STANISŁAW WALIGÓRSKI

LOGO
NA SINCLAIR SPECTRUM

CZĘŚĆ 2 • dla zaawansowanych

INSTYTUT WYDAWNICZY ZWIĄZKÓW ZAWODOWYCH
WARSZAWA 1987

Książka została wydana we współpracy z Redakcją
miesięcznika matematyczno-fizyczno-astronomicznego „Delta".
Konsultacja z ramienia „Delty" *Zbigniew Odrowąż-Sypniewski*

Projekt graficzny okładki, strony tytułowej,
układ graficzno-typograficzny tekstu
oraz wykonanie ilustracji *Witold Gidzewicz*

Zdjęcia na okładkę wykonał *Adam Płuciennik*

Redaktor Wydawnictwa *Tomasz Majchrowski*

Redaktor techniczny *Barbara Charewicz*

Korekta *Zespół*

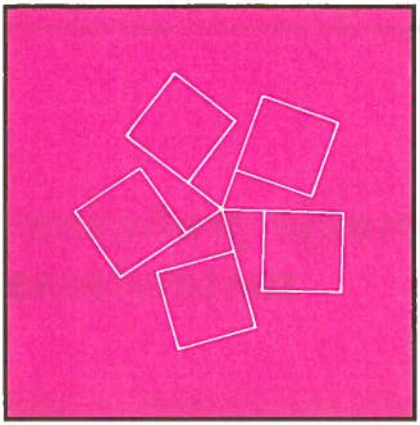
ISBN 83-202-0542-5

INSTYTUT WYDAWNICZY ZWIĄZKÓW ZAWODOWYCH
WARSZAWA 1987
Wydanie I. Nakład 49 700+300 egz.
Ark. wyd. 5,1. Ark. druk. 4,5.
Papier offset kl. III, 80 g. A1.
Oddano do składania 3 IX 1986 r.
Podpisano do druku w lipcu 1987 r.
Druk zakończono w sierpniu 1987 r.
Nr. prod. Ww/1043/KE/86. Zam. 480/86. K-28
DRUKARNIA INSTYTUTU WYDAWNICZEGO
ZWIĄZKÓW ZAWODOWYCH

Komendy i operacje graficzne: zestawienie

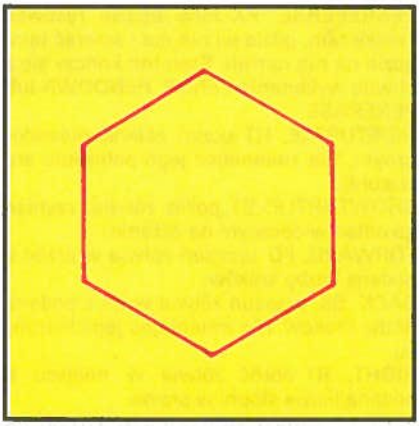
Pierwsza część książki była poświęcona przede wszystkim metodom tworzenia rysunków i dlatego większość użytych w niej procedur była związana z rysowaniem na ekranie. Dla przypomnienia wprowadzonych w pierwszej części komend i operacji graficznych Logo ujmiemy je tu w jednym zestawieniu, uzupełnionym jeszcze kilkoma, które nie były dotychczas użyte. Otrzymamy w ten sposób przegląd wszystkich komend i operacji graficznych Sinclair Logo. Obok pełnej nazwy podany jest skrót (o ile jest) i potem krótkie objaśnienie. Rysunki ilustrujące tę część tekstu (rys. 1–8) są niewielkimi modyfikacjami rysunków z I części – Czytelnik może spróbować narysować je w tej wersji samodzielnie. Ci, którzy nie pamiętają, jak się tworzy procedury, korzysta z REPEAT lub innych konstrukcji, mogą znaleźć odpowiednie wzory na dalszych stronach tej książki.

CLEARSCREEN, CS oczyść ekran, ustaw żółwia w środku ekranu i skieruj go w górę.
CLEAN oczyść ekran, nie zmieniając położenia żółwia.
TEXTSCREEN, TS oczyść ekran, przeznaczając go w całości na teksty. Użycie komendy lub operacji dotyczącej żółwia lub rysowania spowoduje powrót do sytuacji jak po CLEARSCREEN.
SETBG przyjmij kolor o podanym numerze jako kolor tła rysunku.
SETPC przyjmij kolor o podanym numerze jako kolor rysowanej kreski oraz wizerunku żółwia.
SETBORDER, SETBR przyjmij kolor o podanym numerze jako kolor ramki rysunku.
SETTC przyjmij podaną parę liczb, ujętą w nawiasy kwadratowe, jako numer koloru tła znaków (pierwsza z liczb) oraz numer koloru samych znaków (druga liczba), które będą pisane na ekranie po wykonaniu tej komendy.
PENDOWN, PD opuść pisak żółwia tak, aby ten, poruszając się, zostawiał ślad na ekranie. Kolor kreski jest określony przez ostatnio wykonaną komendę SETPC.
PENUP, PU podnieś pisak żółwia tak, aby jego przesuwanie nie powodowało rysowania kreski na ekranie.
PENERASE, PE po wykonaniu tej komendy żółw będzie ścierać kreskę, po której będzie się poruszać. Stan ten kończy się z chwilą wykonania PENUP, PENDOWN albo PENREVERSE.
PENREVERSE, PX żółw będzie rysować kreskę tam, gdzie jej nie ma i ścierać tam, gdzie na nią natrafi. Stan ten kończy się z chwilą wykonania PENUP, PENDOWN lub PENERASE.
HIDETURTLE, HT uczyń żółwia niewidocznym, nie zmieniając jego położenia ani kierunku.
SHOWTURTLE, ST pokaż żółwia, czyniąc go odtąd widocznym na ekranie.
FORWARD, FD przesuń żółwia w przód o podaną liczbę kroków.
BACK. BK przesuń żółwia w tył o podaną liczbę kroków, nie zmieniając jego kierunku.
RIGHT, RT obróć żółwia w miejscu o podaną liczbę stopni w prawo.
LEFT, LT obróć żółwia w miejscu o podaną liczbę stopni w lewo.
HOME przesuń żółwia do środka ekranu,



Rys. 1. Pęk chorągiewek

czyli do początku układu współrzędnych i ustaw go pionowo, czyli pod kątem 0°.
SETPOS przesuń żółwia do miejsca o podanych współrzędnych. Wartością argumentu jest ujęta w nawiasy kwadratowe para liczb – pierwsza jest wartością współrzędnej poziomej x, druga wartością współrzędnej pionowej y miejsca, w którym ma się znaleźć żółw.
SETX przesuń żółwia poziomo do miejsca o podanej współrzędnej x, nie zmieniając jego współrzędnej y ani kierunku.
SETY przesuń żółwia pionowo do miejsca o podanej współrzędnej y, nie zmieniając jego współrzędnej x ani kierunku.
SETHEADING, SETH ustaw żółwia pod podanym kątem, nie zmieniając jego położenia.
SETCURSOR, SETCUR przenieś wskaźnik początku tekstu drukowanego na ekranie do miejsca o podanych współrzędnych tekstowych. Wartością argumentu jest para liczb ujęta w nawiasy kwadratowe: numery, kolumny: wiersze. Położenie i kierunek żółwia – bez zmian.
DOT umieść na ekranie kropkę w miejscu o podanych współrzędnych. Wartością argumentu jest para liczb ujęta w nawiasy kwadratowe – pierwsza jest wartością współrzędnej x, druga wartością współrzędnej y stawianej kropki. Położenie i kierunek żółwia pozostają nie zmienione.
WRAP przejdź do sposobu rysowania, w którym żółw przekraczający brzeg ekranu pojawia się natychmiast na brzegu przeciwległym, zachowując swój kierunek, tak jakby brzegi przeciwległe – górny i dolny oraz lewy i prawy – były sklejone ze sobą parami.
FENCE uniemożliw wychodzenie żółwia poza brzeg ekranu – każda taka próba będzie nieskuteczna i spowoduje sygnalizowanie błędu.
WINDOW przejdź do sposobu rysowania, w którym ekran jest traktowany jako część płaszczyzny, po której porusza się żółw – jego wyjście poza brzeg ekranu jest tylko



Rys. 2. Sześciokąt, który można przesuwać i obracać

Rys. 3. Sześć sześciokątów

zniknięciem z pola widzenia, do którego może swobodnie wrócić, jeśli się odpowiednio pokieruje jego ruchem, a żadna współrzędna nie przekroczyła 32767.

POSITION, POS operacja, której wynikiem jest para liczb, będących aktualnymi współrzędnymi x i y żółwia, ujęta w nawiasy kwadratowe.

XCOR operacja wyznaczająca aktualną współrzędną x żółwia.

YCOR operacja wyznaczająca aktualną współrzędną y żółwia.

HEADING aktualny kąt żółwia.

PENCOLOUR, PC numer aktualnego koloru kreski, rysowanej przez żółwia i koloru jego wizerunku.

BACKGROUND, BG numer koloru tła.

CURSOR ujęta w nawiasy kwadratowe para liczb, będących współrzędnymi tekstowymi wskaźnika początku tekstu.

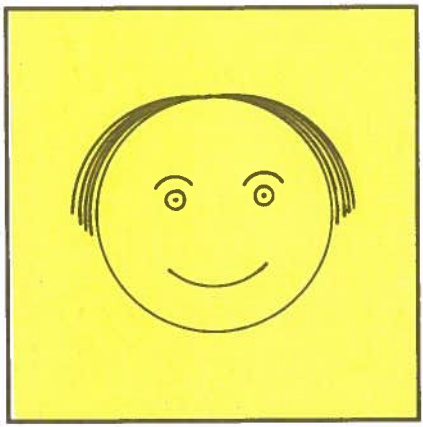
TC ujęta w nawiasy kwadratowe para liczb: numer koloru tła znaków i numer koloru znaków tekstu.

TOWARDS operacja podająca kąt, pod jakim ma się ustawić żółw, gdybyśmy chcieli skierować go na punkt, którego współrzędne, ujęte w nawiasy kwadratowe, są argumentem tej operacji.

Liczby, działania na liczbach i relacje w zbiorze liczb

Liczby w Logo mogą być całkowite albo rzeczywiste, w zapisie stało- lub zmiennopozycyjnym. Sens tych nazw, używanych we wszystkich językach programowania, stanie się jasny, gdy przyjrzymy się, jak liczby są zapisywane i jak należy rozumieć te zapisy.

Rys. 4. Osiem ośmiokątów



Rys. 5. Rysunek z okręgów i ćwierćokręgów rysowanych w prawo i w lewo

Liczba całkowita w zapisie stałopozycyjnym jest to ciąg cyfr, który ewentualnie może być poprzedzony znakiem —. Sinclair Logo nie dopuszcza poprzedzania liczby znakiem +. Liczba rzeczywista w zapisie stałopozycyjnym różni się tym od liczby całkowitej, że występuje w niej kropka dziesiętna. Zapis liczby całkowitej (rzeczywistej) w wersji zmiennopozycyjnej składa się z następujących części: cechy, która jest liczbą całkowitą (rzeczywistą) w zapisie stałopozycyjnym, następującej bezpośrednio po niej litery E (albo e, wszystko jedno) oraz następującej po tej literze mantysy, która jest liczbą całkowitą. Symbol E z następującą po nim liczbą oznacza mnożenie przez 10 podniesione do tej potęgi. Tak na przykład 47E2 oznacza to samo co 4700, —2.714E6 oznacza —2714000, zaś 3123.7E—3 jest tym samym, co 3.1237. Znaki liczby nie mogą być oddzielane od siebie odstępami ani przecinkami.

Znaki czterech działań arytmetycznych są takie same, jak w większości języków pro-

gramowania: + − * / i oznaczają odpowiednio: dodawanie, odejmowanie, mnożenie i dzielenie. Znak − może być także używany jako operator jednoargumentowy − umieszczenie go bezpośrednio przed liczbą lub zmienną o wartości liczbowej oznacza pomnożenie jej przez −1. Taki operator jednoargumentowy nie może być oddzielony od swego argumentu odstępem. Błędem byłoby napisać − 2 zamiast −2 albo − :X zamiast −:X. Co prawda Sinclair Logo odsuwa w pewnych przypadkach ten minus od swego argumentu (w procedurach zapisywanych w pamięci), ale programista sam nie powinien tego naśladować, bo zbyt wiele jest możliwych sytuacji, w których to mogłoby prowadzić do poważnych nieporozumień. Natomiast − jako znak odejmowania należy zawsze oddzielać od obu swoich argumentów odstępami. Zwłaszcza zgubienie odstępu z prawej strony znaku − jest szkodliwe, gdyż sugeruje, że chodzi o mnożenie przez −1 tej liczby, a nie odejmowanie. Dla jednolitości przyjmuje się zasadę, że wszystkie cztery operatory (dodawania, odejmowania, mnożenia i dzielenia) umieszcza się pomiędzy odstępami.

Oprócz tych operatorów mamy jeszcze następujące działania na liczbach:

SUM operacja 2- lub więcej argumentowa, obliczająca wartość sumy swoich argumentów. Jeśli liczba argumentów jest większa niż 2, to należy słowo SUM razem ze wszystkimi argumentami ująć w nawiasy okrągłe. Na przykład: wartością SUM 2 3 jest 5, wartością (SUM 2 2 7) jest 11, ale gdybyśmy napisali SUM 4 6 9 bez nawiasów, to oznaczałoby to dwie liczby, napisane obok siebie, 10 i 9, bo operacja SUM bez nawiasów jest dwuargumentowa.

Takie operacje i komendy, które w zasadzie mają określoną liczbę argumentów, ale ujęte w nawiasy okrągłe zaczynają traktować jako swoje argumenty wszystko to, co jest wewnątrz tych nawiastów, nazywane są w Logo zachłannymi. Spotkamy jeszcze kilka przykładów takich zachłannych procedur: PRODUCT, PRINT, SENTENCE i inne.

PRODUCT operacja 2- lub więcej argumentowa (zachłanna) obliczająca iloczyn wartości swoich argumentów. Na przykład PRODUCT 6 4 daje 24, (PRODUCT 2 3 4 8) daje 192, ale gdyby nie było tych nawiasów, byłyby to trzy liczby zapisane obok siebie: 6 4 8.

DIV operacja dwuargumentowa obliczająca iloraz pierwszego argumentu przez drugi. Jeśli wartość drugiego argumentu jest równa 0, jest sygnalizowany błąd.

REMAINDER operacja dwuargumentowa – jeśli wartości jej argumentów są całkowite, to wyznacza resztę z dzielenia pierwszego przez drugi.

INT operacja jednoargumentowa, wyznaczająca część całkowitą wartości swego argumentu.

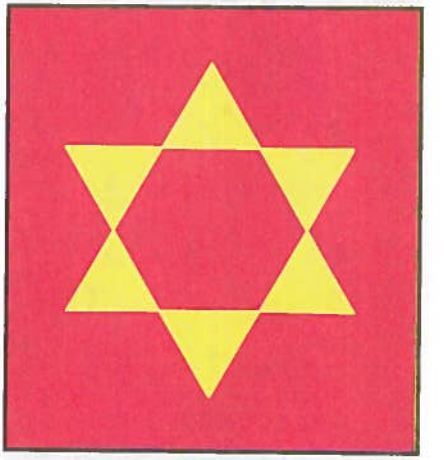
ROUND operacja jednoargumentowa, zaokrąglająca do najbliższej liczby całkowitej.

RANDOM operacja jednoargumentowa. Jeśli wartość argumentu jest liczbą całkowitą dodatnią, to operacja ta wyznacza liczbę, wylosowaną w równomiernym rozkładzie prawdopodobieństwa z przedziału pomiędzy 0 a wartością argumentu pomniejszoną o 1. Na przykład RANDOM 3 powoduje wylosowanie liczby spośród 0, 1, 2.

SQRT pierwiastek kwadratowy. Jeśli wartość argumentu jest ujemna, sygnalizowany jest błąd.

SINE, SIN sinus.

COSINE, COS cosinus.

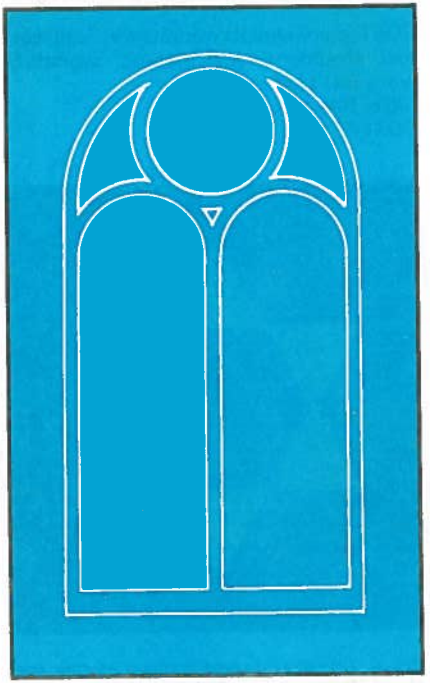


Rys. 6. Czy zamalowanie każdego trójkąta kreskami kładzionymi obok siebie wymaga jakichś obliczeń?

TANGENT, TAN tangens.
COTANGENT, COT cotangens.
ARCSIN arcsin.
ARCCOS arccos.
ARCTAN arctg.
ARCCOT arcctg.

Przy wykonywaniu operacji na liczbach obowiązują następujące zasady pierwszeństwa: jeżeli układ nawiasów nie wyznacza innej kolejności, to najpierw wykonuje się wszystkie dzielenia, potem mnożenia, potem odejmowania, potem dodawania i na końcu inne operacje. Jeśli występują nawiasy, to mogą one określić kolejność wykonywania działań w zwykły sposób. W wyrażeniach opisujących działania na liczbach można używać tylko nawiasów okrągłych: (i).

Na przykład: 6 / 3 * 4 jest zgodnie z tymi zasadami równe 2 * 4, a nie 6 / 12. PRODUCT 2 3 + 8 daje 22, bo dodawanie ma pierwszeństwo przed operacją o nazwie literowej. Gdybyśmy jednak napisali PRODUCT 2 3 8, byłaby to po obliczeniu iloczynu para napisanych obok siebie liczb: 6 8. Wobec tego SUM PRODUCT 2 3 8 jest wyrażeniem poprawnym i daje w wyniku 14. Pamiętając, że SUM i PRODUCT są zachłanne, możemy zbadać skutki różnego rozmieszczania nawiasów: (SUM PRODUCT 6 4 2 10) jest równa 6 * 4 + 2 + 10, bo PRODUCT bez nawiasów jest dwuargumentowy, natomiast SUM (PRODUCT 6 4 2) 10 jest równa 6 * 4 * 2 + 10. Dalej, SIN : X*2 należy odczytywać jako SIN (:X * 2), czyli sinus podwojonej wartości zmiennej :X. Gdybyśmy chcieli wziąć dwukrotną wartość sinusa tej zmiennej, trzeba by napisać (SIN :X) * 2 albo 2 * SIN :X.

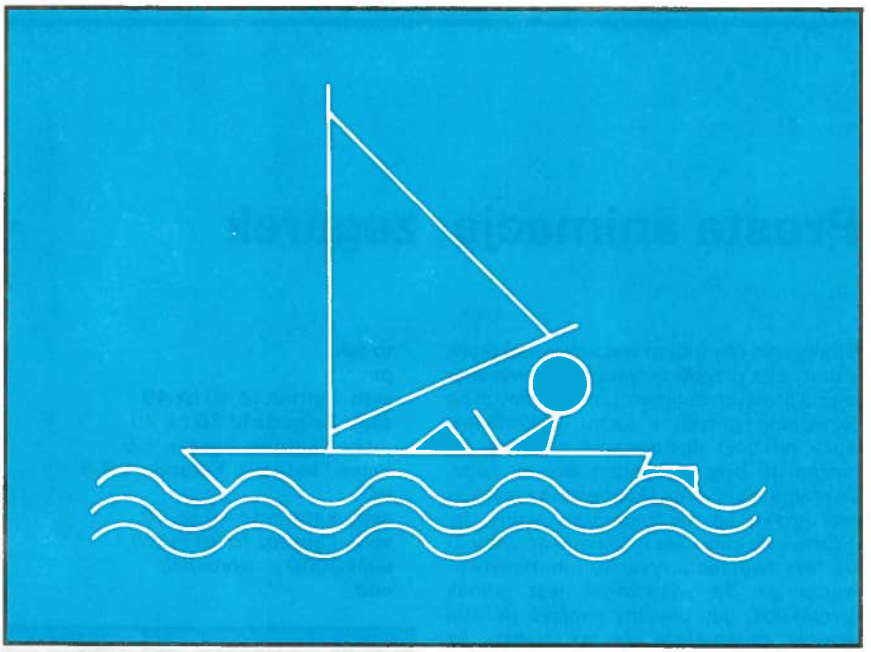


Rys. 7. Tu trzeba będzie obliczyć niektóre kąty

Przy pewnym doświadczeniu posługiwanie się tymi zasadami nie jest trudne, ale dobrze jest kierować się przede wszystkim naczelną zasadą: dbać o czytelność zapisów. Jeżeli więc wstawienie nawiasów ułatwi odczytywanie wyrażenia lub pomoże usunąć wątpliwości co do kolejności wykonania działań, to należy je wstawić.

Nawiasy mogą obejmować każde poprawnie zbudowane wyrażenie: dowolną liczbę, zmienną albo symbol operacji razem z zapisanymi poprawnie argumentami, które też są wyrażeniami. Jeżeli używamy wieloargumentowego operatora różnego od: + − * /, to pisze się go przed sekwencją jego argumentów: na przykład DIV 215 33, i wtedy można ująć w nawiasy całe to wyrażenie, to znaczy napisać (DIV 215 33). Byłoby błędem ująć w nawiasy tylko same argumenty na podobieństwo używanej w matematyce notacji f (x, y) – tutaj operator musi być zawsze wewnątrz nawiasów. Nie jest sprzeczne z tą zasadą ujmowanie w nawiasy argumentu operacji jednoargumentowej, które samo jest wyrażeniem, np. COS (:X). Można się bez tych nawiasów obejść, ale one nie szkodzą. Za to zdecydowanie niepoprawnym byłoby napisanie SUM (:X / :Y 1), bo nawiasy obejmują tu parę napisanych obok siebie wyrażeń, a nie jedno wyrażenie, i oddzielają argumenty od operatora wieloargumentowego, czego nie należy robić. Logo nie zrozumie, dlaczego właśnie te dwa wyrażenia zostały poddane takiej operacji.

Mając dwie liczby lub, ogólniej, dwa prawidłowo zbudowane wyrażenia przyjmu-



Rys. 8. Ustalenie położenia fal względem siebie wymaga pewnych obliczeń (por. cz. I)

jące wartości liczbowe, można je porównać stosując znaki < (mniejsze), = (równe) lub > (większe). Znaków odpowiadających słabym nierównościom nie stosuje się. Jeżeli dwa dowolne prawidłowo zbudowane wyrażenia, przyjmujące wartości liczbowe napiszemy obok siebie przedzielając jednym z tych znaków, to otrzymamy wyrażenie, przyjmujące wartość TRUE (prawda) lub FALSE (fałsz) zależnie od wyniku porównania. Oto parę przykładów – obok wyrażeń są zapisane ich wartości:

3 < 7	TRUE.
1.2293 < −6.2E2	FALSE.
0.628E2 = 2 * 31.4E−1	TRUE.
:X * :X + 1 > 0	TRUE, o ile zmienna :X ma wartość liczbową.
RANDOM 3 > 0	TRUE, o ile została wylosowana liczba dodatnia (z prawdopodobieństwem 2/3), lub FALSE, jeśli zostało wylosowane 0 (z prawdopodobieństwem 1/3).

Prosta animacja: zegarek

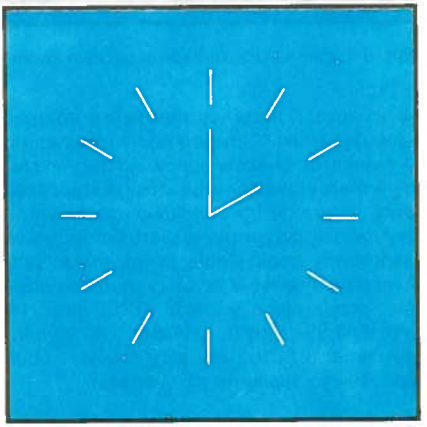
Rysowanie zegarka ze wskazówkami może służyć jako przykład wymagający wykonywania prostych obliczeń, pozwalających na określenie pozycji i ruchu wskazówek. Ruch ten jest dostatecznie powolny, by można go było zaprogramować w Logo. Operacje rysowania i kasowania kresek w tym języku są tak zaprogramowane, by można było cały czas obserwować wszystkie fazy tworzenia rysunku lub modyfikowania go. Ta właściwość jest jednak przeszkodą, gdy chcemy tworzyć złożone rysunki zmieniające się tak szybko, by stworzyć wrażenie ruchu. Poprzestańmy więc na prostym przykładzie*, w którym powolność procesu rysowania i kasowania linii nie jest specjalnym utrudnieniem:

```
to zegar :godz :min :szybkość
cs ht pu
repeat 12 [fd 50 pd fd 10 pu bk 60 rt 30]
make "kątmin :min * 6
make "kątgodz : godz * 30 + :min * 0.5
repeat 1000 [tyk]
st
end
```

Najpierw rysuje się samą tarczę zegara z równomiernie rozmieszczonymi kreskami oznaczającymi godziny, potem określa się początkowe kąty wskazówki minutowej i godzinowej (rys. 9) i uruchamia zegar na 1000 tyknięć, każde trwające jedną minutę czasu tego zegara – jego szybkość można regulować.

* W tej części książki rezygnujemy z zaznaczania przenoszonych wierszy znakiem! Wiersze te kontynuowane są bez stosowania akapitu, co jest równie czytelne.

```
to tyk
pe
seth :kątmin fd 40 bk 40
seth :kątgodz fd 20 bk 20
make "kątmin :kątmin + 6
make "kątgodz :kątgodz + 0.5
pd
seth :kątmin fd 40 bk 40
seth :kątgodz fd 20 bk 20
wait 3000 / :szybkość
end
```



Rys. 9. Zegar

Po skasowaniu rysunków wskazówek i narysowaniu ich w nowym położeniu akcja zatrzymuje się na pewien czas, odwrotnie proporcjonalny do wartości zmiennej :szybkość. Komenda wait powoduje odczekanie bezczynnie podanej liczby pięćdziesiątych części sekundy. Jeśli przyjmiemy szybkość

równą 1, zegar powinien się tylko nieznacznie późnić, bo do tak wyznaczonego czasu oczekiwania dochodzi jeszcze krótki czas wykonania samej procedury tyk, a dokładniej jej pozostałych komend. Trzeba więc przyjąć szybkość nieco większą, dobierając prawidłową doświadczalnie. Jeśli jednak nie starczy nam cierpliwości, możemy sprawdzić działanie zegara przy znacznie większej szybkości, ale wtedy wpływ czasu wykonania procedury tyk będzie większy, bo przy przyspieszonym działaniu zegara jego tempo nie będzie już całkiem proporcjonalne do wartości paremetru :szybkość.

Może się zdarzyć, że tarcza tak narysowanego zegara nie będzie okrągła, lecz nieco spłaszczona. Jeśli tak jest, wynika to z niewłaściwej proporcji skal na osiach współrzędnych: pionowej i poziomej. Można ją poprawić komendą setscrunch. Jej argument jest parą liczb x, y ujętą w nawiasy kwadratowe. Jeśli argumentem setscrunch jest [100 100], otrzymujemy skale takie, jak na początku pracy zaraz po włączeniu komputera. Zmiana pierwszej z tych liczb oznacza zmianę skali osi x, drugiej – osi y. Na przykład: setscrunch [50 100] oznacza pozostawienie bez zmian skali na osi y i skrócenie dwukrotnie skali osi x dla wszystkich odtąd rysowanych elementów. Wszystko, co było przedtem narysowane, pozostanie bez zmian. Wykonanie setscrunch [50 50] powoduje dwukrotne zmniejszenie skali x i y, czyli dwukrotne zmniejszenie rozmiarów wszystkich elementów rysunku, które potem powstaną.

Aby dowiedzieć się, jaka jest aktualna wartość pary tych współczynników, używamy operacji scrunch, pisząc:

```
print scrunch
```

W następnym wierszu otrzymamy tę parę liczb z pominiętymi nawiasami kwadratowymi, na przykład:

```
50 50
```

Ewidencja oraz przechowywanie procedur i danych

Logo stwarza bardzo dobre możliwości kontroli tego, co jest zapisane w pamięci komputera – zarówno treści procedur, jak danych. Sprawdzenie zawartości pamięci można wykonać w każdym momencie, gdy tylko pytajnik jest na ekranie. Zależnie od potrzeb, otrzymamy informacje o wszystkim albo tylko o tych rzeczach, które nas w tej chwili interesują.

Komenda pots powoduje wypisanie tytułów (nagłówków) wszystkich procedur, które zostały zdefiniowane i są aktualnie w pamięci. Nie wypisuje jednak komend i operacji języka. Przed podaniem tej komendy dobrze jest przejść do trybu tekstowego komendą ts. Jeśli w pamięci są tylko procedury i dane związane z zegarem z poprzedniego rozdziału, to reakcja komputera może wyglądać tak:

```
ts
pots
to tyk
to zegar :godz :min :szybkość
```

Komenda pons powoduje podanie nazw i wartości wszystkich zmiennych, wprowadzonych przy pomocy make. Możemy więc mieć w dalszym ciągu następującą wymianę informacji:

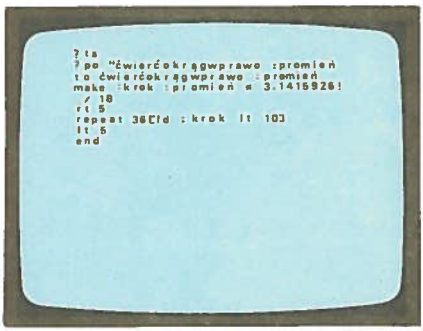
```
pons
make "kątgodz "374
make "kątmin "168
```

Komenda poall powoduje pokazanie wszystkiego, to znaczy nagłówków i treści procedur oraz nazw i wartości zmiennych. Są one wypisywane jednym ciągiem, czyli w miarę napływania nowych wierszy poprzednie mogą znikać pod górną krawędzią ekranu. Aby zatrzymać ten przepływ wierszy tekstu, trzeba nacisnąć SYMBOL SHIFT i S. Naciśnięcie dowolnego klawisza uruchamia ten przepływ ponownie.

Jeśli chcemy zobaczyć nagłówki i treści wszystkich procedur, użyjemy komendy pops. Natomiast po z nazwą procedury, poprzedzoną cudzysłowem jako znakiem dosłowności, spowoduje pokazanie nagłówka i treści tylko tej wybranej procedury. Na przykład:

```
po "zegar
```

Jeśli argumentem po jest lista nazw procedur (czyli ciąg tych nazw ujęty w nawiasy kwadratowe), na ekranie ukażą się wszystkie te procedury w całości, jedna za drugą. (np. rys. 10). Sprawdźmy na przykład na komputerze, jaki będzie skutek



Rys. 10. Wynik po "ćwierćokrągwprawo, gdy ta procedura jest w pamięci

po [zegar tyk]

Dla zmiennych taki odpowiednik po nie istnieje, to znaczy można tylko obejrzeć je wszystkie razem, stosując komendę pons. Innym sposobem oglądania procedur i zmiennych z jednoczesną możliwością poprawiania jest skorzystanie z edytora. Do przeglądania i redagowania procedur służy komenda ed, opisana szczegółowo w pierwszej części tej książki. Przypomnijmy, że powoduje ona przejście do trybu redagowania wskazanej procedury lub procedur, przy czym główną rolę przy poprawianiu odgrywają kombinacje klawiszy EXTENDED MODE ze strzałkami i klawisze DELETE (zakładamy, że Czytelnik potrafi się już posługiwać trybem EXTENDED MODE nawet na Spectrum z gumową klawiaturą). Wyjście z trybu redagowania, to: EXTENDED MODE C, gdy chcemy zapamiętać wersję poprawioną, i BREAK, gdy chcemy przerwać redagowanie bez zapamiętania jego wyników.

Zupełnie podobnie można redagować zapisy zmiennych wprowadzonych przy pomocy make. Służy do tego jednoparametrowa komenda edns, której argumentem jest nazwa zmiennej lub ujęty w nawiasy kwadratowe ciąg takich nazw.

Oprócz procedur definiowanych oraz zmiennych wprowadzonych przy pomocy make w pamięci są zapisane nazwy i treść komend oraz operacji języka. Nie można ich redagować ani obejrzeć ich treści, gdyż są one zapisane w inny sposób, niż procedury i dane wprowadzane przez nas. Można tylko zobaczyć wszystkie ich nazwy przy pomocy komendy .contents. Powoduje ona wypisanie wszystkich nazw, rozumianych w tej chwili przez Logo – zarówno procedur języka, jak definiowanych. Komendą tą trzeba posługiwać się ostrożnie, bo w pewnych przypadkach może ona spowodować zmiany zawartości pamięci i zniszczenie części wyników dotychczasowej pracy. O tym niebezpieczeństwie ostrzega kropka przed nazwą komendy – tak w Logo oznaczane są te komendy, które są niebezpieczne i wymagają ostrożności w posługiwaniu się nimi.

Wprowadzanie każdej procedury lub danej i wykonanie każdego obliczenia wymaga użycia pewnej liczby miejsc pamięci komputera. Przydzielanie tej pamięci i zwalnianie jej po wykorzystaniu odbywa się całkowicie automatycznie. Co pewien czas odbywa się operacja odśmiecania, polegająca na tym, że użyte kiedyś, ale już wolne miejsca pamięci są ponownie przystosowane do pracy. Jest ona wykonywana automatycznie i bez wiedzy programisty. O tym, że takie odśmiecanie właśnie się odbywa, możemy się dowiedzieć, gdy nastąpi mała nie przewidziana przerwa w ruchu żółwia po ekranie. Trwa ona krótko – tylko tyle, ile trzeba czasu na odzyskanie miejsc pamięci – po czym zostaje wznowione działanie żółwia i wszystkie towarzyszące mu obliczenia.

Odśmiecanie pamięci można również wywołać, pisząc komendę recycle. Z pozoru nie wywołuje ona żadnych widocznych skutków, ale teraz możemy zapytać o liczbę wolnych miejsc pamięci, pisząc:

print nodes

Otrzymana liczba będzie właśnie informować, ile tych wolnych miejsc zostało. Są to miejsca w sensie Logo – każde z nich zajmuje po kilka bajtów i ci, którzy są przyzwyczajeni do sposobów podawania objętości wolnej pamięci stosowanych w innych językach, muszą zdawać sobie sprawę z tego, że tutaj to robi się inaczej. Gdybyśmy napisali:

print nodes

kilkakrotnie, za każdym razem otrzymywalibyśmy coraz mniejszą liczbę. Nic dziwnego – każda operacja, nawet badanie liczby wolnych miejsc pamięci powoduje zajęcie kilku miejsc. Gdybyśmy natomiast pisali:

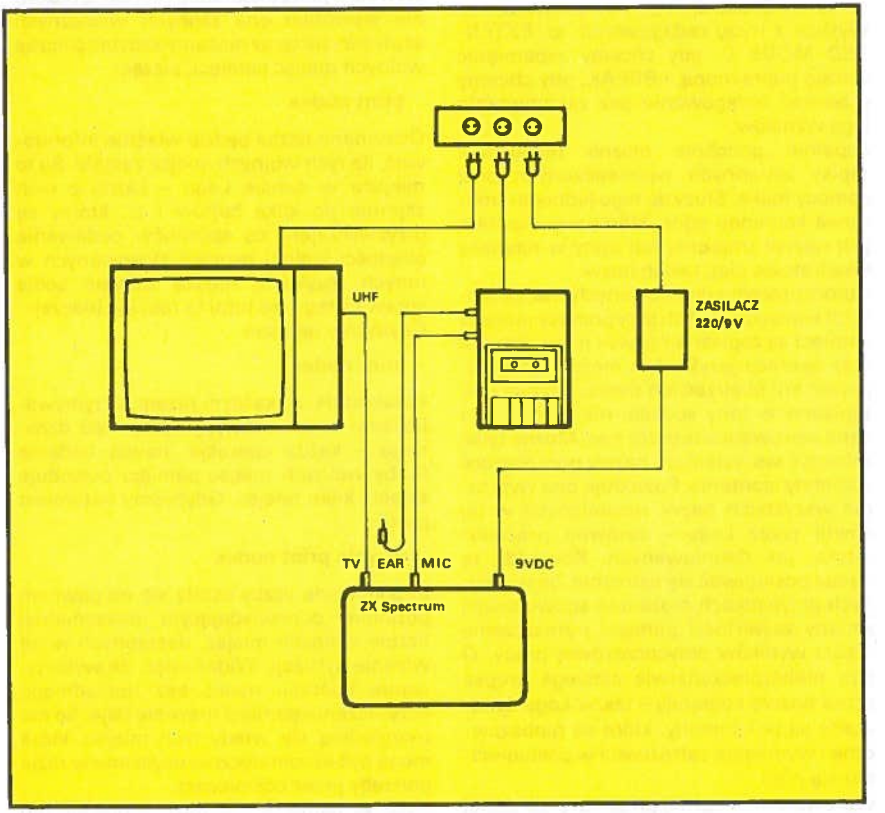
recycle print nodes

to podawane liczby ustalą się na pewnym poziomie, odpowiadającym maksymalnej liczbie wolnych miejsc, dostępnych w tej właśnie sytuacji. Widać więc, że wykonywanie operacji nodes bez uprzedniego oczyszczenia pamięci niewiele daje, bo nie uwzględnia się wtedy tych miejsc, które mogą być automatycznie usyskane w razie potrzeby przez odśmiecacz.

Jeśli w pamięci jest dużo procedur lub danych, może się okazać, że liczba pozostałych miejsc wolnych pamięci jest za mała i

działanie Logo będzie przerwane, a na ekranie ukaże się komunikat „out of memory", czyli stwierdzenie, że jest za mało pamięci, by kontynuować pracę. Nie zdarza się to przy prostych procedurach – spośród wszystkich podanych w tej książce dopiero te, które są na samym końcu, mogą doprowadzić do całkowitego zapełnienia pamięci. Warto jednak wiedzieć, co w takich przypadkach robić. Najczęściej przepełnienie pamięci następuje dlatego, że przechowujemy w niej zbyt wiele procedur, to znaczy oprócz aktualnie potrzebnych pozostały w niej jakieś już nie wykorzystywane, których się jeszcze nie skasowało. Zanim procedury się skasuje, warto je zapisać w pamięci zewnętrznej. Ponieważ w tej książce zajmujemy się w zasadzie tylko najprostszą konfiguracją Spectrum, ograniczymy się do zapisywania ich na taśmie magnetofonowej.

Do zapisywania na taśmie używa się komend: saveall, savescr, saved oraz save. Pierwszym argumentem każdej z nich jest nazwa pliku na taśmie, nadawana mu w momencie zapisywania. Pod tą nazwą znajdzie się później ten plik, gdy zechcemy go wprowadzić ponownie do komputera. Dla wszystkich komend z wyjątkiem ostatniej nazwa pliku jest jedynym argumentem. Saveall zapisuje wszystko to, co można obejrzeć przy pomocy poall. Savescr



Rys. 11. Połączenie Spectrum z magnetofonem. Przy nagrywaniu wtyczka powinna być z gniazdka EAR wyciągnięta

zapisuje aktualny obraz ekranu. Saved służy do zapisywania tego, co jest aktualnie redagowane i ma być przechowane na później (komenda ta umożliwia przerwanie pracy z komputerem nawet wtedy, gdy nie wszystkie dane lub procedury mają prawidłową postać i będą musiały być jeszcze redagowane przed użyciem). Save służy do zapamiętania wybranej procedury lub kompletu procedur. Tylko ona jest dwuargumentowa – jej drugim argumentem jest nazwa procedury (poprzedzona cudzysłowem ") albo ujęty w nawiasy kwadratowe ciąg nazw procedur, które mają być zapamiętane.

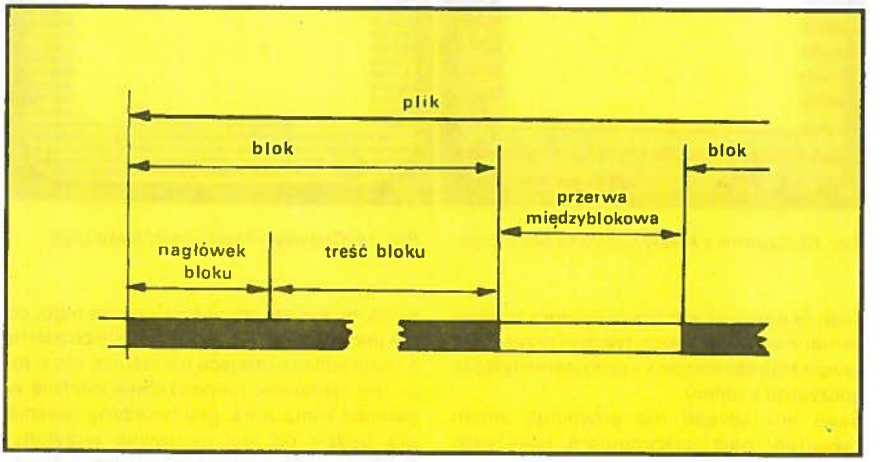
Jeżeli magnetofon ma gniazdka na wtyki 3,6 mm, należy włączyć tylko jeden wtyk – drugi ma pozostać nie wetknięty do gniazdka (rys. 11). Oczywiście zarówno w magnetofonie, jak w komputerze do gniazdek muszą być wetknięte wtyczki tego samego koloru. W większości magnetofonów jest obojętne, czy to będzie gniazdko MIC, czy EAR. Dla magnetofonów bez gniazdek 3,6 mm trzeba mieć właściwą wtyczkę i znać zasady posługiwania się nią – w zasadzie wtedy ona po prostu powinna być we właściwym gniazdku.

Napisanie którejkolwiek z tych komend prawidłowo i naciśnięcie klawisza ENTER powoduje ukazanie się na ekranie napisu:

Start tape, then press any key

czyli: uruchom magnetofon i naciśnij dowolny klawisz. Naciskamy klawisze nagrywania, zwracając uwagę na to, by część rozbiegowa taśmy przeszła poza głowicę. Jeśli w magnetofonie jest mikrofon, możemy doń powiedzieć wyraźnie nazwę nagrywanego pliku. Ułatwi to późniejsze odszukiwanie go na taśmie. Naciśnięcie dowolnego klawisza rozpoczyna nagrywanie, co jest sygnalizowane pojawieniem się ruchomych pasków na obrzeżu ekranu. Zaniknięcie tych pasków i pojawienie się pytajnika oznacza koniec nagrywania – zatrzymujemy magnetofon.

Nagrywana na taśmę treść pliku jest automatycznie dzielona na tzw. bloki. Każdy z nich jest oddzielony od następnego niewielką przerwą (rys. 12). Praktyka wskazuje, że czasami ta przerwa jest zbyt krótka i przy odczytywaniu taśmy do komputera część zapisanych informacji ginie, czyniąc cały zapis bezużytecznym. Aby temu zapobiec, trzeba poprawić zapisaną w pamięci pod adresem 34624 długość przerwy międzyblokowej. W tym celu wystarczy przed



Rys. 12. Struktura zapisów na taśmie magnetofonowej

podaniem komendy save wykonać komendę:

.deposit 34624 60

Tak wpisana nowa długość przerwy międzyblokowej pozostanie w pamięci aż do wyłączenia komputera.
Odczytywanie z taśmy zapewniają komendy:load (czytanie procedur i danych), loadscr (czytanie obrazu na ekran), loadd (czytanie redagowanych procedur lub danych). Każda z nich jest jednoargumentowa i wartością argumentu jest nazwa pliku, poprzedzona cudzysłowem. Połączenie magnetofonu z komputerem jest takie samo, jak przy nagrywaniu Logo – dwie wtyczki w gniazdkach EAR i MIC komputera i podobnie wtyczki w gniazdkach magnetofonu (choć w większości typów magnetofonów może wystarczyć tylko wtyczka w gniazdku EAR). Czytanie z taśmy jest sygnalizowane ruchomymi paskami na obrzeżu ekranu, przed znalezieniem właściwego pliku na taśmie barwa obrzeża zmienia się okresowo (rys. 13 i 14).

odpowiednich komend. Są to: erall – usuń wszystko, co pokazuje poall; erps – usuń wszystko, co pokazuje pops (czyli wszystkie procedury); erns – usuń wszystko, co pokazuje pons (czyli wszystkie zmienne); erase, er – usuń wszystko, co pokaże po o tym samym argumencie (czyli wskazane procedury).
Na przykład:

er [zegar tyk]

albo

er "tyk

Do usuwania nazw zmiennych służy ern, nie mająca swego odpowiednika wśród komend o nazwach zaczynających się od po. Na przykład:

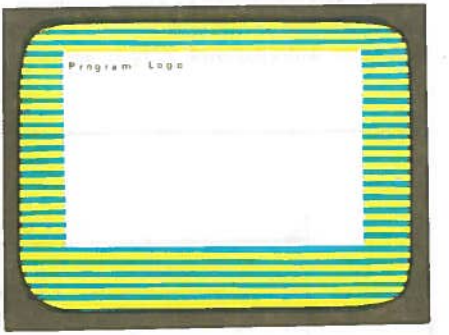
ern [kątgodz kątmin]

usunie obie te zmienne, podczas gdy:

ern "kątmin

tylko jedną z nich.
Korzystając z saveall warto zwrócić uwagę

Rys. 13. Czytanie z kasety nagłówka bloku Logo



Rys. 14. Czytanie z kasety treści bloku Logo

Jeśli w pamięci jest już procedura o takiej samej nazwie, jak wczytywana przez load, to zostanie skasowana – pozostanie tylko ta odczytana z taśmy.
Save ani saveall nie powodują zmian zawartości pamięci komputera. Jeżeli więc po zapisaniu procedur do pamięci zewnętrznej zechcemy je skasować, trzeba użyć na to, by nie zapisywać na taśmie tego, co nie jest koniecznie potrzebne. Nie chodzi tu o oszczędność miejsca na taśmie, ale o to, by nie zajmować niepotrzebnie miejsca w pamięci komputera, gdy tworzony właśnie plik będzie do niej ponownie wczytany. Gdybyśmy na przykład chcieli zapisać procedury związane z tworzeniem zegara, nie

ma potrzeby zapisywać razem z nimi nazw i wartości zmiennych kątgodz, kątmin, bo i tak komendy make, które tworzą te zmienne i nadają im nowe wartości, są wykonywane przy każdym wywołaniu procedury zegar. Przechowywanie starych, a więc już nieaktualnych i do niczego niepotrzebnych wartości tych zmiennych nie miałoby więc żadnego sensu. Można więc w takiej sytuacji przed wykonaniem saveall wykonać erns, kasujące wszystkie zapisane w pamięci zmienne, stworzone przez make.

Tworzenie złożonych rysunków

Rysunki, którymi zajmowaliśmy się dotychczas, były tak proste, że narysowanie ich nie wymagało głębszego zastanowienia. Przykłady, którymi się teraz zajmiemy, także nie będą bardzo złożone, ale już rysunki będą składać się z pewnych części, które z kolei, być może, będą budowane z jakichś innych fragmentów, prostszych i łatwiejszych do wykonania. Posłużymy się tymi przykładami dla pokazania pewnej ogólnej metody, zwanej zstępującą (po angielsku top down), szczególnie przydatnej wtedy, gdy chce się użyć komputera do rozwiązywania zadań skomplikowanych.

Mając jakieś zadanie do rozwiązania, musimy znaleźć sposób, w jaki można by to uczynić, przy czym opis postępowania, które do tego doprowadzi, powinien być możliwie ścisły. Taki ścisły opis metody rozwiązania zadania nazwiemy algorytmem. Mając algorytm musimy z kolei zastanowić się, jak będziemy mogli go zrealizować przy pomocy komputera i jak wobec tego należy zaprogramować działania tego komputera.

Jeżeli zadanie jest na tyle trudne lub skomplikowane, że nie byłoby łatwo opisać od razu całość rozwiązania, to należy je podzielić na zadania częściowe, te z kolei ewentualnie na jeszcze mniejsze zadania składowe i tak dalej, aż dojdzie się do zadań elementarnych, których rozwiązywanie i opisanie procesu rozwiązywania nie będzie już sprawiać trudności. Przez cały czas nie możemy jednak stracić kontroli nad całym przebiegiem rozwiązywania zadania początkowego. Trudność polega między innymi na tym, że w całym tym postępowaniu mogą się zdarzyć różne pomyłki. Trzeba więc postępować na tyle systematycznie i przezornie, żeby nawet w razie popełnienia błędów można je było zlokalizować, poprawić i w końcu dojść do prawidłowych wyników.

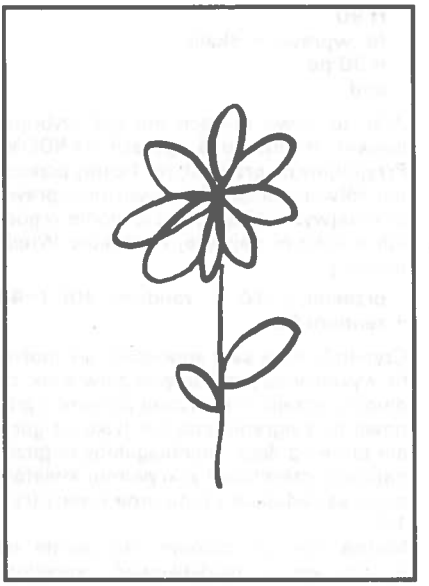
Jedną z typowych trudności jest to, że wskutek jakiejś pomyłki wyniki zadań częściowych nie pasują do siebie i nie dadzą się złożyć ze sobą w sposób pozwalający uzyskać prawidłowe rozwiązanie zadania nadrzędnego. Zapobiega temu konsekwentne stosowanie metody zstępującej – zobaczymy na przykładach, jak to się robi.

Pierwsze z omawianych tu zadań zostało wzięte z książki S. Paperta „Mindstorms – children, computers, and powerful ideas". Spróbujmy narysować kwiat według szkicu z rys. 15.

Gdybyśmy zaczęli od zastanawiania się, jak należy rysować najdrobniejsze części składowe – płatki lub listek – postawilibyśmy się w trudniejszej sytuacji, bo wtedy trzeba by także niezwłocznie powiedzieć, jak te elementy mają być ze sobą montowane, aby prawidłowo utworzyć pożądaną całość. O wiele łatwiej będzie zacząć od wyodrębnienia głównych części rysunku kwiatu, a więc nie tych najbardziej elementarnych, i ustalenia, jak one są połączone ze sobą. Taki rozbiór nie stwarza żadnych trudności, a wszystko, co trzeba, wiemy od razu bez żadnych niedomówień. Możemy więc od razu napisać odpowiednią procedurę w języku Logo:

```
to kwiat
łodyga
korona
powrót żółwia
end
```

Procedura powrotu żółwia została tutaj wprowadzona po to, by nie pozostawiać go



Rys. 15. Projekt kwiatka

po narysowaniu kwiatu gdzieś w okolicy korony. Gdybyśmy chcieli dalej rysować następne kwiaty, to wygodniej będzie mieć żółwia w dogodnej pozycji wyjściowej, na przykład na poziomie ziemi przy dolnym końcu łodygi.
Postępując dalej w tym samym stylu, dzielimy łodygę na części. Tym razem są one całkiem elementarne, jeśli chodzi o odcinki łodygi, natomiast listek jest bardziej skomplikowany:

```
to łodyga
fd 20
listek
fd 30
end

to listek
ćwierćokrągwprawo 12
rt 90
ćwierćokrągwprawo 12
rt 90
end
```

Korona może być zbudowana z płatków o takim samym kształcie, jak listek. Trzeba tylko zdecydować, ile ich będzie – na przykład 12:

```
TO KORONA
REPEAT 12 [RT 30 LISTEK]
END
```

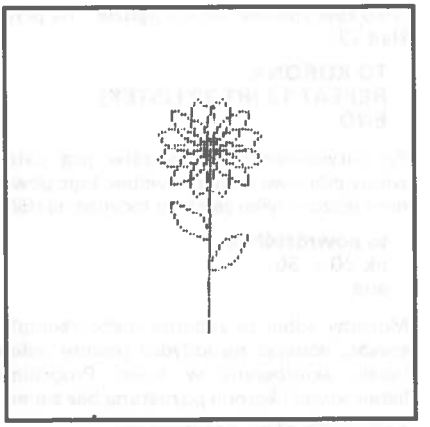
Po narysowaniu korony żółw jest ustawiony pionowo do góry i wobec tego powinien jeszcze tylko zejść po łodydze na dół:

```
to powrótżółwia
bk 20 + 30
end
```

Możemy sobie ta zadanie nieco skomplikować, dodając na łodydze jeszcze jeden listek, skierowany w lewo. Procedury listek, kwiat i korona pozostaną bez zmian, a pozostałe będą wyglądać tak:

```
to łodyga
fd 20
listek
fd 10
lewylistek
fd 30
end

to lewylistek
lt 90
listek
rt 90
end

to powrótżółwia
bk 20 + 10 + 30
end
```

Wywołanie procedury: kwiat da obraz pokazany na rys. 16.
Gdyby nam nie odpowiadała wielkość tego kwiatka albo gdybyśmy chcieli rysować kwiaty różnych rozmiarów, możemy wprowadzić nową zmienną :skala, przez którą pomnożymy argumenty wszystkich fd, bk i ćwierćokrągwprawo w procedurach **łodyga, listek i powrótżółwia**. Przed wywołaniem procedury kwiat trzeba będzie nadać tej zmiennej wartość: np. make " skala.8 cs kwiat
Czytelnik może teraz sam zastanowić się, jak narysować łączkę lub ogródek, gdzie kwiatki rosną chaotycznie lub w rządkach, jak powiązać wielkość kwiatu z jego odległością od widza i tak dalej.
Przesuwanie żółwia do nowego miejsca, w



Rys. 16. Kwiat z dwoma listkami

którym ma wyrosnąć kolejny kwiat, zapewni procedura:

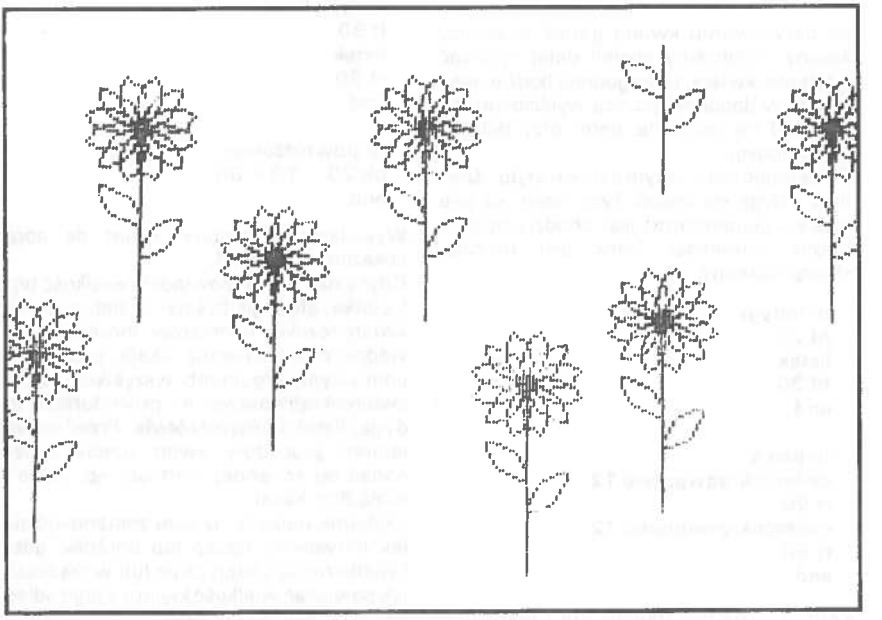
```
to przesuń :wgórę :wprawo
pu
fd :wgóre * :skala
rt 90
fd :wprawo * :skala
lt 90 pd
end
```

Jeśli to nowe miejsce ma być wybrane losowo, można użyć operacji RANDOM. Przyjmijmy na przykład, że chcemy przesunąć żółwia w poziomie w lewo lub w prawo o co najwyżej 40 kroków i w pionie w górę lub w dół o co najwyżej 10 kroków. Wtedy użyjemy:

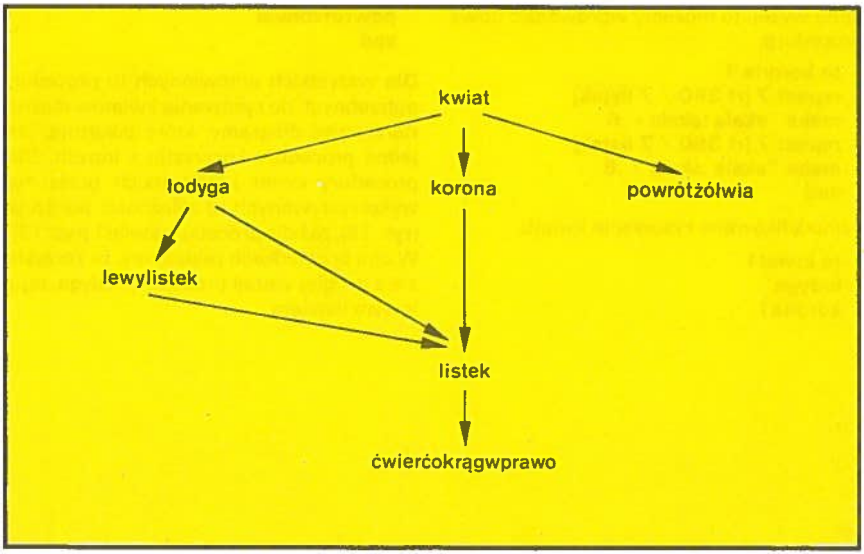
```
przesuń (−10 + random 20) (−40 + random 80)
```

Czytelnik może sam sprawdzić, jak można by wykonywać przesunięcia żółwia tak, by długość przejścia mierzona poziomo i pionowo była ograniczona nie tylko od góry, ale także od dołu. Zapobiegałoby to (przynajmniej częściowo) zakrywaniu kwiatów przez sąsiadujące z nimi inne kwiaty (rys. 17).

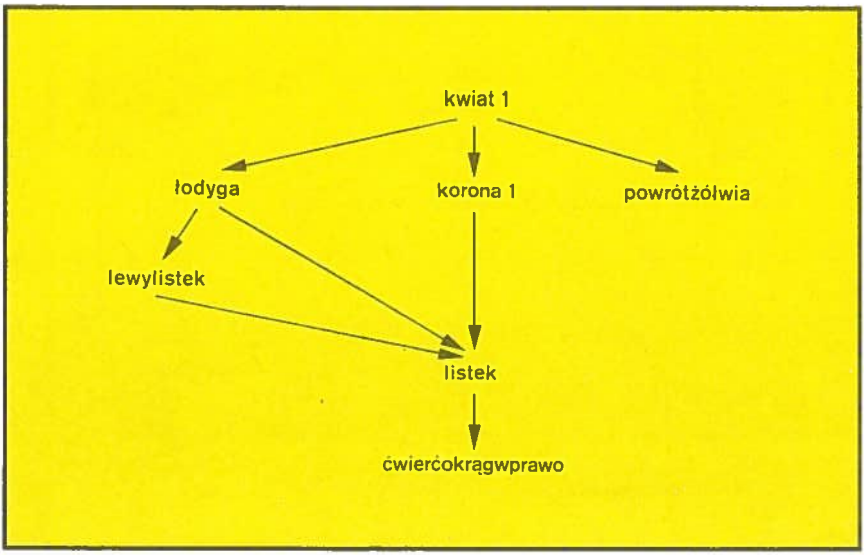
Można również całkiem niezależnie od innych zmian modyfikować procedurę rysującą koronę. Jeśli już mamy wprowadzoną zmienną :skala, tak jak to zostało opi-



Rys. 17. Łączka



Rys. 18. Struktura powiązań procedur rysujących kwiat



Rys. 19. Struktura powiązań procedur rysujących kwiat1

sane wyżej, to możemy wprowadzić nową procedurę:

```
to korona 1
repeat 7 [rt 360 / 7 listek]
make "skala :skala * .6
repeat 7 [rt 360 / 7 listek]
make "skala :skala / .6
end
```

i zmodyfikowane rysowanie kwiatu:

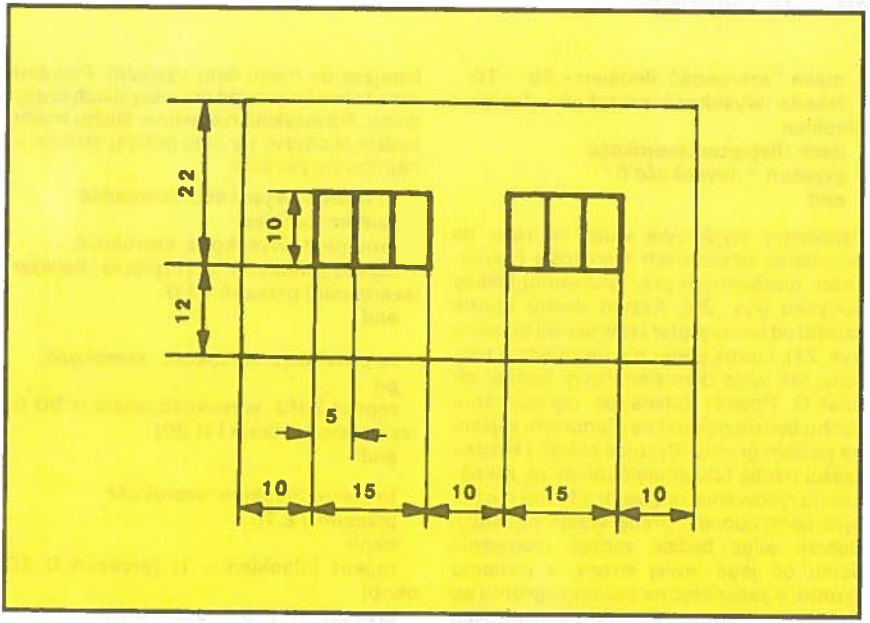
```
to kwiat1
łodyga
korona1
powrótżółwia
end
```

Dla wszystkich omówionych tu procedur, potrzebnych do rysowania kwiatów można narysować diagramy, które pokazują, jak jedne procedury korzystają z innych. Dla procedury kwiat i wszystkich przez nią wykorzystywanych te zależności pokazuje (rys. 18), zaś dla procedury kwiat1 (rys. 19). W obu przypadkach zakładamy, że korzysta się z drugiej wersji procedury łodyga, tej z lewym listkiem.

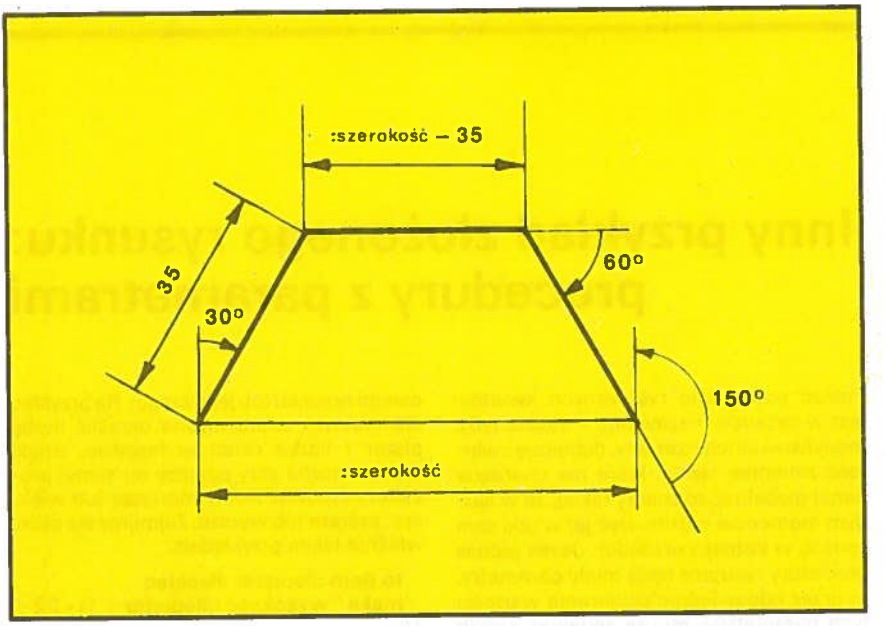
Inny przykład złożonego rysunku: procedury z parametrami

Postać poprzednio rysowanych kwiatów jest w zasadzie niezmienna – można tylko modyfikować ich rozmiary, dobierając wartość zmiennej :skala, która ma charakter danej globalnej, to znaczy takiej, że w każdym momencie można użyć jej w taki sam sposób w każdej z procedur. Jeżeli jednak procedury rysujące będą miały parametry, to przez odpowiednie dobieranie wartości tych parametrów można zmieniać kształt całego rysunku lub jego części. Na przykład dla rysunku domu można określić liczbę pięter i liczbę okien w fasadzie, dzięki czemu można przy pomocy tej samej procedury rysować domy mniejsze lub większe, szersze lub wyższe. Zajmijmy się bliżej właśnie takim przykładem:

```
to dom :ilepięter :ileokien
make "wysokość (:ilepięter + 1) * 22 +
12
```



Rys. 20. Wymiary elementów piętra



Rys. 21. Konstrukcja dachu

```
make "szerokość :ileokien * 25 + 10
fasada :wysokość :szerokość :ilepięter
:ileokien
dach :ilepięter :szerokość
przesuń –:wysokość 0
end
```

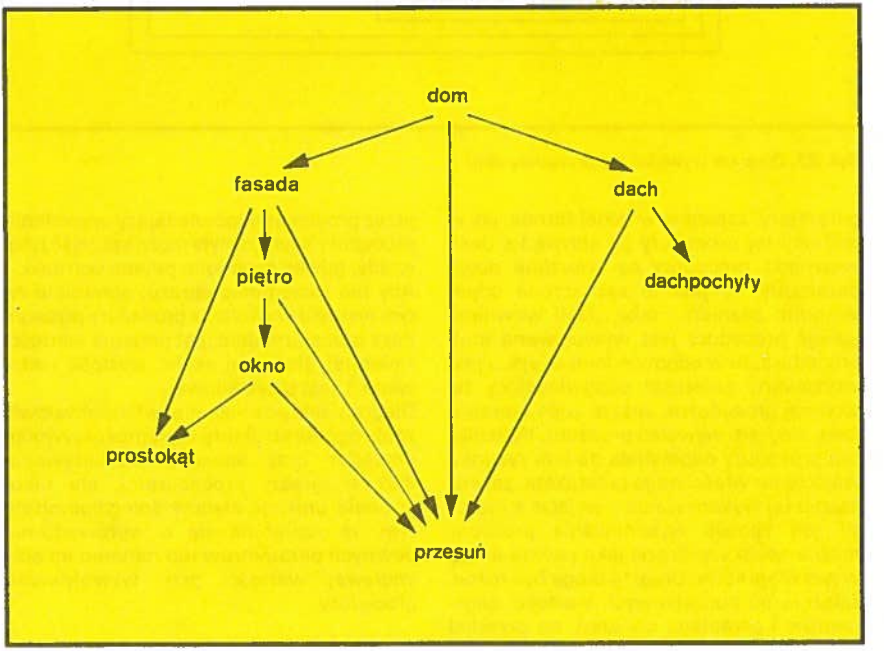
Parametry wyjściowe służą od razu do określenia następnych szerokości i wysokości, niezbędnych przy rysowaniu fasady budynku (rys. 20). Kształt dachu będzie zależał od liczby pięter i szerokości gmachu (rys. 21). Liczba pięter nie uwzględnia parteru, tak więc dom parterowy będzie ich miał 0. Powrót żółwia po narysowaniu dachu będzie polegał na pionowym zejściu na poziom gruntu. Rysując całość i każdą z części trzeba tak ustawić żółwia po zakończeniu rysowania, aby bez trudności można było kontynuować pracę wedle potrzeby. Dobrze więc będzie zacząć rysowanie domu od jego lewej strony, z poziomu gruntu, a zakończyć na poziomie gruntu po jego prawej stronie, to jest w miejscu, skąd można zacząć rysować bezpośrednio przylegający do niego dom sąsiedni. Podobne zasady trzeba przyjąć dla wszystkich części domu. Na przykład rysowanie dachu trzeba będzie skończyć po jego prawej stronie w najniższym punkcie.

```
to fasada :wysokość :szerokość
:ilepięter :ileokien
prostokąt :wysokość :szerokość
repeat (:ilepięter + 1) [piętro :ileokien
:szerokość] przesuń 12 0
end

to prostokąt :wysokość :szerokość
pd
repeat 2 [fd :wysokość :skala rt 90 fd
:szerokość * :skala l rt 90]
end

to piętro :ileokien :szerokość
przesuń 12 10
okno
repeat (:ileokien - 1) [przesuń 0 10
okno]
przesuń 10 (10 – :szerokość)
end
```

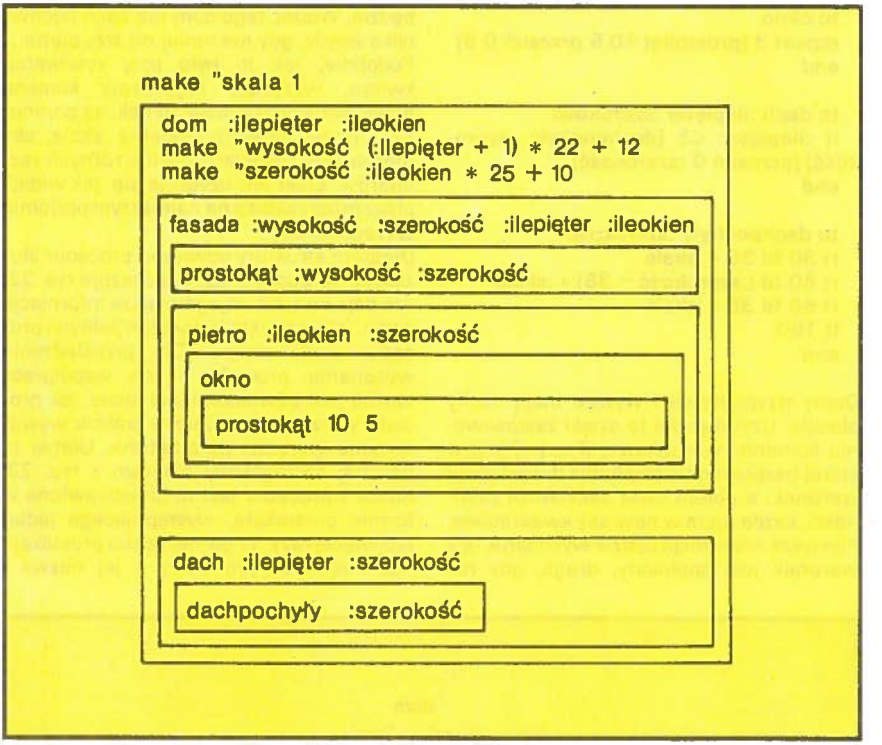
```
to okno
repeat 3 [prostokąt 10 5 przesuń 0 5]
end

to dach :ilepięter :szerokość
if :ilepięter <3 [dachpochyły :szerokość] [przesuń 0 :szerokość]
end

to dachpochyły :szerokość
rt 30 fd 35 * :skala
rt 60 fd (:szerokość − 35) * :skala
rt 60 fd 35 * :skala
lt 150
end
```

Domy trzypiętrowe i wyższe mają dachy płaskie. Uzyskuje się to dzięki zastosowaniu komendy warunkowej if ... [...] [...], w której bezpośrednio po słowie if występuje warunek, a potem dwie sekwencje czynności, każda ujęta w nawiasy kwadratowe. Pierwsza sekwencja będzie wykonana, gdy warunek jest spełniony, druga, gdy nie będzie. Wobec tego dom ma dach pochyły tylko wtedy, gdy ma mniej niż trzy piętra. Podobnie, jak to było przy rysowaniu kwiatu, wszystkie argumenty komend przesuwających żółwia, fd i bk, są pomnożone przez zmienną globalną :skala, aby można było rysować budynki różnych rozmiarów. Efekt ten uzyskuje się, jak widać, przez prosty zabieg na najniższym poziomie szczegółowości.

Diagram struktury powiązań procedur służących do budowy domu pokazuje rys. 22. Ale daje on tylko najogólniejsze informacje o tym, jaki jest układ wywołań jednych procedur przez drugie. Dla prześledzenia wykonania procedur i ich współpracy ważne jest uświadomienie sobie, jak procedury przekazują sobie w trakcie wywoływania wartości parametrów. Ułatwi to bardziej szczegółowy diagram z rys. 23. Każda z procedur jest tu przedstawiona w formie prostokąta, występującego jeden lub więcej razy. W górnej części prostokąta jest nagłówek procedury – jej nazwa i



Rys. 22. Struktura powiązań procedur rysujących dom



Rys. 23. Diagram wywołań dla procedury dom

parametry, zapisane w takiej formie, jak w definicji tej procedury po słowie to. Jeśli wewnątrz procedury są tworzone nowe parametry, to jest to zaznaczone odpowiednim zdaniem make. Jeśli wewnątrz jakiejś procedury jest wywoływana inna procedura, to w odpowiednim miejscu jest wrysowany prostokąt odpowiadający tej wołanej procedurze. Jest to więc szczegółowy diagram wywołań procedur. Wywołaniu procedury odpowiada na tym rysunku wejście do właściwego prostokąta, zakończeniu jej wykonywania – wyjście z niego. W ten sposób wykonywanie procedur można sobie wyobrazić jako pewną drogę w tym diagramie. Drogi te mogą być różne, zależnie od początkowych wartości parametrów i przebiegu obliczeń; na przykład przez prostokąt odpowiadający wywołaniu procedury dachpochyły można przejść tylko wtedy, gdy są spełnione pewne warunki.
Aby nie zaciemniać obrazu, pominięto na tym rysunku wywołania procedury przesuń. Poza procedurą dom jest nadanie wartości zmiennej globalnej skala; wartość nadawana 1 jest przykładowa.
Diagram taki pozwala nie tylko uzmysłowić sobie ogólną strukturę i kolejność wywołań procedur oraz sposoby przekazywania danych między procedurami, ale także pozwala uniknąć błędów polegających na tym, że zapomina się o wprowadzeniu pewnych parametrów lub nadaniu im prawidłowej wartości przy wywoływaniu procedury.

Procedury rekurencyjne

Jeżeli chcemy narysować płot jako ciąg określonej liczby sztachet, to możemy postąpić tak:
Jeśli ta liczba sztachet jest równa 0, to nic nie rób, kończąc tym samym rysowanie. W przeciwnym przypadku:

- narysuj sztachetę,
- narysuj płot z liczbą sztachet o jeden mniejszą.

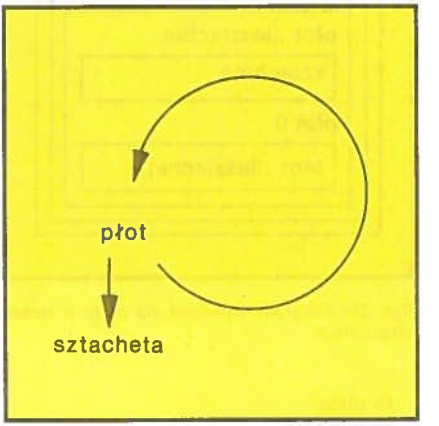
W Logo zapiszemy to:

```
to płot :ilesztachet
if :ilesztachet = 0 [stop]
sztacheta
płot :ilesztachet − 1
end
```

Komenda stop powoduje zakończenie wykonywania procedury. Jeśli więc liczba sztachet jest dodatnia, to warunek po if nie jest spełniony i wykonywane są następne komendy procedury. Jeśli jest 0, to wykonywana jest komenda stop i działanie procedury się kończy.
Tak natomiast opiszemy pojedynczą sztachetę:

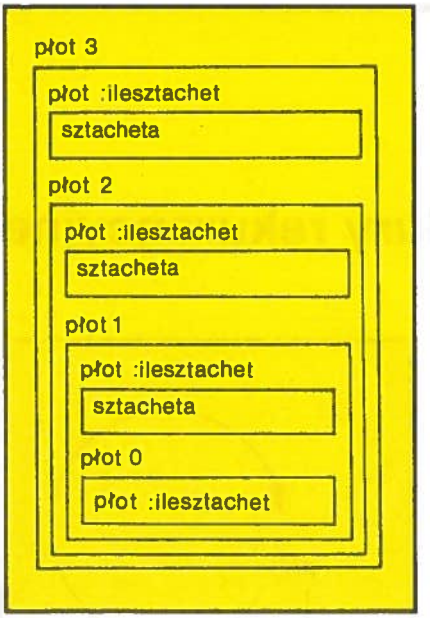
```
to sztacheta
prostokąt 7 3
przesuń 0 6
end
```

Ponieważ procedura płot może wywoływać siebie samą, graf na rys. 24 ma pętlę – jest to charakterystyczna cecha takich zestawów procedur, w których występuje rekurencja. To wywoływanie procedur przez same siebie komplikuje bardzo diagramy wywołań, których postać zależy w tym przypadku od tego, jakie są wartości parametrów procedur – zjawisko, które dla procedur bez rekurencji w ogóle nie występuje. Rys. 25 pokazuje diagram wywołań dla płotu budowanego z trzech sztachet. Dla ułatwienia bezpośrednio nad prostokątem odpowiadającym wywołaniu procedury jest podawana każdorazowo aktualna wartość parametru.



Rys. 24. Struktura powiązań procedur płot i sztacheta

Konstrukcja procedury płot jest typowa dla prostych procedur rekurencyjnych i początkujący powinni kierować się tym wzorem – najpierw sprawdzić, czy jest spełniony warunek zakończenia działania procedury przez użycie komendy stop, jeśli nie, wykonać czynność elementarną, charakterystyczną dla tej procedury, po czym przejść do wywołania rekurencyjnego z odpowiednio zmienionym parametrem.
Mając możliwość rysowania płotów możemy tworzyć krajobrazy bardziej urozmaicone (rys. 26).



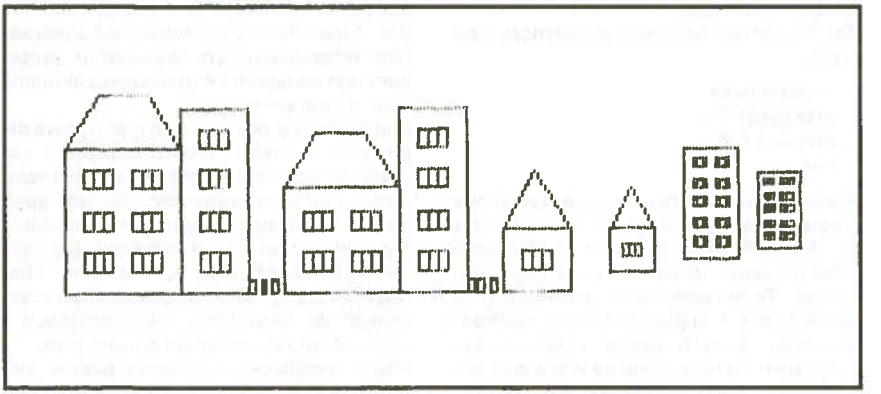
Rys. 25. Diagram wywołań dla płotu o trzech sztachetach

```
to ulica
cs window
make "skala 1
przesuń -80 -120
make "skala 0.6
  repeat 4 [dom random 5 1 + random 3
płot * random 2]
  repeat 6 [dom random 5 1 + random 2
przesuń 10 20 make "skala :skala * 0.7]
  end
```

Ponieważ rozmiary tych dziesięciu domów są dobierane losowo, trudno przewidzieć szerokość całego rysunku – może on nie zmieścić się na ekranie. Z tego powodu na początku jest przejście do trybu window, w którym żółw może swobodnie poruszać się poza ekranem bez szkody dla działania procedury i dla tej części rysunku, która na ekranie się zmieściła (rys. 27 i 28).
Możemy wprowadzić do tych krajobrazów drzewa. Z liściastymi nie ma większych kłopotów. Może to być na przykład (rys. 29):

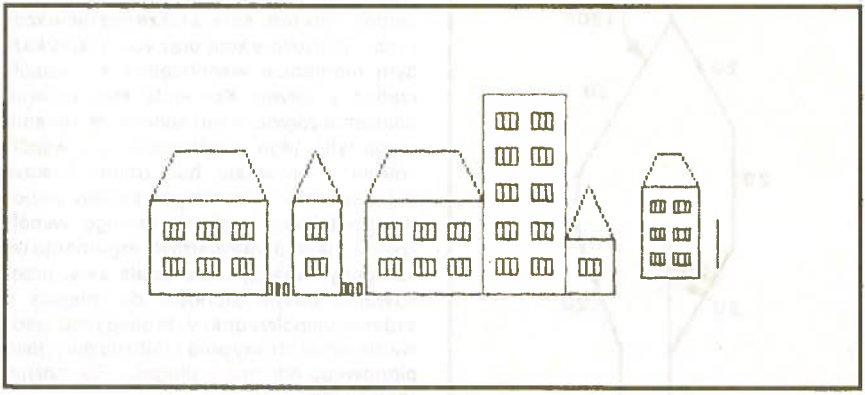
```
to drzewo
fd 30 * :skala
lt 30 fd 20 * :skala
rt 30 fd 20 * :skala
rt 30 fd 20 * :skala
rt 120 fd 30 * :skala
rt 30 fd 10 * :skala
rt 30 fd 20 * :skala
lt 30 fd (40 - 10 * cos 30) * :skala
lt 180
end
```

Tak jak we wszystkich poprzednich przypadkach dbamy o to, by żółw zaczynał rysowanie obiektu z poziomu gruntu, z jego lewej strony i by kończył również na poziomie gruntu po prawej stronie tego obiektu.



Rys. 26. Ulica może wyglądać tak

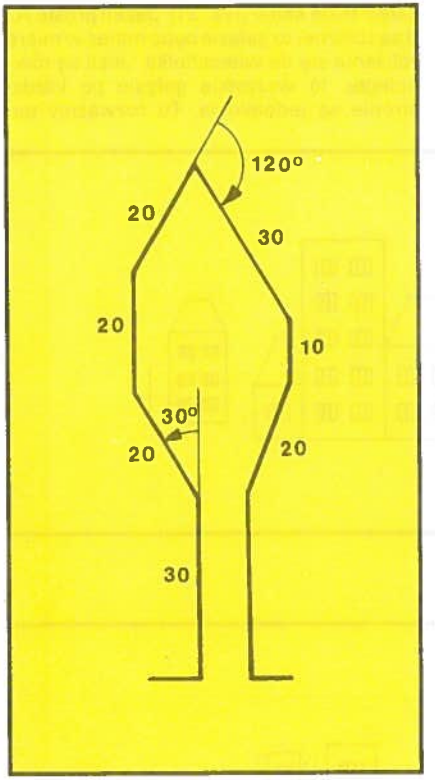
Przed rysowaniem i po jego zakończeniu żółw jest skierowany do góry, pod kątem 0. Narysowanie sosny w postaci stylizowanej jak na rys. 30 jest nieco trudniejsze; wykorzystajmy tę okazję do stworzenia kolejnego przykładu procedury rekurencyjnej. Przyjmujemy, że sosny mogą mieć różne liczby gałęzi, ale zawsze powinny być symetryczne. Kształt sosny będzie w dużej mierze zależał od położenia prostych A, B i prostych do nich symetrycznych, a także od nachylenia dolnych i górnych brzegów gałęzi. Możemy założyć, że te nachylenia są po każdej stronie sosny dla wszystkich gałęzi takie same (rys. 31). Jeżeli proste A i B są zbieżne, to gałęzie będą maleć w miarę zbliżania się do wierzchołka. Jeśli są równoległe, to wszystkie gałęzie po każdej stronie są jednakowe. Tu rozważmy ten drugi przypadek, jako prostszy, pozostawiając Czytelnikowi ten trudniejszy, pierwszy. Przyjmijmy, że sosna powinna mieć przynajmniej po dwie gałęzie z każdej strony:



Rys. 27. Albo tak

Rys. 28. Albo jeszcze inaczej

```
to sosna :ilegałęzi
if :ilegałęzi < 2 [stop]
lewastronapnia
gałęzie :ilegałęzi
```



Rys. 29. Drzewo liściaste

```
prawastronapnia
end
```

Rysowanie gałęzi zaprogramujemy rekurencyjnie (rys. 32):

```
to gałęzie :ilegałęzi
if :ilegałęzi = 0 [stop]
lewagałąź
gałęzie :ilegałęzi − 1
prawagałąź
end
```

Ponieważ założyliśmy, że wszystkie prawe gałęzie i lewe gałęzie są jednakowe, rysujące je procedury nie muszą mieć parametrów. Musiałyby mieć, gdyby rozmiar gałęzi miał zmieniać się wraz z wysokością, na której byłyby osadzone.
Przy rysowaniu gałęzi i stron pnia moglibyśmy posłużyć się komendami fd i bk, tak jak to robiliśmy w poprzednich przypadkach. Ponieważ jest to sposób już znany i chyby nie nastręczający żadnych trudności, spróbujmy zrobić to inaczej. Wykorzystajmy do przesuwania żółwia komendy setpos, setx oraz sety, a także operacje xcor i ycor. Wartością xcor oraz ycor jest w każdym momencie współrzędna x i współrzędna y żółwia. Komenda setx zmienia położenie żółwia w ten sposób, że zmianie ulega tylko jego współrzędna x, a współrzędna y pozostaje bez zmian. Inaczej mówiąc, żółw zostanie przesunięty poziomo do takiego miejsca, którego współrzędna x jest równa wartości argumentu tej komendy. Analogicznie działa sety, przesuwając żółwia pionowo do miejsca o zadanej współrzędnej y. Wobec tego rysowanie lewej strony pnia z dołu do góry, jako pionowego odcinka o długości 30 można, zapisać tak:

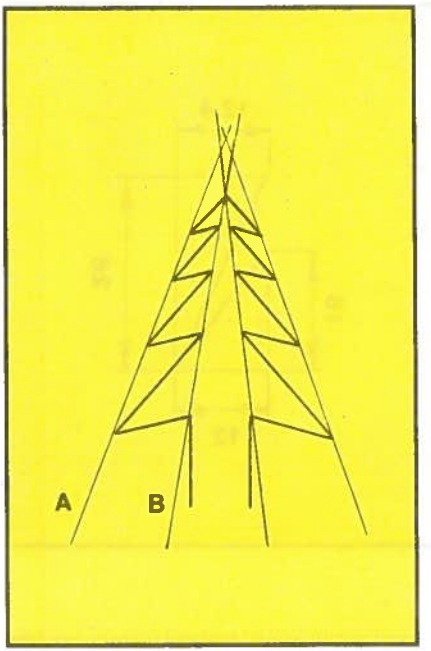
```
to lewastronapnia
sety ycor + 30 * :skala
end
```

Wprowadzenie mnożnika :skala ułatwi nadawanie sosnie różnych rozmiarów, stosownie do potrzeb. Dzięki temu, że w argumencie sety występuje ycor, odcinek pionowy o zadanej długości jest rysowany od aktualnego położenia żółwia.
Prawa strona pnia będzie rysowana z góry na dół, od gałęzi do ziemi:

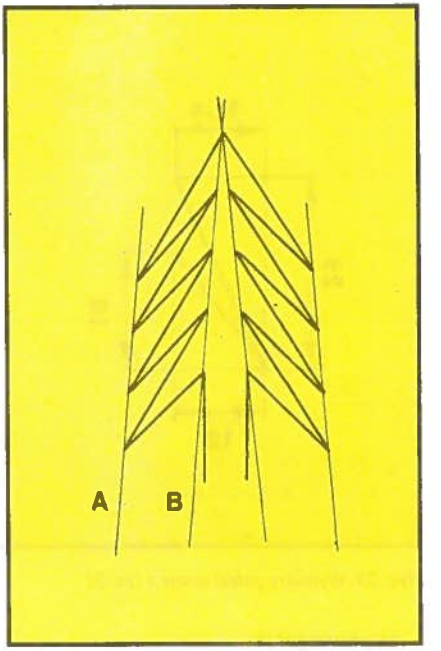
```
to prawastronapnia
sety ycor − 30 * :skala
end
```

Przy rysowaniu gałęzi mamy do czynienia z odcinkami skośnymi, a więc musimy użyć setpos, komendy doprowadzającej żółwia do miejsca o podanych współrzędnych. Para tych współrzędnych musi być ujęta w nawiasy kwadratowe – setpos jest jednoargumentowa. Przyjrzyjmy się, jak to można zrobić (rys. 33):

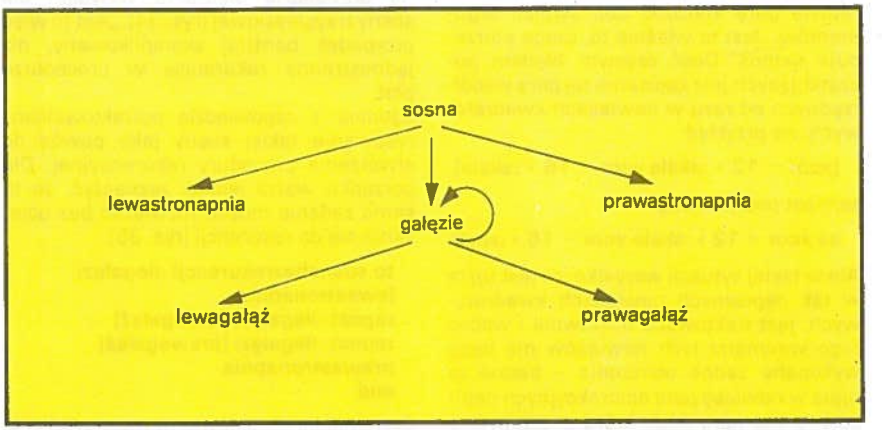
```
to lewagałąź
setpos se xcor − 12 * :skala ycor − 16 *
:skala
setpos se xcor + 12.4 * :skala ycor + 26
* :skala
end
```



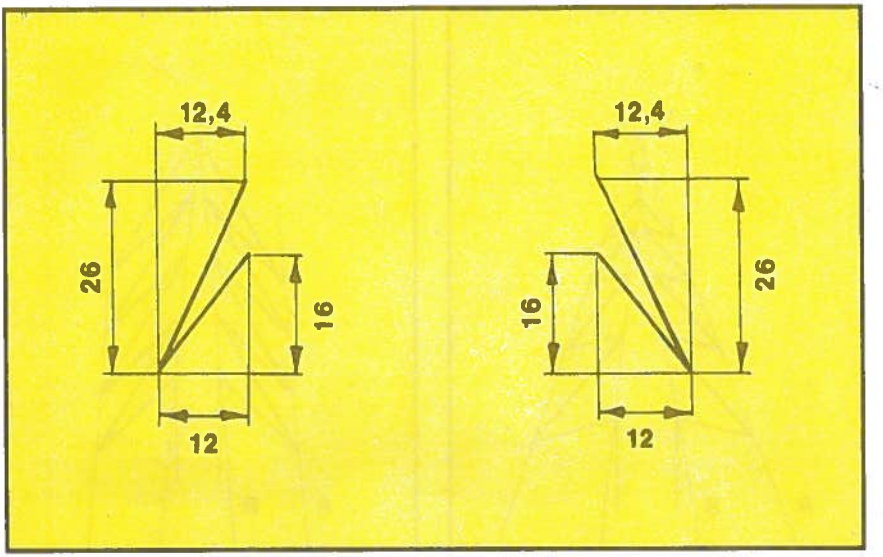
Rys. 30. Sosna schematycznie



Rys. 31. Schemat sosny z równoległymi liniami wierzchołków trójkątów



Rys. 32. Struktura powiązań procedur wywoływanych przez procedurę sosna z rekurencyjnym rysowaniem gałęzi



Rys. 33. Wymiary gałęzi sosny z rys. 31

```
to prawagałąź
setpos se xcor + 12.4 * :skala ycor - 26
* :skala
setpos se xcor - 12 * :skala ycor + 16 *
:skala
end
```

Operacja se tworzy ujętą w nawiasy kwadratowe parę wartości obu swoich argumentów. Jest to właśnie to, czego potrzebuje setpos. Dość częstym błędem początkujących jest zapisanie tej pary współrzędnych od razu w nawiasach kwadratowych, na przykład:

```
[xcor - 12 * :skala ycor - 16 * :skala]
```

zamiast poprawnego:

```
se xcor - 12 * :skala ycor - 16 * :skala
```

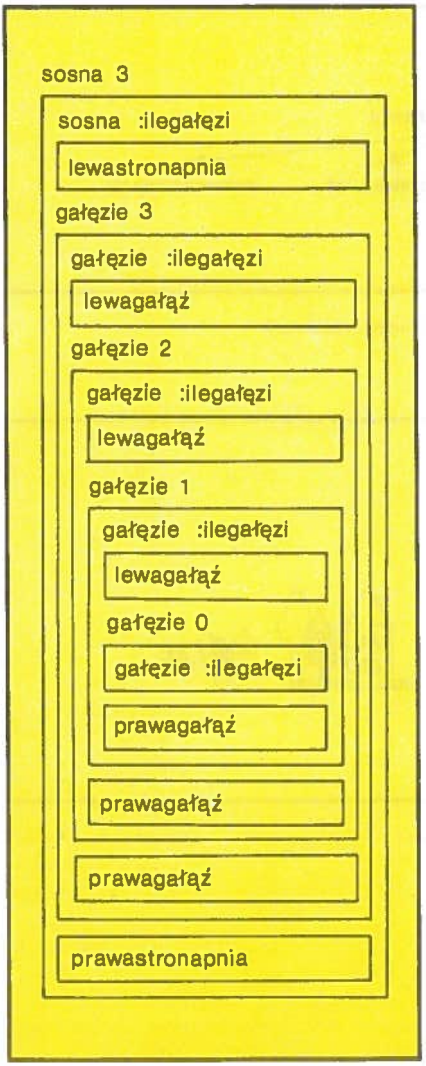
Ale w takiej sytuacji wszystko, co jest ujęte w tak napisanych nawiasach kwadratowych, jest traktowane dosłownie i wobec tego wewnątrz tych nawiasów nie będą wykonane żadne obliczenia - będzie to ujęta w nawiasy para abstrakcyjnych napisów, a nie para liczb, którą chcielibyśmy mieć. Użycie se w tej sytuacji zapewnia wykonanie wszystkich niezbędnych obliczeń zanim wyniki zostaną ujęte w nawiasy kwadratowe i to będzie właśnie to, o co w tym przypadku chodzi.

W odróżnieniu od procedury rysującej rekurencyjnie płot, tutaj wykonuje się pewne czynności zarówno przed wywołaniem rekurencyjnym, jak po nim. Widać to na przykładzie diagramu wywołań dla sosny trzygałęziowej (rys. 34). Jest to więc przypadek bardziej skomplikowany, niż jednostronna rekurencja w procedurze płot.

Zgodnie z zapowiedzią potraktowaliśmy rysowanie takiej sosny jako powód do stworzenia procedury rekurencyjnej. Dla porządku warto jednak zaznaczyć, że to samo zadanie można rozwiązać bez uciekania się do rekurencji (rys. 35):

```
to sosnabezrekurencji :ilegałęzi
lewastronapnia
repeat :ilegałęzi [lewagałąź]
repeat :ilegałęzi [prawagałąź]
prawastronapnia
end
```

Czytelnikom pozostawiamy rozstrzygnięcie kwestii, czy można w podobny sposób zaprogramować rysowanie sosny z gałę-



Rys. 34. Diagram wywołań procedur dla sosny z trzema gałęziami

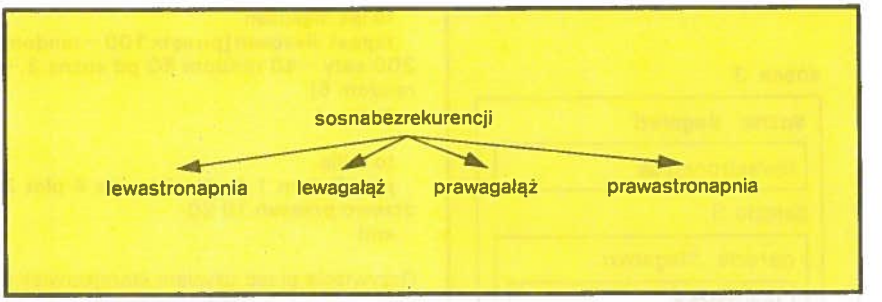
ziami, które maleją w miarę przesuwania się w kierunku wierzchołka.
Teraz możemy użyć sosen jako elementów krajobrazu:

```
to las :ilesosen
repeat :ilesosen [pu setx 100 - random
200 sety -40 random 50 pd sosna 3 +
random 6]
end

to willa
płot 2 dom 1 1 płot 3 sosna 4 płot 3
drzewo przesuń 10 20
end
```

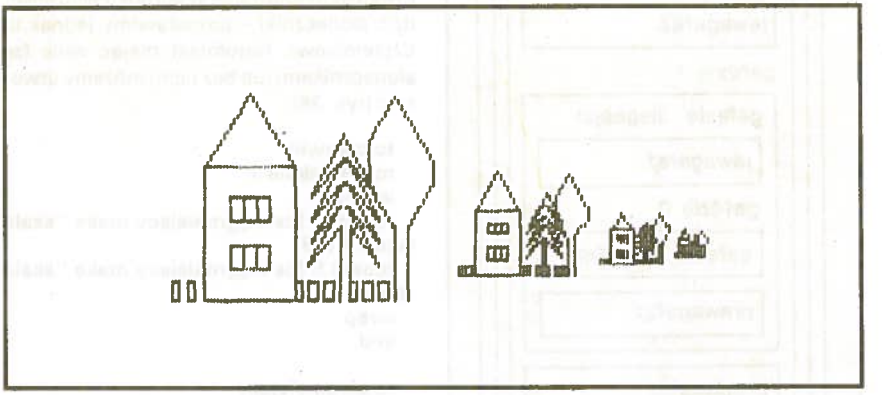
Oczywiście przed użyciem którejkolwiek z tych procedur warto upewnić się, że zmienna :skala ma już nadaną wartość i że ta wartość nam odpowiada.
Mając już opanowaną technikę rysowania kwiatów moglibyśmy w ogródku willi posadzić słoneczniki – pozostawimy jednak to Czytelnikowi. Natomiast mając wille (ze słonecznikami lub bez nich) możemy utworzyć (rys. 36):

```
to ciągwill
make "skala 1
window
repeat 5 [cs ciągmalejący make "skala
:skala * 0.8]
repeat 5 [cs ciągmalejący make "skala
:skala * 1.5]
wrap
end

to ciągmalejący
ht make "staraskala :skala
przesuń 0 -110
repeat 4 [willa make "skala :skala / 2]
przesuń -40 -100
make "skala :staraskala st
end
```

Procedura ciągwill pokazuje jakby sekwencję przezroczy z obrazami czterech will widzianych z różnych odległości – najpierw obrazy się zmniejszają, a potem z powrotem zwiększają. Wadą tych procedur jest może to, że wille na każdym obrazku wyglądają dokładnie jednakowo. Można poprawić tę kiepską architekturę modyfikując odpowiednio procedury ciągmalejący i willa tak, by te wille z ogródkami miały lepszy wygląd i nie były jak spod jednej sztancy.



Rys. 35. Struktura powiązań procedur rysujących sosnę bez rekurencji



Rys. 36. Ciąg will rysowanych w coraz mniejszej skali

Rekurencja a zmienne globalne

Miłą rozrywką może być rysowanie różnych zawijasów. W literaturze dotyczącej Logo można często napotkać następującą procedurę:

```
to spiral :krok :skok :kąt
fd :krok rt :kąt
spiral :krok + :skok :skok :kąt
end
```

Procedura ta nie ma badania warunku zakończenia i wobec tego sama nie może się zatrzymać. Można przerwać jej działanie ręcznie, naciskając klawisz BREAK. Jeśli nie zrobi się tego dość wcześnie, to żółw wyleci niepotrzebnie poza krawędź ekranu. Poza tym na ekranie nie można w sensowny sposób rozmieścić więcej niż jedną taką spiralę: nie ma możliwości komponowania z tych spiralek bardziej złożonych obrazów, bo żółw, startujący ze środka spirali, kończy rysowanie w przypadkowym miejscu, tam, gdzie go zastanie naciśnięcie klawisza BREAK. No a po naciśnięciu tego klawisza i tak cała akcja jest przerwana i wszystko trzeba zacząć od początku.

Zastanówmy się, jak zapobiec tym wszystkim kłopotom. Jako radę na wybieganie żółwia poza ekran można wprowadzić rysowanie spirali od zewnątrz do środka, a nie do środka na zewnątrz. Wtedy także zmniejszenie kroku poniżej pewnej granicy, na przykład poniżej wartości skoku, będzie sygnałem zakończenia rysowania spirali. O położeniu żółwia przed rysowaniem spirali i po nim można decydować rozmaicie. Przyjmijmy, że rysowanie zaczyna się od tego miejsca, gdzie żółw stoi, i w tym kierunku, w jakim jest skierowany, a po ukończeniu rysowania powinien powrócić na to samo miejsce i ustawić się pod tym samym kątem, jak na początku. Wycofywanie się do tego punktu ze środka spirali powinno być wykonane w sposób ekonomiczny – nie warto na przykład, by żółw wycofywał się wstecz po spirali dokładnie tą samą drogą, po której rysował. Wobec tego wycofywanie się po tej skróconej drodze nie może być częścią procedury rekurencyjnej, a dane o położeniu punktu, do którego trzeba wrócić, nie mogą być ustalane wewnątrz tej procedury, bo wtedy groziłoby to tym, że w każdym wywołaniu tej procedury na każdym poziomie rekurencji te wartości początkowe ustawiane byłyby na nowo, a więc albo niepotrzebnie, albo źle, skoro trzeba je wyznaczyć tylko raz. Oznacza to, że dane o początkowym położeniu i kącie żółwia muszą być zapisane jako wartości zmiennych, które dla procedury rekurencyjnej są globalne – były utworzone na zewnątrz procedury i nie używa się parametrów procedury do przekazywania ich wartości. Postąpimy więc w sposób następujący:

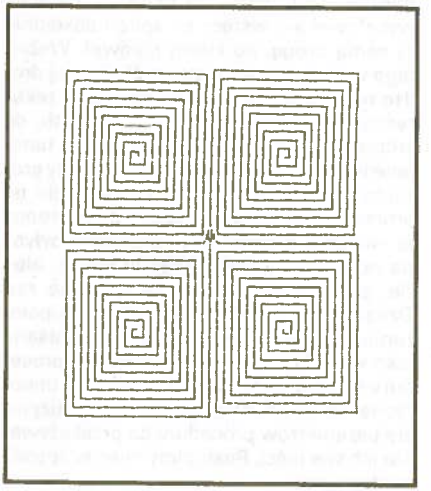
```
to spirala :krok :skok :kąt
make "pozycjapocz pos
make "kątpocz heading
spiralarek :krok :skok :kąt pu
setpos :pozycjapocz pd
seth :kątpocz
end

to spiralarek :krok :skok :kąt
if :krok < :skok [stop]
fd :krok rt :kąt
spiralarek :krok − :skok :skok :kąt
end
```

Wartością przekazywaną przez pos jest ujęta w nawiasy kwadratowe para współ-

rzędnych żółwia, czyli właśnie to, co będzie później potrzebne komendzie setpos. Heading jest kątem położenia żółwia, mierzonym w stopniach; podanie tej liczby komendzie seth powoduje obrócenie żółwia z powrotem w takim kierunku, w jakim był poprzednio.

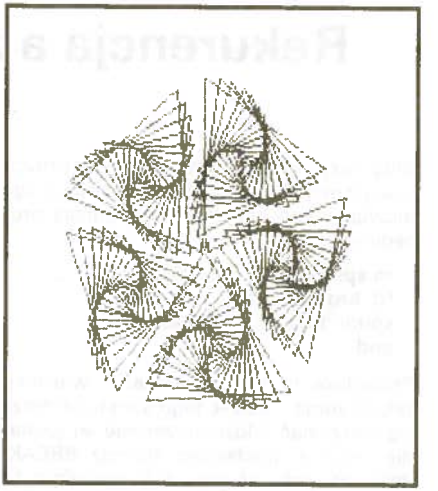
Mając taką procedurę spirala możemy tworzyć różne kompozycje, jak na przykład (rys. 37):

```
to pękspiral :ilespiral :krok :skok :kąt
window cs st
repeat :ilespiral [spirala :krok :skok :kąt
rt 360/:ilespiral]
wrap
end
```

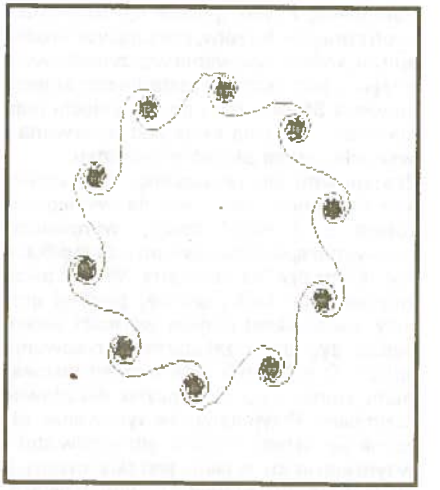


Rys. 37. Pękspiral 4 80 2 90

Znajomość warunku zakończenia działania procedury spiralarek umożliwia oszacowanie liczby wywołań rekurencyjnych niezbędnych dla narysowania konkretnej spirali o zadanych parametrach. Podobnie jak to było w przypadku sosny, mając tę liczbę (wyrażoną w zależności od wartości parametrów) możemy znaleźć inną formę procedury, bez rekurencji, ale za to z komendą repeat. Polecamy Czytelnikowi napisanie jej jako ćwiczenia. Nie znaczy to, że wersja z repeat jest lepsza od wersji z rekurencją. Obie formy są zasadniczo równie dobre, ale takie przyjrzenie się różnym możliwym formom procedur i metodom rozwiązywania zagadnień pozwala lepiej zrozumieć ich naturę i daje większą swobodę wyboru rozwiązań wtedy, gdy napotkamy zadania naprawdę trudne (np. rys. 38).



Rys. 38. Pękspiral 5 80 2 122



Rys. 39. Wynik inspi 6 2 6

Nie zawsze ustalenie warunku zakończenia wykonania procedury rekurencyjnej przychodzi z równą łatwością. Oto inny przykład popularnej rozrywkowej procedury rekurencyjnej bez zakończenia (por. rys. 39, 40, 41):

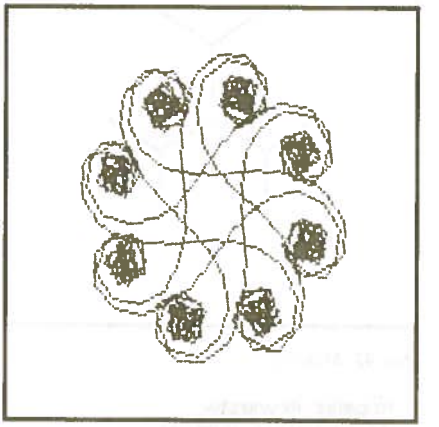
```
to inspi :krok :kąt :skokkąta
fd :krok rt :kąt
inspi :krok :kąt + :skokkąta :skokkąta
end
```

Znowu możemy przyjąć, że po zakończeniu rysowania żółw powinien mieć takie samo położenie i kąt, jak przed rysowaniem, co oznacza wprowadzenie odpowiednich zmiennych globalnych, tak jak w poprzednim przykładzie.

```
to spiralki :krok :kąt :skokkąta
make "pozycjapocz pos
make "kątpocz heading
spiralkirek :krok :kąt :skokkąta
pu setpos :pozycjapocz pd
seth :kątpocz
end
```

Rys. 40. Inspi 6 0 7

```
to spiralkirek :krok :kąt :skokkąta
if ... [stop]
fd :krok rt :kąt
spiralkirek :krok :kąt + :skokkąta
:skokkąta
end
```

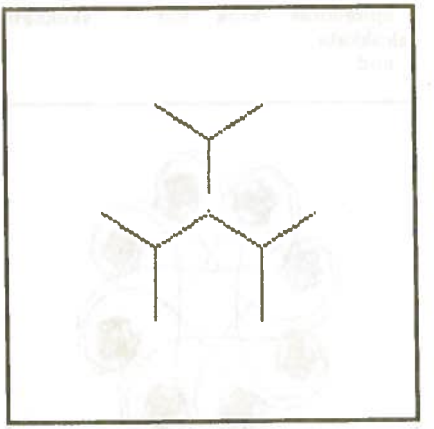


Rys. 41. Inspi 6 5 8

Co należy wstawić na miejsce kropek? Pozostawiamy również to zadanie Czytelnikowi. Warto w każdym razie przyjrzeć się dokładnie sposobowi wykonywania oryginalnej inspi dla różnych wartości parametrów. Wartość parametru krok określa tylko skalę rysunku i na początek można przyjąć ją około 6. Kształt rysunku zależy od wartości pozostałych dwu parametrów. Na początek można je dobierać tak, by nie były zbyt wielkie, później – według uznania.

I wreszcie przykład może najtrudniejszy do zanalizowania, w którym występuje zarówno rekurencja, jak repeat. Zadaniem procedury miotła przy podanej liczbie warstw n jest narysowanie wychodzących z jednego punktu i symetrycznie rozmieszczonych n gałęzi, z których każda rozwidla się na n−1 gałązek, każda z nich z kolei na n−2 i tak dalej. Rysunki wychodzą dość dobrze dla liczby warstw nie przekraczającej 8, potem już ekran staje się za ciasny (rys. 42, 43, 44).

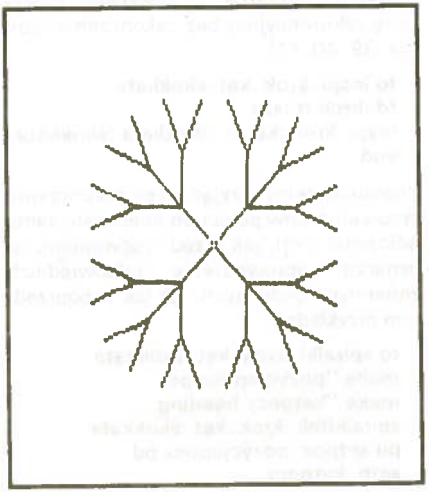
```
to miotła :ilewarstw
make "kąt 360 * (ilewarstw − 1) /
:ilewarstw
make "długość 80 / :ilewarstw
cs gałąź :ilewarstw − 1
end
```



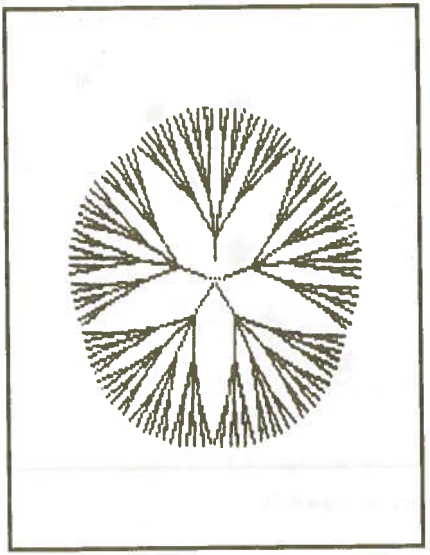
Rys. 42. Miotła 3

```
to gałąź :ilewarstw
if :ilewarstw = 0 [stop]
if :kąt <12 [fd :długość gałąź :ile-
warstw − 1 bk :długość stop]
lt :kąt / 2
make "kąt :kąt / :ilewarstw
repeat :ilewarstw [fd :długość gałąź
:ilewarstw − 1 bk :długość rt :kąt]
fd :długość gałąź :ilewarstw − 1 bk
:długość
make "kąt :kąt * :ilewarstw
lt :kąt / 2
end
```

Jest to przykład, w którym wartości zmiennych globalnych są modyfikowane wewnątrz procedury rekurencyjnej. Jeśli się to robi, to należy zachować dużą ostrożność, bo nietrudno o pomyłkę, a skutki błędów mogą być dość skomplikowane, bo będą mnożyć się tyle razy, ile będzie wywołań rekurencyjnych (w najgorszym przypadku oczywiście). Dobrą zasadą jest, że jeśli już procedura modyfikuje wartość zmiennej globalnej, to po wykonaniu właściwych czynności powinna przywracać starą wartość, tak by po wykonaniu procedury była ona taka sama, jak przed jej rozpoczęciem. Widzimy zastosowanie tej reguły w procedurze gałąź: jeśli wartość zmiennej globalnej zostaje podzielona przez wartość parametru :ilewarstw, to przed zakończeniem wykonania procedury przywraca się starą wartość zmiennej :kąt, mnożąc go przez tę samą wartość parametru.



Rys. 43. Miotła 4



Rys. 44. Miotła 5

Geometria żółwia

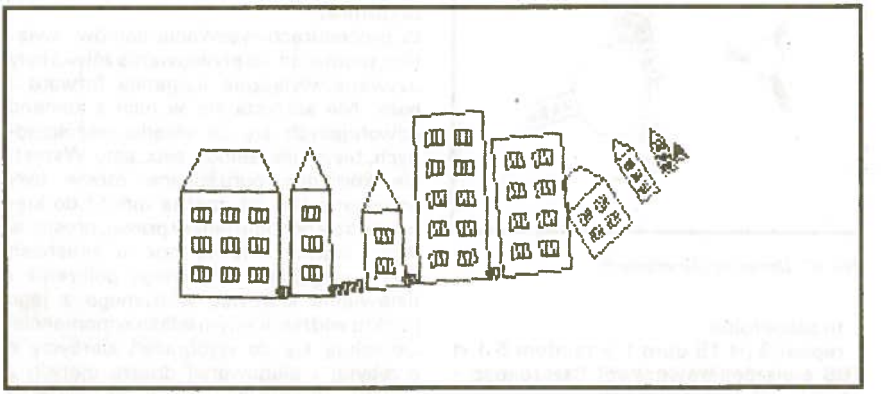
Dotychczas wszystkie krajobrazy były na terenie równinnym. Zobaczymy teraz, jak może wyglądać ulica na terenie pagórkowatym, modyfikując nieco znaną nam już procedurę o tej nazwie (np. rys. 45):

```
to ulicanagórkach
cs window pu setpos [-120 -80] pd
make "skala 0.8
repeat 4 [dom random 5 1 + random 3
rt (-10 + random 5) płot 2 + 3 *
random 2 rt 10 - random 5]
repeat 4 [rt (-10 + random 5) dom
random 5 1 + random 2 rt 10 -
random 5 przesuń 10 20 make "skala :s
kala * 0.7]
end
```

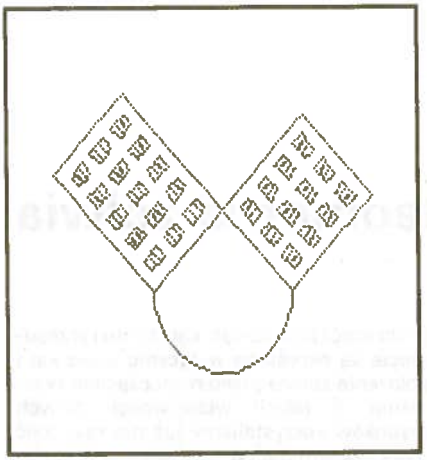
Rysunek jest nieco rozchwiany. Jak widać, można rysować domy i płoty pod różnymi kątami i w różnych położeniach, a ich kształt nie ulegnie przez to zmianie. Można je obracać i przesuwać; kąt obrotu i przesunięcie są określone wyłącznie przez kąt i położenie żółwia przed rozpoczęciem rysowania. Z takich właściwości różnych rysunków korzystaliśmy już nie raz, choć może czasem nie zwracaliśmy na to wyraźnie uwagi. Teraz możemy zobaczyć, jak mogłyby wyglądać bloki mieszkaniowe wybudowane na planecie Małego Księcia (rys. 46):

```
to asteroid
cs make "skala 0.5
repeat 2 [lt 45 dom 3 + random 2 3 rt
135]
repeat 2 [ćwierćokrągwprawo
:szerokość * :skala * cos 45]
end
```

albo budownictwo w nieco innym stylu rozprzestrzenione na planetoidach o mniejszym zaludnieniu (rys. 47):

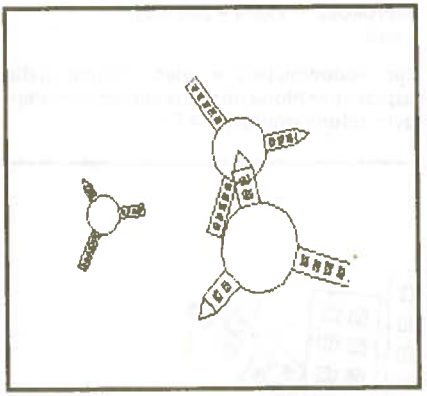


Rys. 45. Ulica na górkach



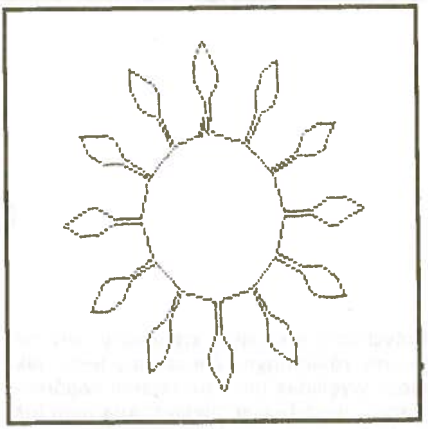
Rys. 46. Bloki na asteroidzie Małego Księcia

```
to planetoidy :ile
repeat :ile [pu setx 100 − random 200
sety (−50 + random 100) seth random
360 make "skala (1 + random 8) / 20 pla-
netoida]
end
```



Rys. 47. Domy na planetoidach

```
to planetoida
repeat 3 [rt 15 dom 1 + random 5 1 rt
105 ćwierćokrągwprawo! (:szerokość *
:skala) / (2 * sin 15) lt 90]
end
```



Rys. 48. Park

Na takiej planetoidzie można założyć park (rys. 48):

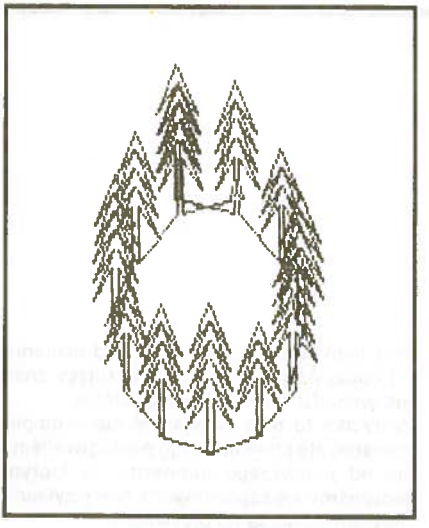
```
to park
repeat 12 [drzewo rt 105 fd 35 * :skala
lt 75]
end
```

Posadzone w nim drzewa liściaste będą rosły prosto, zgodnie z prawem grawitacji. Nie uda się to jednak z sosnami (rys. 49):

```
to lasek
repeat 12 [sosna 3 + random 3 rt 105
fd 35 * :skala lt 75]
end
```

i Logo nie będzie działać prawidłowo. Skąd ta różnica?

W procedurach rysowania domów, kwiatów, płotów itd, do przesuwania żółwia były używane wyłącznie komendy forward i back. Nie korzysta się w nich z komend odwołujących się do układu współrzędnych, takich jak: setpos, setx, sety. Wszystkie komendy poruszające żółwia były podawane tak, jak można mówić do kierowcy samochodu: jedź w prawo, prosto, w lewo, wstecz. Mówią one o zmianach ruchu względem aktualnego położenia i ustawienia kierowcy widzianego z jego punktu widzenia w tym właśnie momencie. Odwołują się do wyobrażeń kierowcy o przebytej i planowanej drodze ujętych z punktu widzenia jego aktualnego położenia i kierunku ruchu; do zestawu pojęć prze-



Rys. 49. Próba zasadzenia sosen, których nie można obrócić: zniekształcenie rysunku jest widoczne zwłaszcza w górnej części

strzennych, które można by nazwać geometrią kierowcy. Analogicznie można mówić o „geometrii" sterowanego w taki sposób żółwia. Operuje ona opisami przebywanej drogi ujmowanymi z punktu widzenia kolejnych położeń żółwia, a nie w odniesieniu do jakiegokolwiek sztywnego układu współrzędnych.

Dla wyraźniejszego podkreślenia genezy takiego sposobu opisywania elementów przestrzeni można ten rodzaj geometrii żółwia nazwać geometrią wewnętrzną.

Procedura sosna jest przykładem innego podejścia. Tu każdy odcinek jest określony przez parę przyrostów współrzędnych x i y w ustalonym układzie współrzędnych ekranu. Przyrosty te nie zmienią się, jeżeli przesunie się początkowy punkt, z którego zaczyna się rysować odcinek. Są więc możliwe przesunięcia równoległe i położenie rysowanego elementu zależy tylko od pozycji żółwia w momencie rozpoczęcia rysowania. Ale nie są możliwe obroty, bo każdy obrót (o kąt różny od wielokrotności 360°) musiałby wiązać się ze zmianą proporcji przyrostów wartości zmiennych x i y, co jest niemożliwe. Kąt ustawienia żółwia przed rozpoczęciem rysowania nie ma wpływu na jego przebieg, bo wykonanie setpos, setx, sety nie zależy od wartości heading ani jej nie zmienia.

I wreszcie trzecim sposobem jest sztywne przywiązanie rysunku do układu współrzędnych ekranu, gdy przy rysowaniu i przesuwaniu żółwia używa się setpos, setx, sety, w których argumentami są bezwzględne wartości współrzędnych punktów, a nie przyrosty dodawane do wartości xcor lub ycor.

Wybór jednego z tych sposobów rysowania zależy od tego, jakim celom rysunek ma służyć i jakim ewentualnym przekształceniom miałyby podlegać jego elementy. W szczególności, czy chcemy zapewnić swobodę wykonywania przesunięć i obrotów, czy umożliwić wykonywanie wszystkich przekształceń, czy tylko niektórych, czy w ogóle żadnych. Wracając do przykładu z elementami krajobrazów, nie jest może źle, że sosna nie rośnie prostopadle do zbocza, a zawsze pionowo, ale za to dobrze, że można ją posadzić na dowolnej wysokości. Co innego będzie, jeśli napotkamy małe planetoidy.

Słowa

Słowo w sensie Logo jest to ciąg dowolnych znaków, który nie zawiera
- nawiasów kwadratowych [ ]
- odstępów
- znaków + − * / = <> ( ) { } nie poprzedzonych bezpośrednio przez " lub \.

Znaki operacji arytmetycznych + − * /, relacji = <> oraz nawiasy ( ) { } [] Logo próbuje zawsze zinterpretować zgodnie z ich znaczeniem, niezależnie od kontekstu, w jakim się znajdują. Na przykład ciąg znaków :a+:b nie będzie potraktowany jako spójna całość, ale jako suma :a + :b, przy czym te odstępy po obu stronach znaku dodawania mogą być dostawione automatycznie. To samo może zdarzyć się z dowolnym ciągiem zawierającym = lub jakikolwiek inny znak operacji arytmetycznej lub relacji. Przy nawiasach nie pojawiają się odstępy, ale Logo wykona odpowiednią analizę napisu, żeby ustalić, gdzie jest początek i koniec domniemanego wyrażenia. Próby takie można powstrzymać, jeżeli taki znak poprzedzi się cudzysłowem ". Wtedy + = czy jakikolwiek inny znak operacji, relacji lub nawias będzie potraktowany dosłownie, to znaczy tylko jako znak, ale poprzedzający go cudzysłów będzie także znakiem tego napisu. Nie dotyczy to nawiasów kwadratowych [ ], których nie można w ten sposób zneutralizować: napisanie "[nie powstrzyma Logo od poszukiwania odpowiadającego temu nawiasowi nawiasu kwadratowego zamykającego.

Każdy znak, łącznie z nawiasami kwadratowymi i odstępami, będzie wzięty dosłownie (to znaczy tylko jako ten znak, bez nadawania mu jakiejkolwiek interpretacji), jeśli będzie bezpośrednio poprzedzony odwrotną skośną kreską \. Dotyczy to również tego samego znaku \. W odróżnieniu od cudzysłowu " ten neutralizujący znak nie wchodzi w skład samego słowa.

Wszystko to może wydawać się skomplikowane, ale słowami Logo posługiwaliśmy się od pierwszego momentu, w którym zaczęliśmy się zapoznawać z tym językiem. Słowami Logo są na przykład:

```
CS
home
co??
z_podkreśleniami_zamiast_odstępów
stara.wartość.x
x
:x
"x
"+
:a"+:b
:a\+:b
tak\ też\ można\ ale\ nie\ warto\]
```

Nie są natomiast słowami:

czarno-biały	(znak − w środku, nie zneutralizowany " ani \)
+	(znak operacji arytmetycznej)
[]	(nawiasy kwadratowe)
lewa > prawa	(znak porównania w środku)
lewa\\> prawa	(to samo: pierwszy\ neutralizuje działanie drugiego)

Słowami są wszystkie liczby, które można użyć w Logo. Jeśli na początku słowa jest cyfra albo − (jest to jedyny przypadek, gdy użycie tego znaku w słowie jest w zasadzie dozwolone) i długość słowa przekroczy 36

znaków, to sygnalizowany jest błąd: za duża liczba. Wynika to stąd, że Logo ocenia wstępnie, że słowo tej postaci powinno być liczbą i bada przede wszystkim, czy ta domniemana liczba nie przekroczy dopuszczalnego zakresu. Stwierdzenie przekroczenia długości 36 znaków blokuje bliższe badanie słowa. Nie jest to poprawne, ale może niezbyt szkodliwe: dość rzadko musi się używać słów dłuższych od 36 znaków i na dodatek zaczynających się od cyfry. Dla słów zaczynających się od znaku różnego od cyfry lub — ten defekt nie występuje i ich długość może przekroczyć 36.

Dla słów będących liczbami mamy oczywiste wyjątki od zakazu stosowania znaków + i — wynikające z zasad budowy liczb w Logo. W słowach nie będących liczbami tylko w bardzo wyjątkowych przypadkach konieczność zmusza do wprowadzania znaków zakazanych do słów. Możemy więc w dalszym tekście założyć, że jeśli słowo nie jest liczbą, to nie zawiera żadnego ze znaków wymienionych w określeniu na początku tych rozważań: nie ma w nim ani odstępów, ani żadnych nawiasów, ani znaków operacji arytmetycznych, ani znaków relacji, ani \. W konsekwencji możemy także założyć, że w środku słowa nie wystąpi znak cudzysłowu ". Uprości to znacznie dalsze rozważania.

Wszystkie nazwy, używane w Logo, są słowami. Są więc nimi nazwy procedur języka i procedur definiowanych, nazwy parametrów procedur i nazwy zmiennych, a także stałe, takie jak na przykład "true lub "false. Interpretacja i sposób potraktowania słowa zależy od kontekstu, w jakim się ono znajdzie. Jeśliby ktoś chciał użyć tej samej nazwy dla zmiennej i definiowanej przez siebie procedury, to może to zrobić. Logo nie stawia tu żadnych ograniczeń. Inną jest sprawą czy warto: jeśli się już na to zdecydujemy, to trzeba bardzo uważać, by nie paść ofiarą stworzonych przez siebie nieporozumień.

Nazwa nie poprzedzona dwukropkiem ani cudzysłowem jest z reguły traktowana jako nazwa procedury i jeśli nie da się jej prawidłowo zinterpretować w tej roli, to sygnalizowany jest błąd. Jeśli chcemy mieć wartość zmiennej, należy przed jej nazwą umieścić dwukropek. Dlatego na przykład umieszcza się dwukropki przed nazwami parametrów procedur — będziemy później operować wartościami tych parametrów. Poprzedzenie słowa cudzysłowem oznacza, że należy je wziąć dosłownie jako stałą napisową. Widzimy więc, że dwukropek i cudzysłów służą jako operatory jednoargumentowe, działające na słowach. Tak samo, jak to było z operatorem jednoargumentowym —, nie można ich oddzielać odstępem od ich argumentów. Napisanie : x z odstępem, zamiast :x, byłoby błędem. Istnieje jeden wyjątek od tych reguł. Słowo znajdujące się wewnątrz pary nawiasów kwadratowych (czyli, jak zobaczymy niżej, będące elementem listy) jest zawsze brane dosłownie jako stała napisowa i, na przykład, poprzedzenie go w tej sytuacji dwukropkiem byłoby bezskuteczne – nie spowodowałoby pobrania wartości zmiennej o tej nazwie. Tak samo słowo w tej sytuacji nie może być zinterpretowane jako wywołanie procedury.

Procedura word służy do składania słów. Jest ona zasadniczo dwuargumentowa, ale jest zachłanna w tym samym sensie jak poznanie poprzednio sum czy product – jeśli ujmiemy w nawiasy okrągłe operator word i następujący po nim dowolny ciąg dwu lub więcej słów, oddzielonych od siebie odstępami, to wartością takiego wyrażenia będzie słowo powstałe ze zlepienia wszystkich tych słów, albo ich wartości, jeśli to były zmienne lub wywołania procedur. Na przykład:

```
print word "x 15
```

wypisze na ekranie

```
x15
```

zaś

```
print (word "Kon "stan "ty "no "pol.)
```

wypisze

```
Konstantynopol.
```

Aby zapis był łatwiejszy do odczytania, dobrze jest nawias zamykający oddzielić odstępem od ostatniego argumentu. Tu robimy to tylko dla czytelności – w niektórych dialektach Logo dla innych komputerów jest to obowiązkowe.

Weźmy jeszcze inny przykład – generowanie nazw zmiennych. Komenda:

```
pons
```

powoduje pokazanie na ekranie nazw i wartości wszystkich zmiennych globalnych, to znaczy utworzonych przy pomocy komendy make. Przypuśćmy, że nie ma wśród nich zmiennych o nazwie zaczynającej się od zmie z następującą po tym liczbą. Wykonajmy:

```
make "i 0
repeat 5 [make "i :i + 1 make word
"zmie :i :i * :i]
pons
```

Pojawiło się pięć nowych zmiennych, od zmie1 do zmie5, których wartości są kwadratami współczynników, czyli liczb kończących nazwy tych zmiennych. Możemy teraz wypisać sobie tabelkę kwadratów liczb od 1 do 5:

```
make "i 0
repeat 5 [make "i :i + 1 (print :i thing
word "zmie:i)]
```

Użyta tu komenda thing wyznacza wartość zmiennej, której nazwa jest wartością jej argumentu. Ponieważ w tym przypadku nazwa była utworzona przy pomocy procedury word, użycie operatora : byłoby nieskuteczne. Jeżeli natomiast nazwa już istnieje i ją znamy, to można użyć zarówno dwukropka (np. :i), jak thing (np. thing "i). Wartościami zmiennych mogą być nie tylko liczby, ale także listy lub słowa; w szczególności wartością zmiennej może być nazwa innej zmiennej. Oto przykład:

```
make "i 0
repeat 4 [make "i :i + 1 make word
"zmie :i word "zmie :i + 1]
pons
```

Widzimy skutek – wartością zmiennej zmie1 jest nazwa zmie2 i tak dalej, tylko wartość zmie5 pozostaje nie zmieniona. Operację thing można superponować, otrzymując wartość wartości zmiennej i tak dalej:

```
print thing "zmie5
pront thing thing "zmie4
```

dają w tej sytuacji ten sam wynik. Można to napisać prościej, używając dwukropka:

```
print :zmie5
print thing :zmie4
```

W drugim z tych wierszy thing być musi, bo operatora : nie można superponować i zapis : :zmie4 byłby odczytany jako próba wzięcia wartości zmiennej :zmie4, a zmiennej o takiej nazwie, zaczynającej się od dwukropka, nie ma.
Ponieważ teraz wszystkie te zmienne stworzyły łańcuch, do tej samej liczby, będącej wartością zmie5, można dotrzeć, zaczynając od pierwszej z nich:

```
print thing thing thing thing :zmie1
```

W Logo nie ma tablic i zmiennych wskaźnikowych, ale możliwość stosowania takiej techniki pozwala się bez nich obejść.
Następne przykłady stosowania zmiennych generowanych zobaczymy niżej, teraz zajmijmy się drugim ważnym pojęciem Logo – listą. Z listami, jak to zaraz zobaczymy, mieliśmy do czynienia już od momentu, kiedy zaczęliśmy stosować REPEAT, czyli kiedy pojawiły się sekwencje słów ujęte w nawiasy kwadratowe.

Listy

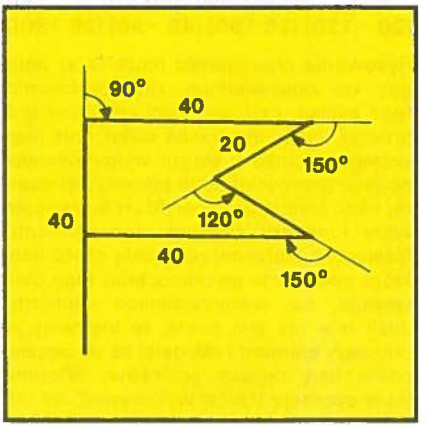
Lista jest to ciąg słów lub list ujęty w nawiasy kwadratowe albo para nawiasów kwadratowych. W tym ostatnim przypadku mówimy, że lista jest pusta. Jeżeli w liście słowa następują po sobie, to muszą być oddzielone odstępami, jak np.:

[Ta lista ma 7 różnych elementów !]
[[Ta lista] [[ma] [ich [tylko 3]] czyli aż [o 4] mniej] !]

Listy są wartościami niektórych operacji. Na przykład pos podaje parę współrzędnych żółwia, ujętą w nawiasy kwadratowe, czyli listę dwuelementową. Jeśli wartością argumentu operacji text jest nazwa procedury definiowanej, to wynikiem tej operacji jest lista list zbudowanych z kolejnych wierszy procedury o podanej nazwie. O listach w komendach Logo, takich jak repeat, już wspomnieliśmy, ale warto zauważyć, że można tu stosować także listy list, jak chociażby:

repeat 6 [rt 60 repeat 6 [fd 40 rt 60]]

Jeśli przedtem było wykonane cs, to otrzymamy obraz jak na rys. 50. W każdym razie jest to łamana. Wszystkie rysunki, rysowane przez żółwia faktycznie składają się z łamanych; jak widzieliśmy, nawet łuki okręgu są przybliżane fragmentami wielokąta. Po każdej takiej łamanej żółw przechodzi z piórem opuszczonym, rysując ją, a przy przejściu od jednej łamanej do innej podnosi pisak. Wtedy opis rysunku można przedstawić jako listę opisów łamanych. Opis łamanej, jak się o tym zaraz przekonamy, też może być listą. W ten sposób opis rysunku na ekranie można przedstawić w postaci listy list. Jest to coś nowego. Dotychczas rysunek był określany przy



Rys. 50. Rysowanie chorągiewki

pomocy procedury, która go rysowała. Teraz otwierają się przed nami nowe możliwości.
Opisać łamaną przy pomocy listy można na wiele sposobów. Pokazany tutaj należy do najprostszych. Pomysłowy Czytelnik może sam wypróbować inne.
Chorągiewka z rys. 50 jest jedną łamaną i rysująca ją procedura może wyglądać na przykład tak:

to chorągiewka
fd 40 rt 90 fd 40 rt 150 fd 20 rt −120
fd 20 rt 150 fd 40 rt −90 fd 20 rt 180
end

Dla uproszczenia pomijamy w tych rozważaniach takie szczegóły, jak kolor rysowanych kresek.

W tym ujęciu można każdemu odcinkowi łamanej przyporządkować parę liczb: długość odcinka oraz kąt, o jaki żółw ma obrócić się w prawo, gdy już ten odcinek przejdzie. Jeśli każda taka para liczb zostanie ujęta w nawiasy kwadratowe, to opis łamanej w tej konwencji będzie listą list dwuelementowych. Dla naszej chorągiewki będzie to:

```
[[ 40 90] [40 150] [20 -120] [20 150]
[40 -90] [20 180]]
```

Możemy tę listę zapamiętać:

```
make "proporczyk [[40 90] [40 150]
[20 -120] [20 150] [40 -90] [20 180]]
```

Rysowanie chorągiewki może teraz polegać na odpowiednim zinterpretowaniu tego zapisu, czyli wartości zmiennej proproczyk. Najpierw trzeba wziąć opis pierwszego odcinka (i skrętu wykonywanego po jego przebyciu), czyli pierwszy element tej listy. Trzeba wykonać fd i rt biorąc pierwszy i ostatni element tego elementu. Następnie bierzemy pozostałą część listy, która powstanie po odrzuceniu tego pierwszego, już wykorzystanego elementu. Jeśli ona nie jest pusta, to bierzemy jej pierwszy element i tak dalej aż do wyczerpania listy opisów odcinków. Widzimy, jakie operacje trzeba wykonywać, by taki listowy opis łamanej można było właściwie zinterpretować. Logo je ma.
Następujące operacje jednoelementowe można stosować do list i słów (obok nazwy procedury podano jej wartość)

first	**pierwszy element**
last	**ostatni element**
butfirst, bf	**pozostałość po odrzuceniu pierwszego elementu**
butlast, bl	**pozostałość po odrzuceniu ostatniego elementu**

Oto procedura rysująca łamaną:

```
to rysunekłam :łamana :skala
if :łamana = [] [stop]
fd :skala * first first :łamana
rt last first :łamana
rysunekłam butfirst :łamana :skala
end
```

Wprowadzenie skali jako dodatkowego parametru pozwoli otrzymać obrazki w różnej skali. Równie łatwo można uzyskać odbicie zwierciadlane takiej łamanej – wystarczy tylko zastąpić fd przez bk oraz rt przez lt:

```
to odbicielam :łamana :skala
if :łamana = [] [stop]
bk :skala * first first :łamana
lt last first :łamana
odbicielam bf :łamana :skala
end
```

Zgodnie z zasadami geometrii żółwia, z faktu, że w tych procedurach były użyte tylko fd, bk, lt i rt wynika, że położenie początku łamanej i jej nachylenie zależą od początkowego położenia żółwia. Dzięki temu można rysować łamaną razem z jej odbiciem lustrzanym:

```
to lustro :łamana :skala
pd st rysunekłam :łamana :skala
pu ht
sety -ycor
seth -heading
pd odbicielam :łamana :skala
pu
sety -ycor
seth -heading
pd st
end
```

Możemy teraz porozrzucać na ekranie proporczyki różnych rozmiarów w różnych położeniach; zobaczymy nie tylko je, ale także ich odbicia lustrzane (rys. 51):

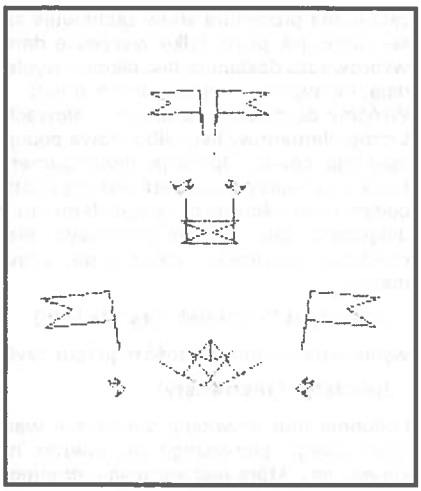
```
to proporczyki :ile :łamana
repeat :ile [pu home przesuń random
60 (-100 + random 200) rt random 360
lustro :łamana (0.2 * (1 + random 4))]
end
```

Oczywiście jeśli zamiast chorągiewki z rys. 50 weźmiemy łamaną opisującą coś innego, obraz będzie wyglądał inaczej.
A jak narysować rysunek składający się z wielu łamanych?

```
to rysunek :lista :skala
if :lista = [] [stop]
rysunekłam first :lista :skala
rysunek bf :lista :skala
end
```

Jego odbicie zwierciadlane:

```
to odbicie :lista :skala
if :lista = [] [stop]
```



Rys. 51. Losowo rozmieszczone chorągiewki i ich odbicia lustrzane

```
odbiciełam first :lista :skala
odbicie bf :lista :skala
end
```

Zwróćmy uwagę na to, że wszystkie te procedury, których zadaniem jest przeglądanie listy i branie jej kolejnych elementów, są rekurencyjne. Jest to rekurencja stosunkowo prosta, tworzona zawsze według tego samego schematu. Jest to najłatwiejszy sposób zaprogramowania takiego przeglądania list od początku do końca, połączonego z wykonywaniem jakichś czynności na jej elementach.

Tak jak to było w przypadku łamanych, o położeniu rysunku lub odbicia decyduje początkowe położenie żółwia, które wyznacza przesunięcie rysunku względem środka ekranu, oraz kąt początkowy żółwia, określający obrót rysunku (lub jego odbicia) względem kierunku pionowego. Dobierając jeszcze skalę można otrzymywać rysunek lub jego odbicie w różnych wersjach, dowolnie go kopiować, by mieć go w większej liczbie egzemplarzy ułożonych na ekranie w jakiś sposób. Wszystko to jest podobne do tego, co widzieliśmy poprzednio w przypadku pojedynczej łamanej i wobec tego nie warto się tym szczegółowo zajmować, pozostawiając resztę pomysłowości Czytelnika.

Dotychczas mówiliśmy, jak wydobyć elementy listy lub jej części. Teraz zajmiemy się tym, jak można listy tworzyć.

Zachłanna operacja sentence, w skrócie se, dwu- lub więcej argumentowa, służy do łączenia list w jedną listę, której elementami są elementy list składowych. Np. wartością:

```
sentence [hej hej] [la la la]
```

jest lista [hej hej la la la]. Jeśli natomiast wartością któregoś argumentu se jest słowo, to będzie ono brane jako element nowej listy, jak to widać na następującym przykładzie składania listy z trzech części:

```
(sentence "dnia [29 grudnia] 1986)
```

Wynikiem będzie lista

```
[dnia 29 grudnia 1986]
```

Podobnie wartością wywołania:

```
(sentence [być [albo [nie być]]] "oto [jest pytanie])
```

łączącego listę dwuelementową, słowo i listę dwuelementową, jest lista pięcioelementowa:

```
[być [albo [nie być]] oto jest pytanie]
```

Lista zapisana w postaci ciągu, ujętego w nawiasy kwadratowe, jest zawsze brana dosłownie i jej elementy nie podlegają żadnym obliczeniom ani badaniom wartości. Wyjątkiem są tylko te listy, które występują w komendach Logo zgodnie z zasadami budowania tych komend, jak na przykład w repeat lub if, co widać z przykładów występujących licznie w tej książce. Natomiast przed utworzeniem listy przy pomocy sentence wartości wszystkich jej argumentów, zarówno zmiennych, jak procedur, są obliczane i dopiero z tych wartości buduje się wynikową listę. Jeśli więc chcemy zbudować listę dwuelementową z wartości zmiennych, dajmy na to, :i oraz :zmie5, to należy napisać sentence :i :zmie5. Byłoby natomiast błędem użycie tu konstrukcji [:i :zmie5], bo to jest lista zbudowana z tych właśnie dwu poprzedzonych dwukropkami słów i nic więcej. Można również użyć list :i :zmie5.

Operacja list, także zachłanna, dwuargu-

mentowa, służy do łączenia wartości swoich argumentów w listę, ale w odróżnieniu od sentence nie zdejmuje zewnętrznych nawiasów kwadratowych z tych wartości argumentów, które są listami. Tak więc:

(list [być [albo [nie być]]] "oto [jest pytanie])

udostępniłaby wartość:

[[być [albo [nie być]]] oto [jest pytanie]]

Print zdejmuje, podobnie jak sentence, zewnętrzne nawiasy, jeśli wartość argumentu jest listą. Na przykład:

print pos

wypisze w jednym wierszu dwie liczby, mimo że wynikiem pos jest lista. Podobnie byłoby, gdybyśmy zastosowali print do wyprowadzania wyników wykonania sentence lub list z przykładów podanych tutaj (lub jakichkolwiek innych).
Print lub w skrócie pr jest procedurą jednoargumentową, ale zachłanną. Ujęcie print i następującego po nim ciągu argumentów w nawiasy okrągłe powoduje wydrukowanie całego tego ciągu, po czym zmianę wiersza. Na przykład:

(print [to będzie] "wydrukowane [bez nawiasów])

Samo przejście do następnego wiersza zapewnia komenda:

print "

Można o niej powiedzieć, że wypisuje wiersz pusty.
Jeśli trzeba uniknąć przejścia do następnego wiersza po wypisaniu żądanego tekstu, należy użyć komendy type. Jednoargumentowa zachłanna procedura type ma dokładnie takie same własności jak print, z tym wyjątkiem, że po jej wykonaniu nie przechodzi się do następnego wiersza. W ten sposób komenda:

type "

nie spowoduje żadnego widocznego skutku. Można o tej komendzie powiedzieć, że wypisuje napis pusty.
Aby uniknąć zdejmowania zewnętrznych nawiasów z list wypisywanych na ekranie trzeba użyć show. Jednoargumentowa zachłanna procedura show zachowuje się tak samo jak print, tylko wszystkie dane wyprowadza dokładnie tak, jak one wyglądają, nie wprowadzając żadnych zmian.
Wróćmy do działań na listach i słowach. Liczbę elementów listy albo słowa podaje operacja count. Operacja dwuargumentowa fput tworzy nową listę złożoną z listy, będącej wartością jej drugiego elementu, z dołączoną, jako nowym pierwszym elementem, wartością pierwszego argumentu:

show fput "początek [i reszta listy]

wypisze na ekranie wartość tego fput, czyli:

[początek i reszta listy]

Podobnie lput powoduje dołączenie wartości swego pierwszego argumentu na koniec listy, która jest wartością drugiego argumentu tej operacji:

show lput "koniec [część początkowa i]

wypisze:

[część początkowa i koniec]

Oczywiście pierwsze argumenty lput i fput nie muszą być słowami. Mogą to być listy i wtedy zostaną wprowadzone w nie zmienionej postaci jako ostatni lub pierwszy element listy będącej wartością operacji.
Operacja member bada, czy wartość jej pierwszego elementu występuje jako element listy będącej wartością drugiego argumentu tej operacji. Jeśli tak, wartością member jest "TRUE, w przeciwnym przypadku "FALSE.
Dwuargumentowa operacja item ma jako wartość pierwszego argumentu liczbę, a wartością drugiego argumentu jest lista. Jeśli ta liczba n nie przekracza długości listy, to wartością item jest n-ty element listy. W przeciwnym przypadku jest sygnalizowany błąd.
Pewna liczba predykatów, czyli operacji, przyjmujących tylko wartości "TRUE i "FALSE, służy do badania list i słów. Wszystkie one są jednoargumentowe. Są to:

listp	czy to jest list?
emptyp	czy to jest lista pusta lub słowo puste?

wordp	czy to jest słowo?
numberp	czy to jest liczba?
namep	czy to jest zmienna?
primitivep	czy to jest nazwa procedury Logo?
definedp	czy to jest nazwa procedury definiowanej?

Operacja wordp jest nieco zdradliwa, bo może dać wartość "TRUE nawet wtedy, gdy z pozoru nie powinna tego zrobić. Na przykład:

```
pr wordp "2+2=4
```

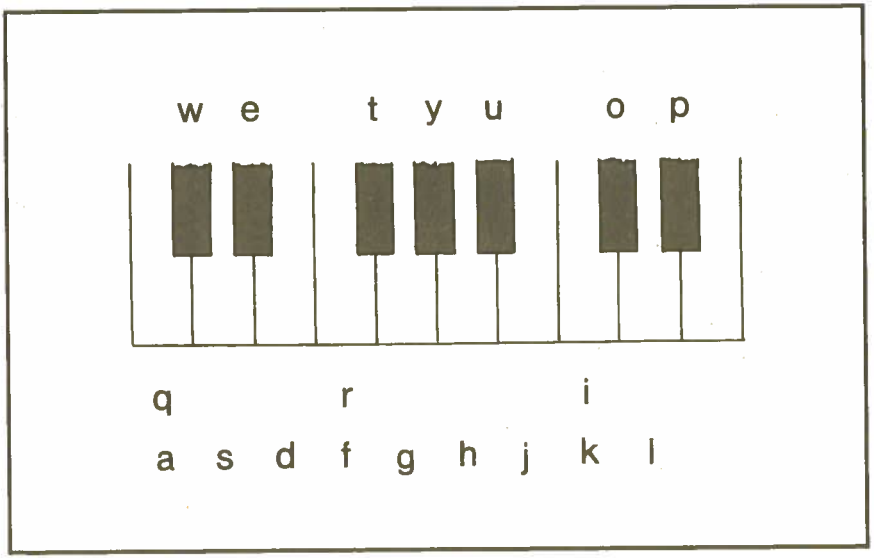
daje "TRUE, ale nie dlatego, że to, co tu widzimy, jest poprawnym słowem. Logo, widząc + i = oblicza wynik dodawania i porównania, wskutek czego wartością argumentu jest "TRUE i oczywiście nie ma wątpliwości, że jest prawidłowo zbudowane słowo. Wartością wordp "TRUE jest więc także "TRUE.

Operacja readlist, w skrócie rl, zeroargumentowa, powoduje wczytanie jednego wiersza z klawiatury, czyli sekwencji słów lub znaków aż do naciśnięcia klawisza ENTER. Wartością operacji jest ta sekwencja, opakowana w nawiasy kwadratowe.

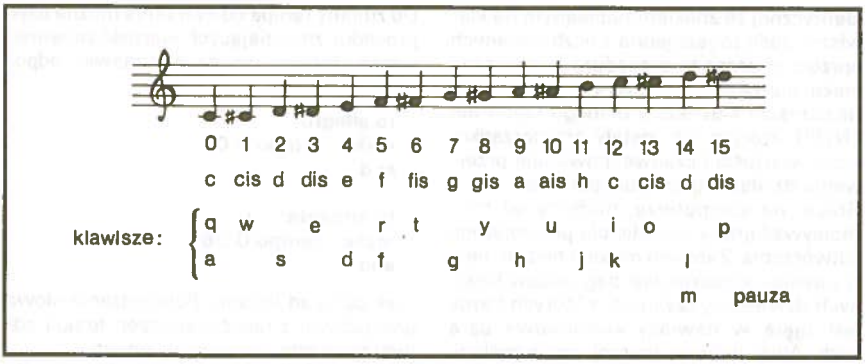
Trochę muzyki

Procedura powodująca generowanie dźwięku, sound, jest jednoargumentowa. Jej argumentem musi być ujęta w nawiasy kwadratowe para liczb, określających czas trwania dźwięku w sekundach oraz jego wysokość. Pierwsza z tych liczb, czas trwania, musi się mieścić pomiędzy 0 a 255, druga, wysokość, ma być między −62 a 75. Przekroczenie tych zakresów jest sygnalizowane jako błąd. Zmiana wysokości o 1 oznacza pół tonu, wysokość 0 oznacza środkowe C.

Jeśli chcemy, by naciśnięcie klawisza powodowało wydanie dźwięku o określonej wysokości, możemy użyć operacji rc, której wartością jest właśnie znak, wprowadzany przez aktualnie naciśnięty klawisz. Przypodrządkujmy klawiszom rzędu asdfghjkl całe tony, takie jak na białych klawiszach fortepianiu, cdefgahcd, a klawiszom rzędu wyższego, qwertyuiop, półtony odpowiadające właściwym czarnym klawiszom (rys. 52). Co prawda, jeśli tonem klawisza w jest cis a klawisza e jest dis, to na fortepianie nie ma czarnego klawisza między białymi dla tonów e i f, czyli na naszej klawiaturze r mógłby zostać bez przydziału. Niech to będzie jednak eis, czyli f i podobnie dru-



Rys. 52. Klawiatura literowa zamiast zwykłej klawiatury fortepianu

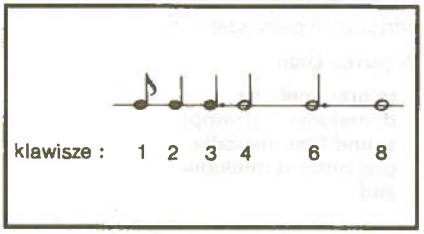


Rys. 53. Wysokości tonów granych na klawiaturze

giemu klawiszowi, nie mającemu swego odpowiednika na fortepianie, z literą i, przyporządkujemy ton c^1 (rys. 53):

```
to pianino
make "a 0
make "s 2
make "d 4
make "f 5
make "g 7
make "h 9
make "j 11
make "k 12
make "l 14
make "q 0
make "w 1
make "e 3
make "r 5
make "t 6
make "y 8
make "u 10
make "i 12
make "o 13
make "p 15
make "tempo 0.1
make "długość 0.5
make "melodia
repeat 1000 [make "klawisz rc if
numberp :klawisz [make „długość
0.5 * klawisz] [make "wysokość
thing :klawisz make "dźwięk se
:długość * :tempo :wysokość sound
:dźwięk make "melodia lput :dźwięk
:melodia]]
end
```

Najwyższy rząd klawiatury, z cyframi, został wykorzystany do określania czasów trwania dźwięków przez nadawanie wartości zmiennej długość. Jeśli przyjmiemy, że naciśnięcie klawisza 8 przestawia klawiaturę na granie całych nut, to czas trwania takich dźwięków będzie równy 0.5 * 8 * 0.1 = 0.4 sekundy. Nuta ósemka, grana po naciśnięciu klawisza 1, będzie więc trwać 0.05 sekundy. Klawisz 2 przełącza na ćwiartki, zaś 4 na półnuty. Jeśli zmienimy wartość zmiennej tempo, proporcjonalnie zmienią się te wszystkie czasy (rys. 54).



Rys. 54. Wykorzystanie klawiszy cyfrowych do określania czasu trwania dźwięków

O tym, czy został naciśnięty klawisz cyfrowy, czy inny, mówi nam operacja numberp badająca wartość zmiennej klawisz – przekazuje ona wartość TRUE, czyli prawda, gdy wartość jest liczbą, i FALSE, czyli fałsz, w każdym innym przypadku. Jeśli więc został naciśnięty klawisz nie oznaczony liczbą, oblicza się przy pomocy operacji thing wartość zmiennej o nazwie

identycznej ze znakiem napisanym na klawiszu. Jeśli to jest jedna z liczb nadanych uprzednio przez tę procedurę komendami make, bierze się ją jako wysokość dźwięku. Naciśnięcie klawisza z dolnego rzędu lub ENTER, którym nie zostały przyporządkowane wartości liczbowe, powoduje przerwanie działania procedury pianino.

Grając na komputerze, możemy od razu zapisywać graną melodię dla późniejszego odtworzenia. Zapisem melodii będzie ujęty w nawiasy kwadratowe ciąg opisów kolejnych dźwięków, czyli nut, z których każdy jest ujętą w nawiasy kwadratowe parą liczb. Albo, mówiąc inaczej, zapis melodii jest listą list dwuelementowych. Operacja lput powoduje umieszczenie nowego elementu na końcu listy; jej wartością jest tak uzupełniona nowa lista. Po ukończeniu działania procedury zapis całej melodii jest wartością zmiennej o tej nazwie. Jeśli jednak chcemy go zachować na dłużej, a mamy zamiar znów grać na tym pianinie, to musimy temu zapisowi nadać inną nazwę – inaczej ponowne uruchomienie procedury pianino go skasuje.

Procedura odtwarzająca melodię może być rekurencyjna, zbudowana na zasadzie:

Jeśli zapis jest pusty (nie ma żadnej nuty), to stop.
Zagraj pierwszą nutę.
Graj melodię z zapisu pozostałego po odrzuceniu pierwszej nuty.

W języku Logo:

```
to graj :melodia
if :melodia = [] [stop]
sound first :melodia
graj butfirst :melodia
end
```

Równie łatwo możemy zagrać melodię od końca do początku, jeśli w tej procedurze zastąpimy operację first (weź pierwszy element) przez last (weź ostatni element) oraz butfirst (weź listę powstałą po odrzuceniu pierwszego elementu) przez butlast (weź listę powstałą po odrzuceniu ostatniego elementu):

```
to grajwstecz :melodia
if :melodia = [] [stop]
sound last :melodia
grajwstecz butlast :melodia
end
```

Do zmiany tempa odtwarzania można użyć procedur zmieniających wartość zmiennej tempo, którym można ponadawać odpowiednie nazwy:

```
to allegro
make "tempo 0.01
end

to andante
make "tempo 0.06
end
```

i tak dalej ad libitum. Poprzedzenie słowa graj jednym z takich oznaczeń tempa odtwarzania odpowiednio go zmieni.

A teraz coś dla pomysłowych. Niezależnie od procedury odtwarzającej może działać procedura wypisująca na ekranie zapis nutowy odpowiadający temu, co jest zapisane jako wartość zmiennej melodia (lub jakiejkolwiek innej, której przypiszemy listę opisującą melodię). Przecież ekran w czasie wykonywania procedury graj jest właściwie nie wykorzystany. Aby to uzyskać, wystarczy napisać komplet procedur, które umożliwią rysowanie pięciolinii, klucza wiolinowego, nut oraz właściwe rozmieszczanie pięciolinii z kluczami i nut na ekranie. Jak już to będziemy mieli, możemy zorganizować sobie różne ćwiczenia. Na przykład można grać na klawiaturze kolejne nuty wskazywane przez żółwia na zapisie melodii na ekranie, a odpowiednia procedura może kontrolować poprawność wykonania, to znaczy długość trwania i wysokość granych dźwięków. Można również ćwiczyć rozpoznawanie wysokości dźwięku – po zagraniu go przez komputer (bez pokazywania nuty na pięciolinii) trzeba nacisnąć klawisz odpowiadający temu dźwiękowi. Zarówno poprawne, jak niepoprawne czynności muszą być jawnie oceniane, to znaczy komputer powinien zawsze reagować w sposób zrozumiały dla ćwiczącego. Po opanowaniu umiejętności prawidłowej oceny wysokości dźwięków można ćwiczyć rozpoznawanie i powtarzanie na klawiaturze dłuższych fraz, z zachowaniem zarówno prawidłowych wysokości, jak czasów trwania dźwięków. Rzecz jasna, ograniczone możliwości tego komputera nie pozwalają na zagranie prawidłowo wszystkiego, co można zagrać na instrumencie

muzycznym i sposób generowania dźwięków przez Spectrum może dla muzykalnego ucha pozostawiać sporo do życzenia, ale opanowanie prostych w gruncie rzeczy sposobów programowania takich ćwiczeń może być pożyteczne na przyszłość, gdyby nas to zainteresowało i gdybyśmy zetknęli się z komputerami połączonymi z urządzeniami akustycznymi wyższej jakości.

Dla bardziej biegłych muzycznie istnieje jeszcze inny rodzaj rozrywki – programowanie generowania różnych wariacji muzycznych na zadany temat, podobnie jak wiele obrazków opisywanych w tej książce powstało na zasadzie tworzenia wariacji rysunkowych na zadany temat rysunkowy. Do najprostszych chwytów należy zmiana rytmu melodii, ale nietrudno wymyślić bardziej skomplikowane modyfikacje. Poważnym ograniczeniem jest tu jednak niemożność generowania akordów, tak że pewne pomysły będą musiały pozostać w sferze teorii. Można skłonić komputer do wygenerowania prostego akompaniamentu do jakiejś melodii, ale nie będzie on mógł zagrać melodii i akompaniamentu jednocześnie, tylko albo jedno, albo drugie. Przy bardziej skomplikowanych procedurach może również okazać się niewystarczającym przyjęty tu sposób zapisywania melodii w postaci listy par odpowiadających nutom. Już choćby rozmieszczanie bemoli lub krzyżyków przy kluczu wiolinowym byłoby utrudnione, nie wiadomo nic z góry o podziale na takty i tak dalej. Ale te niedostatki można usunąć, modyfikując konwencję zapisu – na przykład na początku melodii kilka elementów listy można poświęcić na podawanie zapisanych zgodnie z jakimiś regułami wiadomości dodatkowych, które ułatwią rysowanie nut na pięciolinii, modyfikacje melodii, generowanie nowych wariantów i tak dalej. Porządny zapis melodii powinien dopuszczać zapisywanie pauz, czego w tym prostym sposobie podanym wyżej nie ma, ale nie trudno wprowadzić, poświęcając na przykład na pauzę klawisz M – jej czas trwania byłby określony przez naciśnięty poprzednio klawisz z cyfrą (różną od 0), tak samo jak dla nut. Do grania pauzy można wykorzystać komendę wait. Jest ona jednoargumentowa i powoduje bezczynne oczekiwanie komputera przez podaną liczbę 1/50 części sekundy. Zmiennej m przyporządkujemy jakąś wartość spoza dopuszczalnego zakresu wysokości dźwięków, na przykład 76. Wtedy w procedurze pianino do rzędu komend make dodamy jeszcze jedną:

```
make "m 76
```

a we wszystkich trzech procedurach: pianino, graj, grajwstecz zastąpimy słowo sound przez zagraj wprowadzając jednocześnie nową procedurę:

```
to zagraj :dźwięk
if last :dźwięk <76 [sound :dźwięk] [wait 50 * first :dźwięk]
end
```

Jeśli już zaczynamy wykorzystywać nie zajęte jeszcze klawisze jako przełączniki, to możemy to kontynuować. Na przykład naciśnięcie pewnych klawiszy może zmieniać wysokości tonów klawiszy dźwięków o oktawę wyżej lub niżej. Wymagało by to dalszych przeróbek procedury pianino, ale wszystkie pozostałe procedury już żadnych zmian by nie wymagały, skoro i tak można odtwarzać przy pomocy sound dźwięki w granicach prawie dwunastu oktaw, czyli znacznie przekraczających praktyczne potrzeby.

Jak trawa rośnie: procesy losowe

Przyjrzyjmy się działaniu procedury trawa, korzystającej z procedur histogram i zerohistogramu:

```
to trawa :ileźdźbeł :ilekroków
zerujhistogram :ileźdźbeł
repeat :ilekroków [histogram random
:ileźdźbeł 1]
end

to zerujhistogram :ilepunktów
make "nrpktu 0
repeat :ilepunktów [make (word "p
:nrpktu) 0 make "nrpktu ! :nrpktu +1]
end

to histogram :nrpunktu :przyrost
make "punkt word "p nrpunktu
make :punkt :przyrost +thing :punkt
pu
setx 3 * :nrpunktu − 100
sety (thing :punkt) − 50
pd
sety −50
end
```

Wygląda to tak, jakby na pustej początkowo grządce zaczęły rosnąć źdźbła trawy. Jeśli żółw jest widoczny, to biega od źdźbła do źdźbła, jakby wyciągał każde w górę.
Gdybyśmy powtórzyli

```
cs trawa 30 300
```

kilkakrotnie, zobaczylibyśmy, że za każdym razem wygląda to trochę inaczej, jedne źdźbła rosną w górę szybciej, inne wolniej, choć po dłuższej obserwacji widać, że na ogół ta trawa rośnie mniej więcej równomiernie na całej grządce.
Jest to przykład procesu przypadkowego, czyli losowego. Procedura trawa wybiera losowo przy pomocy operacji random źdźbła, a dokładniej ich numery, mieszczące się w zakresie od 0 do :ileźdźbeł − 1, i przy pomocy procedury histogram przedłuża wybrane źdźbło o 1 w górę. Czynność tę powtarza zadaną liczbę razy i stąd pochodzi efekt obserwowany na ekranie.
Biorąc trawę szybko rosnącą, dajmy na to bambus, otrzymamy nieco inny proces losowy. Teraz widać większe zróżnicowanie długości łodyg, choć powtarzanie eksperymentów pozwoli dojść do przekonania, że nie ma miejsc, w których by łodygi rosły zawsze gorzej lub zawsze lepiej od innych:

```
to bambus :ilełodyg :ilekroków
zerujhistogram :ilełodyg
repeat :ilekroków [histogram random
:ilełodyg random 6]
end
```

Dobierając wartość parametru :ilekroków trzeba uważać, by ten gaj bambusowy zmieścił się na ekranie i nie przekroczył jego górnego brzegu. Teraz źdźbła rosną przeciętnie około trzy razy szybciej niż w przypadku zwykłej trawy, ale może się zdarzyć, że niektóre łodygi wybiją się wyraźniej nad inne – mamy przecież do czynienia z procesem losowym.
Aby zbadać rozkład długości łodyg, możemy nieco rozbudować rysunek na ekranie.
Zanim to zrobimy, spróbujmy przyjrzeć się zasadzie działania już wprowadzonych procedur.
Rysowanie histogramów wykorzystujemy także jako przykład stosowania zmiennych generowanych. Aby móc rysować histogram, trzeba wiedzieć, jaka jest wysokość każdego słupka w każdym momencie procesu. Służą do tego zmienne p0, p1, p2, ...

Nazwa zmiennej jest składana przy pomocy operacji word z litery p oraz liczby (numeru punktu), obliczanej stosownie do potrzeb. W procedurze histogram ten numer punktu jest wartością pierwszego parametru. Procedura powoduje zwiększenie o :przyrost (wartość drugiego parametru) zmiennej o tym numerze; jej nazwa jest składana przez operację word i zapamiętywana jako wartość zmiennej pomocniczej punkt. Wyrażenie thing :punkt daje wartość wartości zmiennej punkt, czyli wartość zmiennej generowanej, o którą chodzi. Można ją teraz wykorzystać do obliczenia nowej wartości tej zmiennej (porównaj trzeci wiersz procedury histogram), albo do ustawienia żółwia we właściwej pozycji do rysowania nowego słupka.
Działanie procedury histogram możemy obejrzeć na ekranie. Napiszmy:

```
cs zerujhistogram 75
```

Większej liczby punktów dać nie można, gdyż spowodowałoby to przekroczenie zakresu ekranu. Ekran został oczyszczony, a procedura zerujhistogram utworzyła przy pomocy operacji word siedemdziesiąt pięć zmiennych i nadała im wszystkim wartość 0, przygotowując w ten sposób rysowanie nowego histogramu.
Komenda:

```
histogram 1 10
```

powoduje pojawienie się w lewej części ekranu słupka o wysokości 10. Przyglądając się treści procedury histogram, możemy wyliczyć wykonane przez nią następujące czynności: Wartością zmiennej pomocniczej punkt staje się nazwa p1, utworzona przy pomocy operacji word. Skądinąd wiemy, że wartością zmiennej p1 jest 0 wskutek wykonania procedury zerujhistogram. Wobec tego :przyrost + thing :punkt jest równe 10 + 0 i ta liczba zostaje zapisana jako wartość wartości zmiennej punkt, czyli jako wartość zmiennej o nazwie p1. Dalej żółw z podniesionym pisakiem przesuwa się poziomo do punktu o współrzędnej x równej 3 * 1 − 100 czyli −97, po czym przesuwa się pionowo do punktu o współrzędnej y równej pomniejszonej o 50 wartości zmiennej punkt, czyli, inaczej mówiąc, pomniejszonej o 50 aktualnej wartości zmiennej p1. Opuszczenie pisaka i przesunięcie żółwia pionowo w dół na poziom o współrzędnej y −50 kończy procedurę narysowaniem nowego słupka.

```
histogram 0 20
```

narysuje obok słupek dwa razy wyższy. Jest to lewy kraniec histogramu.

```
histogram 1 20
```

podwyższy nam sąsiadujący słupek do wysokości 30. Możemy tak dalej poeksperymentować z rysowaniem i podwyższeniem różnych słupków wykresu.
Napisanie po tym wszystkim:

```
cs
```

skasuje obraz na ekranie, ale wartości wszystkich zmiennych, odpowiadających punktom histogramu są nadal zapamiętane. Jeśli już znamy działanie procedury histogram, to możemy teraz utworzyć procedurę, która narysuje nam na ekranie wykres ze słupkami o odpowiednich wysokościach. Można tę procedurę pisać od nowa albo otrzymać ją przez redagowanie procedury histogram. Jak?

```
to rysujhistogram :ilepunktów
make "nrpunktu 0
repeat :ilepunktów [pu setx 3 *
:nrpunktu − 100 sety (thing word "p
:nrpunktu) − 50 pd sety −50 make
"nrpunktu :nrpunktu + 1]
end
```

Rzeczywiście:

```
rysujhistogram 75
```

odtworzy nam poprzedni wykres. Punkty, w których zmienne opisujące wysokości słupków mają wartość 0 są zaznaczone kropkami.

```
cs rysujhistogram 30
```

pokaże tylko część poprzedniego obrazu.
Wróćmy teraz do naszego zagadnienia: jak narysować wykres obrazujący rozkład długości łodyg narysowanych na ekranie. Stwórzmy nową kępkę trawy, tym razem mniejszą, aby było szybciej:

```
cs trawa 10 100
```

Jak ta trawa wyrośnie, będziemy mogli zaraz obok niej pokazać, ile jest źdźbeł o długości 0, 1, 2 i tak dalej. W tym celu

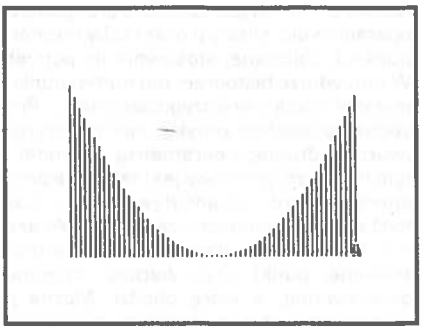
badamy długości kolejnych trawek tej kępki i dla każdej podwyższamy słupek histogramu o 1 w miejscu odpowiadającym jej długości.

```
to długości :ilepunktów
make "nrpunktu 0
repeat :ilepunktów [histogram
:ilepunktów + (thing word "p
:nrpunktu + 1]
end
```

Wykrzyknik, tak samo jak w poprzednich przypadkach, oznacza tylko przekroczenie wiersza druku i nie należy do treści procedury. Jest tylko znakiem kontynuacji.
Mając tę kępkę trawy na ekranie i pisząc:

```
długości 10
```

otrzymamy obok niej rozkład długości.
W ten sposób można obserwować nie tylko sam proces losowy, ale także badać pewne jego własności. Ci, którzy wiedzą więcej o rachunku prawdopodobieństwa, mogą sobie te eksperymenty modyfikować tak, aby badać inne zjawiska losowe.
Terminologia związana ze wzrostem traw została tu użyta tylko dlatego, że takie można mieć skojarzenia, patrząc na zmiany histogramów na ekranie. W rzeczywistości możemy w ten sposób badać procesy, które z roślinami mają niewiele wspólnego.
Rzucając kostką do gry możemy wyrzucić dowolną liczbę oczek od jednego do sześciu. Trawa 6 20 pokaże, ile wypadło jedynek, dwójek itd. w 20 rzutach, a długości 6 da dodatkowo informację, jaki był rozkład długości słupków tego histogramu, to znaczy po ile razy w tych 20 rzutach wypadały jednakowe liczby oczek. Ta ostatnia informacja jest zresztą mniej istotna w grach, do których używa się kostek. Jeżeli już mamy do czynienia w grze z większą liczbą rzutów, to bardziej interesująca jest suma oczek uzyskiwanych w kolejnych rzutach. To też można badać przy pomocy osobnej procedury; jej napisanie pozostawimy Czytelnikowi. Podobnie można by rozważać inne przykłady: rzut monetą, wyciąganie kart z dobrze potasowanej talii itd.
Zupełnie innym zastosowaniem histogramów będą wykresy funkcji jednej zmiennej. Na przykład pisząc:

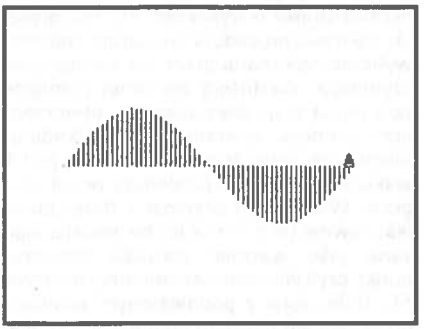


Rys. 55. Wykres słupkowy paraboli

```
to parabola
make "x 0
repeat 60 [histogram :x (:x − 30)
* (:x − 30) / 10 make "x :x + 1]
end
zerujhistogram 60
cs parabola
```

uzyskamy obraz jak na rys. 55.
Można wprowadzać procedury z parametrami, jak na przykład:

```
to sinusoida :amplituda :częstotliwość
make "x 0
repeat 60 [histogram :x :amplituda *
sin (:częstotliwość * :x * 6)
make "x :x + 1]
end
```



Rys. 56. Sinusoida

Rys. 56 pokazuje wykres powstały po:

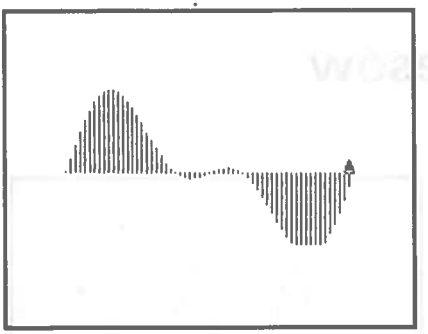
```
zerujhistogram 60 cs sinusoida 30 1
```

Widać, że dla ujemnych wartości funkcji

słupki są także rysowane prawidłowo. Pisząc teraz:

```
cs sinusoida 20 2
```

otrzymamy wykres sumy dwóch funkcji (rys. 57). Gdybyśmy chcieli mieć wykres

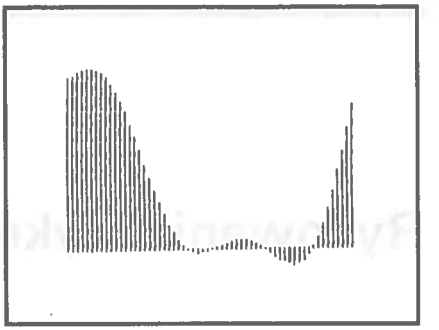


Rys. 57. Suma dwóch różnych sinusoid

tylko tej ostatniej, trzeba by było przedtem napisać zerujhistogram 60. Gdy jest tak jak teraz, możemy jeszcze dodać do tego funkcję kwadratową:

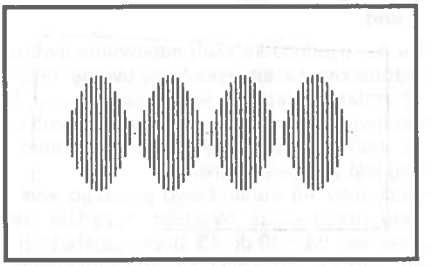
```
cs parabola
```

Wynik będzie taki, jak na rys. 58.
Tworząc inne procedury rysowania takich wykresów prążkowych dla rozmaitych funkcji, warto pamiętać o właściwym doborze parametrów, tak aby rysunek mieścił się prawidłowo na ekranie. W obu powyższych procedurach zakłada się zmianę parametru :x od 0 do 59 po jednym słupku na każdą wartość całkowitą – razem 60 słupków. Przy położeniu wykresu narzuconym przez procedurę histogram funkcja musi się mieścić pomiędzy −38 a 138,



Rys. 58. Suma funkcji z rys. 57 i 55

jeśli nie chcemy, by słupki wyszły poza ekran. To samo dotyczy oczywiście wykresów sum funkcji. Po skasowaniu obrazu komendą cs zapisy wysokości słupków zachowują się i wykres można odtworzyć przy pomocy procedury rysujhistogram 60. Ponieważ raz narysowany słupek nie może być skrócony, dodawanie funkcji o ujemnych wartościach powinno być poprzedzone komendą cs. Można jednak uzyskiwać efekty specjalne, jak na rys. 59, właśnie rezygnując z cs.



Rys. 59. Złożenie dwu wykresów razem

Rysowanie wykresów

Operacja run powoduje wykonanie kolejno wszystkich elementów listy, która jest jej argumentem. W szczególności jeśli ta lista zawiera tylko jedno wyrazenie, run obliczy jego wartość. Wykorzystamy to do rysowania wykresów dowolnych funkcji, opisywanych wyrażeniami:

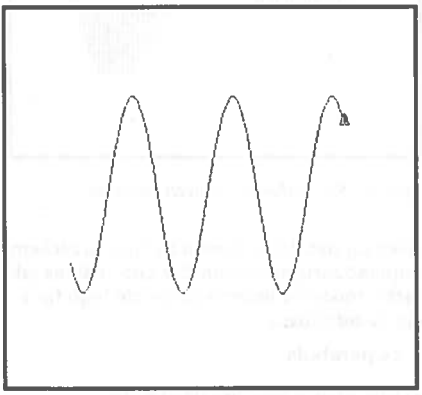
```
to wykres
print [podaj wyrażenie opisujące
funkcję zmiennej :x]
make "f readlist
make "x -50.51
pu setpos [-101 0]
repeat 100 [make "x :x + 1 setpos se
xcor + 2 run :f pd]
end
```

Na osi x jednostka skali odpowiada dwóm jednostkom ekranu (zwróćmy uwagę na to, że zmianie wartości współrzędnej :x o 1 odpowiada zmiana pozycji xcor żółwia o 2). Jednostką skali wykresu na osi pionowej jest jednostka ekranu.

Zacznijmy od stosunkowo prostego wykresu funcji sinus. Wartość :x będzie się zmieniać od −49 do 49 stopni (dokładniej: od −49.51° do 49,51°) i, aby narysować kilka fal sinusoidy, trzeba będzie wziąć wyrażenie 60 * sin 10 * :x. Mnożnik 60 jest konieczny, by krzywa ładnie wyszła, bo bez niego funkcja zmieniałaby się od −1 do 1, co na wykresie w tej skali wyglądałoby jak pozioma prosta.

Cały dialog z komputerem będzie wyglądać tak:

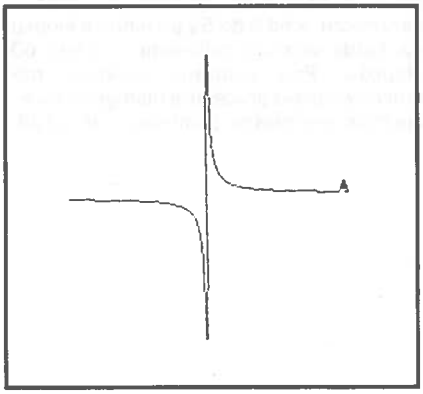
```
cs
wykres
podaj wyrażenie opisujące funkcję
zmiennej :x
60 * sin 10 * :x
```



*Rys. 60. Wykres funkcji 60 * sin 10 * :x*

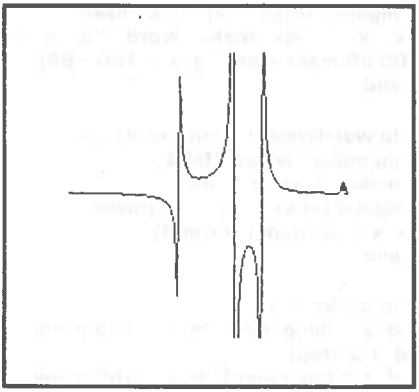
Wynik będzie taki, jak na rys. 60.

Readlist, wczytujący podane wyrażenie,



Rys. 61. Wykres 60 / :x

ujmuje go w nawiasy kwadratowe i wartość nadana zmiennej f ma postać [60 * sin 10 * :x], czyli taką, jaka jest potrzebna do wykonania operacji run. Ponieważ w każdym kroku dobiera się odpowiednią wartość :x, run wyznacza wartość wyrażenia właśnie dla tej wartości. Użycie setpos powoduje ustawienie żółwia w kolejnym punkcie wykresu.
Wykres funkcji 60 / :x wygląda jak na rys. 61. Dzięki temu, że wartości :x są brane z poprawką 0.51, nie są one nigdy całkowite i dla każdej funkcji uniknęliśmy dzielenia przez zero. Trzeba jednak rysować w trybie window, bo w okolicy zera żółw wykracza poza krawędzie ekranu, najpierw dolną, potem górną. Przechodząc od dolnej skrajnej wartości do górnej rysuje linię pionową, nie należącą oczywiście do krzywej, którą można potraktować jako pionową asymptotę – prostą, do której wykres się zbliża, gdy :x zbliża się do zera.



*Rys. 62. Funkcja wymierna 20000 / ((:x + 10) * (:x − 10) * (:x − 20))*

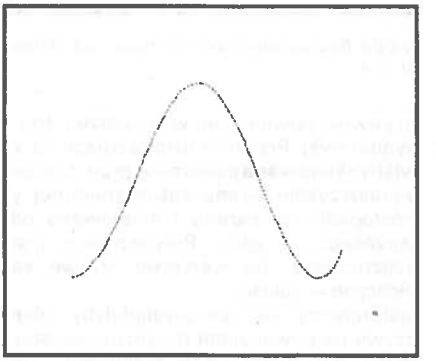
Rys. 62 pokazuje wykres bardziej skomplikowany funkcji wymiernej 20000 / ((:x + 10) * (:x − 10) * (:x − 20)). Jak w poprzednich przypadkach trzeba było dobrać odpowiedni mnożnik, by wykres zmieścił się na ekranie i był czytelny.
Gdybyśmy chcieli rysować rodziny krzywych ze zmieniającym się w pewnym zakresie parametrem, na przykład rodzinę cosinusoid 60 * cos 2 * (:x + :y) dla :y zmieniającego się od −10 do 10 co 2, to znacznie wygodniej będzie posłużyć się procedurą, dla której wyrażenie opisujące funkcję jest parametrem. Możemy także stworzyć sobie możliwość określenia zakresu zmienności :x przez podanie wartości minimalnej, maksymalnej i skoku tej zmiennej.

```
to krzywa :f :xmin :max :dx
pu
make "x :xmin − :dx − 0.51
repeat (:xmax − :x) / :dx [make "x :x
+ :dx setpos se :x run :f pd]
end
```

Dla uproszczenia przyjmujemy tutaj (inaczej niż w procedurze wykres), że jednostki skali na osiach poziomej i pionowej wykresu są jednakowe i równe jednostce odległości ekranu. Tempo rysowania i dokładność rysunku zależą od doboru wartości zmiennej dx – im jest ona mniejsza, tym dokładniejszy jest rysunek, ale także tym wolniej powstaje. Poprawka 0.51 ma, tak samo jak poprzednio umożliwić rysowanie większości funkcji wymiernych bez narażania się na dzielenie przez zero.
Pisząc:

```
cs window make "y 10
krzywa [60 * cos 2 * (:x + :y)]
−100 100 2
```

otrzymamy krzywą jak na rys. 63. Ponieważ wyrażenie opisujące funkcję jest podawane bezpośrednio jako parametr, musi być ujęte w nawiasy kwadratowe. W pro-



*Rys. 63. Krzywa [60 * cos 2 * (:x + :y)] −100 100 2 dla :y równego 10*

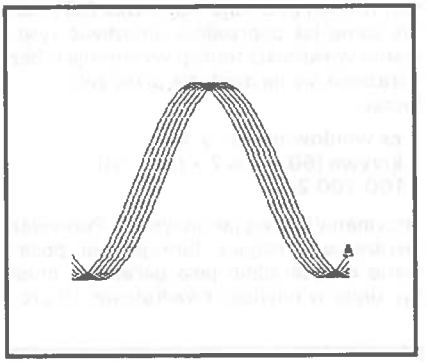
cedurze wykres uniknęliśmy tego, bo wartość zmiennej f była wczytywana przez readlist, które te nawiasy samoczynnie dodawało. Tutaj pominięcie nawiasów będzie sygnalizowane jako błąd.
Procedura:

```
to rodzina :f :xmin :xmax :dx :ymin
:ymax :dy
  make "y :ymin − :dy
  repeat (:ymax − :y) / :dy [make "y :y
+ :dy krzywa :f :xmin :xmax :dx]
  end
```

umożliwi rysowanie całych rodzin krzywych. Może to być ta opisana wyżej rodzina cosinusoid:

```
rodzina [60 * cos 2 * (:x + :y)] −100
100 2 −10 10 4
```

Patrząc na otrzymany w ten sposób rys. 64, można sobie wyobrazić, że te krzywe leżą



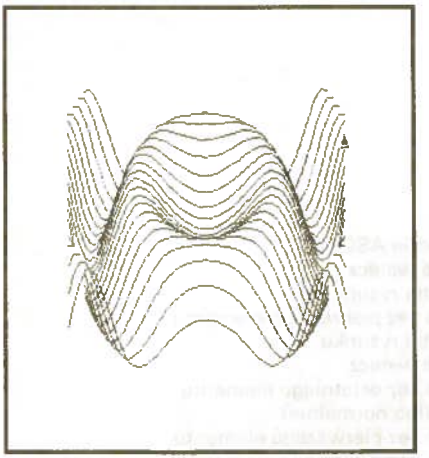
Rys. 64. Rodzina takich krzywych dla :y od −10 do 10 co 4

na pewnej powierzchni w przestrzeni trójwymiarowej. Przy tych oznaczeniach oś :x byłaby skierowana poziomo w prawo, leżąc w płaszczyźnie ekranu, zaś oś zmiennej :y prostopadła do ekranu i skierowana od patrzącego w głąb. Powierzchnia jest przezroczysta, bo wszystkie krzywe są widoczne w całości.
Zastanówmy się, jak wyglądałyby takie krzywe na powierzchni nie przezroczystej. Pewne ich części byłyby zasłonięte – im głębiej leży krzywa, tym większa jej część może znikać za występującymi bliżej częściami powierzchni. Krzywa leżąca najbliżej nie jest zasłonięta niczym. Dla drugiej, leżącej tuż za nią, można założyć to samo (dlaczego?). Trzecia z kolei jest widoczna tylko tam, gdzie jest rysowana powyżej lub poniżej obydwu z nich. Każda następna, idąc w głąb, jest widoczna tylko tam, gdzie jest powyżej lub poniżej wszystkich krzywych poprzedzających. Zasada ta została wykorzystana w procedurach, które umożliwiają rysowanie obrazów warstwicowych dla dość dużej klasy powierzchni. Np.:

```
to powierzchnia :f :xmin :xmax :dx
:ymin :ymax :dy
  ograniczenia :xmin :xmax :dx
  make "y :ymin − :dy
  repeat (:ymax − :y) / :dy [make
"y :y + :dy warstwica :f :xmin :xmax :dx]
  end

  to ograniczenia :xmin :xmax :dx
  make "x :xmin − :dx
  repeat (:xmax − :x) / :dx [make
"x :x + :dx make word "d :x +
100 85 make word "g :x + 100 −85]
  end

  to warstwica :f :xmin :xmax :dx
  pu make "widąć "false
  make "x :xmin − :dx
  repeat (:max − :x) / :dx [make
"x :x + :dx punkt :x run :f]
  end

  to punkt :x :z
  if :z < thing word "d :x + 100 [rysuj
"d :x :z stop]
  if :z > thing word "g :x + 100 [rysuj
"g :x :z stop]
  pu make "widać "false setpos se :x :z
  end

  to rysuj :litera :x :z
  make word :litera :x + 100 :z
  if :widać [pd] [make "widać ,,true]
  setpos se :x :z
  end
```

Procedura ograniczenia służy do ustawienia początkowych wartości ograniczeń górnych i dolnych obszaru zasłoniętego. W trakcie rysowania każdej warstwicy są one odpowiednio korygowane. Rysuje się tylko

części warstwic, dla których wartość funkcji f jest większa od aktualnego ograniczenia górnego lub mniejsza od ograniczenia dolnego.
Ta metoda eliminacji linii zasłoniętych jest bardzo prosta, nawet prymitywna, ale dlatego niedokładna. Opiera się na badaniu sytuacji na końcach odcinków rysowanych krzywych – jeśli taki punkt trafi pomiędzy ograniczenie górne i dolne, to ma być niewidoczny, a z nim cały przylegający doń odcinek. W ten sposób niewidoczne stają się nawet te części krzywych, które przy precyzyjnym rysowaniu byłyby przynajmniej częściowo odsłonięte. Z kolei, jeśli oba końce takiego odcinka krzywej wypadają poza strefą niewidoczności, to cały odcinek krzywej będzie rysowany, gdyby nawet jego część miała być zasłonięta i dlatego niewidoczna. W szczególności może się zdarzyć dla krzywych rysowanych bardzo stromo, że jeden koniec odcinka wypadnie poniżej ograniczenia dolnego, drugi powyżej ograniczenia górnego i wobec tego cały odcinek będzie narysowany, przecinając fragment rysunku, który faktycznie powinien go zasłaniać.
Te defekty możemy zobaczyć na rys. 65. Ale to są tylko szczegóły. Tam, gdzie krzywe przebiegają bardziej poziomo, takie błędy nie występują i dlatego cały rysunek jest dostatecznie wyraźny i czytelny.
Defekty te można stosunkowo łatwo usunąć, stosując dokładniejsze badanie położeń punktów kolejnych warstwic. Skomplikowałoby to nieco procedurę punkt. Jednak takie ulepszenia spowodowałyby spowolnienie rysowania, które i tak trwa już dość długo i wymaga sporej cierpliwości. Innym sposobem poprawienia dokładności rysunku jest zwiększenie gęstości punktów na osi x. Ale to też kosztuje: dwa razy gęściej umieszczone punkty powodują dwukrotne zwiększenie czasu rysowania. Potrzebny jest więc jakiś kompromis i będzie on zależał w dużej mierze od tego, do czego te rysunki będą nam potrzebne. W każdym razie raz utworzony rysunek może być zapisany przy pomocy savescr i odtwarzany potem (przez loadscr) znacznie szybciej, niż powstawał.



*Rys. 65. Rysunek warstwicowy:powierzchnia [:y + 30 * sin 0.01 * (3.7 * :x * :x + 6 * :y * :y)] −100 100 4 −55 55 5*

W tym miejscu można już zakończyć prezentowanie walorów i możliwości Logo na Sinclair Spectrum. Kto dokładnie poznał oba tomy tej książki, będzie mógł sam kontynuować pracę nad rozwinięciem i udoskonaleniem swojej z nim współpracy, by potem już łatwo i bez złych nawyków przejść na wyższy poziom kontaktów z informatyką.

Wykaz komend Logo

Ten rozdział zawiera spis wszystkich komend języka Logo (wersji angielskiej) z krótkimi objaśnieniami. Był on już zamieszczony w cz. I tej książki, powtarzamy go jednak, by umożliwić korzystanie z niego tym, którzy kupią jedynie *Logo dla zaawansowanych*.

Słowo op przy nazwie komendy oznacza operację, to znaczy taką procedurę języka, przy której wykonaniu jest obliczana wartość, przypisywana nazwie tej procedury. Dla każdej komendy jest również podawana liczba argumentów; jeśli ona jest zmienna, to podana jest najmniejszą możliwą.

and	op	2	logiczne i
arccos	op	1	arccos
arccot	op	1	arcctg
arccotangent	op	1	arcctg
arcsin	op	1	arcsin
arctan	op	1	arctg
arctangent1	op	1	arctg
ascii	op	1	nr znaku w kodzie ASCII
back		1	przesuń żółwia wstecz
background	op	1	numer koloru tła rysunku
bf	op	1	lista lub słowo bez pierwszego elementu
bg	op	1	numer koloru tła rysunku
bk		1	przesuń żółwia wstecz
bl	op	1	lista lub słowo bez ostatniego elementu
bright		1	pisz jaskrawo (lub normalnie)
butfirst	op	1	lista lub słowo bez pierwszego elementu
butlast	op	1	lista lub słowo bez ostatniego elementu
bye		0	koniec pracy Logo
catalog		0	katalog dyskietki lub microdrive'u
char	op	1	znak o podanym kodzie ASCII
clean		0	zmaż ekran, nie ruszając żółwia
clearscreen		0	czyść ekran
cleartext		0	zmaż tekst z całego ekranu
copydef		2	powiel definicję procedury
copyscreen		0	drukuj obraz z ekranu na drukarce
cos	op	1	cos
cosine	op	1	cos
cot	op	1	ctg
cotangent	op	1	ctg
count	op	1	długość słowa lub listy
cs		0	czyść ekran

ct		0	zmaż tekst z całego ekranu
cursor	op	0	położenie wskaźnika
define		2	określ procedurę
definedp	op	1	czy procedura jest określona?
div	op	2	iloraz
dot		1	rysuj punkt
ed		1	redaguj
edit		1	redaguj
edns		1	redaguj wszystkie nazwane tu zmienne
emptyp	op	1	czy puste słowo lub lista?
end		0	zakończenie definicji procedury
equalp	op	2	czy równe?
er		1	usuń procedurę
erall		0	usuń wszystko
erase		1	usuń procedurę
erasefile		1	usuń plik z dyskietki lub microdrive'u
ern		1	usuń wszystkie nazwane tu zmienne
erns		0	usuń wszystkie zmienne
erps		0	usuń wszystkie procedury
false		0	fałsz (jako stała logiczna)
fd		1	przesuń żółwia naprzód
fence		0	postaw płot wokół ekranu
first	op	1	pierwszy element listy lub słowa
flash		0	pisz migający napis
forward		1	przesuń żółwia naprzód
fput	op	2	wstaw pierwszy element
heading	op	0	kąt położenia żółwia
hideturtle		0	schowaj żółwia
home		0	wróć żółwiem do początku układu
ht		0	schowaj żółwia
if		2	jeśli
int	op	1	część całkowita liczby
inverse		0	pisz napis odwracając kolory (negatyw)
item	op	2	element listy lub słowa
keyp	op	0	czy był naciśnięty klawisz?
last	op	1	ostatni element listy lub słowa
left		1	obróć żółwia w lewo
list	op	2	stwórz listę
listp	op	1	czy jest listą?
load		1	ładuj z pamięci zewnętrznej
loadd		1	ładuj redagowane procedury lub dane
loadscr		1	ładuj obraz z pamięci zewnętrznej
lput	op	2	wstaw ostatni element
lt		1	obróć żółwia w lewo
make		2	przypisz zmiennej podaną wartość
memberp	op	2	czy jest elementem listy lub słowa?
name		2	nazwij
namep		1	czy jest nazwą zmiennej?
nodes	op	0	liczba wolnych miejsc pamięci (w sensie Logo)
normal		0	pisz napis bez odwracania kolorów (pozytyw)
not	op	1	logiczne nie
numberp	op	1	czy jest liczbą?
op		1	wynik operacji
or	op	2	logiczne lub

output		1	wynik operacji
over		1	pisz napis, nadrukowując na poprzednim
pc	op	0	kolor pisaka
pd		0	opuść pisak
pe		0	włącz ścieranie
pencolour	op	0	kolor pisaka
pendown		0	opuść pisak
penerase		0	włącz ścieranie
penreverse		0	włącz odwracanie kreski
penup		0	podnieś pisak
po		1	pokaż procedurę
poall		0	pokaż wszystkie procedury i zmienne
pons		0	pokaż wszystkie zmienne
pops		0	pokaż wszystkie procedury
pos	op	0	pozycja żółwia
position	op	0	pozycja żółwia
pots		0	pokaż tytuły wszystkich procedur
pr		1	pisz na ekranie
primitivep	op	1	czy jest nazwą procedury pierwotnej?
print		1	pisz na ekranie
printoff		0	wyłącz druk na drukarce
printon		0	włącz druk na drukarce
product	op	2	iloczyn
pu		0	podnieś pisak
px		0	włącz odwracanie kreski
random	op	1	losuj liczbę
rc	op	0	czytaj znak z klawiatury
readchar	op	0	czytaj znak z klawiatury
readlist	op	0	czytaj listę
recycle		0	odśmieć pamięć
remainder	op	2	reszta z dzielenia
repeat		2	powtórz
right		1	obróć żółwia w prawo
rl		0	czytaj listę
round	op	1	zaokrąglij liczbę
rt		1	obróć żółwia w prawo
run	op	1	zrób to, co jest podane w argumencie
save		2	zapisz w pamięci zewnętrznej
saveall		1	zapisz wszystko w pamięci zewnętrznej
saved		1	zapisz redagowane procedury lub dane
savescr		1	zapisz obraz w pamięci zewnętrznej
scrunch	op	0	proporcja skali osi y do x
se	op	2	stwórz listę z elementów argumentów
sentence	op	2	stwórz listę z elementów argumentów
setbg		1	określ kolor tła rysunku
setborder		1	określ kolor ramki (obrzeża) ekranu
setbr		1	określ kolor ramki (obrzeża) ekranu
setcur		1	ustaw wskaźnik (kursor)
setcursor		1	ustaw wskaźnik (kursor)
setdrive		1	wybierz dyskietkę, microdrive lub magnetofon
seth		1	ustaw żółwia pod podanym kątem
setheading		1	ustaw żółwia pod podanym kątem
setpc		1	określ kolor pisaka
setpos		1	przesuń żółwia na podaną pozycję

setscr		1	określ proporcje skali osi y do x
setscrunch		1	określ proporcje skali osi y do x
settc		1	określ kolory tekstu (tła i znaków)
setx		1	przesuń żółwia poziomo
sety		1	przesuń żółwia pionowo
show		1	pokaż listę lub słowo
shownp	op	0	czy żółw jest widoczny?
showturtle		0	pokaż żółwia
sin	op	1	sin
sine	op	1	sin
sound		1	graj podaną nutę
sqrt	op	1	pierwiastek kwadratowy
st		0	pokaż żółwia
startrobot		0	uruchom robota
stop		0	stop (zakończenie wykonania procedury)
stoprobot		0	zatrzymaj robota
sum		2	suma
tan	op	1	tg
tangent	op	1	tg
tc	op	0	kolory tekstu
text	op	1	treść procedury
textcolour	op	0	kolory tekstu
textscreen		0	przejście do trybu pisania tekstów
thing	op	1	wartość zmiennej
to		1	początek definicji procedury
toplevel		0	przerwij rekurencję
towards	op	0	kąt widzenia punktu przez żółwia (azymut)
true	op	0	prawda (jako stała logiczna)
ts		0	przejście do trybu pisania tekstów
type		1	wpisz tekst, nie zmieniając wiersza
wait		1	czekaj
window		0	określ ekran jako okno na płaszczyznę
word	op	2	stwórz słowo
wordp	op	1	czy jest słowem?
wrap		0	zwiń ekran (połącz brzeg górny z dolnym i lewy z prawym)
xcor	op	0	współrzędna x żółwia
ycor	op	0	współrzędna y żółwia
+	op	2	dodawanie
-	op	1	mnożenie przez –1 lub odejmowanie
*	op	2	mnożenie
/	op	2	dzielenie
=	op	2	równość
<	op	2	mniejsze
>	op	2	większe
"	op	1	operator dosłowności
:	op	1	wartość
\	op	1	blokada interpretacji znaku
.bload		2	ładuj binarnie z pamięci zewnętrznej
.bsave		2	zapisz binarnie w pamięci zewnętrznej
.call		1	wywołaj binarnie
.contents		0	zawartość listy nazw procedur i danych
.deposit		2	włóż do pamięci pod podany adres
.examine		1	zobacz co jest w pamięci pod podanym adresem
.primitives		0	zawartość listy nazw procedur pierwotnych

.reserve	1	zajmij pamięć
.reserved	0	zajęte
.serialin	0	przyjmij przez wejście szeregowe
.serialout	1	wyślij przez wyjście szeregowe
.setserial	1	prędkość transmisji

UWAGA: komendami z kropką należy posługiwać się ostrożnie, gdyż nieuważne ich użycie może doprowadzić do nieprzewidzianych skutków, takich jak uszkodzenie potrzebnych zapisów w pamięci komputera, zablokowanie komputera przepełnieniem pamięci lub zablokowanie działania Logo.

Polskie Logo

Zestawienie polskich odpowiedników komend Sinclair Logo jest ułożone tak samo, jak opisy komend – nazwy angielskie są w porządku alfabetycznym. Składnia, sposób wykonania, zasady redagowania procedur są w obu wersjach takie same. Ten rozdział także był już zamieszczony w I części *Logo na Sinclair Spectrum*, powtórzenie go jednak jest tak samo celowe, jak rozdziału poprzedniego.

and	i
arccos	arccos
arccot	arcctg
arccotangent	arcctg
arcsin	arcsin
arctan	arctg
arctangent	arctg
ascii	ascii
back	wstecz
background	tło↑
bf	bezpierw
bg	tło↑
bk	ws
bl	bezost
bright	jaskrawo
butfirst	bezpierw
butlast	bezost
bye	dość
catalog	katalog
char	znak
clean	zmaż
clearscreen	czyść
cleartext	zmażtekst
copydef	powiel
copyscreen	drukobrazu
cos	cos
cosine	cos
cot	ctg
cotangent	ctg
count	długość
cs	cs
ct	zt
cursor	kursor↑
define	określ
definedp	określone?
div	iloraz
dot	pkt
ed	red
edit	red
edns	redwn
emptyp	puste?
end	już
equalp	równe?
er	us
erall	usw
erase	usuń
erasefile	usuńplik
ern	usn
erns	uswn
erps	uswp
false	fałsz
fd	np
fence	pole
first	pierw
flash	migaj
forward	naprzód
fput	nap
heading	kąt↑
hideturtle	sż
home	wróć
ht	sż
if	jeśli
int	entier, ent
inverse	negatyw
item	element
keyp	klawisz?
last	ost
left	lewo

list	lista
listp	lista?
load	ładuj
loadd	ładujr
loadscr	ładujo
lput	nak
lt	lw
make	przypisz, przyp
memberp	element?
name	nazwij
namep	jest?
nodes	wolne
normal	pozytyw
not	nie
numberp	liczba?
op	wy
or	lub
output	wynik
over	nadruk
pc	pisak↑
pd	opu
pe	ścieranie
pencolour	pisak↑
pendown	opu
penerase	ścieranie
penreverse	odwracanie
penup	podnieśpisak
po	po
poall	pow
pons	pown
pops	powp
pos	poz↑
position	poz↑
pots	potp
pr	pp
primitivep	pierwotne?
print	pisz
printoff	wyłączdruk
printon	włączdruk
product	iloczyn
pu	pod
px	odwracanie
random	losowa
rc	cz
readchar	czytajznak
readlist	czytajlistę
recycle	odśmieć
remainder	reszta
repeat	powtórz
right	prawo
rl	cl
round	zaokr
rt	pw
run	zrób
save	zapiszp
saveall	zapisz
saved	zapiszr
savescr	zapiszo
scrunch	proporcja↑
se	zd
sentence	zdanie
setbg	tło
setborder	ramka
setbr	ramka
setcur	kursor
setcursor	kursor
setdrive	dysk
seth	kąt
setheading	kąt
setpc	pisak
setpos	poz
setscr	proporcja
setscrunch	proporcja
settc	koloryt
setx	xpoz
sety	ypoz
show	pokaż
shownp	widać
showturtle	pż
sin	sin
sine	sin
sound	nuta
sqrt	pierwiastek, pkw
st	pż
startrobot	startrobot
stop	stop
stoprobot	stoprobot
sum	suma
tan	tg
tangent	tg
tc	kt↑
text	treść
textcolour	koloryt↑
textscreen	teksty
thing	wartość, war
to	oto
toplevel	przerwij
towards	azymut
true	prawda
ts	ts
type	wpisz
wait	czekaj
window	okno
word	słowo
wordp	słowo?

wrap	sklej	.serialout	.wyślij
xcor	xpoz ↑	.setserial	.transmisja
ycor	ypoz ↑		
.bload	.ładujbin	Poza tym w polskim Logo są:	
.bsave	.zapiszbin		
.call	.wywołaj	exp	funkcja wykładnicza
.contents	.pamięć	jas 1	jaskrawy ekran
.deposit	.umieść	jas 0	kasowanie jaskrawości ekranu
.examine	.zobacz	ln	logarytm naturalny
.primitives	.pierwotne	mig 1	miganie ekranu
.reserve	.zajmij	mig 0	kasowanie migania ekranu
.reserved	.zajęte	pi	liczba pi
.serialin	.przyjmij	zam	zamaluj, zamalowanie obszaru

Mając jakąkolwiek niepolską wersję Logo, w szczególności Sinclair Logo, można wprowadzić rozwiązanie zastępcze:

1. Zdefiniować w trybie graficznym wprowadzania z klawiatury (GRAPHICS, GRAPH na Spectrum +) wszystkie polskie litery ze znakami diakrytycznymi: ą, ę, ć, ń, ó, ś, ź, ż (oba jako graficzny odpowiednik z), ł. Definiowanie znaków jest opisane w podręcznikach Spectrum, do wpisywania danych do pamięci używa się . deposit.
2. Przedefiniować procedury języka, nadając im nowe nazwy (przy pomocy to ... end lub copydef). Nie da się w ten sposób przemianować procedur definiujących, to i end. Nie da się również zastąpić komunikatów angielskich polskimi. Takie przemianowanie procedur powoduje również dodatkowe zajęcie części pamięci na nowe nazwy, wskutek czego zmniejszy się nieco pamięć dostępna dla obliczeń.

Literatura

- Abelson H.: *Logo for the Apple II.* Byte Books, McGraw Hill 1982.
- Abelson H.: *A beginner's guide to Logo.* „Byte" 7 no. 8 Aug. 1982, ss. 88–112.
- Abelson H., Klotz L. Jr.: *Logo for the Apple II: Technical Manual.* Terrapin Inc.
- Abelson H., Di Sessa A.: *Turtle geometry: the computer as a medium for exploring mathematics.* MIT Press 1981.
- Abelson H.: *Einführung in Logo.* IWT Verlag 1985.
- Allan B.: *Introducing Logo.* Granada 1984.
- Bearden D.: *A bit of Logo magic.* Prentice Hall 1984.
- Bearden D.: *1, 2, 3, my computer and me. A Logo funbook for kids.* Prentice Hall 1983.
- Bearden D., Martin K., Muller J.: *The turtle's sourcebook.* Prentice Hall 1984.
- Bitter G. G., Watson N. R.: *Commodore 64 Logo primer.* Reston 1985.
- Bossuet G.: *Introduire l'ordinateur a l'école?* Presses Universitaires de France 1981.
- Bossuet G., Fournier J., Leguyader C.: *The Logo graphic „Aiguelongue" experience.* W: Lewis R., Tass D. E. eds. *Computers in Education.* North Holland 1981, s. 795.
- Brady J.M.: *Introducing Logo education.* „Computer Education" 19, 1975, ss. 24–27.
- Conlan J., Inman D.: *Sprites, a turtle, and TI Logo.* Prentice Hall 1984.
- Cugola M., Giaccaglini G.: *Il mondo del Logo, un microcosmo in espansione,* „Bit" 6 no. 37 Marzo 1983, ss. 82–89.
- Davidson L. J.: *Apple Logo: Reference manual.* Apple Computer Inc. 1982.
- Du Boulay J.: *Teaching mathematics through programming.* „Int. J. Math. Educ. Sci. Technol." 11 1982 no. 3, ss. 347–360.
- Feurzeig W., Lukas G.: *Logo – a programming language for learning mathematics.* „Educational Technology" March 1972, ss. 33–16.
- Gary G., Bitter G., Watson N. R.: *Apple Logo primer.* Reston 1983.
- Goldberg A., Ross J.: *Is the Smalltalk language for children?* „Byte" 6 no. 8 Aug. 1981, ss. 348–368.
- Goldenberg E. P.: *Special technology for special children.* University Park Press 1979.
- Goldenberg E. P.: *Logo – a cultural glossary.* „Byte" 7 no. 8 Aug. 1982, ss. 210–228.
- Goodyear P.: *Commodore 64 Logo, a learning and teaching guide.* Ellis Horwood 1984.
- Goodyear P.: *Logo, a guide to learning through programming.* Ellis Horwood 1984.
- Gravina R.: *Smalltalk and the personal computer.* „Computer Education" no. 35. ss. 9–13.
- Harvey B.: *Why Logo?* „Byte" 7 no. 8 Aug. 1982, ss. 163–193.
- Hoppe H.U., Löthe H.: *Problemlösen und Programmieren mit Logo.* Teubner 1984.
- Howe J., O'Shea T., Plane F.: *Teaching mathematics through Logo programming: an evaluation study.* W: Lewis R., Tagg E. D. eds. *Computer Assisted Learning.* North Holland 1980, ss. 85–101.
- Kay A., Goldberg A.: *Personal dynamic media.* „Computer" 10 1977 no. 3, ss. 31–41.
- Lawler W.: *Designing computer-based microworlds.* „Byte" 7 no. 8 Aug. 1982, ss. 138–160.
- Lothe H.: *Logo Programmiersprache nicht nur fur Kinder.* „Chip" no. 1 Jan. 1983, ss. 122–124.
- McDougall A., Adams T., Adams P.: *Einstieg in Logo mit MIT-Logo und Apple-Logo.* Hanser 1984.
- McDougall A., Adams T., Adams P.: *Learning Logo on the Apple II.* Prentice Hall 1982.
- Menzel K.: *Logo in 100 Beispielen.* Teubner 1985.
- Nelson H.: *Logo for personal computers.* „Byte" 6 no. 6 June 1981, ss. 36–44.
- Nikolov R.: *Logo, pervi klas.* Sofia 1984.
- Nikolov R., Sendova E.: *Ezik i matematika. Logo, vtori klas.* Sofia 1984.
- O'Shea T.: *Artificial Intelligence and Computer Based Education.* „Computer Education" 1978 no. 30, ss. 25–28.
- Overall T. et al.: *Learning with Logo at the*

Lamplighter School. "Microcompunting" Sept. 1981, ss. 42–47.

– Papert S.: *Mindstorms: children, computers, and powerful ideas.* Basic Books 1980.

– Papert S.: *Redefining childhood: The computer presence as an experiment in developmental psychology.* W: Lavinston S. H. ed. *Information Processing 80.* North Holland 1980, ss. 993–998.

– Papert S.: *Teaching children to be mathematician versus teaching about mathematics.* „Int. J. Math. Educ. Sci. Technol." 3 1977, ss. 249–262; także: Taylor R. P. ed.: *The computer in the school, tutor, tool, tutee.* Teachers College Press 1980, ss. 177–196.

– Parr M.: *DIY Logo.* „Personal Computer World". Sep. 1982, ss. 156–159.

– Ross P., Howe J.: *Teaching mathematics through programming: ten years on.* W: Lewis R., Tagg D. E. eds.: *Computers in Education.* North Holland 1981, ss. 193–148.

– Ross P.: *Introducing Logo for the Apple II computer, Texas Instruments 99/4A and Tandy Color Computer.* Addison Wesley 1983.

– Ross P.: *Logo programming.* Addison-Wesley 1983.

– Solomon C., Harris D.: *Language in the Logo computer culture.* Proc. of the National Educational Computing Conference. U. Iowa 1979.

– Solomon C.: *Apple Logo: introduction to programming through turtle graphics.* Logo Computer Systems Inc. 1982.

– Solomon C.: *Introducing Logo to children.* „Byte" 7 no. 8 Aug. 1982. ss. 196–208.

– Sparer E.: *Sinclair Logo I. Turtle graphics.* Sinclair Research Ltd 1984.

– Sparer E.: *Sinclair Logo 2. Programming reference manual.* Sinclair Research Ltd 1984.

– Tater D., Sobalvarro P.: *The Terrapin Logo language tutorial.* Terrapin Inc.

– Thornburg D. D.: *Discovering Apple Logo.* Addison-Wesley 1983.

– Ury L.: *A teknosbeka-grafika.* „Mikroszamitogep Magazin" no. 1, 1985, ss. 16-19.

– Waligórski S.: *Elementy informatyki w liceum ogólnokształcącym.* „Matematyka" nr 6, 1985, ss. 284–295.

– Waligórski S.: *Język Logo i metody pracy z komputerem w szkole.* I Krajowa Konferencja: *Informatyka w szkole,* Wałbrzych 1985.

– Waligórski S.: *Logo.* „Informatyka" nr 7–9, 1985.

– Waligórski S.: *Wybrane języki i metody programowania w zastosowaniach. Współczesne kierunki rozwoju informatyki.* Materiały Szkoły Jesiennej PTI – Rydzyna 1984.

– Watt D.: *Logo in the schools.* „Byte" 7 no. 8 Aug. 1982, ss. 116–134.

– Weidenfeld G., Mathiew F., Perolat Y.D.: *Logo* Eyrolles 1984.

– Weir S.: *Logo and the exceptional child.* „Microcomputing" Sept. 1981, ss. 76–84.

– Weir S., Watt D.: *Logo: a computer environment for learning disabled students.* „The Computing Teacher" 8 1981 no 5. ss. 11–19.

– Williams G.: *Logo for the Apple II, the TI-99/4A, and the TRS-80 Color Computer.* „Byte" 7 no. 8 Aug. 1982. ss. 230–290.

– Wills S.: *Computer education in Tasmanian schools.* „The Australian Computer Bulletin" Aug. 1980. ss. 22–28.

– Wills S.: *Turtles for teaching computing.* W: Lewis R., Tagg D. E. eds. *Computers in Education.* Nort Holland 1981, ss. 795–796.

W treści