die einzelnen Befehle des Programms sequentiell liest. Die Befehle sind logische Operationen mit einem oder zwei Bits, d.h. AND, OR und Negation. Der Speicher besteht aus Worten der Länge ein Bit. Die zwei Register im Prozessor können ebenfalls jeweils nur ein Bit speichern. Die bedingte Ausführung von Befehlen ist nicht dokumentiert.



Zusammenfassung

Wir haben in diesem Aufsatz die Grammatik und Struktur des Plankalküls beschrieben, sowie die Ursprünge der von Zuse verwendeten Notation. Wir haben die Sprache zum ersten mal implementiert und verschiedene Muster-Programme sind erfolgreich getestet worden.

Was bringt aber die Implementierung einer längst überholten Programmiersprache?
Was können Studenten daraus lernen?

Für einige unsere studentischen Mitarbeiter war dies der erste von ihnen erstellte Compiler, insofern hatte das ganze eine didaktische Komponente. Das wichtigste war aber, das mit solche Projekten Computergeschichte plötzlich und unerwartet lebendig wird. Die Geschichte des Computers ist zu wichtig, um sie allein den Historikern zu überlassen. Man kann heute Informatiker für die Geschichte unseres Faches durch solche Rekonstruktionsprojekte interessieren und begeistern. Für uns war die Erstellung eines Plankalkül-Compilers eine beeindruckende Reise zurück zu den Ursprüngen des Computerzeitalters. Allein deswegen hat sich das Projekt für alle Beteiligten gelohnt. Und es war uns wichtig eine historische und kulturelle Lücke zu füllen: Plankalkül, die erste Programmiersprache der Welt, sollte nicht für immer im Dornröschenschlaf verharren.

Der interessierte Leser kann die Definition der Grammatik von Plankalkül 2000 im ersten Appendix finden. Die frühesten Notizen von Zuse über den Plankalkül haben wir im zweiten Appendix transkribiert. Diese Notizen befinden sich in den von der GMD in den siebziger Jahren archivierten Zuse-Papiere (ZuP). Mehr Information über unser Projekt kann im Konrad-Zuse-Internet-Archiv gefunden werden:

<http://www.zib.de/zuse>

Literatur

1. Arefi, F., Hughes, Ch., Workman, D.: Automatically Generating Visual Syntax-Directed Editors. *Communications of the ACM* 33, 349-360 (1990).
2. Bauer, F.L., Wössner, H.: The "Plankalkül" of Konrad Zuse: A Forerunner of Today's Programming Languages. *Communications of the ACM* 15, 678-685 (1972).
3. Giloi, W.: Konrad Zuses Plankalkül: The First High-Level, "non von Neumann" Programming Language. *IEEE Annals of the History of Computing* 19, (1997).
4. Grosch, J.: Generators for High-Speed Front-Ends. LNCS, 371, Berlin, Heidelberg: Springer-Verlag, 1988, pp. 81-92.
5. Grosch, J., Emmelmann, H.: A Tool Box for Compiler Construction. LNCS, 477, Berlin, Heidelberg: Springer-Verlag, 1990, pp. 106-116.
6. Hohmann, J.: Der Plankalkül im Vergleich mit algorithmischen Sprachen. stmv-Verlag: Darmstadt (1979).
7. Petzold, H.: Moderne Rechenkünstler. München: C.H. Beck (1992).
8. Rojas, R. (Hrsg.): Die Rechenmaschinen von Konrad Zuse. Berlin, Heidelberg: Springer-Verlag, 1998.
9. Rojas, R., Göktekin, C., Friedland, G., Krüger, M., Langmack, O., Kuniß, D.: Plankalkül: The First High-Level Programming Language and its Implementation. Technical Report B-3/2000, Freie Universität Berlin (2000).
10. Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax Directed Programming Environment. *Communications of the ACM* 24, 563-573, (1981).
11. Zuse, K.: Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relais-schaltungen. Unveröffentlichtes Manuskript, Zuse-Papiere 045/018, 1943.
12. Zuse, Konrad: Der Plankalkül. *Berichte der Gesellschaft für Mathematik und Datenverarbeitung*, Nr. 63, Sankt Augustin, 1972.
13. Zuse, K.: Der Computer - Mein Lebenswerk. Berlin, Heidelberg: Springer-Verlag (1986).

Appendix A

Syntax von Plankalkül 2000

In diesem Appendix fassen wir die Syntax der Untermenge des Plankalküls zusammen, die von uns implementiert wurde. Das Symbol "|" trennt Optionen voneinander. Ein Stern bedeutet keine oder mehrere Wiederholungen des Ausdrucks in geschweiften Klammern.

Zulässige Symbole:

digit	::= 0 1 2 ... 9	
number	::= digit {digit}*	
letter	::= a b ... z A B ... Z	
type-letter	::= a b ...h j ... z	\\ d.h. ohne "i"
identifizier	::= letter {letter digit}*	
log-constant	::= + - + log-constant - log-constant	\\ - bedeutet "Falsch"
constant	::= number log-constant	
dot	::= "."	
comma	::= ","	

Programme bestehen aus Programmnummer, Randauszug und einer Sequenz von Befehlen:

program	::=	P number randauszug {statement }* END
---------	-----	---

Randauszug (definiert Bezeichner für die Funktion, Eingabe und Ausgabevariablen): Hier wird angenommen, daß nicht mehr als vier Ein- und Ausgabevariablen verwendet werden.

randauszug	::=	identifizier v-tuple => r-tuple
v-tuple	::=	(V0[:type]) (V0[:type],V1[:type]) (V0[:type],V1[:type],V2[:type]) (V0[:type],V1[:type],V2[:type],V3[:type])
r-tuple	::=	(R0[:type]) (R0[:type],R1[:type]) (R0[:type],R1[:type],R2[:type]) (R0[:type],R1[:type],R2[:type],R3[:type])

Befehle (eigentlich nur drei Arten davon, dazu Blöcke und Sonderbefehle):

statement ::= assignment | if-then | while | block | built-ins
 block ::= [statement {statement}*]
 built-ins ::= FIN | FIN number

Zuweisung:

assignment ::= term2 => variable |
 tuple => variable
 tuple => var-tuple
 var-tuple ::= (variable, variable {comma variable}*)
 tuple ::= (term2, term2 {comma term2}*)

Bedingte Anweisungen:

if-then ::= term2 -> statement

Wiederholungen:

while ::= w block |
 w [number] block |
 w1 (arith-arg) block |
 w1 [number] (arith-arg) block

Datentypen:

simple-type ::= 0
 tuple-type ::= (type, type {comma type}*)
 type ::= simple-type | tuple-type | number dot type
 var-type ::= type-letter dot type
 all-type ::= type | var-type

Variablen:

variable ::= {V | Z | R} number [component: type] |
 {V | Z | R} number [: all-type]
 generic-variable ::= type-letter | "i" | "i" number
 component ::= number | variable | generic-variable | term2 |
 component dot component

Funktionsaufrufe:

```

call          ::=  R number [:type] ( term2 {,term2}*) |
                 identifier [:type] ( term2 {,term2}*) |
                 R number [component : type] ( term2 {,term2}*) |
                 identifier [component : type] ( term2 {,term2}*)

```

Arithmetische und logische Operationen:

```

term          ::=  constant | variable | generic-variable | call |
                 (operation) | (condition) | (term2)

term2         ::=  term | - term | ! term | operation | condition

operation     ::=  term2 {+ | - | × | / | & | "|" | ~ | ~\ } term

```

Logische Bedingungen:

```

condition     ::=  term2 = term2 |
                 term2 > term2 |
                 term2 < term2 |
                 tuple = tuple

```

Laufzeit-System:

- V-Variablen können nicht überschrieben werden.
- R-Variablen können nicht gelesen werden.
- Nur Daten derselben Dimension (n.0) können miteinander arithmetisch kombiniert oder verglichen werden. Die einzige Ausnahme sind generische Variablen, die mit den anderen Typen kombiniert werden können.
- Numerische Konstanten haben generischen Typ.
- Logische Konstanten haben den passenden Typ (z.B. +++ hat Typ 3.0).
- Logische Operatoren können nur auf Daten vom Typ "0" angewendet werden.
- Bei Prozeduraufrufen werden generische Typvariablen instanziiert.
- Partiiell aufgebaute Strukturen können nicht als Argument für Prozeduraufrufe verwendet werden.

Appendix B: Früheste Notizen von Konrad Zuse zum "Plankalkül"

Dieser Text wurde von Konrad Zuse 1939 in sein Notizbuch eingetragen.

25.5.39

Stand der Arbeiten an Versuchsmodellen: Rechenwerk fertig aber funktioniert schlecht.¹

Mechanisches Hirn: inzwischen Schaltungsmagnetik entwickelt (allgemeine Dyadik). Seit etwa einem halben Jahr allmähliches Einführen in die formale Logik. Viele meiner früheren Gedanken habe ich dort wieder gefunden. (Bedingungskombinatorik = Aussagenlogik; Lehre von den Intervallen = Gebietenkalkül). Ich plane jetzt die Aufsetzung des "Plankalküls". Hierzu sind eine Reihe von Begriffen zu klären.

- 1.) Aufgabenstruktur. Formel z.B. behandelt die äußere Gestalt der Angaben.
- 2.) Index Ordnung: $a_{1,2,n,3}$
- 3.) Vielfach gegliederte Indizes. Variable Indizes usw. "Ausklammerung" von Indizes.
- 4.) Numerische Formeln. Alle Formeln mit dehnbare Struktur, z.B. $E(x) f(x); (x) f(x)$ usw.
- 5.) Befehlsformalismus. Algebraische Abwandlungen. Zur Kontrolle. Funktionsbezeichnung. Allgemeine Form (= Addition) Kennzeichnung bestimmter Werte usw.
- 6.) Schachspiel: selbst einfache Beziehungen nur schwer darstellbar, jedoch große Möglichkeiten vorauszusehen. Entwicklung des Schachspiels

Grundbegriff:

I. Mengen, Relationen, Prädikate usw.

II. Einfache Rechenoperationen mit ganzen Zahlen
(Anzahl Begriff, Zahlen, Addition, Subtraktion $>$, $<$, $=$)

III. Geometrie des Punktes. Nachbarbezeichnung, diagonale Beziehung,
Richtungsbegriff, Begriff der dazwischen liegenden Punkte

IV. Grundregeln der Brettspiele, Parteien, Begriffe der Parteien, Steine setzen
a) auf das Feld setzen (Spiel-Anfang)
b) während des Spiels setzen
Schlagen, Ersetzen (Bauer Umwandlung), Begriff des Zuges, Begriff des Gewinnens und Verlierens

¹ Es handelt sich um die mechanische Rechenmaschine Z1.

V. Schachspiel

- A. Felderklärung
- B. Aufzählung der Steine
- C. Setz- und Schlagregeln der Steine
- D. Sonderregeln
- E. Schachregeln.

Minimalform der rein universellen Programmiersprache

Allgemeine Form: $F(v) \Rightarrow R$

Bedingungen:

- 1) Bedingter Sprung (Schlußzeichen)
- 2) Errechneter Index.

Die Begriffe "Minimalform" und "Optimalform" sind sehr relativ. Vom Standpunkt:

- 1) des Aufwandes an Regeln bzw. Formelzeichen
- 2) der Kürze der Darstellung
- 3) der Beschränkung auf notwendige Information
- 4) der eleganten und übersichtlichen Darstellung
- 5) der minimalen Rechenzeit (Speicheraufwand usw.)
- 6) der günstigen Übersetzungsmöglichkeit (einfacher Compiler)

Programmiersprachen: Explizite Form

Allgemeine Formalsprachen: auch implizite Darstellung

Generelles aber nicht allgemein lösbares Problem: Übersetzung der impliziten in die explizite Form (hierbei wieder Optimierungsproblem)

Raúl Rojas ist Professor für Künstliche Intelligenz am Institut für Informatik der FU Berlin. Er ist Herausgeber des Springer Buchs: *Die Rechenmaschinen von Konrad Zuse* (1998). Er hat die Grammatik vom PK-2000 definiert und das Projekt geleitet.

Cüneyt Göktekin ist Student an der FU Berlin und hat verschiedene Java-Simulationen der Schaltungen der Rechenmaschine Z3 erstellt (siehe <http://www.zib.de/zuse>). Er schreibt noch die endgültige Version des PK-Editors.

Gerald Friedland ist Student an der FU Berlin. Er war 1998 Sieger des Landeswettbewerbs "Jugend Forscht" mit einem Programm, das im Laufe der Zeit zu Java-Radio geworden ist (siehe <http://www.javaradio.de>). Er hat die PK-Zwischensyntax definiert und die PK-Speicherverwaltung geschrieben.

Mike Krüger ist Student an der FU Berlin. Er hat das Laufzeitsystem für unsere Plankalkül-Implementierung geschrieben.

Ludmila Scharf ist Studentin an der FU Berlin. Sie hat die PK-Schach-Programme von Zuse in Java implementiert und die LaTeX-Makros für den Plankalkül geschrieben.

Olaf Langmack ist Diplom-Informatiker der Universität Bremen und war Mitarbeiter am Institut für Informatik der FU Berlin. Er ist Gründer des Netzwerks "Feinarbeit" und der auf Compilerbau spezialisierte Firma "Transformal" in Berlin. Olaf Langmack hat an der Definition der Grammatik mitgewirkt.

Denis Kuniß ist Diplom-Informatiker der Technischen Universität Berlin und bei der Firma Transformal in Berlin verantwortlich für Compilerbau. Er hat den PK-Parser geschrieben und an der Definition der Grammatik mitgewirkt.