Compilerbau: 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.

Unsere Sprache:

1.) **Zeichen-Befehle:**

`walk` *expr* … Gehe *expr* Schritte in die aktuelle Richtung,
zeichne dabei einen Strich
`walk back` *expr* … Dasselbe, entgegen der aktuellen Richtung
`jump` *expr* und **`jump back`** *expr* … Dasselbe, ohne zeichnen
`turn left` *expr* … Drehe dich um *expr* Grad nach links
`turn` [**`right`**] *expr* … Drehe dich um *expr* Grad nach rechts
`direction` *expr* … Stelle die Richtung auf *expr* Grad (wie in Mathe:
`0` ist *„geh nach rechts“*, *positive* Werte drehen *gegen den Uhrzeigersinn*)
`clear` … Lösche den Schirm
`walk home` und **`jump home`** … Setze Position und Richtung
auf den Ausgangspunkt (Koordinaten **`0/0`**, Richtung 0 = nach oben)
`stop` … Bleib stehen (das Programm tut nichts mehr, Fenster bleibt offen)
`finish` … Programmende (Fenster geht zu)
`path` *name* [**`(`** [*args*] **`)`**] … führe einen selbst definierten Weg aus

2.) **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* (stack-artig)
`walk mark` und **`jump mark`** … geh *zur letzten gespeicherten Position*,
drehe dich dann in die gespeicherte Richtung

Markierungen, die im aktuellen Statement-Block gesetzt werden, werden *gelöscht*, wenn man den aktuellen *Statement-Block verlässt*. (wird noch genauer definiert)

4.) **Bedingungen:**

`if` *cond* **`then`** *statements* [**`else`** *statements*] **`endif`**

5.) **Schleifen:**

`do` *expr* **`times`** *statements* **`done`** … Zählschleife ohne Zählvariable
`counter` *var* **`from`** *expr* (**`to`** | **`downto`**) *expr* [**`step`** *expr*] **`do`** *statements* **`done`**
`while` *cond* **`do`** *statements* **`done`**
`repeat` *statements* **`until`** *cond*

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

`path` *name* [**`(`** [*params*] **`)`**] *statements* **`endpath`**

Rekursion ist erlaubt!

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

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

Rekursion ist erlaubt!

8.) **Source-File und Hauptprogramm**

{ *pathdef* | *calcdef* } **`begin`** *statements* **`end`**

9.) *statements* ist eine **Folge von Befehlen**, *ohne* Trennzeichen

10.) *params* ist eine **Parameter-Liste**:
Keine, eine oder mehrere Variablen-Namen *var* , getrennt durch **`,`**

Alle Parameter und Argument-Werte sind **`double`**
und werden *„by value“* übergeben.

Hat ein *Path* *keine* Parameter, können sowohl *Definition* als auch *Aufruf* wahlweise *ohne* **`( )`** oder mit *leeren* **`( )`** erfolgen. Eine *Calculation* muss in Definition und Aufruf *immer* **`( )`** haben, auch wenn sie leer sind.

11.) *args* ist eine **Argument-Liste**:
Keine, eine oder mehrere Rechnungen *expr* , getrennt durch **`,`**

12.) **Ausdrücke** *expr* sind Rechnungen wie üblich:
- *Zahlen* (optional mit Kommapunkt, aber ohne Zehnerpotenz-**E**)
- *Variablen* *var*
- Die *Rechenzeichen* **`+ - * / ^`** („hoch“) sowie ein unäres **`-`**
- *Runde Klammern* und *Betragsstriche*
- Einige *vordefinierte Funktionen*: **`sin cos tan sqrt`**
  sowie **`rand(`***from*`,` *to*`)`** … **`double`**-*Zufallszahl* in den angegebenen Grenzen
- Aufruf *eigener Funktionen* *name* **`(`** [*args*] **`)`**

Alle Rechnungen liefern **`double`**-Werte.

Alle *Winkel*-Angaben werden *in Grad* gemacht (Orientierung wie in Mathe), sowohl für die **`turn`**-Befehle als auch für die *Winkelfunktionen*.

13.) *cond* sind **Bedingungen**:
- *Vergleiche* mit **`> < >= <= = <>`**
- **`and or not`** mit den üblichen *Vorrang*-Regeln
- *Klammern*

14.) **Variablennamen** *var* und Funktionsnamen *name* seien wie in C definiert:
Buchstabe oder **_** gefolgt von beliebig vielen Buchstaben, Ziffern oder **_** .
Außerdem sei an erster Stelle das **@** erlaubt,
für *globale bzw. vordefinierte* Variablen.

Namen und Schlüsselworte sind *case-sensitiv*,
Schlüsselworte und vordefinierte Namen werden *klein* geschrieben.

15.) **Variablen** werden durch die erste Zuweisung *implizit deklariert*,
je nach Name als lokale oder globale Variablen. Alle Variablen sind **`double`** .

16.) **Vordefinierte Variablen** sind:

`@dir` ... Aktuelle *Blickrichtung* in Grad (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)
`@max_x` und **`@max_y`** ... *maximale Werte* für x- und y-Koordinaten (auch schreibbar)
(gültiger Bereich von **`-@max_x`** bis **`@max_x`**)
`@delay` ... *Wartezeit* nach jedem Zeichen-Befehl, in Millisekunden
(auch schreibbar)

17.) **White Space** 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.

Implementierungshinweise Compiler

- Ob du den Compiler *händisch* mittels rekursiver Syntax-Analyse implementierst
oder *Lex+Yacc* bzw. Flex+Bison verwendest, ist dir überlassen.
- Auch bei einem handgeschriebenen Compiler ist sauber
zwischen lexikalischer Analyse und Syntax-Analyse zu trennen!
- Normalerweise verwendet ein Compiler zur effizienten *Speicherung von Namen*,
Schlüsselworten usw. eine *Hashtable*.

  Ich freue mich natürlich über eine Hashtable, aber am Anfang darfst Du einfach
ein *lineares, ungeordnetes Array von Strings* für die Namensspeicherung und
Namenssuche verwenden und dann den *Index oder Pointer* in dieses Array
zur weiteren Identifizierung des Namens benutzen.

  Der Lexer sollte *unterschiedliche Token* für jedes einzelne *Schlüsselwort*
und einen *gemeinsamen Token-Typ* für *alle anderen Namen* liefern.
- Da unsere Sprache *Schleifen und Funktionen* enthält, wäre es unklug,
die erkannten Programmbefehle gleich in der Syntaxanalyse auszuführen:
Wir müssten dazu bei der Syntaxanalyse im Source-Code hin und her springen
und Sourcecode-Teile wiederholt analysieren und ausführen.

  Das Ergebnis der Syntax-Analyse soll daher ein *Syntax-Baum* sein,
der *erst nach der kompletten Analyse* des Sourcecodes ausgeführt
bzw. interpretiert wird.
- Der Compiler sollte *alle unzulässigen Programme erkennen* und
eine *Fehlermeldung* ausgeben, aber es reicht eine ganz einfache Fehlerbehandlung.

Implementierungshinweise Ausführung:

- Verwende zur Anzeige *meine SDL-Grafik-Funktionen*.
- Die *internen Koordinaten* und der *Blickwinkel* der Turtle sind **`double`**-Werte.

  Eine *Umrechnung* auf Pixel-Koordinaten erfolgt *erst unmittelbar beim Zeichnen*, wobei **`0/0`** *in der Mitte* des Grafik-Fensters ist (*positive* y-Achse nach *oben*).
- Nach jedem echten Zeichenbefehl soll das Programm **`@delay`** Millisekunden *warten*.