Rozdział 4

Programowanie wysokopoziomowe

Języki wysokiego poziomu można podzielić na różne sposoby. Jednym z kryteriów jest to, czy wyraża się w nich algorytm (sposób rozwiązania problemu), czy jedynie opisuje się problem jaki należy rozwiązać bez podawania sposobu jego rozwiązania:

- *języki algorytmiczne (imperatywne)* wyrażają **JAK** rozwiązać problem
- *języki deklaratywne* opisują **CO** należy rozwiązać

Najbardziej popularna jest rodzina języków imperatywnych i na początek przedstawimy pokrótce podstawowe koncepcje jakie w nich spotykamy.

4.1 Podstawowe koncepcje w językach algorytmicznych

W językach imperatywnych spotyka się te same koncepcje. Są one wspólne dla tych języków. Jak można się przekonać często kod w różnych językach wygląda podobnie i różni się tylko szczegółami składni charakterystycznej dla danego języka.

Duże różnice można dopiero zauważyć przy zmianie paradygmatu programowania. Trzy programy liczące to samo ale np. za pomocą programowania strukturalnego, programowania obiektowego i programowania współbieżnego będą istotnie się różnić.

4.1.1 Typy

Przez *typ* rozumie się zbiór wartości wraz z operacjami na tych wartościach i relacjami jakie można sprawdzać między wartościami tego typu.

Dla przykładu typ `bool` z języka C/C++ posiada dwie możliwe wartości `true` i `false`. Na wartościach tych można wykonywać np. operację logicznej koniunkcji `&&`, alternatywy `||` czy negacji `!`.

Można również sprawdzać, między innymi, czy dwie wartości logiczne są równe `==` albo różne `!=`.

Języki programowania można podzielić na takie, które bardzo dokładnie kontrolują zgodność typów, tzw. języki *silnie typowane*, i takie, które przy niezgodności typów dokonują niejawnej konwersji.

Dla przykładu, w języku **C**/**C**++, kompilator natrafiając na niezgodność typów dokona konwersji:

```
double d;
int n;
...
d = n;
```

wartość całkowita zmiennej `n` zostanie przekonwertowana do liczby rzeczywistej w postaci zmiennopozycyjnej podwójnej precyzji.

Analogiczny fragment kodu w języku silnie typowanym jakim jest **Ada**:

```
declare
  N : Integer;
  D : Long_Float;
begin
  ...
  D := N;
  ...
end;
```


powoduje błąd kompilacji i konieczność wpisania jawnej konwersji:

```
  D := Long_Float (N);
```

W języku **C**/**C**++ również możliwe jest wyrażenie jawnej konwersji dzięki operacji rzutowania:

```
d = (double) n;
```

Silne typowanie umożliwia wykrycie wielu błędów w programie już na etapie kompilacji. Z tego powodu w krytycznych zastosowaniach (np. lotnictwo, astronautyka, energetyka jądrowa, kolej, metro, aparatura medyczna), w których od poprawności działania zależy zdrowie i życie ludzkie, powinno stosować się języki silnie typowane.

4.1.2 Zmienne i stan obliczeń

Wartości na których operuje program przechowuje się w zmiennych. Zmienna oprócz ustalonego typu (determinuje możliwe wartości jakie może przyjąć), to ma

również swoją nazwę oraz położenie w pamięci komputera gdzie jej wartość jest przechowywana.

W językach imperatywnych typową cechą jest to, że zmienne mogą zmieniać swoją bieżącą wartość na inną.

Jako ciekawostkę można podać, że w szerokiej rodzinie języków deklaratywnych (np. programowania funkcyjnego albo programowania w logice) zmienną są (sic) niezmienne. Choć wydaje się to pozbawione sensu, to jednak takie „niezmienne" zmienne ułatwiają tworzenie programów współbieżnych oraz przeprowadzanie analizy poprawności programu.

Bieżące wartości wszystkich zmiennych programu stanową bieżący stan obliczeń. Podczas działania programu stan taki zmienia się i na koniec zawiera on wyliczone wartości stanowiące rozwiązanie problemu, który program rozwiązywał.

Szczególną rolę odgrywają tablice. Dzięki nim wygodne jest przetwarzanie wielu danych tego samego typu. Dzięki odwołaniom się do nich przez indeks łatwo jest skrócić kod do postaci pętli przebiegającej każdy element tablicy.

W języku **C**/**C++** tablice indeksowane są od 0. Tak prawdę mówiąc, to nawet nie ma w nim tablic w takim rozumieniu jak w innych językach. Jest jedynie arytmetyka na wskaźnikach:

$$\texttt{x[2]} \equiv \texttt{*(x+2)} \equiv \texttt{*(2+x)} \equiv \texttt{2[x]}.$$

W językach takich jak np. **Basic**, tablice indeksowane są od 1:

```
1000 DIM T(10)
1010 FOR I = 1 TO 10
1020 LET T(I) = I*I
1030 NEXT I
```

W języku **Pascal** i **Ada** tablice mogą mieć dowolny zakres. Przykład w **Adzie**:

```
declare
  T : array (-2 .. 7) of Integer := (-1 => 3, 1 => 2, others => 0);
begin
  T (3 .. 7) := T (-2 .. 2);
  for I in T'Range loop
    Put (T (I)); -- 0 3 0 2 0 0 3 0 2 0
  end loop;
end;
```

Można w nich również indeksować tablice typem wyliczeniowym. Przykład w **Adzie**:

```
type DzienTygodnia is (Poniedzialek, Wtorek, Sroda, Czwartek,
                       Piatek, Sobota, Niedziela);

LiczbaGodzinKonsultacji : array (DzienTygodnia) of Integer :=
    (Poniedzialek => 3, Wtorek => 3, Sroda => 1, Piatek => 1,
     others => 0);
```

4.1.3 Instrukcja przypisania

Główną instrukcją, która może zmienić stan obliczeń jest instrukcja podstawienia. W różnych językach może ona mieć różną postać ale najczęściej składa się z symbolu oznaczającego przypisanie (np. `:=` w językach Pascal i Ada) łączącego przypisywaną zmienną (po lewej stronie) i wyrażenie (po prawej stronie), którego wartość jest przypisywana zmiennej.

Po wykonaniu przypisania program osiąga nowy stan obliczeń. Zmiana stanu możliwa jest również w wyniku wykonania instrukcji wejścia, która pobiera wartość spoza pamięci (np. wpisana przez użytkownika na klawiaturze) umieszczą ją w zmiennej.

Zdarzają się języki, w których zmiana stanu może nastąpić w innych sytuacjach jak np. podczas obliczania wartości wyrażenia (w C można napisać wyrażenie `i + j++`, którego wartością jest suma wartości zmiennych `i` oraz `j` ale dodatkowo zmienna `j` zmieni swoją wartość na o jeden większą. Sytuacje takie nazywa się *efektami ubocznymi*. Możliwość ich pojawiania się w niektórych językach bardzo utrudnia analizę programów w nich napisanych. Są jednak takie języki jak np. Ada, gdzie efekty uboczne są całkowicie zakazane a to ze względu na zastosowanie tego języka w tworzeniu krytycznego oprogramowania od poprawności działania którego zależy zdrowie a nawet życie ludzkie.

4.1.4 Sekwencja instrukcji

W językach imperatywnych można tworzyć sekwencję (ciąg) instrukcji. W tym celu należy je oddzielić średnikami (jak np. w języku Pascal) albo umieścić je w kolejnych wierszach (jak np. w języku FORTRAN czy Python).

W niektórych językach jak np. **Pascal** średnik służy do oddzielania instrukcji zatem w bloku `begin ... end` nie ma potrzeby stawiania średnika po ostatniej instrukcji (gdy się go postawi, to interpretuje się go jako oddzielenie ostatniej instrukcji w sekwencji od instrukcji pustej znajdującej się przed słowem kluczowym `end`).

W innych językach jak np. **Ada** czy **C/C++** średnik kończy instrukcję zatem trzeba stawiać go po każdej instrukcji, nawet przed klamrą zamykającą blok `{ ... }`.

W języku C/C++ jest jednak wyjątek od tej reguły i nie trzeba stawiać średnika po instrukcji złożonej jeśli zawiera ona blok `{ ... }`. Z kolei w języku

Ada usunięto ten wyjątek i każda instrukcja kończy się średnikiem (stawia się go po końcu bloku instrukcji).

4.1.5 Instrukcje rozgałęzienia

Kolejnym podstawowym pojęciem w językach wysokiego poziomu jest rozgałęzienie. Dzięki niemu program może wybrać jedną z dwóch ścieżek obliczeń. Z instrukcjami rozgałęzienia spotkaliśmy się już w języku assemblera (np. instrukcja `BCC` mikroprocesor 6502, w której skok odbywa się w przypadku gdy znacznik przeniesienia C jest wyzerowany).

W językach wysokiego poziomu najczęściej rozgałęzienie występuje w postaci instrukcji `if-then-else`. W pełnej swojej postaci składa się z trzech elementów:

1. Warunku na podstawie którego następuje wybór jednej z dwóch ścieżek obliczeń.
2. Bloku instrukcji wykonywanych w przypadku spełnienia warunku (tzw. blok `then`).
3. Bloku instrukcji wykonywanych w przypadku niespełnienia warunku (tzw. blok `else`).

W języku C/C++ instrukcja rozgałęzienia ma następująca postać:


```
if(warunek)
  blok instrukcji
else
  blok instrukcji
```

Jeśli blok instrukcji składa się z tylko z jednej instrukcji, to można pominąć nawiasy klamrowe ją otaczające.

Należy zwrócić uwagę na to, że nie występuje w niej żaden znacznik gdzie się ona kończy. Zatem, kiedy zagnieżdża się jedną instrukcję rozgałęzienia w drugiej, jak np. w poniższym przykładzie:

```
z = 1; if(x > 0) if(y < 0) z = 2; else z = 3;
```

może dojść do pozornej niejednoznaczności. Czy słowo kluczowe `else` dotyczy instrukcji `if` z warunkiem `x > 0` czy raczej instrukcji `if` z warunkiem `y < 0`.

Standard języka rozstrzyga te wątpliwości wiążąc słowo `else` z najbliższą instrukcją `if`.

Powyższy przykład należy rozumieć następująco:

```
z = 1;
if(x > 0)
{
```

```
  if(y < 0)
  {
    z = 2;
  }
  else
  {
    z = 3;
  }
}
```

Aby program był czytelniejszy i nie zmuszał nas do zastanawiania się jak interpretować instrukcję, zaleca się stosowanie nawiasów klamrowych nawet w sytuacjach gdy wydają się zbędne.

Z takimi wątpliwościami nie ma np. w języku Ada, w którym specjalnie zaznacza się koniec instrukcji rozgałęzienia:

```
if warunek then
  blok instrukcji
else
  blok instrukcji
end if;
```


W języku Ada nie stosuje się nawiasów klamrowych do otaczania bloku instrukcji. Rolę takich nawiasów pełni para słów kluczowych `then` i `else` oraz para słów kluczowych `else` i `end if`.

Przykład z języka C możemy zapisać w języku Ada następująco:

```
Z := 1;
if X > 0 then
  if Y < 0 then
    Z := 2;
  else
    Z := 3;
  end if;
end if;
```

Inną ciekawą własnością instrukcji rozgałęzienia w języku Ada jest słowo kluczowe `elsif`, dzięki któremu zamiast zagnieżdżać instrukcję rozgałęzienia jedną w drugiej, można kontynuować ciągle tę samą instrukcję:

```
-- bez użycia elsif
if W1 then
  S1;
else
  if W2 then
```

```
-- z użyciem elsif
if W1 then
  S1;
elsif W2 then
  S2;
```

```
    S2;                        elsif W3 then
  else                           S3;
    if W3 then                 else
      S3;                        S4;
    else                       end if;
      S4;
    end if;
  end if;
end if;
```

Podobne rozwiązania spotyka się w innych językach (np. słowo `elif` w języku **Python**).

Inną instrukcją rozgałęzienia jest instrukcja `case`, znana np. z języków **Pascal** albo **Ada**, lub odpowiadająca jej instrukcja `switch` występująca w języku **C/C++**.

W języku **Ada** ma ona następującą postać:

```
case wyrażenie is
  when jedna lub więcej wartość =>
    blok instrukcji
  when jedna lub więcej wartość =>
    blok instrukcji
  ...
  when others =>
    blok instrukcji
 end case;
```


Działanie jej polega na wyliczeniu wartości wyrażenia i odnalezienie odpowiedniego przypadku `when` zawierającego tę wartość. Jeśli taki przypadek istnieje, to wykonywany jest odpowiadający mu blok instrukcji. W przeciwnym przypadku wykonywany jest blok instrukcji domyślnych podanych w przypadku `when others`.

Przykład:

```
case Digit is
  when 0 .. 4 =>
    Kind := Small_Digit;
  when 6 | 8 =>
    Kind := Big_Even_Digit;
  when others =>
    Kind := Big_Odd_Digit;
end case;
```

W języku C/C++ powyższy przykład wyglądałby następująco:

```
switch(cyfra)
{
  case 0:
  case 1:
  case 2:
  case 3:
  case 4:
    kind = small_digit;
    break;
  case 6:
  case 8:
    kind = big_even_digit;
    break;
  default:
    kind = big_odd_digit;
}
```

Zwróć uwagę na konieczność użycia instrukcji `brake` dzięki którym następuje wyskok na zewnątrz instrukcji `switch` po wykonaniu odpowiedniego bloku instrukcji (nie jest ona potrzebna na końcu ostatniego bloku).

Jako ciekawostkę można podać przykład ilustrujący jak nieelegancko zaprojektowano instrukcję `switch` w języku **C/C++**. Otóż fraza `case` jest tylko etykietą, do której następuje skok w przypadku danej wartości.

Spróbuj odpowiedzieć jakie liczby wydrukuje poniższy program:

```
// main.c
#include <stdio.h>
int main(void)
{
  switch(1)
  {
    for(int i = 0; i < 10; i++)
    {
      case 1:
        printf("%d\n", i);
    }
  }
  return 0;
}
```

Okazuje się, że mimo poprawności składniowej powyższego programu, pozbawiony jest on sensu. Za każdym jego uruchomieniem, drukuje on inne liczby (najczęściej duże więc tylko jedną):

```
$ gcc -o main main.c
$ ./main
348155941
$ ./main
211533861
$ ./main
531132453
$ ./main
256765989
$ ./main
307249189
```

Aby było jeszcze ciekawiej, program zaczyna drukować cyfry od 0 do 9 gdy skompiluje się go z opcją optymalizacji kodu:

```
$ gcc -O -o main main.c
$ ./main
0
1
2
3
4
5
6
7
8
9
```


4.1.6 Instrukcje pętli

Każdy nietrywialny program zawiera jakąś pętlę. Powtarza on wykonywanie pewnego fragmentu kodu ustaloną liczbę razy lub tak długo dopóki dane spełniają zadany warunek.

Podstawową instrukcją pętli jest instrukcja `while`. W języku **C**/**C++** ma ona postać:

```
while( warunek )
  blok instrukcji
```

Powtarza ona wykonywanie bloku instrukcji dopóki spełniony jest warunek.

W podpunkcie 4.4.1, w którym prezentowane będą pierwsze języki wysokiego poziomu, przy opisie języka **FORTRAN** podano przykład najdroższego błędu w programie (zamiana kropki na przecinek). Okazuje się, że w języku **C**/**C++** też możliwe jest popełnienie podobnego błędu.

Gdyby w poniższej pętli:

```
while(x < 1.2)
{
  ...
}
```

omyłkowo wpisano przecinek zamiast kropki w literale 1.2, to pętla nigdy by się nie zakończyła.

To dlatego, że przecinek w wyrażeniu `x < 1,2` jest zwykłym operatorem i nakazuje policzenie po kolei wartości dwóch wyrażeń `x < 1` i `2`, przy czym wartością jest wartość drugiego wyrażenia. Wartość `2`, będąc różną od zera, oznacza w miejscu warunku sterującego pętlą logiczną prawdę.

W języku **Ada** blok instrukcji otoczony jest słowami kluczowymi `loop` i `end loop`:

```
while warunek loop
  blok instrukcji
end loop;
```

Zaskakującą postać ma pętla `while` w języku **Python**. Otóż po warunku sterującym jest dwukropek a instrukcje składające się na blok instrukcji wpisuje się w kolejnych wierszach z wcięciem wykonanym tabulatorem:


```
while warunek:
    instrukcja
    instrukcja
    ...
    instrukcja
```

Jest to o tyle dziwna postać, że naciśnięcie lub nie naciśnięcie jednego znaku tabulacji zmienia sens pętli:

```
i = 1
while i < 10:
    print(i)
i = i + 1
```

Brak wcięcia w ostatnim wierszu powoduje, że instrukcja `i = i + 1` nie jest wykonywana wewnątrz pętli i pętla nie kończy pracy (w nieskończoność drukuje wartość `1`).

Drugą często używaną instrukcją pętli w językach algorytmicznych jest pętla `for`.

W języku **C**/**C++** ma ona nietypową postać:

```
for(inicjowanie; warunek; aktualizacja)
  blok instrukcji
```

i można ją traktować jako skrót następującej pętli `while`:

```
inicjowanie;
while( warunek )
{
  blok instrukcji
  aktualizacja;
}
```

Aby było ciekawiej, każda z części `inicjowanie`, `warunek` i `aktualizacja` jest opcjonalna i może być opuszczona. W skrajnym przypadku opuszczenie wszystkich trzech części oznacza nieskończoną pętlę:

```
for(;;)
  blok instrukcji
```

Pętla `for` języku **Ada** ma następującą postać:

```
for Zmienna in Zakres loop
  blok instrukcji
end loop;
```


Gdyby chcieć zapisać w **Adzie** nieskończoną pętlę, to można to zrobić następująco:

```
loop
  blok instrukcji
end loop;
```

4.1.7 Funkcje i procedury

W językach algorytmicznych istnieje możliwość wydzielenia fragmentu kodu stanowiącego logiczną całość, nazwanie go i wykorzystywanie w różnych miejscach programu wywołując ten kod.

Działanie kodu może zależeć od parametrów formalnych, których aktualne wartości ustalane są w chwili wywołania kodu.

Jeśli wydzielony kod służy do wyliczenia wartości, to nazywa się go funkcją i jego wywołanie może wystąpić w kontekście wyrażenia do którego zwróci wyliczoną wartość.

Gdy kod nie wylicza wartości, to nazywa się go procedurą i jego wywołanie może wystąpić w kontekście instrukcji.

Oto przykład funkcji zapisanej w języku **C**/**C**++ służącej wyliczeniu największego wspólnego dzielnika dwóch jej argumentów:

```
int nwd(int x, int y)
{
  while(y > 0)
  {
    int r = x % y;
    x = y;
    y = r;
  }
  return x;
}
```

W języku **Python** istnieje możliwość jednokrotnego przypisania wartości większej liczbie zmiennych. Powyższą funkcję zapisać możemy następująco bez pomocniczej zmiennej `r`:

```
def nwd(x, y):
    while y > 0:
        x, y = y, x % y
    return x
```


W języku **Ada** przyjęto natomiast, że parametry przez które nie odbiera się wyniku (parametry w trybie `in`) zachowują się wewnątrz funkcji jak stałe i aby je zmieniać, to trzeba utworzyć ich lokalne kopie w zmiennych zadeklarowanych wewnątrz funkcji:

```
function NWD (X, Y : in Integer) return Integer is
  A : Integer := X; -- lokalna kopia parametru X
  B : Integer := Y; -- lokalna kopia parametru Y
begin
  while B > 0 loop
    declare
      R : Integer := A rem B;
    begin
      A := B;
      B := R;
    end;
  end loop;
  return A;
end NWD;
```

Jako przykład procedury użyjemy fragment kodu, który dla całkowitego parametru `x` oddaje parametrem `y` wartość o jeden mniejszą od `x` a parametrem `z` oddaje wartość o jeden większą od `x`.

Procedury w języku **C**/**C++** zapisuje się w postaci funkcji, których typ wyniku jest `void` czyli pusty (nie składający się z żadnych wartości):

```
void p(int x, int* y, int* z)
{
  *y = x - 1;
  *z = x + 1;
}
```

Konieczne było użycie operatora dereferencji (gwiazdka), ponieważ w **C**/**C**++ dozwolony jest tylko jeden sposób przekazywania parametru: przez wartość. Aby odebrać z procedury wartość należy zainicjować parametr wskaźnikiem, gdzie ta wartość ma być odłożona:

```
int a = 10, b, c;
p(a, &b, &c);
```

W języku **Ada** mamy trzy różne tryby przekazywania parametrów:

- `in` – dostarczana wartość parametru,
- `out` – oddawana wartość parametru,
- `in out` – modyfikowana wartość parametru.

Powyższy przykład możemy zapisać w **Adzie** następująco:

```
procedure P (X : in Integer; Y, Z : out Integer) is
begin
  Y := X - 1;
  Z := X + 1;
end P;
```

A tak może wyglądać wywołanie procedury `P`:

```
declare
  A : Integer := 10;
  B : Integer;
  C : Integer;
begin
  P (A, B, C);
end;
```

4.2 Który język programowania jest najlepszy?

Czasami dochodzi do sporów, który język jest lepszy od którego. Moglibyśmy przyjąć za lepszy ten język programowania, w którym da się wyrazić więcej możliwych do zaprogramowania obliczeń.

Z punktu widzenia teorii informatyki takie spory są bezsensowne. Okazuje się, że wszystko to co może policzyć komputer da się zaprogramować na maszynie Turinga. Zatem języki programowania, w których można wyrazić to co za pomocą maszyną Turinga zwykło się nazywać Turing-zupełnymi (ang. Turing-complete).

Większość języków programowania jest Turing-zupełnymi. Jedynie nie są nimi te, w których nie można wyrazić niekończącej się pętli (jedynie pętle powtarzane skończoną liczbę razy). Przykładem takiego języka jest język **Coq**.

Ciekawym zagadnieniem jest pytanie o najmniejszy (co do liczby możliwych instrukcji) język programowania, który jest Turing-zupełny.

Jednym z takich małych języków jest zbiór trzech instrukcji języka Basic:

- `LET X = 0` – wyzerowanie zmiennej;
- `LET X = X + 1` – zwiększenie zmiennej o jeden;
- `IF X = Y THEN GOTO` – skok gdy wartości dwóch zmiennych są sobie równe.

Aby uprościć korzystanie z programu, dodamy jeszcze dwie instrukcje:

- `INPUT X` – wczytaj z klawiatury wartość zmiennej;
- `PRINT X` – wydrukuj na ekranie wartość zmiennej.

Poniższy program oblicza dla podanej wartości n wartość $n!$ (silnia n równa $1 \times 2 \times \cdots \times n$).

```
1000 INPUT N
1010 LET U=0
1020 LET U=U+1
1030 LET R=0
1040 IF R=N THEN GOTO 1230
1050 LET R=R+1
1060 IF R=N THEN GOTO 1230
1070 LET V=0
1080 IF V=U THEN GOTO 1110
1090 LET V=V+1
1100 IF V=V THEN GOTO 1080
1110 LET S=0
1120 LET S=S+1
1130 LET T=0
1140 IF T=V THEN GOTO 1180
1150 LET U=U+1
1160 LET T=T+1
1170 IF T=T THEN GOTO 1140
```

Rysunek 4.1: Obliczenia silni w języku Basic na komputerze Atari 800.

```
1180 IF S=R THEN GOTO 1210
1190 LET S=S+1
1200 IF S=S THEN GOTO 1130
1210 IF R=N THEN GOTO 1230
1220 IF R=R THEN GOTO 1050
1230 PRINT U
```


Na rysunku 4.1 przedstawiono przykłady uruchomienia powyższego programu na komputerze **Atari 800**. Czas wykonywania programu bardzo szybko rośnie wraz ze wzrostem wczytywanej wartości.

Jeszcze ciekawszym językiem Turing-zupełnym jest jedna instrukcja `MOV` procesora `x86`. Więcej o tym w artykule: `http://drwho.virtadpt.net/files/mov.pdf`

Jak widać spór o to który język programowania może wyrazić więcej programów jest bezsensowny, bo wszystkie języki Turing-zupełne potrafią wyrazić to samo (to co może policzyć maszyna Turinga).

Zastanawiać się można jedynie, który język programowania jest lepszy do konkretnego zastosowania. Na przykład gdybym miał pisać program dokonujący obliczeń symbolicznych na wzorach, to zdecydowanie bardziej nadają się do tego takie języki jak np. **LISP** albo **Prolog** niż np. **C/C++** albo **FORTRAN**.

4.3 Formalna weryfikacja poprawności programu

Przedstawimy na przykładzie języka **SAPRK**, będącego podzbiorem języka **Ada**, na jakie problemy można natrafić przy projektowaniu najprostszej funkcji. Postaramy się napisać program, który da się dowieść formalnie, że wykonuje to co zaplanowaliśmy.

Jako przykładowy problem rozpatrzmy znajdowanie największej wartości w tablicy jednowymiarowej.

Wersja 1

Zdefiniujmy w języku **SPARK** pakiet `Max`, który dostarcza nam typ `Vector` (tablica jednowymiarowa) i funkcję `Find` znajdującą w zadanej tablicy największą wartość.

W pliku `max.ads` umieścimy specyfikację tego pakietu:

```
1  package Max with SPARK_Mode is
2
3    type Vector is array (Integer range <>) of Integer;
4
5    function Find (V : Vector) return Integer
6    with
7      Post => (for all I in V'Range =>
8                 V(I) <= Find'Result);
9
10 end Max;
```

Napisano w niej w warunku końcowym `Post`, że funkcja `Find` gwarantuje, że zwrócona prze nią wartość będzie będzie większa lub równa od każdej wartości w zadanej tablicy `V`.

O tym jak słaba jest ta specyfikacja można przekonać się implementując funkcję `Find` tak jak w poniższym pliku `max.adb`:

```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := Integer'Last;
5    begin
6      return M;
7    end Find;
8
9  end Max;
```

Jak widać funkcja `Find` nie zagląda do elementów tablicy `V` a jedynie zwraca największą wartość w typie `Integer`.

O tym, że tak źle napisana funkcja spełnia specyfikację można przekonać się wywołując następująco program `gnatprove` dowodzący poprawność programu:

```
$ gnatprove -P default.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /TI/src/spark/wersja1/gnatprove/gnatprove.out
```

Potrzebny do wykonania powyższego polecenia plik `default.gpr` zawiera dwie linijki:

```
project Default is
end Default;
```

Wersja 2

Poprawimy specyfikację funkcji `Find` dodając dodatkowy warunek, że zwracana wartość musi być jedną z wartości w tablicy:

```
1  package Max with SPARK_Mode is
2
3    type Vector is array (Integer range <>) of Integer;
4
5    function Find (V : Vector) return Integer
6    with
7      Post => (for all I in V'Range =>
8                  V(I) <= Find'Result) and
9              (for some I in V'Range =>
10                 V(I) = Find'Result);
11
12 end Max;
```



```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := Integer'Last;
5    begin
6      return M;
7    end Find;
8
9  end Max;
```

Tym razem powyższa implementacja funkcji `Find` nie spełnia specyfikacji:

```
$ gnatprove -P default.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
max.ads:7:13: medium: postcondition might fail (e.g. when
Find'Result = Integer'Last and V = (others => 1) and
V'First = 1 and V'Last = 0)
Summary logged in /TI/src/spark/wersja2/gnatprove/gnatprove.out
```

Program `gnatprove` podał nawet przypadek kiedy będzie ona działała niepoprawnie. Wystarczy, że tablica `V` będzie pusta (nie będzie miała elementów). Taka pusta tablica ma początkowy indeks `V'First` większy od ostatniego indeksu `V'Last`.

Wersja 3

Dopiszmy w specyfikacji funkcji `Find` dodatkowy warunek, że tablica dostarczana do niej nie jest pusta (warunek wstępny `Pre`):

```
1  package Max with SPARK_Mode is
2
3    type Vector is array (Integer range <>) of Integer;
4
5    function Find (V : Vector) return Integer
6    with
7      Pre => V'First <= V'Last,
8      Post => (for all I in V'Range =>
9                  V(I) <= Find'Result) and
10               (for some I in V'Range =>
11                 V(I) = Find'Result);
12
13 end Max;
```



```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := Integer'Last;
5    begin
6      return M;
7    end Find;
8
9  end Max;
```

Tym razem program `gnatprove` stwierdza, że nie jest spełnione warunek końcowy:

```
$ gnatprove -P default.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
max.ads:8:13: medium: postcondition might fail (e.g. when
Find'Result = Integer'Last and V = (others => 0) and
V'First = 0 and V'Last = 0)
Summary logged in /TI/src/spark/wersja3/gnatprove/gnatprove.out
```

Podaje nawet w jakim przypadku funkcja źle zadziała. Wystarczy, że tablica ma jeden element równy 0 a zwracana wartość jest największą wartością w typie `Integer`.

Wersja 4

Poprawmy więc implementację funkcji `Find` tak aby wykonywała pętlę po elementach tablicy i faktycznie poszukiwała największy z jej elementów:

```
1  package Max with SPARK_Mode is
2
3    type Vector is array (Integer range <>) of Integer;
4
5    function Find (V : Vector) return Integer
6    with
7      Pre => V'First <= V'Last,
8      Post => (for all I in V'Range =>
9                  V(I) <= Find'Result) and
10               (for some I in V'Range =>
11                 V(I) = Find'Result);
12
13 end Max;
```

```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := V(V'First);
5    begin
6      for I in V'First + 1 .. V'Last loop
7        if V(I) > M then
8          M := V(I);
9        end if;
10     end loop;
11     return M;
12   end Find;
13
14 end Max;
```

Tym razem program `gnatprove` stwierdza, że możliwe jest błędne działanie programu jeśli początkowy indeks tablicy ma wartość równą największej liczbie w typie `Integer` (nie da się policzyć wartości o jeden większej, gdyż wykracza ona poza zakres tego typu):

```
$ gnatprove -P default.gpr
```

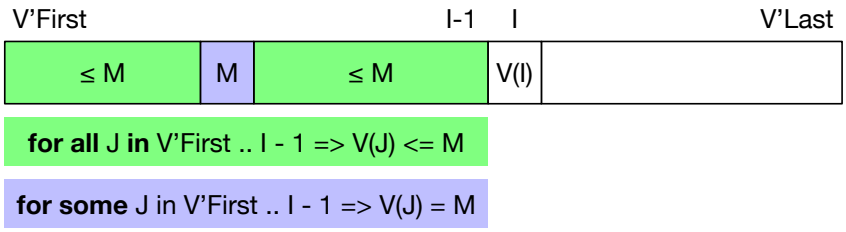
```
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
max.adb:6:22: medium: overflow check might fail (e.g. when
V = (Integer'Last => 0, others => Integer'Last) and
V'First = 2147483647 and V'Last = 2147483647)
max.ads:9:16: medium: postcondition might fail, cannot prove
V(I) <= Find'result (e.g. when Find'Result = -1 and I = 0 and
V = (1 => -1, others => 0) and V'First = 0 and V'Last = 1)
Summary logged in /TI/src/spark/wersja4/gnatprove/gnatprove.out
```

Wersja 5

Dodajmy w specyfikacji funkcji `Find` dodatkowy warunek wstępny, że pierwszy indeks tablicy `V` jest mniejszy od największej liczby w typie `Integer`:

```
1  package Max with SPARK_Mode is
2
3    type Vector is array (Integer range <>) of Integer;
4
5    function Find (V : Vector) return Integer
6    with
7      Pre => V'First <= V'Last and
8             V'First < Integer'Last,
9      Post => (for all I in V'Range =>
10                V(I) <= Find'Result) and
11              (for some I in V'Range =>
12                 V(I) = Find'Result);
13
14 end Max;
```

```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := V(V'First);
5    begin
6      for I in V'First + 1 .. V'Last loop
7        if V(I) > M then
8          M := V(I);
9        end if;
10     end loop;
11     return M;
12   end Find;
13
14 end Max;
```



Rysunek 4.2: Niezmiennik pętli poszukującej największego elementu.

O dziwo tym razem program `gnatprove` nadal nie był w stanie udowodnić poprawności naszej funkcji, mimo tego, że jest już ona napisana poprawnie:

```
$ gnatprove -P default.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
max.ads:10:16: medium: postcondition might fail, cannot prove
V(I) <= Find'result (e.g. when Find'Result = -1 and I = 0 and
V = (1 => -1, others => 0) and V'First = 0 and V'Last = 1)
Summary logged in /TI/src/spark/wersja5/gnatprove/gnatprove.out
```


Wersja 6

Teraz dopiero robi się ciekawie. Mamy poprawny program i musimy pomóc programowi dowodzącemu. W tym celu trzeba podpowiedzieć mu jaki warunek jest spełniony przy każdym przebiegu pętli. Warunek taki nazywa się *niezmiennikiem pętli*.

Niezmiennik pętli faktycznie wyraża to co wykonuje pętla. Pisząc pętlę mamy go na myśli, nawet jeśli nie wiemy, że nazywa się on niezmiennikiem i pełni tak ważną rolę w formalnym dowodzeniu poprawności programów.

W naszym przykładzie wykonujemy iteracje po kolejnych elementach tablicy. Kiedy jesteśmy przy elemencie `V(I)` na pozycji `I`, to wszystkie wcześniejsze elementy były już zbadane i zmienna `M` jest największą wartością spośród wartości na pozycjach od `V'First` do `I - 1` (patrz rysunek 4.2).

Niezmiennik pętli wpisuje się w implementacji funkcji przy pomocy pragm `Loop_Invariant`. W naszym przykładzie jest to prosty warunek i wręcz przypomina warunek końcowy ze specyfikacji funkcji z tą różnicą, że w specyfikacji dotyczył całej tablicy a niezmiennik mówi tylko coś o dotychczas przejrzanym kawałku tablicy.

```
1 package Max with SPARK_Mode is
2
```

```
3   type Vector is array (Integer range <>) of Integer;
4
5   function Find (V : Vector) return Integer
6   with
7     Pre => V'First <= V'Last and
8            V'First < Integer'Last,
9     Post => (for all I in V'Range =>
10               V(I) <= Find'Result) and
11             (for some I in V'Range =>
12               V(I) = Find'Result);
13
14 end Max;
```

```
1  package body Max with SPARK_Mode is
2
3    function Find (V : Vector) return Integer is
4      M : Integer := V(V'First);
5    begin
6      for I in V'First + 1 .. V'Last loop
7        pragma Loop_Invariant
8          (for all J in V'First .. I - 1 => V(J) <= M);
9        pragma Loop_Invariant
10         (for some J in V'First .. I - 1 => V(J) = M);
11       if V(I) > M then
12         M := V(I);
13       end if;
14     end loop;
15     return M;
16   end Find;
17
18 end Max;
```


Tym razem program `gnatprove` stosując indukcję matematyczną po liczbie wykonanych iteracji pętli potrafi udowodnić poprawność podanego niezmiennika i wywnioskować z niego, że implementacja funkcji `Find` spełnia zadaną specyfikację:

```
$ gnatprove -P default.gpr
Phase 1 of 2: generation of Global contracts ...
Phase 2 of 2: flow analysis and proof ...
Summary logged in /TI/src/spark/wersja6/gnatprove/gnatprove.out
```

Na rysunku 4.3 przedstawiono pełny raport z dowodzenia (znajduje się on w pliku `gnatprove.out`). Można z niego odczytać, że program miał do udowodnienia 15 warunków i wszystkie udało się udowodnić.

```
$ more gnatprove/gnatprove.out
Summary of SPARK analysis
=========================

-----------------------------------------------------------------------------------------------
SPARK Analysis results        Total      Flow   Interval   CodePeer      Provers   Justified   Unproved
-----------------------------------------------------------------------------------------------
Data Dependencies                 .         .          .          .            .           .          .
Flow Dependencies                 .         .          .          .            .           .          .
Initialization                    .         .          .          .            .           .          .
Non-Aliasing                      .         .          .          .            .           .          .
Run-time Checks                  10         .          .          .   10 (CVC4)           .          .
Assertions                        4         .          .          .    4 (CVC4)           .          .
Functional Contracts              1         .          .          .    1 (CVC4)           .          .
LSP Verification                  .         .          .          .            .           .          .
-----------------------------------------------------------------------------------------------
Total                            15         .          .          .   15 (100%)           .          .


max steps used for successful proof: 1

Analyzed 1 unit
in unit max, 2 subprograms and packages out of 2 analyzed
  Max at max.ads:1 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (0 checks)
  Max.Find at max.ads:5 flow analyzed (0 errors, 0 checks and 0 warnings) and proved (15 checks)
```


Rysunek 4.3: Raport z dowodzenia poprawności pakietu `Max`.

4.4 Historia języków programowania

Dokonamy przeglądu najważniejszych języków programowania jakie powstały na przestrzeni kilkudziesięciu lat.

4.4.1 Lata 1950-te (początki)

1955 FLOW-MATIC Język opracowany przez zespół pod kierownictwem Grace Hopper. Z języka tego wywodzi się język COBOL.

1957 FORTRAN Język opracowany przez zespół w firmie IBM pod kierownictwem Johna Backusa. Nazwa języka jest akronimem od `FORmula TRANslator`. Głównym przeznaczeniem języka są obliczenia numeryczne. Język FORTRAN jest najstarszym używanym do dzisiaj językiem wysokiego poziomu.

Ciekawostka: oto najdroższy błąd w programie. Zmiana przecinka na kropkę (sąsiednie klawisze) kosztowała 135 milionów dolarów.

```
DO 10 K = 1. 3
...
10    CONTINUE
```


Kompilator języka FORTRAN zanim przystąpi do analizowania linii kodu, usuwa z niej wszystkie spacje. Zatem pierwszy wiersz przykładu wygląda podczas analizy następująco: `DO10K=1.3`.

Gdyby zamiast kropki był przecinek, to kompilator rozpoznałby instrukcję pętli, która kończy się w wierszu z etykietą 10 i ma wykonać się dla trzech kolejnych wartości `K = 1, 2, 3`.

Wpisanie kropki zmieniło sens instrukcji. Kompilator rozpoznał ją jako podstawienie wartości 1.3 pod zmienną o nazwie `DO10K`. Zmienne w FORTRANIe nie muszą być deklarowane a instrukcja `CONTINUE` oznacza pustą instrukcję, zatem program jest poprawny składniowo i wykona raz treść pętli dla nieokreślonej wartości zmiennej `K`.

W roku 1962, sonda Mariner, udająca się w kierunku Wenus, była sterowana programem z takim błędem i zaraz po starcie zmieniła trajektorię lotu zagrażając życiu ludzi. Z tego powodu konieczne było jej zdalne zniszczenie.

1958 LISP Język opracowany przez zespół na uczelni MIT pod kierownictwem Johna McCarthy'ego. Nazwa jego jest akronimem od `LISt Processor`. Język opiera się na rachunku lambda opracowanym w latach trzydziestych przez Alonzo Churcha (abstrakcyjnym model obliczeń równoważny maszynom Turinga). Głównym zastosowaniem języka LISP są obliczenia symboliczne i sztuczna inteligencja. Jest drugim po FORTRANIE najdłużej używanym współcześnie językiem programowania.

W języku danymi są listy i programy zapisuje się w postaci list. Zatem dane i programy nie są w tym języku rozróżnialne. W prosty sposób można napisać programy operujące na programach (metaprogramowanie).

Przykład funkcji rekurencyjnej obliczającej wyrazy ciągu Fibonacciego:

```
>>> (define fib (lambda (n)
(cond ((eq n 0) 0)
((eq n 1) 1)
(t (+ (fib (- n 1)) (fib (- n 2)))))))
>>> (fib 10)
55
```

1958 ALGOL 58 Język opracowany przez europejski zespół uczonych. Nazwa jest akronimem od `ALGOrithmic Language`. Przez ponad trzydzieści lat był używany w podręcznikach i artykułach jako ogólnie znany sposób zapisywania algorytmów. To z niego wywodzą się tzw. języki *algolo-podobne*, jak np. `Pascal`, `Modula` czy `Ada`.

1959 COBOL Język opracowany przez zespół pod kierunkiem Grace Hopper. Nazwa jego jest akronimem od `COmmon Business Oriented Language`. Głównym zastosowaniem języka są aplikacje biznesowe przetwarzające bazy danych. Instrukcje w tym języku zapisywane były w postaci zbliżonej do zdań języka angielskiego:

```
ADD A TO B.
ADD B TO C GIVING D.
```

4.4.2 Lata 1960-te

1962 APL Język do obliczeń algebraicznych. Nazwa jego jest skrótem od `A Programming Language`. Żartuje się, że programy w nim napisane są tak nieczytelne, że nawet ich autorzy nie są w stanie po godzinie zrozumieć jak one działają. Programy to ciągi symboli i greckich liter.

1962 Simula Pierwszy język programowania obiektowego. Głównym jego zastosowaniem były, jak sama nazwa wskazuje, symulacje komputerowe.

1962 SNOBOL Język do przetwarzania napisów. Jego nazwa to akronim od `StriNg Oriented and symBOlic Language`.

Poniższy program kopiuje wiersze ze standardowego wejścia na standardowe wyjście. Jeśli próba czytania zakończy się niepowodzeniem, to nastąpi skok do wiersza z etykietą `END`. Po wydrukowaniu wiersza następuje bezwarunkowy skok do wiersza z etykietą `LOOP`:

Rysunek 4.4: Program w języku SNOBOL uruchomiony w przeglądarce.

```
LOOP    X = INPUT       :F(END)
        OUTPUT = X      :(LOOP)
END
```

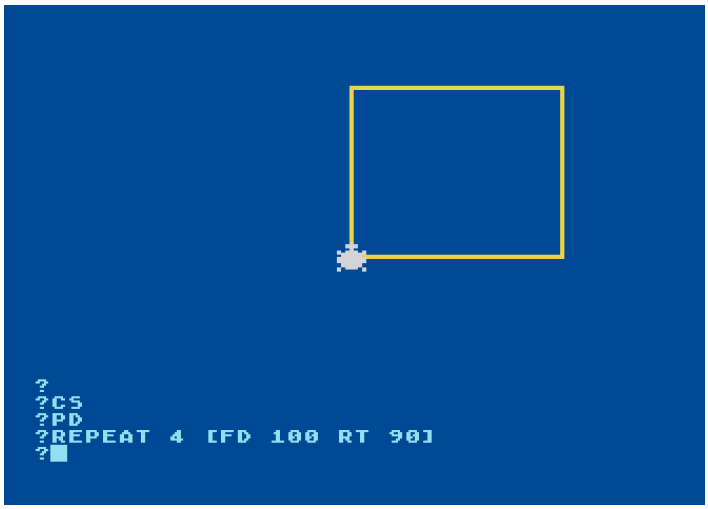
Językiem **SNOBOL** można pobawić się na stronie `https://tio.run/#snobol4` (patrz rysunek 4.4).

1963 CPL Pierwszy poprzednik języka **C**.

1964 BASIC Nazwa języka jest akronimem od `Beginner's All-purpose Symbolic Instruction Code`. Język ten był szczególnie popularny w latach 80-tych na mikrokomputerach (np. Apple II, ZX-81, ZX-Spectrum, Atari, Commodore).

Poniższy program drukuje tabliczkę mnożenia:

```
10 FOR I=1 TO 10
20 FOR J=1 TO 10
30 PRINT I*J; " ";
```



Rysunek 4.5: Rysowanie w języku Logo

```
40 NEXT J
50 PRINT
60 NEXT I
```


1964 PL/1 Język programowania o niezbyt skromnej nazwie `Programming Language One`. Żartuje się, że nie ma na świecie człowieka, który znałby cały ten język. Opracowany był przez duży zespół ludzi i dokładano do niego wszystko to, co spodobało się w innych językach.

1967 BCPL Drugi poprzednik języka **C**.

1968 Logo Język opracowany na MIT dla dzieci do nauki programowania. Jego charakterystyczną cechą jest łatwe tworzenie nawet skomplikowanej grafiki (fraktale) za pomocą prostych instrukcji grafiki żółwia.

Na rysunku 4.5 przedstawiono program rysujący kwadrat. Instrukcja `CS` czyście ekran (ang. Clear Screen), następnie opuszcza pióro (ang. Pen Down) a potem powtarza czterokrotnie dwie instrukcje: rysowanie linii długości 200 (ang. ForwarD) i obrót w prawo o 90 stopni (ang. RighT).

1969 B Bezpośredni poprzednik języka `C`.

4.4.3 Lata 1970-te (nowe paradygmaty)

1970 Pascal Niklaus Wirth, wielki propagator programowania strukturalnego, opracował język Pascal do nauki tego paradygmatu programowania. Jeszcze w latach 90-tych język ten był używany w wielu firmach programistycznych. Na komputerach PC pod systemami DOS i Windows, szczególnie popularne były kompilatory tworzone przez firmę Borland (Turbo-Pascal i Borland-Pascal).

Język ten miał duży wpływ na inne języki programowania jak np. **Ada** albo **Modula**.

1970 Forth Ciekawy język pierwotnie opracowany do sterowania teleskopem astronomicznym. Wszystkie obliczenia oraz sterowanie odbywa się w nim dzięki operacjom na stosach. Dzięki bardzo dużej szybkości wykonywania kodu był popularny na ośmiobitowych mikrokomputerach nawet do tworzenia grafiki w czasie rzeczywistym.

```
2 2 + . <enter> 4 ok

: fib 2dup + dup . ; <enter> ok
1 1 <enter> ok
fib <enter> 2 ok
fib <enter> 3 ok
fib <enter> 5 ok
fib <enter> 8 ok

: test 10 0 ?do i . loop ;  <enter> ok
test <enter> 0 1 2 3 4 5 6 7 8 9  ok
```


1972 C Język stworzony przez Dennisa Ritchiego do programowania systemowego.

1972 Smalltalk Język programowania obiektowego. Dodano go do języka **C** tworząc w ten sposób język **Objective-C**.

W języku **Smalltalk** wszystko jest obiektem. Nawet obliczenie wartości wyrażenia `1 + 2`, to wysłanie do obiektu `1` komunikatu `+` a argumentem `2`.

1972 Prolog Język programowania w logice opracowany przez zespół profesora Alaina Colmerauera z uniwersytetu w Marsylii. Nie miał on żadnego poprzednika. Wywodzi się bezpośrednio z rachunku predykatów (rachunku kwantyfikatorów). Jest przykładem języka deklaratywnego. Za pomocą formuł logicznych opisuje się w nim co należy policzyć a system stosując metodę rezolucji (metoda dowodzenia twierdzeń) wnioskuje co jest rozwiązaniem.

Dla przykładu rozpatrzmy następujący przykład dotyczący operowania na listach.

Listy w języku Prolog zapisuje się wymieniając między kwadratowymi nawiasami elementy oddzielone przecinkami.

Przykłady list:

```
[]     lista pusta
[a]    lista jednoelementowa
[2, b] lista dwuelementowa
```

Pionową kreską można oddzielić początkowe elementy listy od listy pozostałych elementów:

```
[1, 2, 3] = [1 | [2, 3]] = [1, 2 | [3]] = [1, 2, 3 | []]
```

Elementami listy mogą być inne listy:

```
[1, [2, [3, 4], 5], 6]  lista trójelementowa
```

Program w Prologu składa się z predykatów (warunków). Jednym z nich jest `append/3`. Ma on trzy argumenty będące listami i wyraża warunek, że trzecia lista jest połączeniem dwóch pierwszych.

Przykładowe zapytania:

```
?- append([1, 2], [3, 4, 5], [1, 2, 3, 4, 5]).
true.

?- append([1, 2], [3, 4, 5], X).
X = [1, 2, 3, 4, 5].
```

W pierwszym zapytaniu polecamy sprawdzić czy lista `[1, 2, 3, 4, 5]` jest połączeniem list `[1, 2]` i `[3, 4, 5]`.

W drugim szukamy takiej listy `X`, która jest połączeniem list `[1, 2]` i `[3, 4, 5]`.

Predykat `append/3` nie jest zwykłą funkcją oczekującą argumentów i wyliczającą wynik. To logiczny warunek zatem możemy mu dostarczać dane w dowolny sposób:

```
?- append([1, 2], X, [1, 2, 3, 4, 5]).
X = [3, 4, 5].

?- append(X, [3, 4, 5], [1, 2, 3, 4, 5]).
X = [1, 2] ;
false.
```

W pierwszym przypadku polecamy znaleźć taką listę, która dołączona na końcu listy `[1, 2]` stworzy listę `[1, 2, 3, 4, 5]`.

Podobnie w drugim szukamy listy `X`, którą należy wydłużyć o listę `[3, 4, 5]` aby otrzymać listę `[1, 2, 3, 4, 5]`. Po znalezieniu pierwszej odpowiedzi Prolog jest gotowy szukać kolejnej ale po naciśnięciu przez użytkownika znaku średnik nie znajduje innej listy (druga odpowiedź **false**).

Aby było ciekawiej możemy zapytać się o dwie listy, których połączeniem jest lista `[1, 2, 3]`:

```
?- append(X, Y, [1, 2, 3]).
X = [],
Y = [1, 2, 3] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 2, 3],
Y = [] ;
false.
```


Jak widać Prolog rozerwał listę `[1, 2, 3]` na wszystkie cztery różne sposoby.

Czy definicja predykatu `append/3` jest skomplikowana jeśli potrafi tyle różnych rzeczy?

Otóż nie. Dzięki temu, że wyrażamy **co to znaczy, że lista jest połączeniem dwóch list** a nie **sposób (algorytm) łączenia list**, definicja jest bardzo prosta:

```
append([], X, X).
append([X | T], Y, [X | Z]) :- append(T, Y, Z).
```

Pierwsza warunek stwierdza fakt, że połączeniem listy pustej z dowolną listą `X` jest lista `X`.

Drugi warunek to reguła w postaci implikacji **jeśli ..., to ...**. Należy go rozumieć następującą (implikację `:-` czytamy z prawa na lewo): *jeśli lista Z jest połączeniem listy T z listą Y, to lista [X | Z] jest połączeniem listy [X | T] z listą Y.*

1973 ML

1975 Scheme

1978 SQL

4.4.4 Lata 1980-te

1980 C++

1983 Ada

1984 Common Lisp

1984 MATLAB

1985 Eiffel

1986 Objective-C

1986 Erlang

1987 Perl

4.4.5 Lata 1990-te (era internetu)

1990 Haskell

1991 Python

1991 Visual Basic

1993 R

1995 Ruby

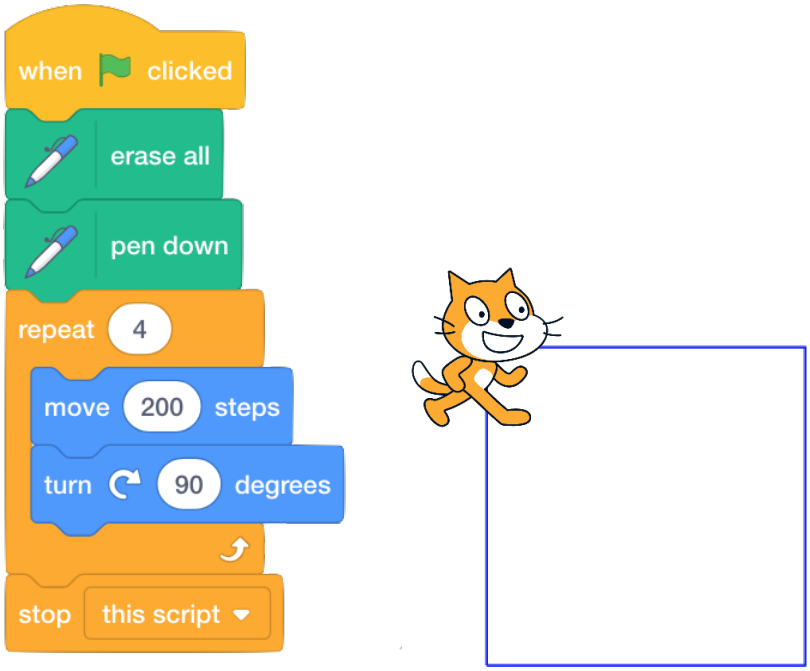
1995 Ada 95

1995 Java

1995 Object Pascal

1995 Javascript

1995 PHP



Rysunek 4.6: Rysowanie kwadratu w języku Scratch.

4.4.6 Lata 2000-ne

2001 C#

2001 D

2002 Scratch

2003 Scala

2005 Ada 2005

2005 F#

2007 Clojure

2009 GO

4.4.7 Lata 2010-te (programowanie funkcyjne, współbieżne i rozproszone)

2010 Rust

2012 Ada 2012

2012 Julia

2014 Swift

4.4.8 Lata 2020-te

2022 Ada 2022 Język zostanie rozszerzony między innymi o wygodne wyrażanie algorytmów równoległych.

Rozpatrzmy poniższy przykład:

```
1  declare
2    SUM : Integer := 0;
3    T   : array (1 .. 10) of Integer :=
4              (4, 2, 5, 7, 2, 1, 4, 5, 7, 5);
5  begin
6    for X of T loop
7      SUM := SUM + X;
8    end loop;
9    Put (SUM);
10 end;
```


Oblicza on sumę elementów tablicy. Jeśli nasz komputer posiada więcej niż jeden procesor (jeden rdzeń), to obliczenia sumy można przyspieszyć rozkładając pracę na poszczególne procesory (rdzenie):

```
1  declare
2    SUM : Integer := 0;
3    T   : array (1 .. 10) of Integer :=
4              (4, 2, 5, 7, 2, 1, 4, 5, 7, 5);
5  begin
6    parallel for X of T loop
7      SUM := SUM + X;
8    end loop;
9    Put (SUM);
10 end;
```