High-Level-Programmiersprachen in den sechziger Jahren: Lisp und Algol 60

Jürgen Nickelsen

1 Historischer Kontext

Lisp und Algol 60 wurden beide Ende der fünfziger Jahre entwickelt. Zu dieser Zeit war die Programmierung von Computern zwar noch etwas ziemlich neues, aber es hatte sich trotzdem schon eine gewisses Umfeld auch im Bereich Programmiersprachen entwickelt.

1.1 Assembler und Präprozessoren

Nachdem am Anfang Computer nur in der reinen Maschinensprache programmiert wurden, entwickelten sich bald *Assemblersprachen*, in denen die einzelnen Instruktionen der Maschine durch für Menschen leichter verständliche Kürzel (*Mnemonics*) repräsentiert wurden.

Vor 1954 hatten die Computer noch keine Indexregister, was die Bearbeitung großer Gleichungssysteme sehr umständlich machte. Um dieses zu vereinfachen, wurden *algebraische Systeme* entwickelt, die einen Programmtext mit Index-Ausdrücken in Assemblertext umwandelten. Weiterentwicklungen dieser Systeme verarbeiteten algebraische Formeln (vermutlich etwa in der Art von FORTRAN-Ausdrücken) zu Assemblertext vor.

1.2 FORTRAN

Eine Sprache für „FORmula TRANslation“ wurde von John Backus innerhalb von IBM 1953/54 vorgeschlagen. Die erste als „benutzbar“ bezeichnete Version gab es 1957 mit FORTRAN III; erste Implementierungen außerhalb von IBM 1960. Ab 1962 versuchte man, einen gemeinsamen Standard zu schaffen, der 1966 mit FORTRAN IV erreicht wurde. (Die Weiterentwicklung von FORTRAN nach 1960 war zum Teil darauf zurückzuführen, daß Algol als Alternative nicht interessant schien.)

1.3 COBOL

Planungen zur COmmon Business Oriented Language begannen 1959. Wesentliche Ziele beim Sprachdesign waren Portabilität und die Einfachheit der Programmierung, zu fördern durch eine möglichst große Nähe zur englischen Sprache.

1.4 Weitere

In den fünfziger Jahren wurden noch einige weitere Programmiersprachen entwickelt, die uns nicht bis heute erhalten geblieben sind. Beispiele dafür sind IT, MATH-MATIC, FLOW-MATIC, AIMACO, COMTRAN, FLPL und IPL.

2 Algol 60

2.1 Ziele

Die „ALGOrithmic Language“ Algol sollte eine Programmiersprache sein, die alle bisherigen Entwicklungen ablöst. Man war sich darüber im klaren, daß die parallele Entwicklung von vielen rein lokal verbreiteten Sprachen, die gerade im akademischen Umfeld ein gewisses Ausmaß angenommen hatte, der als sehr wichtig eingeschätzten Portabilität von Programmen und damit dem Austausch von Ideen nicht förderlich war.

Numerische Probleme sollten so weit wie möglich in mathematischer Standardnotation beschrieben werden können. Andere Aspekte des Rechnereinsatzes als numerische Berechnungen befanden sich damals erst in der Entwicklung und waren noch nicht weit verbreitet; die am Standardisierungsprozeß von Algol 60 Beteiligten waren selbst hauptsächlich mit Numerik befaßt.

Algol sollte für die Beschreibung von Rechenverfahren in Publikationen gut geeignet sein. Deshalb wurde auf die Lesbarkeit der Sprache auch ohne Kommentare sowie auf Einfachheit und mathematische Eleganz besonderen Wert gelegt.

Die Sprache sollte sich gut in Maschinensprache übersetzen lassen. An dieser Stelle trat ein Konflikt zwischen der mathematisch orientierten Sicht von Funktionen und einem mehr berechnungsoriertierten Konzept von Prozeduren auf.

2.2 Entstehungsgeschichte

Die Entstehungsgeschichte von Algol 60 ist geprägt von Kommitees, Konferenzen, Bulletins und Diskussionen.

In Europa wurde seit 1954 an einer einheitlichen algorithmischen Sprache gearbeitet. Bei der deutschen Gesellschaft für angewandte Mathemetik und Mechanik (GAMM) wurde 1955 unter Beteiligung von verschiedenen europäischen Forschungseinrichtungen und Firmen ein Kommitee gegründet, das daran arbeitete.

1957, als die Arbeiten des GAMM-Kommitees schon nahezu abgeschlossen waren, kam die Idee zu einer weltweit einheitlichen Programmiersprache auf, die die bisher existierenden Sprachen ablösen sollte. Dementsprechend wurde der amerikanischen Association for Computing Machinery (ACM) vorgeschlagen, eine gemeinsame Konferenz zu diesem Thema abzuhalten.

Zur gleichen Zeit war eine Arbeitsgruppe bei der ACM, die sich auch mit der Planung einer einheitlichen Programmiersprache beschäftigte, zu dem Schluß gekommen, daß die zu schaffende Sprache eine „algebraische“, also in diesem Sinne FORTRAN-ähnliche, Hochsprache sein sollte, da ein einheitlicher Instruktionssatz für alle Computer als nicht erreichbar angesehen wurde.

FORTRAN selbst kam aber zu diesem Zweck nicht in Frage, da FORTRAN eine IBM-Erfindung war und nur auf IBM-Hardware zur Verfügung stand. Damals war noch nicht abzusehen, daß FORTRAN auch für andere Plattformen verfügbar sein würde.

Die von der GAMM vorgeschlagene Konferenz fand 1958 an der ETH Zürich statt. Als Diskussionsgrundlage für die weitere Arbeit wurde der Standard Algol 58 verabschiedet.

Obwohl Algol 58 eindeutig noch nicht als das Endprodukt der Bemühungen angesehen wurde, gab es schon bald eine Implementierung eines Compilers bei IBM. Diese Entwicklung wurde aber zugunsten von FORTRAN wieder eingestellt, da Algol 58 als Alternative noch nicht attraktiv genug schien.

Nachdem in der Zwischenzeit von beiden Gruppen weiter an Algol gearbeitet worden war, fand 1960 in Paris eine weitere Konferenz statt, in der der endgültige Standard Algol 60 verabschiedet wurde.

2.3 Aussehen der Sprache

Da die Entwicklung der imperativen Programmiersprachen von Algol 60 stark beeinflußt worden ist, können wir rückblickend sagen, daß Algol 60 zu vielen unserer heutigen Sprachen sehr ähnlich war.

Dieses sind die wesentlichen Merkmale von Algol 60:

- Es gibt einfache Variablen und Arrays der Typen `integer`, `real` und `boolean`. Jede Variable muß vor ihrem Gebrauch deklariert werden.
- Werte können als Konstanten deklariert werden.
- An Kontrollstrukturen gibt es `switch`, `for`, `while`, `until`, `if/else`.
- Statements können verschachtelt sein. Dadurch ist es möglich, folgendes zu schreiben:

```
if f then begin h := y ; z := w end
     else if h ≠ z then h := y
```

- Die Verschachtelung von Statements wird durch die Möglichkeit von Blöcken mit lokalen Variablendeklarationen unterstützt. Die Größe eines lokalen Arrays kann dabei vom Wert einer Variablen abhängen. Variablen haben einen lexikalischen Scope.
- Prozeduren haben formale Parameter und können rekursiv sein. Parameter werden per *call-by-value* oder *call-by-name* übergeben. Das unvermeidliche Fakultäts-Beispiel folgt:

```
integer procedure factorial(n);
  value n; integer n;
  begin
    if n = 0 then
      factorial := 1
    else
      begin integer i;
        i := factorial(n - 1) ;
        factorial := n * i;
      end
  end
```

Was fehlt:

- Operationen zur Verarbeitung von Zeichen und Zeichenketten
- Definition von Ein- und Ausgabeoperationen (Diese Operationen sollten der jeweiligen Hardware angepaßt definiert werden.)
- Möglichkeit zur dynamischen Speicherallozierung (außer lokalen Arrays variabler Größe)

- Modulkonzept
- höhere Datentypen
- abstrakte Datentypen

2.4 Weiterentwicklung

Eine Weiterentwicklung der Sprache führte zu Algol 68. Im Gegensatz zur Entwicklung von Algol 60 verlief der Standardisierungsprozeß hier sehr zäh. Implementierungen gab es nur wenige. Algol 68 wird aber von seinen Fans sehr hoch geschätzt.

Ein weiterer direkter Nachfolger von Algol 60 ist die Sprache Pascal, die von Niklaus Wirth entwickelt wurde, nachdem er sich von der weiteren Algol-Entwicklung wegen Meinungsverschiedenheiten abgewandt hatte. Im Gegensatz zu Algol 68 beschränken sich die Erweiterungen von Pascal gegenüber Algol 60 auf einige wenige Kernkonzepte, zu denen als wichtige Elemente ein mächtiges Typkonzept und dynamische Speicherverwaltung gehören.

Pascal war vor allem als Lehr- und Ausbildungssprache gedacht, verbreitete sich aber durch die populären Implementierungen UCSD-Pascal (von der University of California at San Diego) und Turbo-Pascal (von der Firma Borland) vor allem auf Mikrocomputern stark. Offensichtliche Mängel der Sprache, vor allem bei Ein-/Ausgabeoperationen, führten zu implementierungsspezifischen und gegenseitig inkompatiblen Erweiterungen.

Weiterhin hatte Algol 60, teilweise auf dem Umweg über Pascal, erheblichen Einfluß auf die Entwicklungen der Sprachen Simula, Ada, C, Modula 2 u.a.

3 Lisp

3.1 Ziele

Lisp war von vornherein für den praktischen Einsatz in der KI-Forschung vorgesehen. Hierbei kam es weniger auf effiziente numerische Operationen an, sondern mehr auf die Verarbeitung und Manipulation von symbolischen Ausdrücken. Um Strukturen aus der Logik und der natürlichen Sprache angemessen modellieren zu können, wurde auf die Verarbeitung von Listen besonderer Wert gelegt.

Die Sprache sollte einfach und mathematisch klar sein, teils aus ästhetischen Gründen, teils aus der Überlegung, daß es für eine kompakte und orthogonale Syntax leichter sein würde, Methoden für Korrektheitsbeweise zu entwickeln.

3.2 Entstehungsgeschichte

Auf der Basis von FLPL (FORTRAN List Processing Language, eine Erweiterung von FORTRAN für den Umgang mit Listen) entwickelte John McCarthy 1958 Konzepte für konditionale Ausdrücke, rekursive Funktionen und rekursive Manipulation von Listen. Um Funktionen als Argumente zu benutzen, verwendete er die Lambda-Notation von Church, benutzte aber dessen weitergehende Ideen nicht. (Church benutzte Funktionen höherer Ordnung statt konditionaler Ausdrücke.)

Für die Verwirklichung dieser Konzepte reichte FLPL als Basis nicht aus. Ursprünglich war vorgesehen, einen Lisp-Compiler zu entwickeln, aber da dieses Vorhaben als schwierig angesehen wurde, begann man zunächst, einzelne Funktionen per Hand zu kodieren. Als die *eval-*

Funktion fertiggestellt war, die Lisp-Ausdrücke auswertet, hatte man plötzlich einen Interpreter zur Verfügung. Das hatte einen starken Einfluß auf die Sprache, da nun eine Implementierung vorhanden war, mit der man experimentieren konnte.

Durch ein Versehen in der Implementierung wurde die Auswertung von Variablen mit *dynamischem Scope* durchgeführt, während eigentlich lexikalischer Scope gewünscht war. Das wurde zwar als ein Fehler betrachtet, aber trotzdem nicht umgestellt.

Für die interne Darstellung von Lisp-Ausdrücken als Lisp-Daten wurde eine Listenform gewählt, bei der im Gegensatz zur üblichen Infix-Notation bei mathematischen Ausdrücken der Operator an erster Stelle stand. Als „externe" Darstellung von Lisp-Programmen war ursprünglich eine andere Notation vorgesehen; diese wurde aber fallengelassen, weil sie erst noch in die interne Darstellung hätte kompiliert werden müssen und außerdem nie vollständig definiert worden war.

Die ursprüngliche Idee von „mathematisch reinen" Funktionen ohne Seiteneffekte wurde fallengelassen zugunsten imperativer Konstrukte, die das Programmieren vereinfachten. So kamen ein Konstrukt für sequentielle Auswertung von Ausdrücken und das GOTO[1] dazu.

Das erste „LISP Programmer's Manual" wurde 1960 veröffentlicht. Das 1962 erschienene „LISP 1.5 Programmer's Manual" hatte eine Reihe von Implementierungen auf verschiedenen Plattformen zur Folge; 1963 gab es den ersten interaktiv benutzbaren Interpreter.

Das bei Bolt Beranek and Newman entstandene *BBN-Lisp* (ab 1964, später *Interlisp*) brachte eine Reihe von neuen Ideen bei der Programmierung und Methodik von Lisp, vor allem im Bereich Tool Support.

Weiterentwicklungen am MIT führten über LISP 2 („an implementation in search of a language") Ende der sechziger Jahre zu *MacLisp*, das bis in die frühen achtziger Jahre das Rückgrat der KI-Forschung am MIT darstellte.

3.3 Aussehen der Sprache

Im Gegensatz zu Algol sind Lisp-Programme von denen der meisten heute verwendeten Programmiersprachen sehr verschieden. Das rührt daher, daß Lisp-Programme selbst in Datenstrukturen der Sprache notiert werden. Lisp-Programme bestehen aus *symbolischen Ausdrücken*. Ein symbolischer Ausdruck kann ein *Atom* sein oder eine *Liste* von symbolischen Ausdrücken. Ein Atom ist entweder eine Zahl oder sieht ungefähr so aus wie eine Variable in anderen Sprachen.

Bei der Auswertung von Ausdrücken wird ein Atom zu seinem Wert ausgewertet (ein Atom kann somit als Variable benutzt werden). Eine Liste wird entweder als *Funktion* oder als *Special Form* ausgewertet, wobei das erste Element der Liste angibt, um welche Funktion oder Special Form es sich handelt.

Bei der Auswertung einer Funktion werden die Argumente der Funktion zuerst ausgewertet (*Applicative Order*, strikte Semantik). Bei Special Forms werden die Argumente nicht ausgewertet; die Auswertung bleibt der entsprechenden Form überlassen (zum Beispiel bei konditionalen Ausdrücken und Funktionsdefinitionen). Die Auswertung von Ausdrücken kann unterbunden werden, wenn der Ausdruck literal verwendet werden soll (*Quoting*).

Lisp hat eine automatische Speicherverwaltung, die bei der Konstruktion von Listen Speicherelemente implizit alloziert. So allozierte Bereiche haben eine unbegrenzte Lebensdauer.

[1] Considered harmful

Bereiche, die vom Programm nicht mehr referenziert werden können, werden von der *Garbage Collection* erkannt und als freier Speicher neu zur Verfügung gestellt.

Es folgt wieder das unvermeidliche Fakultäts-Beispiel:

```
(defun fac (n)
  (cond ((zerop n) 1)
        (t (* n (fac (- n 1))))))
```

Hier ist `cond` die Special Form für den konditionalen Ausdruck, `zerop` ist der Vergleich mit Null, `t` das Literal für den Wahrheitswert TRUE.

Ein gutes Beispiel für den Umgang mit Listen ist die Funktion `map`, die eine Funktion auf jedes Element einer Liste anwendet und als Ergebnis eine Liste der erhaltenen Werte liefert.

```
(defun map (func l)
  (cond ((null l) ())
        (t (cons (apply func (list (car l)))
                 (map func (cdr l))))))
```

`null` liefert wahr für eine leere Liste; `car` liefert das erste Element einer Liste, `cdr` die Liste ohne das erste Element; `apply` wendet eine Funktion auf eine Liste von Argumenten an, `cons` konstruiert eine Liste aus dem ersten Element und einer Liste.

Eine Anwendung von `map` kann so aussehen:

```
(map '(lambda (n) (* 2 n)) '(1 2 3 4))
⟶ '(2 4 6 8)
```

Das Zeichen ' steht für Quote, der Ausdruck wird also nicht ausgewertet; `lambda` ist eine Special Form, die (entsprechend zum Lambda-Kalkül) eine anonyme Funktion konstruiert.

3.3.1 Was fehlt:

(Die folgenden Punkte beziehen sich auf die anfängliche Form von Lisp. Die hier genannten Konzepte wurden der Sprache in spaeteren Implementierungen hinzugefügt; in Common Lisp sind sie alle enthalten. In Scheme sind sie teilweise nicht direkt enthalten, können aber mit den zur Verfügung stehenden Sprachmitteln implementiert werden.)

- Modulkonzept
- Ein- und Ausgabeoperationen (Die Möglichkeit zur Eingabe in den Interpreter und die Ausgabe des Ergebnisses reichte zunächst)
- lexikalischer Scope
- höhere Datentypen außer der Liste
- abstrakte Datentypen

3.4 Weiterentwicklung

Zur Erprobung neuer Konzepte wurde ab 1975 *Scheme* als „mathematisch reiner" Lisp-Dialekt entwickelt, nach dem Vorbild von Algol 60 mit expliziter Blockstruktur und lexikalischem Scope. Scheme beschränkt sich bewußt auf wenige, aber sehr mächtige Konzepte (zum Beispiel Funktionen, Continuations und Environments als *first-class*-Objekte). Wichtiges Ziel waren stilistische Reinheit und Orthogonalität.

Die Vielzahl der Lisp-Dialekte führte 1980/81 zu dem Wunsch nach Vereinheitlichung. Gewünscht wurde ein stabiler Lisp-Dialekt für den Produktionseinsatz. Aus diesen Bemühungen entstand 1984 *Common Lisp*. Common Lisp ist im Gegensatz zu Scheme ein sehr umfangreiches System. Wichtige Merkmale sind:

- lexikalischer Scope
- Rückgabe mehrer Werte von Funktionen
- ausgefeilte numerische und Ein-/Ausgabeoperationen (darunter als kurioses Beispiel die Ausgabe von ganzzahligen Werten als römische Zahlen)

Scheme wurde 1991 von ANSI und IEEE standardisiert; ein ANSI-Standard für Common Lisp müßte gerade fertig geworden sein.

4 Gemeinsamkeiten von Algol 60 und Lisp

Sowohl Lisp als auch Algol 60 kennen eine Blockstruktur mit lokalen Variablen. Diese Blockstruktur ist explizit bei Algol 60, implizit bei Lisp durch Funktionen (die auch anonym sein können) mit formalen Parametern. Für Lisp war eigentlich lexikalischer Scope erwünscht, aber zunächst nicht implementiert worden.

Algol 60 und Lisp kennen beide Funktionen und Prozeduren mit formalen Parametern, die rekursiv sein können. Beide haben im Zusammenhang damit etwas obskure Konzepte implementiert; bei Algol 60 ist das call-by-name, bei Lisp dynamischer Scope.

5 Unterschiede

Sehr verschieden zwischen beiden Sprachen ist die Entstehungsgeschichte. Bei Algol 60 dominieren Konferenzen, internationale Kommitees, kursierende Bulletins; Lisp ist am Anfang geprägt durch die Bemühungen einzelner, schnelle Implementierungen und experimentelles Prototyping, dann durch Entwicklung von verschiedenen Gruppen in verschiedene Richtungen. Erst später gab es auch bei Lisp eine Standardisierung.

Während die Definition von Algol 60 bewußt unabhängig von (sowieso nicht vorhandenen) Implementierungen war, wurde Lisp de facto immer wieder durch Implementierungen definiert. Das hängt *auch* damit zusammen, daß Lisp sehr leicht zu implementieren ist (das sogar in dem Maße, daß die Implementierung eines Lisp-Interpreters als übungsaufgabe für Studenten beliebt ist).

Obwohl die Implementierung mathematischer Funktionen bei der Entwicklung von Algol 60 eine große Rolle gespielt hat, ist die Sprache doch durchweg prozedural. Lisp hat dagegen von Anfang an einen funktionalen Kern gehabt, auch wenn die funktionale Reinheit durch Konstrukte mit Seiteneffekten nicht gegeben ist. Es ist in Lisp allerdings (bis auf Ein- und Ausgabe) möglich, rein funktional zu programmieren.

6 Software-Engineering-Aspekte von Algol 60 und Lisp

Ein wichtiger Aspekt aus softwaretechnischer Sicht ist das bei beiden Sprachen fehlende Modulkonzept. Aus heutiger Sicht läßt uns diese Tatsache beide Sprachen in der Form der sechziger Jahre für große Projekte nicht mehr geeignet erscheinen.

Als besonders haarsträubend wird häufig die Möglichkeit von Lisp angesehen, selbstmodifizierenden Code zu produzieren. Das ist natürlich implementierungsabhängig, kann aber sogar so weit gehen, daß eine Funktion in der Lage ist, sich selbst zu modifizieren. Die Benutzung dieses „Features“ ist der Wartbarkeit von Programmen natürlich nicht zuträglich.

Beim Entwurf von Algol 60 wurde besonderer Wert auf gute Lesbarkeit gelegt. Das ist auch heute noch ein wichtiges Ziel beim Sprachentwurf; Algol 60 wirkt in dieser Hinsicht ausgesprochen modern.

Literatur

[AJF88] Anthony J. Field, Peter G. Harrison: *Functional Programming.* Addison-Wesley, 1988.

[Bac78] John Backus: The History of FORTRAN I, II, and III. *ACM SIGPLAN Notices*, 13(8), August 1978.

[GLSJ93] Guy L. Steele Jr., Richard P. Gabriel: The Evolution of Lisp. *ACM SIGPLAN Notices*, 28(3), März 1993.

[Lin93] C. H. Lindsey: A History of ALGOL 68. *ACM SIGPLAN Notices*, 28(3), März 1993.

[McC78] John McCarthy: History of LISP. *ACM SIGPLAN Notices*, 13(8), August 1978.

[Nau78] Peter Naur: The European Side of the Last Phase of the Development of ALGOL 60. *ACM SIGPLAN Notices*, 13(8), August 1978.

[Per78] Alan J. Perlis: The American Side of the Development of ALGOL. *ACM SIGPLAN Notices*, 13(8), August 1978.

[Rit93] Dennis M. Ritchie: The Development of the C Language. *ACM SIGPLAN Notices*, 28(3), März 1993.

[Sam78] Jean E. Sammet: The Early History of COBOL. *ACM SIGPLAN Notices*, 13(8), August 1978.

[Win93] Barbara Maren Winkler: Die Geschichte von FORTRAN. Referat im Seminar „Geschichte der Informatik“ an der Freien Universität Berlin, WS 92/92, Februar 1993.

[Wir93] Niklaus Wirth: Recollections About the Development of Pascal. *ACM SIGPLAN Notices*, 28(3), März 1993.