

Compilerbau: Projekt Turtle Graphics

Klaus Kusche

„Turtle Graphics“-Programmiersprachen werden oft für den Grundschul-Bereich vorgeschlagen, um Schülern einen ersten Eindruck von den Prinzipien des Programmierens zu geben.

Eine „Turtle-Graphics“-Programmiersprache dient dazu, Strichgrafiken am Bildschirm durch einfache Programme zu erzeugen. Die wesentliche Idee dabei ist, dass nicht absolute Pixelkoordinaten angegeben werden, um Striche zu zeichnen, sondern dass ein „Männchen“ (bzw. in den ursprünglichen Implementierungen eine „Turtle“, d.h. eine Schildkröte) programmgesteuert über den Bildschirm läuft und dabei eine Spur hinterlässt. Alle Befehle werden daher ausgehend von der aktuellen Position und der aktuellen Blickrichtung der Turtle ausgeführt.

Der zentrale interne Programmzustand besteht daher aus

- den aktuellen x- und y-Koordinaten
- und dem aktuellen Blickwinkel der Turtle.

Befehle:

1.) Zeichen-Befehle:

walk *expr* ... Gehe *expr* viele Schritte in die aktuelle Richtung,
zeichne dabei einen Strich

walk back *expr* ... Dasselbe, entgegen der aktuellen Richtung
(die interne Blickrichtung wird dabei nicht verändert)

jump *expr* und **jump back** *expr* ... Dasselbe, ohne zeichnen

walk home und **jump home** ... Setze Position und Richtung
auf den Ausgangspunkt (Koordinaten **0/0**, Richtung **0** = nach rechts)

turn left *expr* ... Drehe dich um *expr* Grad nach links (gegen den Uhrzeigersinn)

turn [right] *expr* ... Drehe dich um *expr* Grad nach rechts (im Uhrzeigersinn)

direction *expr* ... Stelle die Richtung auf *expr* Grad (wie in Mathe:

0 ist „geh nach rechts“, die positiven Werte laufen gegen den Uhrzeigersinn)

color *red, green, blue* ... Setze die Zeichen-Farbe
red, green und *blue* sind Werte von **0** bis **100**

clear ... Lösche den Schirm

stop ... Bleib stehen (das Programm tut nichts mehr, das Fenster bleibt offen)

finish ... Programmende (das Fenster geht zu)

path *name* [([*args*])] ... führe einen selbst definierten Pfad aus
(wie ein Funktionsaufruf einer Funktion ohne Returnwert)

2.) Zuweisungen und Rechenbefehle:

store *expr* **in** *var* ... Wert von *expr* in *var* speichern

add *expr* **to** *var* und **sub** *expr* **from** *var* ... wie += und -=

mul *var* **by** *expr* und **div** *var* **by** *expr* ... wie *= und /=

3.) Markierungs-Befehle:

mark ... aktuelle Position und Richtung speichern

walk mark und **jump mark** ... geh zur letzten gespeicherten Position,
drehe dich dann in die gespeicherte Richtung

Die Markierungen werden stack-artig gespeichert: Ein **walk mark** / **jump mark**
springt zur Position des zuletzt ausgeführten mark und entfernt diese Markierung,
sodass der nächste Sprung zur letzten davor gespeicherten Markierung springt.

Möchte man mehrmals zur selben Markierung springen,
muss unmittelbar nach dem Sprung ein neuerliches mark erfolgen.

4.) Schleifen, if's:

if cond then statements [else statements] endif ... wie üblich

do expr times statements done ... Zählschleife ohne Zählvariable

counter var from expr (to | downto) expr [step expr] do statements done
... die klassische Zählschleife mit Zählvariable

while cond do statements done

repeat statements until cond

... wie üblich.

Bei **while** ist *cond* eine „solange“-Bedingung (Schleifenende bei **false**),

bei **repeat** eine „bis“-Bedingung (Schleifenende bei **true**)

while wird eventuell gar nicht durchlaufen, **repeat** mindestens ein Mal

5.) Folge von Befehlen:

statements ist eine Aneinanderreihung von Befehlen aus 1. bis 5., ohne Trennzeichen
statements darf nicht leer sein (muss mindestens einen Befehl enthalten)

Funktionen, Hauptprogramm usw.:

6.) Pfad-Funktions-Definition *pathdef* (nur außerhalb von *statements* erlaubt!):

path name [([params])] statements endpath

Hat ein **path** keine Parameter, können sowohl Definition als auch Aufruf
wahlweise ohne () oder mit leeren () erfolgen.

Ein **path** hat keinen Returnwert.

7.) Rechen-Funktions-Definition *calcdef* (nur außerhalb von *statements* erlaubt!):

calculation name ([params]) [statements] returns expr endcalc

Eine **calculation** muss sowohl in der Definition als auch im Aufruf
immer () haben, auch wenn sie leer sind.

Alle Funktionen müssen einen Returnwert zurückgeben (immer eine Kommazahl),
dieser muss immer am Ende der Funktion nach **returns** angegeben werden.

Besteht die Funktion nur aus einer Rechnung, können die *statements* wegfallen.

8.) *params* ist eine Parameter-Liste:

Kein, ein oder mehrere Variablen-Namen *var* getrennt durch ,

Die Parameter einer Funktion müssen verschiedene Namen haben.

Alle Parameter und Argument-Werte sind Kommazahlen und werden „by value“ übergeben.

Ihr Wert darf in der Funktion verändert werden, sie verhalten sich wie lokale Variablen.

9.) Startsymbol der Sprache = **Inhalt des gesamten Source-Files: program**

{ pathdef | calcdef } begin statements end

Die *statements* stellen das Hauptprogramm dar und werden automatisch beim Start des Programms ausgeführt.

Die *pathdef* und *calcdef* dürfen davor in beliebiger Reihenfolge stehen, ein Pfad oder eine Funktion darf auch vor ihrer Definition aufgerufen werden.

Direkte und indirekte Rekursion bei Pfaden und Funktionen ist erlaubt!

Das Verhalten bei Endlos-Rekursionen ist undefiniert (muss nicht geprüft oder abgefangen werden).

Rechnungen, Bedingungen, Variablen usw.

10.) **Ausdrücke** *expr* sind Rechnungen wie üblich:

- Zahlen (optional mit Kommapunkt, aber ohne Zehnerpotenz-**E**)
- Variablen *var*
- Die Rechenzeichen + - * / ^ („hoch“) sowie ein unäres - mit den üblichen Vorrang-Regeln, ^ ist rechtsbindend
- Runde Klammern und Betragsstriche (Absolutwert)
- Einige vordefinierte Funktionen: **sin cos tan sqrt** sowie **rand(from, to) ... double-Zufallszahl** in den angegebenen Grenzen
- Aufruf eigener Funktionen: *name* ([*args*])

Alle Rechnungen erfolgen mit Kommazahlen (**double**).

Alle Winkel-Angaben werden in Grad gemacht sowohl für die **turn**-Befehle als auch für die Winkelfunktionen.

Die Orientierung ist wie in Mathe: **0** ist „nach rechts“, **90** ist „nach oben“.

11.) *cond* sind **Bedingungen**:

- Vergleiche mit > < >= <= = <>
- **and or not** mit den üblichen Vorrang-Regeln (**not** vor **and** vor **or**)
- Klammern

12.) *args* ist eine **Argument-Liste**:

Keine, eine oder mehrere Rechnungen *expr* getrennt durch ,

13.) **Variablenamen** *var* und **Funktionsnamen** *name*:

Sie sind wie in C definiert:

Buchstabe oder _ gefolgt von beliebig vielen Buchstaben, Ziffern oder _ .
Außerdem ist das @ erlaubt. An erster Stelle zeigt es an, dass es sich um eine globale bzw. vordefinierte Variable handelt.

Namen und Schlüsselworte sind case-sensitiv,

Schlüsselworte und vordefinierte Namen werden klein geschrieben.

Namen von Variablen, Funktionen oder Pfaden dürfen nicht gleich sein.

Verschiedene Funktionen oder Pfade dürfen aber gleichnamige lokale Variablen oder Parameter enthalten.

- 14.) **Variablen** werden durch die erste Zuweisung implizit deklariert, wenn es sie noch nicht gibt, je nach Name als lokale (ohne @) oder globale (mit @) Variablen:

Es ist möglich, selbst beliebige globale Variablen zu definieren, indem man einen Wert in eine neue Variable speichert, deren Name mit @ beginnt.

Alle Variablen sind Kommazahlen (**double**).

- 15.) **Vordefinierte Variablen** sind:

@dir ... Aktuelle Blickrichtung in Grad, im Bereich $0 \leq \text{@dir} < 360$ (nur lesbar)

@dist ... Aktuelle Entfernung vom Ausgangspunkt (nur lesbar)

@x und **@y** ... aktuelle x- und y-Koordinate relativ zum Ausgangspunkt (nur lesbar)

@1 bis **@9** ... **argv[1]** bis **argv[9]** als **double** (nur lesbar)

Auf der Befehlszeile nicht angegebene **@1** bis **@9** haben den Wert **0**.

@pi ... Die Konstante Pi (nur lesbar).

@max_x und **@max_y** ... maximale Werte für x- und y-Koordinaten

(gültiger Bereich von $-\text{@max_x}$ bis @max_x bzw. $-\text{@max_y}$ bis @max_y)

(auch schreibbar, ändert die Größe des sichtbaren Koordinatenbereichs)

@delay ... Wartezeit nach jedem Zeichen-Befehl, in Millisekunden

(auch schreibbar)

@red, **@green** und **@blue** ... RGB-Wert der Farbe, mit der gezeichnet wird

(Bereich **0** ... **100**, auch schreibbar)

- 16.) **White Space** (Zwischenräume, Tabs, Zeilenvorschübe) wird wie üblich ignoriert (abgesehen von seiner Aufgabe, Zahlen, Namen usw. zu beenden bzw. zu trennen), ebenso **Kommentare**, die mit " beginnen und bis zum Zeilenende gehen.

Außer bei Kommentaren haben Zeilenvorschübe und Einrückungen keine Bedeutung.

Implementierungshinweise Compiler

- Da unsere Sprache Schleifen und Funktionen enthält, wäre es unklug, die erkannten Programmbefehle gleich während der Syntaxanalyse auszuführen: Wir müssten dazu bei der Syntaxanalyse im Source-Code hin und her springen und Sourcecode-Teile wiederholt analysieren und ausführen.

Die Grundstruktur des Compilers ist daher, dass die Syntax-Analyse Syntax-Bäume liefert, die erst nach der kompletten Analyse des Sourcecodes interpretiert werden.

- Ich stelle den Syntaxbaum-Interpreter, das Hauptprogramm und viele Typdefinitionen zur Verfügung. Gefragt ist also nur der Lexer und der Parser.

Du darfst aber selbstverständlich auch meine Code-Teile erweitern und ersetzen, wenn du möchtest.

- Ob du den Compiler händisch mittels rekursiver Syntax-Analyse implementierst oder Lex+Yacc bzw. Flex+Bison verwendest, ist dir überlassen.

Hinweise zur Implementierung mit Lex+Yacc sammle ich in einem separaten Dokument.

- Auch bei einem handgeschriebenen Compiler ist sauber zwischen lexikalischer Analyse und Syntax-Analyse zu trennen! (d.h. der Parser soll Tokens verarbeiten und nicht direkt die Input-Zeichen)
- Als Ergebnis liefern bzw. befüllen Lexer und Parser primär zwei Datenstrukturen, die dann vom Interpreter verarbeitet bzw. benutzt werden:
 - Die **Syntaxbäume**.
 - Die **Namenstabelle**.
 Für Funktions- und Pfadnamen verweist diese Tabelle auf zusätzliche Strukturen mit den Parametern und dem Code der Funktion / des Pfades.

Genauer:

- Der Lexer verwendet die Namenstabelle zur Schlüsselwort-Erkennung (zumindest bei meinem handgeschriebenen Compiler, nicht bei Lex+Yacc), und er trägt alle neuen Namen in die Namenstabelle ein.
 Er soll unterschiedliche Token-Typen für jedes einzelne Schlüsselwort (ohne Verweis auf die Namenstabelle) und einen gemeinsamen Token-Typ mit Verweis auf den jeweiligen Namenseintrag für alle anderen Namen liefern.
- Erst der Parser trägt dann den genauen Namens-Typ (Variable, Funktion, ...) ein und hängt gegebenenfalls Funktionsdefinitions-Strukturen an die Namenseinträge in der Tabelle.
- Der Parser soll als Returnwert den Syntaxbaum des Hauptprogrammes liefern.
 Die Syntaxbäume des Codes von Funktionen und Pfaden werden beim Parsen in den Funktionsdefinitionen gespeichert die an der Namenstabelle hängen.
- Normalerweise verwendet ein Compiler zur effizienten Speicherung von Namen, Schlüsselwörtern usw. eine Hashtable (oder einen binären Suchbaum).
 Mein Code verwendet einfach ein lineares, ungeordnetes Array für die Speicherung aller Namen, Schlüsselwörter usw. Du darfst dieses Array beim Einlesen von Namen ganz primitiv sequentiell durchsuchen und neue Namen hinten anfügen.
 Das Array hat eine fixe Größe, die die maximale Anzahl von Namen limitiert, für die Namenstabelle wird also kein dynamisch angelegter Speicher verwendet (die Funktionsdefinitions-Strukturen werden aber schon dynamisch allokiert, ebenso die eigentlichen Namens-Strings).
 Der vordere Teil des Arrays ist bereits mit allen Schlüsselworten und vordefinierten Namen initialisiert.
 Im Syntaxbaum und bei der Ausführung werden alle Namen durch einen Pointer auf ihren Eintrag referenziert, nicht durch ihren Index im Array. Wenn du überschüssige Energien hast, könntest Du das Array also relativ problemlos durch eine bessere Datenstruktur ersetzen, ohne die Typdefinitionen oder den Interpreter aufwändig zu ändern.
- Die Syntaxbäume bestehen aus einzelnen Knoten, die einzeln dynamisch angelegt und durch Pointer verbunden werden. Es ist kein binärer Baum: Jeder Knoten enthält einerseits einen Next-Pointer und andererseits eine fixe Anzahl von Sohn-Pointern.

- Der **Next-Pointer** enthält bei den Befehlen einer Statement-Liste einen Zeiger auf den jeweils nächsten Befehl dieser Statement-Liste. Bei allen anderen Knotentypen ist er unbenutzt.
- Die **Sohn-Pointer** zeigen entweder auf die Syntaxbäume der Bestandteile eines Befehls oder der Operanden eines Operators oder die Argumente eines Funktions- oder Pfadaufrufes.

Welche Knotentypen es gibt und was genau die Bedeutung der Sohn-Pointer für jeden einzelnen Knotentyp ist, steht im Header mit den Typdefinitionen.

- Der Evaluator speichert die Werte der gerade aktiven lokalen Variablen und Parameter in einem einzigen Stack, der ebenfalls sequentiell nach dem Wert zu einer Variable durchsucht wird. Auch hier würde man bei einem professionellen Compiler eine effizientere Datenstruktur wählen.
- Der Compiler sollte alle unzulässigen Programme erkennen und eine Fehlermeldung ausgeben, aber es reicht eine ganz einfache Fehlerbehandlung.
- Tipp: Du musst nicht gleich den vollen Sprachumfang implementieren. Beginne mit ein paar Grundkonstrukten und erweitere deinen Compiler schrittweise!

Abgabe-Informationen

- Die Abgabe sollte bis Montag 12. Mai morgens elektronisch bei mir eingetroffen sein, und zwar eine Abgabe pro Gruppe (nicht pro Person).
- Wir haben am Di 15. April einen ganzen Tag Lehrveranstaltung. Ich würde mich sehr freuen, wenn jede Gruppe Ihren Ist-Stand präsentiert! (Vortrag + Demo)
- Wenn mit der Abgabe keine klare und eindeutige Zuordnung von jedem einzelnen Bestandteil der Lösung zu einem bestimmten Gruppenmitglied mitkommt, gibt es eine einheitliche Beurteilung für alle Mitglieder der Gruppe.
- Die Abgabe sollte enthalten:
 - Alle Sourcen (keine Exe-Files!).
 - Turtle-Graphics-Test- und Beispielprogramme.
 - Falls nötig: Eine Doku für alles,
 - was an der Lösung nicht selbsterklärend bzw. offensichtlich ist,
 - oder was von meinen Vorgaben abweicht (und warum).
 - Optional: Ganz toll wäre natürlich eine kurze Doxygen-Doku oder Ähnliches für alle selbst geschriebenen Funktionen usw..
 - Eine Build-Anleitung, falls der Build nicht offensichtlich bzw. trivial ist.
 - Ein kurzer Bericht zum Projekt und seinem Verlauf, beispielsweise:
 - Was erwies sich als besonders schwierig / arbeitsaufwändig, was ging schief, was ging unerwartet leicht? Zeitaufwände pro Projektteil?
 - Wo liegen besondere Features oder kreative Leistungen der jeweiligen Lösung? Enthält die Lösung Erweiterungen gegenüber der Aufgabenstellung? Was fehlt oder funktioniert unbefriedigend?
 - Welche Verbesserungen an der Turtle-Sprache, den Angaben und Hilfestellungen, und am bereitgestellten Code usw. wären sinnvoll? Was war unklar, sind Fehler oder Defizite in meinem Code? Was war besonders hilfreich bzw. nützlich?
 - Was hat Freude gemacht?