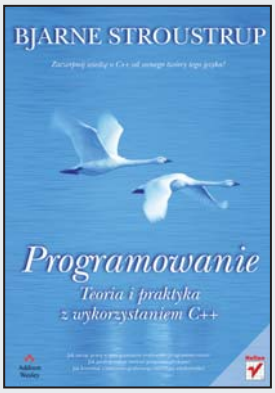
Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl


# Programowanie. Teoria i praktyka z wykorzystaniem C++




Autor: Bjarne Stroustrup
Tłumaczenie: Łukasz Piwko
ISBN: 978-83-246-2233-7
Tytuł oryginału: Principles and Practice Using C++
Format: 172×245, stron: 1112

**Zaczerpnij wiedzę o C++ od samego twórcy języka!**

- Jak zacząć pracę w zintegrowanym środowisku programistycznym?
- Jak profesjonalnie tworzyć programy użytkowe?
- Jak korzystać z biblioteki graficznego interfejsu użytkownika?

Jeśli zależy Ci na tym, aby zdobyć rzetelną wiedzę i perfekcyjne umiejętności programowania z użyciem języka C++, powinieneś uczyć się od wybitnego eksperta i twórcy tego języka – Bjarne Stroustrupa, który jako pierwszy zaprojektował i zaimplementował C++. Podręcznik, który trzymasz w ręku, daje Ci szansę odkrycia wszelkich tajników tego języka, obszernie opisanego w międzynarodowym standardzie i obsługującego najważniejsze techniki programistyczne. C++ umożliwia pisanie wydajnego i eleganckiego kodu, a większość technik w nim stosowanych można przenieść do innych języków programowania.

Książka „Programowanie w C++. Teoria i praktyka" zawiera szczegółowy opis pojęć i technik programistycznych, a także samego języka C++, oraz przykłady kodu. Znajdziesz tu również omówienia zagadnień zaawansowanych, takich jak przetwarzanie tekstu i testowanie. Z tego podręcznika dowiesz się, na czym polega wywoływanie funkcji przeciążonych i dopasowywanie wyrażeń regularnych. Zobaczysz też, jaki powinien być standard kodowania. Poznasz sposoby projektowania klas graficznych i systemów wbudowanych, tajniki implementacji, wykorzystywania funkcji oraz indywidualizacji operacji wejścia i wyjścia. Korzystając z tego przewodnika, nauczysz się od samego mistrza pisać doskonałe, wydajne i łatwe w utrzymaniu programy.

- Techniki programistyczne
- Infrastruktura algorytmiczna
- Biblioteka standardowa C++
- Instrukcje sterujące i obsługa błędów
- Implementacja i wykorzystanie funkcji
- Kontrola typów
- Interfejsy klas
- Indywidualizacja operacji wejścia i wyjścia
- Projektowanie klas graficznych
- Wektory i pamięć wolna
- Kontenery i iteratory
- Programowanie systemów wbudowanych
- Makra

**Wykorzystaj wiedzę Bjarne Stroustrupa i pisz profesjonalne programy w C++!**

# Spis treści

## Rozdział 4. Wykonywanie obliczeń 103



## Rozdział 5. Błędy 139

## Rozdział 6. Pisanie programu 175



## Rozdział 7. Kończenie programu 217

## Rozdział 8. Szczegóły techniczne — funkcje itp. 247



## Rozdział 9. Szczegóły techniczne — klasy itp. 287

# Część III Dane i algorytmy 523

## Rozdział 17. Wektory i pamięć wolna 525



## Rozdział 18. Wektory i tablice 563

# Dodatki 941

## Dodatek A Zestawienie własności języka 943



## Dodatek B Biblioteka standardowa 991



## Dodatek C Podstawy środowiska Visual Studio 1043

# 6
# Pisanie programu

Pisać program, znaczy rozumieć.

— Kristen Nygaard

Pisanie programu polega na stopniowym modyfikowaniu swojego wyobrażenia na temat tego, co się chce zrobić i jak się chce to wyrazić. W tym i następnym rozdziale zbudujemy program. Zaczniemy od pierwszego mglistego pomysłu, przejdziemy etapy analizy, projektowania, implementacji, testowania, ponownego projektowania, na ponownej implementacji kończąc. Chcemy pokazać proces myślowy, który ma miejsce podczas tworzenia oprogramowania. W międzyczasie omówimy organizację programu, typy definiowane przez użytkownika oraz techniki przetwarzania danych wejściowych.

## 6.1. Problem

Pisanie programu zaczyna się od postawienia problemu. To znaczy, jest problem, do rozwiązania którego chcemy napisać program. Aby ten program był dobry, należy dokładnie zrozumieć problem. Jeśli program będzie rozwiązywał nie ten problem, co trzeba, to nie będzie przydatny bez względu na to, jak może być elegancki. Czasami zdarzają się szczęśliwe przypadki, że program spełnia jakieś przypadkowe pożyteczne zadanie, mimo że został napisany w całkiem innym celu. Lepiej jednak nie liczyć na takie szczęście. My chcemy napisać taki program, który będzie w prosty i jasny sposób rozwiązywał dokładnie ten problem, dla którego został napisany.

Jaki program byłoby najlepiej napisać na tym etapie nauki? Taki, który:

- Ilustruje techniki projektowania i pisania programów.
- Umożliwia zapoznanie się z charakterem decyzji, które programista musi podejmować, oraz implikacjami, które te decyzje pociągają.
- Nie wymaga zastosowania zbyt wielu nowych konstrukcji programistycznych.
- Jest wystarczająco skomplikowany, aby zmusić nas do przemyślenia jego projektu.
- Można napisać na kilka sposobów.
- Rozwiązuje łatwy do zrozumienia problem.
- Rozwiązuje problem wart uwagi.
- Jest na tyle mały, że można go w całości przedstawić i zrozumieć.

Wybór padł na program „zmuszający komputer do wykonywania typowych działań arytmetycznych na wyrażeniach, które mu podamy" — tzn. chcemy napisać prosty kalkulator. Programy tego typu są użyteczne. Można je znaleźć w każdym komputerze biurkowym, a nawet można znaleźć takie komputery, które pozwalają uruchamiać tylko tego typu programy — kalkulatory kieszonkowe.

Jeśli np. wpiszemy

```
2+3.1*4
```

będziemy oczekiwać, że program zwróci wynik

```
14.4
```

Niestety kalkulator taki nie będzie miał żadnych funkcji, których już nie spełnia nasz komputer, ale to by było zbyt wysokie wymaganie wobec pierwszego programu.

## 6.2. Przemyślenie problemu

Od czego więc zacząć? Pomyśl o postawionym problemie i tym, jak go rozwiązać. Po pierwsze zdecyduj się, co program ma robić i w jaki sposób będziesz się z nim komunikować. Później będziesz mógł zastanowić się nad tym, jak napisać odpowiedni kod. Opisz wstępną wersję swojego rozwiązania i zastanów się, co należy poprawić. Możesz spróbować przedyskutować problem i jego rozwiązanie z kolegą lub koleżanką. Próba przedstawienia komuś własnych myśli jest doskonałym sposobem na znalezienie słabych punktów własnego rozumowania,

nawet lepszym niż napisanie ich. Papier (lub komputer) nie będzie z Tobą dyskutować i krytycznie oceniać Twoich poglądów. Idealny etap projektowania to taki, który przeprowadza się w towarzystwie.

Niestety nie ma takiej ogólnej strategii rozwiązywania problemów, która odpowiadałaby wszystkim ludziom i pozwalała rozwiązać każdy problem. Istnieje mnóstwo książek, których autorzy twierdzą, że pomogą Ci efektywniej rozwiązywać problemy, oraz cała masa publikacji na temat projektowania programów. Nasza droga nie wiedzie poprzez nie. W zamian opiszemy garść ogólnych wskazówek, które mogą być pomocne w rozwiązywaniu mniejszych problemów. Następnie szybko przejdziemy do wypróbowywania tych sugestii na naszym małym kalkulatorze.

Czytając treść naszych rozważań na temat kalkulatora, oceniaj to, co widzisz, bardzo sceptycznym okiem. Aby zachować realizm, przedstawimy ewolucję naszego programu poprzez szereg wersji. Opiszemy argumenty, które doprowadziły nas do powstania każdej z nich. Oczywiście znaczna część tej argumentacji musi być niekompletna lub błędna, inaczej szybko byśmy skończyli ten rozdział. Naszym celem jest pokazanie przykładowych problemów i procesów myślowych charakterystycznych dla procesu projektowania i implementowania programu. Wersja, z której jesteśmy ostatecznie zadowoleni, zostanie przedstawiona dopiero na końcu następnego rozdziału.

Pamiętaj, że w tym i następnym rozdziale proces dochodzenia do finalnej wersji programu — droga wiodąca przez niepełne rozwiązania, pomysły i błędy — jest co najmniej tak samo ważny, jak wersja ostateczna i ważniejszy niż narzędzia techniczne języka programowania, których będziemy używać (wrócimy do nich później).

## 6.2.1. Etapy rozwoju oprogramowania

Poniżej przedstawimy nieco terminologii związanej z tworzeniem oprogramowania. Pracując nad rozwiązaniem problemu, wielokrotnie powtarza się następujące fazy:

- *Analiza* — przemyśl, co masz zrobić, i opisz, jak to aktualnie rozumiesz. Taki opis nazywa się **zestawem wymagań** lub **specyfikacją**. Nie będziemy szczegółowo opisywać technik opracowywania i opisywania takich wymogów. Temat tej książki tego nie obejmuje, ale należy pamiętać, że w miarę zwiększania się rozmiaru problemu techniki te nabierają wagi.
- *Projektowanie* — utworzenie ogólnej struktury systemu i podjęcie decyzji, na jakie części podzielić implementację oraz jak powinny się one ze sobą komunikować. W ramach projektowania zastanów się jakie narzędzia — np. biblioteki — możesz wykorzystać do opracowania struktury programu.
- *Implementacja* — napisz kod, usuń błędy oraz sprawdź za pomocą testów, czy robi to, co powinien.

## 6.2.2. Strategia

Oto garść wskazówek, które — jeśli zostaną zastosowane z rozwagą i wyobraźnią — będą pomocne w wielu projektach programistycznych:

- Jaki problem jest do rozwiązania? Przede wszystkim należy starać się dokładnie opisać, co chce się zrobić. Najczęściej oznacza to sporządzenie opisu problemu lub, jeśli ktoś

inny dał nam zadanie do wykonania, próbę odszyfrowania, co dokładnie to oznacza. W tym momencie należy przyjąć punkt widzenia użytkownika (nie programisty czy implementatora). To znaczy, zamiast zastanawiać się, jak rozwiązać problem, pomyśl, co program ma w ogóle robić. Zadawaj pytania typu: „Co ten program może dla mnie zrobić?" albo „W jaki sposób chciałbym komunikować się z tym programem?". Pamiętaj, że większość z nas może korzystać z własnego bogatego doświadczenia jako użytkownika komputerów.

- Czy problem został jasno nakreślony? W realnym świecie nigdy nie jest. Nawet w takim ćwiczeniu trudno jest opisać go wystarczająco precyzyjnie. Dlatego staramy się wszystko wyjaśnić. Szkoda by było, gdybyśmy rozwiązali nie ten problem, co trzeba. Inna pułapka to wygórowane wymagania. Obmyślając, co byśmy chcieli, łatwo możemy ulec chciwości lub nadmiernym ambicjom. Zawsze lepiej jest obniżyć wymagania, aby ułatwić napisanie specyfikacji programu, jego zrozumienie, użytkowanie oraz (taką trzeba mieć nadzieję) implementację. Gdy zadziała, zawsze można napisać wzbogaconą wersję 2.0.

- Czy problem wydaje się możliwy do rozwiązania w przewidzianym czasie oraz przy określonym zasobie umiejętności i dostępnych narzędziach? Nie ma sensu rozpoczynać projektu, którego nie mamy szans ukończyć. Jeśli jest za mało czasu na implementację (włącznie z testowaniem) wszystkich wymaganych funkcji programu, zazwyczaj lepiej jest go w ogóle nie zaczynać. Lepiej zamiast tego zdobyć więcej zasobów (zwłaszcza czasu) lub (idealnie) zmienić wymagania, aby ułatwić zadanie.

- Spróbuj podzielić program na dające się ogarnąć myślami części. Nawet najmniejszy program rozwiązujący realny problem można podzielić na części.

  - Znasz jakieś narzędzia, biblioteki itp., które mogą być pomocne? Odpowiedź prawie zawsze brzmi: tak. Od samego początku nauki programowania masz do dyspozycji zawartość standardowej biblioteki C++. Później poznasz znaczną jej część i dowiesz się, jak poszukiwać jeszcze więcej. Będziesz korzystać z bibliotek graficznych, macierzy itp. Mając odrobinę doświadczenia, będziesz w stanie znaleźć tysiące bibliotek w internecie. Pamiętaj — nie ma sensu wyważać otwartych drzwi, jeśli tworzy się oprogramowanie do realnego użytku. Co innego w czasie nauki programowania. Wówczas takie działania w celu sprawdzenia, jak to zrobili inni, mają głęboki sens. Cały czas, który oszczędzisz dzięki wykorzystaniu istniejących bibliotek, możesz poświęcić na pracę nad innymi częściami problemu albo na odpoczynek. Skąd wiadomo, czy dana biblioteka spełnia nasze wymagania i prezentuje odpowiednią jakość? To trudne pytanie. Można spytać znajomych, popytać na grupach dyskusyjnych i wypróbować kilka krótkich przykładów, zanim się zdecydujemy.

  - Wyodrębnij takie części rozwiązania, które można opisać oddzielnie od reszty (i potencjalnie wykorzystać w kilku miejscach programu, a nawet w innych programach). Umiejętność znajdowania takich części przychodzi z doświadczeniem, dlatego w książce tej przedstawiamy wiele takich przykładów. Używaliśmy już wektorów, łańcuchów i strumieni (`cin` i `cout`). W tym rozdziale po raz pierwszy przedstawimy przykłady projektów, implementacji i wykorzystania części programów dostępnych jako typy

zdefiniowane przez użytkownika (`Token` i `Token_stream`). Jeszcze więcej takich przykładów znajduje się w rozdziałach 8. oraz 13. – 15., w których zostaną opisane techniki ich tworzenia. Na razie zastanów się nad taką analogią — gdybyśmy projektowali samochód, zaczęlibyśmy od opisu jego podzespołów, takich jak koła, silnik, fotele, klamki do drzwi itp. Każdą z nich wyprodukowalibyśmy osobno, aby następnie użyć jej do złożenia całego pojazdu. W nowoczesnym samochodzie wykorzystuje się dziesiątki tysięcy takich części. Realne programy nie różnią się w niczym od samochodów pod tym względem (poza tym, że częściami są fragmenty kodu). Nie przyszło by nam do głowy budować samochodu z surowych materiałów, jak żelazo, plastik i drewno. Analogicznie nie tworzymy poważnych programów, bezpośrednio używając wyrażeń, instrukcji typów wbudowanych w język. Projektowanie i implementowanie takich części jest najważniejszym tematem tej książki, a nawet w ogólne programowania. Zajrzyj też do rozdziałów 9. (typy definiowane przez użytkownika), 14. (hierarchie klas) oraz 20. (typy ogólne).

- Zbuduj niewielką wersję programu z ograniczoną funkcjonalnością, która rozwiązuje najważniejszą część problemu. Rzadko się zdarza, abyśmy od samego początku dokładnie znali charakter problemu. Często nam się tak wydaje (chyba wiadomo, co to jest program kalkulator?), ale w rzeczywistości jest inaczej. Tylko dzięki wytężonemu myśleniu o problemie (analiza) i eksperymentowaniu (projekt i implementacja) można na tyle dokładnie zrozumieć problem, aby napisać dobry program. Dlatego tworzymy ograniczoną wersję, aby:
  - Odkryć własne braki w rozumowaniu, pojmowaniu problemu oraz narzędziach.
  - Sprawdzić, czy nie trzeba zmienić niektórych szczegółów w specyfikacji problemu, aby zadanie było wykonalne. Rzadko się zdarza, aby w czasie analizy i we wstępnej fazie projektowania udało się przewidzieć wszystkie możliwe trudności. Należy skorzystać z informacji, które zyskujemy podczas pisania i testowania kodu.

  Taką wstępną wersję o ograniczonej funkcjonalności czasami nazywa się **prototypem**. Jeśli (co jest prawdopodobne) ta pierwsza wersja nie działa lub jest tak brzydka, że nie mamy ochoty się nią dłużej zajmować, wyrzucamy ją i tworzymy nowy prototyp, tym razem wzbogaceni o nowe doświadczenia. Powtarzamy ten proces aż do uzyskania zadowalającego wyniku. Nie kontynuuj pracy w bałaganie, który z czasem tylko się pogorszy.
- Zbuduj pełną wersję, najlepiej wykorzystując w tym celu części z wersji wstępnej. Chodzi o to, aby program tworzyć z działających części, a nie pisać cały kod na raz. Można oczywiście mieć nadzieję, że jakimś cudem nieprzetestowane fragmenty kodu będą działać i na dodatek robić to, co zaplanowano.

## 6.3. Wracając do kalkulatora

W jaki sposób będziemy komunikować się z kalkulatorem? To łatwe — wiemy już, jak posługiwać się strumieniami `cin` i `cout`, a graficzne interfejsy użytkownika (GUI) zostaną opisane dopiero w rozdziale 16. W związku z tym wybór padł na okno konsoli. Program będzie pobierał z klawiatury wyrażenia, obliczał ich wartość i drukował wynik na ekranie. Na przykład:

```
Wyrażenie: 2+2
Wynik: 4
Wyrażenie: 2+2*3
Wynik: 8
Wyrażenie: 2+3–25/5
Wynik: 0
```

Wyrażenia, tzn. `2+2` i `2+2*3`, powinien wpisywać użytkownik. Reszta należy do programu. Wyświetlenie słowa *Wyrażenie:* będzie zachętą dla użytkownika do wpisania wyrażenia. Moglibyśmy napisać *Proszę wpisać wyrażenie i znak nowego wiersza:*, ale to wydawało nam się zbyt rozwlekłe. Z drugiej strony taki przyjemny znaczek > byłby chyba za bardzo tajemniczy. Takie szkicowanie przykładów użycia we wczesnej fazie pracy jest bardzo ważne. Dzięki temu można się dowiedzieć, jaki jest minimalny zestaw funkcji programu. W projektowaniu i analizie przykłady takie nazywają się **przypadkami użycia**.

Większość ludzi, którzy po raz pierwszy stykają się z problemem kalkulatora, wpada na następujący pomysł, jeśli chodzi o główną logikę programu:

```
wczytaj_wiersz
oblicz      // wykonuje pracę
wydrukuj_wynik
```

Takie zapiski to oczywiście nie jest prawdziwy kod, tylko tzw. *pseudokod*. Stosuje się go we wczesnych fazach projektowania, gdy nie ma jeszcze pewności co do tego, jaką zastosować notację. Np., czy `obliczenia` ma być wywołaniem funkcji? Jeśli tak, to jakie będzie przyjmować argumenty? Jest po prostu za wcześnie na zadawanie takich pytań.

## 6.3.1. Pierwsza próba

Na tym etapie nie jesteśmy jeszcze gotowi napisać programu kalkulatora. Nie przemyśleliśmy jeszcze wszystkiego, ale myślenie to ciężka praca i — jak większość programistów — nie możemy się doczekać, żeby już coś napisać. Spróbujemy więc swoich sił i napiszemy prosty kalkulator, aby zobaczyć, do czego nas to doprowadzi. Nasz pierwszy pomysł wygląda tak:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Wpisz wyrażenie (obsługujemy operatory + i –): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin>>lval>>op>>rval; // Wczytuje coś w rodzaju 1 + 3.
    if (op=='+')
        res = lval + rval; // dodawanie
    else if (op=='–')
        res = lval – rval; // odejmowanie
    cout << "Wynik: " << res << '\n';
    keep_window_open();
    return 0;
}
```

Wczytujemy parę wartości oddzielonych operatorem, np. `2+2`, obliczamy wynik (tu `4`) i drukujemy go na ekranie. Zmienną przechowującą wartość z lewej strony operatora nazwaliśmy `lval`, a z prawej strony `rval`.

To nawet działa! Co z tego, że program nie jest ukończony? To wspaniałe uczucie zrobić coś, co działa! Może to programowanie i informatyka są łatwiejsze, niż głoszą plotki? Możliwe, ale nie dajmy się ponieść emocjom z powodu tego pierwszego sukcesu. Oto lista czynności:

1. Oczyścić kod.
2. Dodać obsługę dzielenia i mnożenia (np. **2*3**).
3. Dodać obsługę wyrażeń zawierających więcej niż jeden operand (np. **1+2+3**).

W szczególności pamiętamy, że zawsze należy sprawdzać, czy użytkownik podał sensowne dane (zapomnieliśmy z pośpiechu wcześniej), oraz że porównywanie jednej wartości z wieloma stałymi lepiej wykonać za pomocą instrukcji `switch` niż `if`.

Łączenie działań w łańcuchy, np. `1+2+3+4`, obsłużymy, sumując wartości w czasie wczytywania. Tzn. wczytujemy `1`, widzimy `+2`, więc dodajemy `2` do `1` (uzyskując w ten sposób wynik `3`). Dalej widzimy `+3`, a więc dodajemy `3` do poprzedniego wyniku itd. Po kilku falstartach i poprawieniu kilku błędów składni uzyskaliśmy następujący rezultat:

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Wpisz wyrażenie (obsługujemy operatory +, –, * oraz /): ";
    int lval = 0;
    int rval;
    char op;
    cin>>lval; // Wczytywanie pierwszego argumentu wyrażenia z lewej.
    if (!cin) error("Na początku nie ma argumentu wyrażenia.");
    while (cin>>op) { // Wczytywanie operatora i prawego argumentu wyrażenia na zmianę.
        cin>>rval;
        if (!cin) error("Nie ma drugiego argumentu wyrażenia.");
        switch(op) {
        case '+':
            lval += rval; // Dodawanie: lval = lval + rval
            break;
        case '–':
            lval –= rval; // Odejmowanie: lval = lval – rval
            break;
        case '*':
            lval *= rval; // Mnożenie: lval = lval · rval
            break;
        case '/':
            lval /= rval; // Dzielenie: lval = lval / rval
            break;
        default: // Koniec operatorów — drukowanie wyniku.
            cout << "Wynik: " << lval << '\n';
            keep_window_open();
            return 0;
        }
```

```
        }
        error("Nieprawidłowe wyrażenie.");
    }
```

Wygląda nieźle, ale gdy wpiszemy wyrażenie `1+2*3`, to ujrzymy wynik `9` zamiast `7`, którego spodziewalibyśmy się na podstawie wiedzy zdobytej w szkole podstawowej. Analogicznie wynikiem wyrażenia `1-2*3` będzie `-3` zamiast spodziewanego `-5`. Kalkulator wykonuje działania w złej kolejności — wyrażenie `1+2*3` jest liczone jako `(1+2)*3` zamiast `1+(2*3)`. Analogicznie `1-2*3` jest liczone jako `(1-2)*3` zamiast `1-(2*3)`. Lipa! Moglibyśmy uznać, że zasada, iż „mnożenie wiąże mocniej niż dodawanie" jest głupią i przestarzałą konwencją, ale nie możemy zignorować wielowiekowej tradycji, aby ułatwić sobie programowanie.

## 6.3.2. Tokeny

Musimy zatem znaleźć sposób na wczytywanie części wiersza „na zapas", aby sprawdzić, czy nie ma tam gdzieś operatora * (albo /). Jeśli jest, musimy zmienić kolejność wykonywania działań. Niestety próbując wczytać nieco danych z wyprzedzeniem, napotkamy kilka trudności:

1. Nie wymagamy, aby wyrażenie znajdowało się w jednym wierszu. Na przykład poniższe też jest poprawne:

   ```
   1
   +
   2
   ```

2. Jak znaleźć znaki * i / wśród cyfr i plusów w kilku wierszach danych wejściowych?

3. Jak zapamiętać, gdzie znajdował się znaleziony znak *?

4. Jak wykonać obliczenia, które nie są ściśle typu „od lewej do prawej"?

Postanowiliśmy być wielkimi optymistami i zająć się tylko punktami 1 – 3. Ostatnim zajmiemy się trochę później.

Poszukamy pomocy. Przecież ktoś na pewno zna typowy sposób wczytywania danych typu liczby i operatory i zapisywania ich w taki sposób, aby można je było łatwo wykorzystać w obliczeniach. Ta konwencjonalna i przydatna technika nazywa się rozbiorem na składniki, czyli tokeny (ang. *tokenize*) — wczytuje się dane i dzieli je na **tokeny**. Na przykład wyrażenie

```
45+11.5/7
```

zostałoby rozłożone na tokeny

```
45
+
11.5
/
7
```

**Token** to sekwencja znaków, która reprezentuje pewną całość, np. liczbę lub operator. Kompilator C++ dzieli na tokeny kod źródłowy. W istocie różne formy rozkładu na czynniki są podstawą analizy większości rodzajów tekstów. W wyrażeniach matematycznych w języku C++ potrzebujemy trzech rodzajów tokenów:

- literały zmiennoprzecinkowe — zgodnie z definicją w C++, np. `3.14`, `0.274e2` i `42`;
- operatory — `+`, `-`, `*`, `/` oraz `%`;
- nawiasy — `(` i `)`.

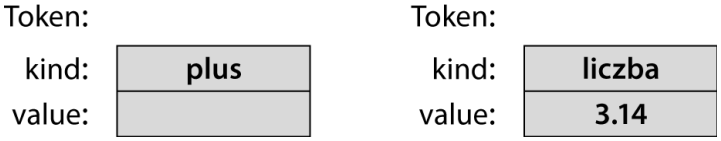
Wydaje się, że trudności mogą nastręczać literały zmiennoprzecinkowe. Wczytanie liczby `12` wydaje się znacznie łatwiejsze niż `12.3e-3`. Ale kalkulatory zazwyczaj wykonują działania na liczbach zmiennoprzecinkowych. Analogicznie podejrzewamy, że aby nasz kalkulator był przydatny, musi obsługiwać nawiasy.

W jaki sposób reprezentuje się takie tokeny w programie? Można spróbować zapamiętywać, gdzie każdy token się zaczyna, a gdzie kończy, ale to może być uciążliwe (zwłaszcza jeśli pozwolimy na wpisywanie wyrażeń obejmujących więcej niż jeden wiersz). Dodatkowo, jeśli będziemy zapisywać wartości jako sekwencje znaków, będziemy musieli później znaleźć sposób na odczytanie tych wartości. To znaczy, jeśli liczbę `42` zapiszemy jako znaki `4` i `2`, później będziemy musieli odgadnąć, że te dwa znaki reprezentują liczbę `42` (tzn. `4*10+2`). Oczywistym i konwencjonalnym rozwiązaniem tego problemu jest przedstawienie każdego tokenu jako pary (**rodzaj**, **wartość**). Pierwszy element informuje o rodzaju tokenu — liczba, operator, nawias. Drugi natomiast np. w przypadku liczb określa dokładną wartość.

Jak więc wykorzystać pomysł par (**rodzaj**, **wartość**) w kodzie? Zdefiniujemy typ `Token` do reprezentowania tokenów. Po co? Przypomnij sobie, po co używamy typów: przechowują potrzebne nam dane i pozwalają wykonywać na nich różne operacje. Na przykład typ `int` pozwala przechowywać liczby całkowite i umożliwia dodawanie, odejmowanie, mnożenie oraz dzielenie tych liczb. Natomiast typ `string` przechowuje łańcuchy znaków i pozwala je np. łączyć. W języku C++ i jego bibliotece standardowej dostępnych jest wiele typów, np. `char`, `int`, `double`, `string`, `vector` i `ostream`. Nie ma jednak typu `Token`. W istocie można wymienić mnóstwo typów — tysiące, a nawet dziesiątki tysięcy — które chcielibyśmy mieć do dyspozycji, a których nie ma w języku ani jego bibliotece standardowej. Do naszych ulubionych typów, które nie są standardowo dostępne, należą `Matrix` (zobacz rozdział 24.), `Date` (zobacz rozdział 9.) oraz reprezentujące go liczby całkowite nieskończonej precyzji (poszukaj w internecie informacji na temat typu `Bignum`). Jeśli przemyślisz to, dojdziesz do wniosku, że język nie może standardowo obsługiwać dziesiątek tysięcy typów — kto by je zdefiniował i zaimplementował, kto by je potem znalazł, nie mówiąc już o tym, jak gruby musiałby być podręcznik do nauki takiego języka. Język C++ wzorem innych nowoczesnych języków programowania rozwiązuje ten problem, pozwalając użytkownikowi definiować własne (niestandardowe) typy (ang. *user-defined type* — typ zdefiniowany przez użytkownika).

## 6.3.3. Implementowanie tokenów

Jak powinien wyglądać token? To znaczy, jakie właściwości powinien mieć nasz typ `Token`? Musi nadawać się do reprezentowania operatorów (np. `+` i `-`) i wartości liczbowych (np. `42` i `3.14`). Oczywistym rozwiązaniem jest zaimplementowanie czegoś takiego, co może zawierać informację na temat rodzaju tokenu i w razie potrzeby jego wartość:



Pomysł ten można zaimplementować w języku C++ na wiele sposobów. Przedstawiamy najprostszy, który wydaje nam się użyteczny:

```
class Token { // Bardzo prosty typ zdefiniowany przez użytkownika.
public:
    char kind;
    double value;
};
```

`Token` to typ (tak samo jak `int` czy `char`), a więc można go używać do definiowania zmiennych i przechowywania wartości. Składa się z dwóch części (nazywanych składowymi) — `kind` (rodzaj) oraz `value` (wartość). Słowo kluczowe `class` oznacza „typ zdefiniowany przez użytkownika". Wskazuje definicję typu z zerem lub większą liczbą składowych. Pierwsza składowa o nazwie `kind` jest znakiem `char`, a więc można jej użyć do przechowywania znaków `'+'` i `'*'`, które będą reprezentowały operatory. Przy użyciu tego typu można tworzyć następujące instrukcje:

```
Token t;          // Zmienna t jest typu Token.
t.kind = '+';     // Zmienna t reprezentuje znak +.
Token t2;         // Zmienna t2 jest innym obiektem typu Token.
t2.kind = '8';    // Cyfra 8 oznacza rodzaj (kind) tokenu będący liczbą.
t2.value = 3.14;
```

Aby uzyskać dostęp do składowej, posługujemy się odpowiednią notacją — **nazwa_obiektu.↪nazwa_składowej**. Tekst `t.kind` można przeczytać jako „rodzaj obiektu `t`", a `t2.value` jako „wartość obiektu `t2`". Obiekty typu `Token` można kopiować tak samo jak typu `int`:

```
Token tt = t;     // Inicjacja kopii
if (tt.kind != t.kind) error("To niemożliwe!");
t = t2;           // przypisanie
cout << t.value; // wydrukuje 3.14
```

Mając typ `Token`, wyrażenie `(1.5+4)*11` można przedstawić za pomocą siedmiu tokenów:

| '(' | '8' | '+' | '8' | ')' | '*' | '8' |
|---|---|---|---|---|---|---|
| | 1.5 | | 4 | | | 11 |

Należy zauważyć, że proste tokeny, jak `+`, nie mają wartości, a więc do ich reprezentowania niepotrzebna jest składowa `value`. Potrzebowaliśmy znaku, który oznaczałby liczbę. Wybór padł na `'8'`, ponieważ nie jest to operator ani znak interpunkcyjny. Wykorzystanie `'8'` w taki sposób jest dosyć tajemnicze, ale na razie może być.

`Token` jest przykładem typu zdefiniowanego przez użytkownika w języku C++. Typy takie poza danymi składowymi mogą też zawierać funkcje (operacje) składowe. Istnieje wiele powodów, dla których się je definiuje. My zdefiniujemy tylko dwie, aby ułatwić sobie inicjowanie obiektów typu `Token`:

```
class Token {
public:
    char kind;      // Rodzaj tokenu
```

```
    double value;  // Dla liczb: wartość.
    Token(char ch) // Tworzy Token ze znaku.
        :kind(ch), value(0) { }
    Token(char ch, double val) // Tworzy Token ze znaku i wartości typu double.
        :kind(ch), value(val) { }
};
```

Nie są to zwykłe funkcje składowe, tylko tzw. **konstruktory**. Mają taką samą nazwę jak ich typ i służą do inicjalizowania (tworzenia) obiektów typu `Token`. Na przykład:

```
Token t1('+');        // Inicjacja zmiennej t1 — t1.kind = '+'.
Token t2('8',11.5); // Inicjacja zmiennej t2 — t2.kind = '8' i t2.value = 11.5.
```

Tekst `:kind(ch), value(0)` w pierwszym konstruktorze oznacza: „Zainicjuj składową `kind` wartością `ch`, a `value` ustaw na `0`". W drugim konstruktorze znajduje się tekst `:kind(ch), value(val)`, który oznacza: „Zainicjuj składową `kind` wartością `ch`, a `value` ustaw na `val`".W obu przypadkach nie trzeba robić nic więcej, aby utworzyć obiekt typu `Token`, dlatego treść funkcji jest pusta — `{ }`. Specjalna składnia inicjująca (ang. *member initializer list* — **lista wartości inicjujących składowe**), która zaczyna się dwukropkiem, jest używana tylko w konstruktorach.

Zauważ, że konstruktor nie zwraca wartości. Dlatego nie trzeba (a nawet nie można) definiować typu zwrotnego konstruktora. Więcej na temat konstruktorów napiszemy w rozdziałach 9.4.2 i 9.7.

## 6.3.4. Używanie tokenów

Może spróbujemy dokończyć nasz kalkulator! Chociaż warto by było najpierw opracować jakiś plan działania. Jak będziemy wykorzystywać obiekty typu `Token` w programie? Możemy wczytać dane wejściowe do wektora takich obiektów:

```
Token get_token(); // Wczytuje token ze strumienia cin.
vector<Token> tok; // Tutaj będziemy zapisywać tokeny.
int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}
```

Teraz możemy wczytać wyrażenie w całości i dopiero po tym obliczyć jego wartość. W przypadku `11*12` otrzymamy coś takiego:

| '8' | '*' | '8' |
|---|---|---|
| 11 | | 12 |

W strukturze tej możemy znaleźć operator i jego operandy. Gdy to zrobimy, z łatwością wykonamy działanie mnożenia, ponieważ liczby `11` i `12` zostały zapisane jako wartości liczbowe, a nie łańcuchy.

Teraz przeanalizujemy bardziej skomplikowane wyrażenie. Dla wyrażenia `1+2*3` `tok` będzie zawierać pięć obiektów typu `Token`:

| '8' | '+' | '8' | '*' | '8' |
|---|---|---|---|---|
| 1 | | 2 | | 3 |

W tym przypadku operator mnożenia można znaleźć za pomocą prostej pętli:

```
for (int i = 0; i<tok.size(); ++i) {
    if (tok[i].kind=='*') { // Znaleźliśmy operator mnożenia!
        double d = tok[i–1].value*tok[i+1].value;
        // Co teraz?
    }
}
```

No dobrze, ale co teraz? Co zrobimy z iloczynem `d`? Jak określić kolejność wykonywania działań w wyrażeniu? Operator + znajduje się przed *, a więc nie możemy po prostu wykonać wszystkich działań od lewej do prawej. Możemy spróbować od prawej do lewej! To by się sprawdziło w przypadku wyrażenia `1+2*3`, ale już nie dla wyrażenia `1*2+3`. Nie wspominając już o takim czymś, jak `1+2*3+4`. To wyrażenie trzeba obliczyć „od środka" — `1+(2*3)+4`. Jak poradzimy sobie z nawiasami? Wydaje się, że utknęliśmy. Musimy się wycofać, przestać na chwilę programować i pomyśleć nad wczytywaniem oraz tym, jak rozumiemy łańcuch wejściowy i jak obliczamy jego wartość jako wyrażenia.

Pierwsza entuzjastyczna próba rozwiązania problemu (napisania kalkulatora) zakończyła się klapą. Pierwsze podejścia często tak się kończą i sytuacje takie są nam potrzebne, ponieważ dzięki nim możemy lepiej zrozumieć naturę problemu. W tym przypadku nawet poznaliśmy przydatne pojęcie tokenu, które samo jest przykładem pojęcia pary (**nazwa**,**wartość**), z którym będziemy jeszcze wielokrotnie się spotykać. Musisz jednak pamiętać, że takie bezmyślne i nieplanowane pisanie kodu nie powinno zajmować zbyt dużo czasu. Przed dokonaniem analizy (próbą zrozumienia problemu) i opracowaniem projektu (opracowaniem ogólnej struktury rozwiązania) powinno się napisać bardzo mało kodu.

### WYPRÓBUJ

Z drugiej strony, czemu nie moglibyśmy znaleźć prostego rozwiązania tego problemu? Nie wydaje się aż taki trudny. Podjęcie próby rozwiązania problemu, nawet zakończone fiaskiem, może doprowadzić nas do lepszego zrozumienia problemu i opracowania rozwiązania. Pomyśl, co możesz zrobić od razu. Jako przykład niech posłuży wyrażenie `12.5+2`. Możemy rozbić je na tokeny, dojść do wniosku, że to bardzo proste wyrażenie i obliczyć wynik. Może byłoby trochę bałaganu, ale rozwiązanie jest proste, więc może warto podążyć w tym kierunku i znaleźć coś, co wystarczy! Pomyśl, co byś zrobił, gdybyś znalazł operatory + i * w wierszu `2+3*4`. To także można obliczyć „na piechotę". Jak byśmy sobie poradzili ze skomplikowanym wyrażeniem typu `1+2*3/4%5+(6-7*(8))`? Jak byśmy radzili sobie z błędami, np. `2+*3` albo `2&3`? Pomyśl nad tym przez chwilę. Możesz zrobić sobie notatki na kartce, nakreśl możliwe rozwiązania oraz wypisz interesujące lub ważne wyrażenia.

### 6.3.5. Powrót do tablicy

Jeszcze raz przeanalizujemy problem, tym razem starając się nie wyrywać z nieprzemyślanymi pomysłami. Jedyne, co odkryliśmy, to fakt, że obliczenie przez program tylko jednego wyrażenia sprawia nam trudności. Chcielibyśmy mieć możliwość obliczenia wielu wyrażeń w jednym uruchomieniu programu. W związku z tym wzbogacamy nasz pseudokod w następujący sposób:

```
while (nie_skończone) {
    wczytaj_wiersz
    oblicz        // wykonaj pracę
    wydrukuj_wynik
}
```

To z pewnością komplikuje sprawę, ale musimy wziąć pod uwagę fakt, że kalkulatorów zazwyczaj używa się do wykonywania kilku obliczeń po kolei. Czy mamy kazać użytkownikowi uruchamiać nasz program ponownie, aby wykonać każde obliczenie? Moglibyśmy, ale w wielu nowoczesnych systemach operacyjnych uruchamianie programów trwa za długo, a więc lepiej tego nie robić.

Kiedy patrzymy na nasz pseudokod, nasze początkowe próby rozwiązania problemu i przykłady użycia, nasuwa się nam kilka pytań (i kilka nieśmiałych odpowiedzi):

1. Jeśli użytkownik wpisze `45+5/7`, jak znajdziemy poszczególne elementy — `45`, `5`, `/` i `7`? Odpowiedź: podzielimy na tokeny!
2. W jaki sposób oznaczymy koniec wyrażenia? Oczywiście znakiem nowego wiersza (zawsze podejrzliwie traktuj zwroty typu „oczywiście” — „oczywiście” to nie żaden powód!
3. Jak zaprezentujemy wyrażenie `45+5/7` jako dane, aby można było obliczyć wynik? Przed wykonaniem dodawania musimy w jakiś sposób zamienić znaki `4` i `5` w liczbę całkowitą `45` (tj. `4*10+5`). Zatem podział na tokeny jest częścią rozwiązania.
4. Jak sprawić, aby wyrażenie `45+5/7` było obliczane jako `45+(5/7)`, a nie `(45+5)/7`?
5. Ile wynosi `5/7`? Około `.71`, a więc to nie jest liczba całkowita. Z doświadczenia wiemy, że użytkownicy kalkulatorów oczekują wyników zmiennoprzecinkowych. Czy powinniśmy pozwolić na wpisywanie liczb zmiennoprzecinkowych? Oczywiście!
6. Czy możemy pozwolić na używanie zmiennych? Moglibyśmy na przykład napisać:

   ```
   v=7
   m=9
   v*m
   ```

   Dobry pomysł, ale zostawimy to na później. Na razie zajmiemy się podstawową funkcjonalnością.

Najważniejsza decyzja to prawdopodobnie odpowiedź na pytanie w punkcie 6. W rozdziale 7.8 zobaczysz, że odpowiedź ta pociągnie za sobą prawie podwojenie rozmiaru wstępnej wersji projektu. To podwoiłoby czas potrzebny na uruchomienie wstępnej wersji programu. Podejrzewamy, że początkujący potrzebowałby nawet cztery razy więcej czasu i niewykluczone, że straciłby w końcu cierpliwość. We wczesnych fazach prac nad projektem należy zawsze unikać

przesady z liczbą funkcji. Wstępna wersja zawsze powinna być prosta i zawierać tylko najważniejsze funkcje. Kiedy uda Ci się zmusić coś do działania, możesz postawić sobie bardziej ambitne wymagania. Budowa programu etapami jest znacznie łatwiejsza niż wszystkiego na raz. Odpowiedź „tak" na pytanie 6. miałaby jeszcze jeden zły wynik: mogłoby być trudno oprzeć się pokusie dodania jeszcze innych funkcji. Może warto pomyśleć o funkcjach matematycznych albo o pętlach? Gdy zacznie się dodawać kolejne „fajne" funkcje, trudno przestać.

Z programistycznego punktu widzenia najbardziej kłopotliwe są punkty 1, 3 i 4. Ponadto są ze sobą powiązane, ponieważ gdy znajdziemy już `45` i `+`, co mamy z nimi zrobić? Tzn., jak zapisać je w programie? Oczywiście częściowym rozwiązaniem tego problemu jest podział na tokeny, ale tylko częściowym.

Co zrobiłby doświadczony programista? Gdy mamy do rozwiązania jakiś trudny techniczny problem, często można znaleźć jakieś standardowe rozwiązanie. Wiemy, że ludzie piszą kalkulatory, przynajmniej od kiedy istnieją komputery przyjmujące dane symboliczne z klawiatury, a więc od 50 lat. Musi być jakieś standardowe rozwiązanie! W takiej sytuacji doświadczony programista konsultuje się z kolegami i przeszukuje dostępną literaturę. Byłoby głupstwem myśleć, że w jeden dzień uda się wymyśleć coś lepszego, niż inni wymyślili przez 50 lat.

## 6.4. Gramatyki

Istnieje standardowa odpowiedź na pytanie, jak rozszyfrować znaczenie wyrażenia: najpierw wprowadzone znaki należy zebrać i podzielić na tokeny (to już sami odkryliśmy). Jeśli użytkownik wpisze:

```
45+11.5/7
```

program powinien utworzyć następującą listę tokenów:

```
45
+
11.5
/
7
```

Token to sekwencja znaków, którą uważamy za jakąś jednostkę, np. operator lub liczbę.

Po utworzeniu tokenów program musi upewniać się, że całe wyrażenie jest poprawnie rozumiane. Na przykład wiemy, że wyrażenie `45+11.5/7` oznacza `45+(11.5/7)`, a nie `(45+11.5)/7`. Sęk w tym, jak nauczyć program tej przydatnej zasady (dzielenie „wiąże mocniej" niż dodawanie)? Standardowa odpowiedź jest taka, że piszemy **gramatykę** definiującą składnię naszych danych wejściowych, a następnie piszemy program, w którym implementujemy zasady tej gramatyki. Na przykład:

```
// Prosta gramatyka wyrażeń:

Expression:
    Term
    Expression "+" Term // dodawanie
    Expression "–" Term // odejmowanie
Term:
    Primary
```

```
        Term "*" Primary // mnożenie
        Term "/" Primary // dzielenie
        Term "%" Primary // reszta z dzielenia (modulo)
    Primary:
        Number
        "(" Expression ")" // grupowanie
    Number:
        floating-point-literal
```

Jest to zestaw prostych zasad. Ostatnią należy czytać następująco: „`Number (liczba)` to literał zmiennoprzecinkowy". Natomiast treść przedostatniej jest taka: „`Primary` (czynnik) jest liczbą lub znakiem `'('`, po którym jest wyrażenie i znak `')'`". Reguły dla `Expression` (wyrażenia) i `Term` (składnika) są podobne. Każda z nich jest zdefiniowana z uwzględnieniem jednej z reguł, które znajdują się dalej.

Jak pamiętamy z podrozdziału 6.3.2, nasze tokeny — zgodnie z definicją w języku C++ — to:

- literał zmiennoprzecinkowy (zgodny z definicją w języku C++, np. `3.14`, `0.274e2` lub `42`);
- `+`, `-`, `*`, `/` oraz `%` — operatory;
- `(` i `)` — nawiasy.

Używając gramatyki i tokenów, zrobiliśmy bardzo duży pojęciowy skok w stosunku do naszego początkowego pseudokodu. Tego rodzaju postępy chcielibyśmy robić zawsze, ale rzadko się to udaje bez pomocy. Do tego właśnie służą doświadczenie, literatura i mentorzy.

Na pierwszy rzut oka gramatyka ta wydaje się bezsensowna. Często tak jest z notacją techniczną. Pamiętaj jednak, że jest to ogólna i elegancka (co w końcu docenisz) notacja do opisu czegoś, co potrafisz robić przynajmniej od czasów szkoły podstawowej. Nie masz problemu z obliczeniem wyrażenia `1-2*3` albo `1+2-3` lub `3*2+4/2`. Potrafisz jednak wyjaśnić, jak to robisz? Umiesz to tak wyjaśnić, aby zrozumiał to nawet ktoś, kto nigdy nie miał styczności z konwencjonalną arytmetyką? Czy Twoje wyjaśnienia będą miały zastosowanie dla wszystkich kombinacji operatorów i argumentów? Aby wystarczająco szczegółowo i precyzyjnie objaśnić coś komputerowi, potrzebna jest odpowiednia notacja — a gramatyka należy do najlepszych konwencjonalnych narzędzi do jej tworzenia.

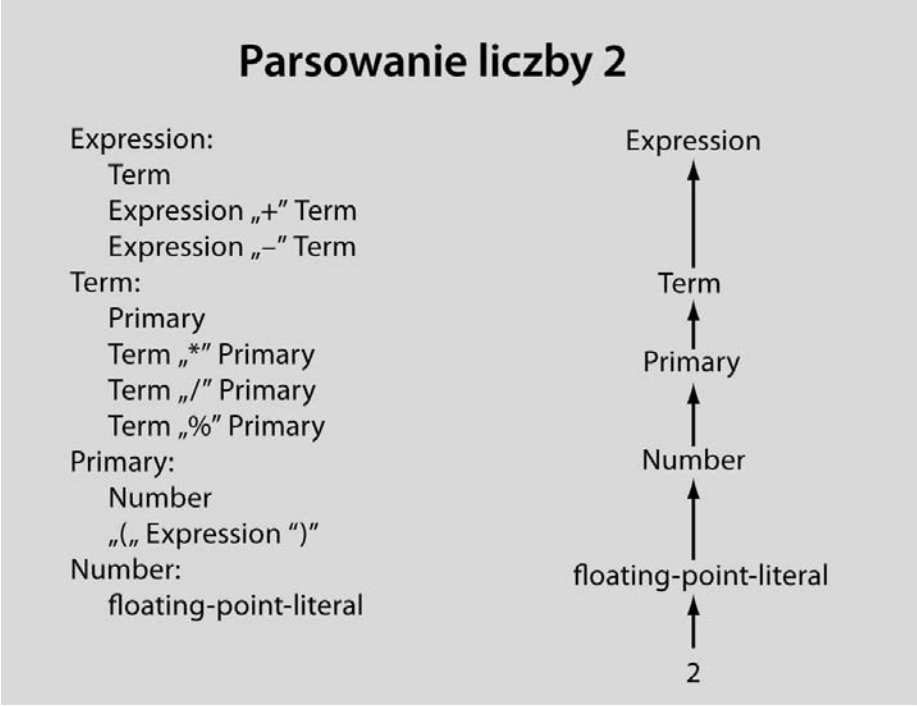
Jak czyta się gramatykę? Mając pewne dane wejściowe, zaczyna się od pierwszej reguły, `Expression` (wyrażenie), i przeszukuje kolejne, znajdując te, które pasują do tokenów w miarę ich wczytywania. Wczytywanie strumienia tokenów zgodnie z zasadami gramatyki nazywa się **parsowaniem** (analizą składniową), a program, który to robi, nazywamy **parserem** (ang. *parser*) lub **analizatorem składni** (ang. *syntax analyzer*). Nasz analizator odczytuje tokeny od lewej do prawej, dokładnie w takiej kolejności, jak je wpisujemy i czytamy. Wypróbujemy jakiś bardzo prosty przykład: czy `2` jest wyrażeniem?

1. Wyrażenie (`Expression`) musi być składnikiem (`Term`) lub kończyć się składnikiem. Składnik musi być czynnikiem (`Primary`) lub kończyć się czynnikiem. Ten czynnik musi zaczynać się znakiem `(` lub być liczbą (`Number`). Oczywiście `2` nie jest znakiem `(`, tylko literałem zmiennoprzecinkowym, a więc liczbą, która jest czynnikiem.

2. Przed tym czynnikiem (liczba 2) nie ma znaku /, * ani %, a więc jest to kompletny składnik (a nie zakończenie wyrażenia z operatorem /, * lub %).

3. Przed składnikiem tym (Primary 2) nie ma znaku + ani -, a więc jest to pełne wyrażenie (Expression), a nie zakończenie wyrażenia z operatorem + lub -.

W związku z tym zgodnie z naszą gramatyką 2 jest wyrażeniem. Przegląd gramatyki można przedstawić graficznie:




Na rysunku została przedstawiona ścieżka, którą przemierzyliśmy przez definicje. Odwracając nasze rozumowanie, możemy powiedzieć, że 2 jest wyrażeniem, ponieważ jest literałem zmiennoprzecinkowym, który jest liczbą, liczba jest czynnikiem, czynnik jest składnikiem, a składnik wyrażeniem.

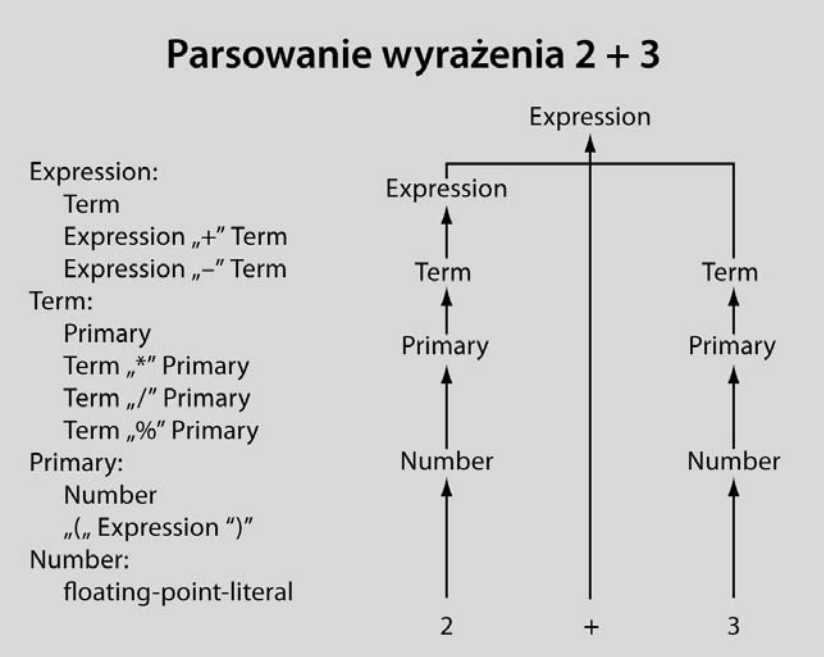
Spróbujmy czegoś bardziej skomplikowanego. Czy 2+3 jest wyrażeniem? Naturalnie znaczna część rozumowania będzie taka sam jak dla 2:

1. Wyrażenie musi być składnikiem lub mieć go na końcu. Składnik musi być czynnikiem lub kończyć się czynnikiem, który z kolei musi zaczynać się od znaku ( lub być liczbą. Oczywiście 2 nie jest znakiem (, ale jest literałem zmiennoprzecinkowym, który jest liczbą, ta z kolei jest czynnikiem.

2. Przed tym czynnikiem (liczba 2) nie ma znaku /, * ani %, a więc jest to kompletny składnik (a nie zakończenie wyrażenia z operatorem /, * lub %).

3. Za składnikiem tym (Primary 2) jest znak +, a więc jest to koniec pierwszej części wyrażenia i musimy poszukać składnika za tym znakiem. Dokładnie w taki sam sposób, jak w przypadku 2 dowiadujemy się, że 3 jest składnikiem. Ponieważ za 3 nie ma znaku + ani -, uznajemy, że jest to pełny składnik, a nie pierwsza część wyrażenia z operatorem + lub -. W związku z tym 2+3 spełnia zasadę Expression+Term, a więc jest wyrażeniem.

Znowu nasz tok rozumowania możemy przedstawić graficznie (pomijamy zasadę literału zmiennoprzecinkowego jako liczby dla uproszczenia):

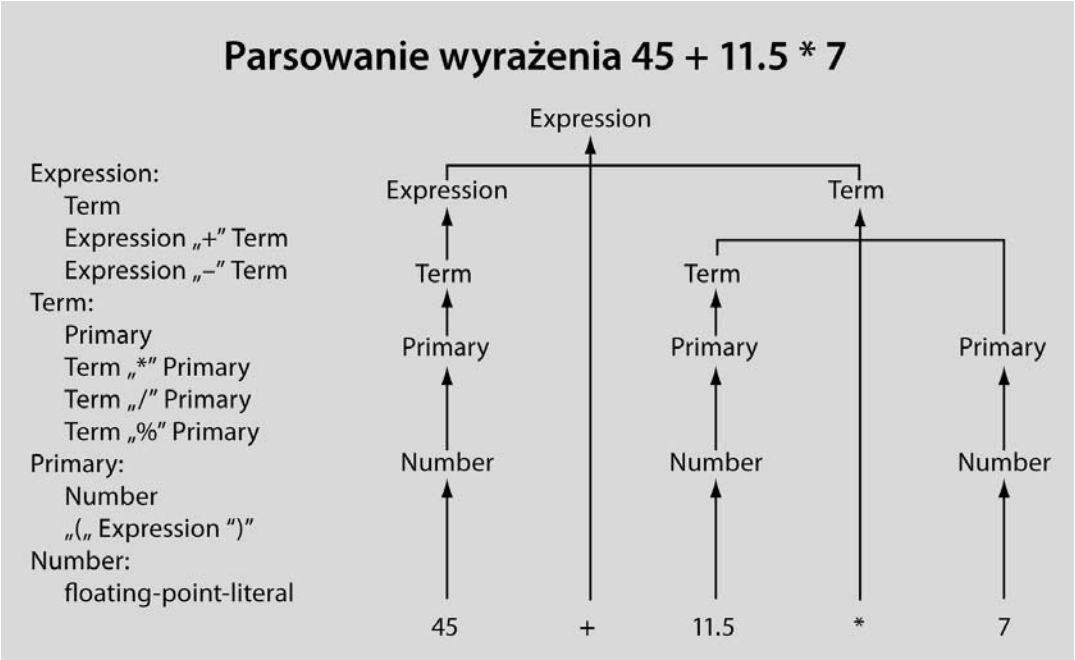



Na rysunku przedstawiliśmy ścieżkę, którą przemierzyliśmy przez definicje. Odwracając nasze rozumowanie, możemy powiedzieć, że `2+3` jest wyrażeniem, ponieważ `2` jest składnikiem, który jest wyrażeniem, `3` jest składnikiem oraz wyrażenie ze znakiem `+` i składnikiem również jest wyrażeniem.

Prawdziwym powodem, dla którego zainteresowaliśmy się gramatykami, jest fakt, że mogą one nam pomóc w rozwiązaniu problemu poprawnego przetwarzania wyrażeń z operatorami `*` i `+`. Spróbujemy zatem przeanalizować wyrażenie `45+11.5*7`. Zabawa w komputer szczegółowo sprawdzający wszystkie reguły byłaby żmudna. Dlatego pominiemy niektóre pośrednie etapy, które opisaliśmy już przy analizie `2` i `2+3`. Oczywiście `45`, `11.5` i `7` to literały zmiennoprzecinkowe, które są liczbami, liczby zaś są czynnikami, a więc możemy zignorować wszystkie reguły poniżej czynnika (`Primary`). Otrzymujemy:

1. `45` jest wyrażeniem, po którym znajduje się znak `+`, a więc szukamy wyrazu zamykającego regułę `Expression+Term`.
2. `11.5` jest składnikiem, po którym znajduje się znak `*`, a więc szukamy czynnika kończącego regułę `Term*Primary`.
3. `7` jest czynnikiem, a więc `11.5*7` jest składnikiem zgodnie z regułą `Term*Primary`. Teraz widzimy, że `45+11.5*7` jest wyrażeniem zgodnie z regułą `Expression+Term`. Mówiąc dokładniej, jest to wyrażenie, w którym najpierw jest wykonywane mnożenie `11.5*7`, a potem dodawanie `45+11.5*7`, dokładnie tak, jak gdybyśmy napisali `45+(11.5*7)`.

Na następnej stronie przedstawiamy graficzną ilustrację naszego rozumowania (znowu pomijamy zasadę literału zmiennoprzecinkowego jako liczby dla uproszczenia).




Powyższy rysunek obrazuje nasz tok rozumowania. Zauważ, jak reguła `Term*Primary` zapewnia, że `11.5` zostanie pomnożone przez `7` zamiast dodane do `45`.

Początkowo może wydawać Ci się to trudne do zrozumienia, ale ludzie czytają gramatyki, a te prostsze łatwo zrozumieć. Naszym celem nie było jednak nauczyć Cię rozumieć wyrażeń `2+2` czy `45+11.5*7`. To już każdy potrafi. Szukaliśmy sposobu na objaśnienie komputerowi, co oznacza wyrażenie `45+11.5*7` i wiele innych skomplikowanych wyrażeń, które mogą zostać mu podsunięte do obliczenia. Skomplikowane gramatyki nie nadają się do odczytu przez człowieka, ale komputery radzą sobie z nimi doskonale. Analizują ich reguły szybko i prawidłowo, a przychodzi im to z łatwością. W tym właśnie komputery są dobre — w dokładnym wykonywaniu poleceń.

## 6.4.1. Dygresja — gramatyka języka angielskiego

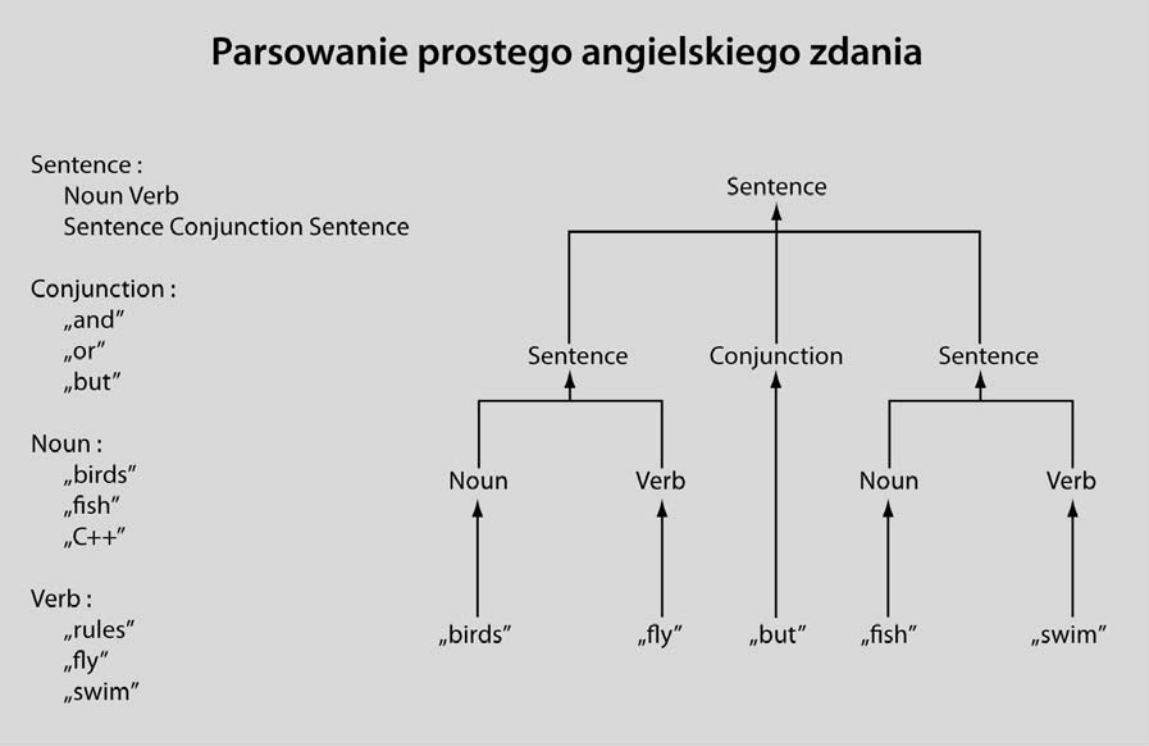
Jeśli nigdy dotąd nie miałeś do czynienia z gramatykami, to pewnie czujesz się oszołomiony. W istocie możesz czuć się niepewnie, nawet jeśli już się z czymś takim spotkałeś wcześniej. Spójrz na poniższy fragment gramatyki języka angielskiego:

```
Sentence :
    Noun Verb                          // np. C++ rules
    Sentence Conjunction Sentence      // np. Birds fly but fish swim
Conjunction :
    "and"
    "or"
    "but"
Noun :
    "birds"
    "fish"
    "C++"
Verb :
    "rules"
    "fly"
    "swim"
```

Zdania są zbudowane z części mowy (np. rzeczowników, czasowników i łączników). Można je przeanalizować pod kątem tych reguł, aby sprawdzić, które słowa są rzeczownikami, czasownikami itd. Ta prosta gramatyka zawiera także semantycznie bezsensowne zdania, tj. „C++ fly and birds rules", ale poprawienie tego to całkiem inna kwestia, którą należałoby się zająć w znacznie bardziej zaawansowanej książce.

Wiele osób uczono tych podstawowych zasad na zajęciach z języka angielskiego w szkole. Istnieją nawet poważne dowody neurologiczne potwierdzające, że takie zasady są zakodowane w naszych mózgach.

Spójrz na podobne do wcześniejszych drzewo parsowania, ale tym razem przedstawiające proste angielskie zdanie:




To nie jest takie skomplikowane. Jeśli miałeś problemy ze zrozumieniem podrozdziału 6.4, wróć do niego teraz i przeczytaj go ponownie. Za drugim razem może być o wiele bardziej zrozumiały!

## 6.4.2. Pisanie gramatyki

Skąd wzięliśmy te zasady gramatyki wyrażeń? Trzeba przyznać, że pomogło nam w tym doświadczenie. Robimy to w taki sposób, w jaki ludzie zazwyczaj piszą gramatyki wyrażeń. Napisanie prostej gramatyki jest łatwe. Wystarczy wiedzieć, jak:

1. Odróżnić zasadę od tokenu.
2. Wstawić jedną zasadę za drugą (**sekwencja**).
3. Wyrazić alternatywne wzorce (**alternacja**).
4. Wyrażać powtarzające się wzorce (**repetycja**).
5. Rozpoznać pierwszą zasadę gramatyki.

W różnych książkach i różnych systemach parsowania stosuje się różne notacje i terminologię. Na przykład niektórzy nazywają tokeny **znakami terminalnymi**, a zasady **znakami nieterminalnymi** lub **produkcjami**. My umieszczamy tokeny w podwójnych cudzysłowach i zaczynamy od pierwszej zasady. Alternatywy znajdują się w osobnych wierszach, np.:

```
List:
    "{" Sequence "}"
Sequence:
    Element
    Element " ," Sequence
Element:
    "A"
    "B"
```

Zatem sekwencja (`Sequence`) jest elementem (`Element`) lub elementem i sekwencją oddzielonymi od siebie przecinkiem. Element to litera `A` lub `B`. Lista (`List`) to sekwencja w nawiasach klamrowych. Możemy generować te listy (jak?):

```
{ A }
{ B }
{ A,B }
{A,A,A,A,B }
```

Poniższe natomiast nie są listami (dlaczego?):

```
{ }
A
{ A,A,A,A,B
{A,A,C,A,B }
{ A B C }
{A,A,A,A,B, }
```

Tej sekwencji nie nauczyłeś się w przedszkolu ani nie zakodowałeś w mózgu, chociaż nadal nie jest to nauka najwyższych lotów. W rozdziałach 7.4 i 7.8.1 objaśnimy sposoby wyrażania zasad syntaktycznych za pomocą gramatyk.

## 6.5. Zamiana gramatyki w kod

Istnieje wiele sposobów na zmuszenie komputera do respektowania zasad gramatyki. Użyjemy najprostszego — napiszemy po jednej funkcji dla każdej zasady i wykorzystamy nasz typ `Token` do reprezentowania tokenów. Program implementujący gramatykę często nazywa się **parserem**.

### 6.5.1. Implementowanie zasad gramatyki

Do zaimplementowania naszego kalkulatora potrzebujemy czterech funkcji — jednej do wczytywania tokenów i po jednej dla każdej zasady w gramatyce:

```
get_token()     // Wczytuje znaki i tworzy tokeny.
                // Wykorzystuje strumień cin.
expression()    // Obsługuje operatory + i –.
```

```
                    // Wywołuje funkcje term() i get_token().
term()              // Obsługuje operatory *, /, i %.
                    // Wywołuje funkcje primary() i get_token().
primary()           // Obsługuje liczby i nawiasy.
                    // Wywołuje funkcje expression() i get_token().
```

Uwaga: każda funkcja zajmuje się określoną częścią wyrażenia i resztę zostawia pozostałym funkcjom. To radykalnie upraszcza kod funkcji. Metodę tę można porównać ze współpracą grupy ludzi, gdzie każdy wykonuje swoje zadanie, a problemy znajdujące się poza jego specjalnością przekazuje do rozwiązania innym.

Co powinny te funkcje robić? Każda funkcja powinna wywoływać inne funkcje gramatyki zgodnie z zasadą, którą implementuje, oraz funkcję `get_token()`, jeśli wymagany jest token. Jeśli na przykład funkcja `primary()` ma postąpić zgodnie ze swoją zasadą (`Expression`), musi wywołać:

```
get_token()     // Do obsługi znaków ( i ).
expression()    // Do obsługi wyrażenia.
```

Co powinny takie funkcje zwracać? Co z odpowiedzią, której w rzeczywistości oczekujemy? Dla wyrażenia `2+3` funkcja `expression()` mogłaby zwrócić `5`. Istotnie, wszystkie potrzebne informacje są dostępne. Tego spróbujemy! Dzięki temu unikniemy odpowiedzi na jedno z naszych najtrudniejszych pytań: „Jak zaprezentować wyrażenie `45+5/7` w postaci danych, aby można było obliczyć jego wartość?”. Zamiast zapisywać reprezentację `45+5/7` w pamięci, obliczamy wartość tego wyrażenia przy wczytywaniu. Ten mały pomysł stanowi prawdziwy przełom! Dzięki temu czterokrotnie zmniejszy się objętość kodu w stosunku do tego, co uzyskalibyśmy, gdyby funkcja `expression()` zwracała coś skomplikowanego do obliczenia później. Właśnie zaoszczędziliśmy sobie około 80 procent pracy.

Do towarzystwa nie pasuje funkcja `get_token()` — ponieważ obsługuje tokeny, a nie wyrażenia, nie może zwracać wartości podwyrażeń. Na przykład znaki `(` i `+` nie są wyrażeniami. Musi zatem zwracać typ `Token`. Dochodzimy do wniosku, że potrzebujemy następujących funkcji:

```
// Funkcje realizujące zasady gramatyczne:
Token get_token()       // Wczytuje znaki i tworzy tokeny.
double expression()     // Obsługuje znaki + i –.
double term()           // Obsługuje znaki *, / i %.
double primary()        // Obsługuje liczby i nawiasy.
```

## 6.5.2. Wyrażenia

Najpierw napiszemy funkcję `expression()`. Gramatyka dla niej jest następująca:

```
Expression:
    Term
    Expression '+' Term
    Expression '–' Term
```

Ponieważ to jest nasza pierwsza próba zamiany zasad gramatyki na kod, przedstawimy kilka błędnych rozwiązań. Tak to zazwyczaj wygląda przy stosowaniu nowych technik, a poza tym jest to dobry sposób na nauczenie się wielu rzeczy. W szczególności początkujący programista

może nauczyć się bardzo dużo, patrząc na dramatycznie różne zachowania podobnych fragmentów kodu. Czytanie kodu to przydatna umiejętność, którą należy pielęgnować.

### 6.5.2.1. Wyrażenia — pierwsza próba

Patrząc na zasadę **Expression'+' Term**, najpierw próbujemy wywołać funkcję `expression()`, następnie znaleźć znak + (i -) oraz wywołać funkcję `term()`:

```
double expression()
{
    double left = expression(); // Wczytuje i oblicza wartość wyrażenia.
    Token t = get_token();      // następny token
    switch (t.kind) {           // Sprawdza, jaki to jest rodzaj tokenu.
    case '+':
        return left + term();       // Wczytuje i sprawdza wartość składnika,
                                    // następnie wykonuje dodawanie.
    case '–':
        return left – term();       // Wczytuje i sprawdza wartość składnika,
                                    // następnie wykonuje odejmowanie.
    default:
        return left;                // Zwraca wartość wyrażenia.
    }
}
```

Wygląda dobrze. To jest prawie naiwna transkrypcja gramatyki. Jest dość prosta — najpierw wczytuje wyrażenie, a następnie sprawdza, czy za nim znajduje się znak + lub - i jeśli tak, wczytuje składnik (`Term`).

Niestety w rzeczywistości to jest bez sensu. Skąd wiadomo, gdzie kończy się wyrażenie, dzięki czemu można poszukać znaków + i -? Przypominam, że nasz program czyta od lewej do prawej i nie może wybiec do przodu, aby sprawdzić, czy nie ma dalej znaku +. W istocie ta wersja funkcji `expression()` nigdy nie wyjdzie poza pierwszy wiersz — zaczyna od wywołania funkcji `expression()`, która zaczyna od wywołania funkcji `expression()` itd. w nieskończoność. Nazywa się to **nieskończoną rekurencją** (ang. *infinite recursion*), która jednak będzie miała koniec — gdy skończy się w komputerze pamięć do przechowywania nigdy niekończących się wywołań funkcji `expression()`. Pojęcie **rekurencji** oznacza wywoływanie funkcji przez siebie samą. Nie wszystkie rekurencje są nieskończone, poza tym technika ta jest bardzo przydatna w programowaniu (zobacz rozdział 8.5.8).

### 6.5.2.2. Wyrażenia — druga próba

Co w takim razie robimy? Każdy składnik jest wyrażeniem, ale nie każde wyrażenie jest składnikiem. Możemy więc zacząć od szukania składnika i przejść do szukania pełnego wyrażenia tylko wówczas, gdy znajdziemy znak + lub -. Na przykład:

```
double expression()
{
    double left = term();       // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token();      // następny token
    switch (t.kind) {           // Sprawdza, jaki to rodzaj tokenu.
    case '+':
```

```
            return left + expression(); // Wczytuje wyrażenie i sprawdza jego wartość,
                                        // a następnie wykonuje dodawanie.
        case '–':
            return left – expression(); // Wczytuje wyrażenie i oblicza jego wartość,
                                        // a następnie wykonuje odejmowanie.
        default:
            return left;                // Zwraca wartość składnika.
        }
    }
```

To jakoś działa. Wypróbowaliśmy ten kod w ukończonym programie i stwierdziliśmy, że przetwarza wszystkie poprawne wyrażenia (i ani jednego niepoprawnego). W większości przypadków nawet zwraca prawidłowe wyniki. Na przykład wyrażenie `1+2` jest wczytywane jako składnik (o wartości `1`), po którym jest znak `+` i wyrażenie (które w tym przypadku jest składnikiem o wartości `2`) — zwracany wynik to `3`. Analogicznie dla wyrażenia `1+2+3` zwracana jest wartość `6`. Możemy długo ciągnąć listę przypadków, w których kod ten zwraca prawidłowe wyniki, ale do rzeczy — co stanie się z wyrażeniem `1-2-3`? Funkcja `expression()` wczyta `1` jako składnik i przejdzie do `2-3` jako wyrażenia (składającego się ze składnika `2` i wyrażenia `3`). Następnie odejmie wartość wyrażenia `2-3` od `1`. Innymi słowy obliczy wartość wyrażenia `1-(2-3)`, która wynosi `2` (liczba dodatnia). Nas jednak w szkole uczono (a może i wcześniej), że `1-2-3` oznacza tyle, co `(1-2)-3`, a więc wynik powinien być `-4` (liczba ujemna).

W ten sposób utworzyliśmy bardzo fajny program, który nie robi tego, co trzeba. Sytuację pogarsza fakt, że w wielu przypadkach zwraca prawidłowy wynik. Na przykład dla wyrażenia `1+2+3` zostanie zwrócony wynik `6`, ponieważ `1+(2+3)` jest równoważne z `(1+2)+3`. Jaki był nasz podstawowy z programistycznego punktu widzenia błąd? Zawsze należy zadać sobie to pytanie, gdy znajdzie się błąd. Dzięki temu będzie można uniknąć powtórzenia go innym razem.

Naszym problemem jest to, że popatrzyliśmy na kod i postanowiliśmy zgadywać. To rzadko wystarcza! Musimy rozumieć, co robi nasz kod, i umieć wyjaśnić, czemu robi to, co trzeba.

Ponadto analizowanie błędów jest często najlepszym sposobem na znalezienie poprawnego rozwiązania. Nasza funkcja `expression()` szuka najpierw składnika, a następnie, jeśli znajduje się za nim znak `+` lub `-`, wyrażenia. Jest to w rzeczywistości implementacja nieco innej gramatyki:

```
Expression:
    Term
    Term '+' Expression // dodawanie
    Term '–' Expression // odejmowanie
```

Różnica między tą a naszą gramatyką polega na tym, że my chcieliśmy, aby wyrażenie `1-2-3` było traktowane jako wyrażenie `1-2`, po którym jest znak `-` i składnik `3`. Natomiast uzyskaliśmy składnik `1`, po którym znajduje się znak `-` i wyrażenie `2-3`. Innymi słowy chcieliśmy, aby `1-2-3` oznaczało `(1-2)-3`, a nie `1-(2-3)`.

Tak, debugowanie bywa żmudne, trudne i czasochłonne, ale w tym przypadku pracujemy nad zasadami wyuczonymi w podstawówce. Sęk w tym, że musimy „wpoić" je komputerowi, który uczy się znacznie wolniej od nas.

Zauważ, że moglibyśmy zdefiniować `1-2-3` jako `1-(2-3)` zamiast `(1-2)-3` i uniknąć całej tej dyskusji. Często najtrudniejszymi problemami w programowaniu są zasady, które zostały ustalone dla ludzi wiele lat przed tym, jak zaczęliśmy używać komputerów.

### 6.5.2.3. Wyrażenia — do trzech razy sztuka

Co teraz? Spójrz jeszcze raz na gramatykę (tę poprawną w podrozdziale 6.5.2) — każde wyrażenie zaczyna się składnikiem, po którym może znajdować się znak + lub -. Musimy więc poszukać składnika, sprawdzić, czy znajduje się za nim jeden z tych znaków, i robić tak aż do wyczerpania plusów i minusów. Na przykład:

```
double expression()
{
    double left = term();                   // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token();                  // następny token
    while ( t.kind=='+' || t.kind=='–') {   // Szuka znaków + i –.
        if (t.kind == '+')
            left += term();                 // Oblicza wartość składnika i wykonuje dodawanie.
        else
            left –= term();                 // Oblicza wartość składnika i wykonuje odejmowanie.
        t = get_token();
    }
    return left; // Jeśli nie ma więcej znaków + lub –, zwraca odpowiedź.
}
```

Ten kod jest nieco bardziej skomplikowany, ponieważ musieliśmy wprowadzić pętlę, aby móc wyszukać wszystkie plusy i minusy. Ponadto powtarzamy się trochę — dwa razy sprawdzamy znaki + i - oraz dwa razy wywołujemy funkcję `get_token()`. Ponieważ to tylko zaciemnia kod, pozbędziemy się nadmiarowego testu znaków + i -:

```
double expression()
{
    double left = term();  // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token(); // następny token
    while(true) {
        switch(t.kind) {
        case '+':
            left += term();         // Oblicza wartość składnika i wykonuje dodawanie.
            t = get_token();
            break;
        case '–':
            left –= term();         // Oblicza wartość składnika i wykonuje odejmowanie.
            t = get_token();
            break;
        default:
            return left;            // Jeśli nie ma więcej znaków + lub –, zwraca odpowiedź.
        }
    }
}
```

Warto zauważyć, że pomijając pętlę, ta wersja kodu jest podobna do pierwszej próby (podrozdział 6.5.2.1). Usunęliśmy wywołanie funkcji `expression()` wewnątrz funkcji `expression()` i zastąpiliśmy je pętlą. Innymi słowy przekonwertowaliśmy wyrażenie z zasad gramatyki na pętlę szukającą składnika ze znakiem + lub -.

### 6.5.3. Składniki

Zasada dotycząca składnika (`Term`) w gramatyce jest bardzo podobna do zasady wyrażenia (`Expression`):

```
Term:
    Primary
    Term '*' Primary
    Term '/' Primary
    Term '%' Primary
```

W konsekwencji kod również powinien być bardzo podobny. Oto pierwsza wersja:

```
double term()
{
    double left = primary();
    Token t = get_token();
    while(true) {
        switch (t.kind) {
        case '*':
            left *= primary();
            t = get_token();
            break;
        case '/':
            left /= primary();
            t = get_token();
            break;
        case '%':
            left %= primary();
            t = get_token();
            break;
        default:
            return left;
        }
    }
}
```

Niestety ten kod nie da się skompilować — operator reszty z dzielenia (`%`) nie jest zdefiniowany dla liczb zmiennoprzecinkowych. Zostaniemy o tym uprzejmie poinformowani. Odpowiadając wcześniej „oczywiście" na pytanie dotyczące tego, czy pozwolić też na wpisywanie liczb zmiennoprzecinkowych, nie przemyśleliśmy sprawy dokładnie i padliśmy ofiarą własnej **zachłanności na funkcje**. Tak jest **zawsze**! Co z tym zrobimy? Możemy sprawdzać, czy argumenty operatora `%` są liczbami całkowitymi i jeśli nie, zgłaszać błąd. Możemy też usunąć ten operator z kalkulatora. Wybierzemy najprostsze rozwiązanie. Dodamy ten operator później (rozdział 7.5).

Po usunięciu operatora `%` funkcja działa — tzn. prawidłowo analizuje składniki i oblicza ich wartości. Jednak doświadczony programista zauważy pewien niepożądany szczegół, który sprawia, że funkcja `term()` jest nie do przyjęcia. Co się stanie, jeśli ktoś wpisze `2/0`? Nie można dzielić przez zero. Jeśli do tego dopuścimy, komputer wykryje to i zamknie program, wyświetlając raczej mało pomocny komunikat o błędzie. Niedoświadczony programista przekona się o tym na własnej skórze. Dlatego lepiej dodać odpowiedni test i wyświetlić lepszy komunikat o błędzie:

```
double term()
{
    double left = primary();
    Token t = get_token();
    while(true) {
        switch (t.kind) {
        case '*':
            left *= primary();
            t = get_token();
            break;
        case '/':
            { double d = primary();
            if (d == 0) error("Dzielenie przez zero.");
            left /= d;
            t = get_token();
            break;
            }
        default:
            return left;
        }
    }
}
```

Dlaczego umieściliśmy instrukcje obsługujące operator / w bloku? Ponieważ kompilator nalegał. Jeśli chcesz definiować i inicjować zmienne w instrukcjach `switch`, musisz umieścić je w blokach.

## 6.5.4. Podstawowe elementy wyrażeń

Zasada gramatyczna definiująca podstawowe czynniki wyrażeń również jest prosta:

```
Primary:
    Number
    '(' Expression ')'
```

Kod służący do jej implementacji jest nieco zagmatwany, ponieważ jest tu więcej okazji do popełnienia błędu składni:

```
double primary()
{
    Token t = get_token();
    switch (t.kind) {
    case '(': // Obsługa reguły '(' expression ')'.
        {  double d = expression();
           t = get_token();
           if (t.kind != ')') error("Oczekiwano ')'.");
           return d;
        }
    case '8':            // Za pomocą znaku '8' reprezentujemy liczby.
        return t.value; // Zwraca wartość liczby.
    default:
        error("Oczekiwano czynnika.");
    }
}
```

W zasadzie nie ma tu nic nowego, czego nie ma w funkcjach `expression()` i `term()`.Użyliśmy tych samych narzędzi językowych, zastosowaliśmy tę samą technikę wyodrębniania tokenów oraz te same techniki programistyczne.

## 6.6. Wypróbowywanie pierwszej wersji

Aby sprawdzić działanie tych funkcji, musimy zaimplementować funkcję `get_token()` i utworzyć funkcję `main()`. To drugie zadanie jest banalne — będziemy wywoływać funkcję `expression()` i drukować zwracany przez nią wynik:

```
int main()
try {
    while (cin)
        cout << expression() << '\n';
    keep_window_open();
}
catch (exception& e) {
    cerr << e.what() << endl;
    keep_window_open ();
    return 1;
}
catch (...) {
    cerr << "Wyjątek \n";
    keep_window_open ();
    return 2;
}
```

Błędy zostaną obsłużone w typowy sposób (rozdział 5.6.3). Opis implementacji funkcji `get_token()` odłożymy do podrozdziału 6.8, a teraz przetestujemy pierwszą wersję naszego kalkulatora.

### WYPRÓBUJ

Pierwsza wersja kalkulatora (z funkcją `get_token()`) znajduje się w pliku o nazwie *calculator00.cpp*. Pobierz go i uruchom.

Wcale nie jesteśmy zaskoczeni tym, że program nie działa dokładnie tak, jak oczekiwaliśmy. Wzruszamy tylko ramionami i pytamy sami siebie: „Czemu nie?” albo lepiej: „Dlaczego działa tak, a nie inaczej?”. Wpisz liczbę `2` i znak nowego wiersza. Brak jakiejkolwiek reakcji. Daj jeszcze jeden znak nowego wiersza, aby sprawdzić, czy może zasnął. Nadal nic. Wpisz `3` i znak nowego wiersza. Zero reakcji! Wpisz `4` i znak nowego wiersza. Jest odpowiedź — `2`! Teraz ekran wygląda tak:

```
2

3
4
2
```

Kontynuujemy, wpisując `5+6`. Program zwraca w odpowiedzi `5`, a więc na ekranie widzimy:

```
2

3
4
2
5+6
5
```

Jeśli nie masz doświadczenia programistycznego, pewnie jesteś bardzo skonsternowany! W istocie nawet doświadczony programista mógłby osłupieć. Co tam się dzieje? W tym momencie chcemy zamknąć program. Jak to zrobić? Zapomnieliśmy zaprogramować polecenie zamykające program, ale można go zmusić do zamknięcia za pomocą błędu. Wpisujemy więc *x* i program zakończy działanie komunikatem *Nieprawidłowy token*. W końcu coś zadziałało zgodnie z planem!

Zapomnieliśmy odróżnić na ekranie dane wejściowe od wyjściowych. Zanim przejdziemy do rozwiązywania zagadki, poprawimy to niedociągnięcie, aby lepiej widzieć, co się dzieje. Na razie wystarczy dodanie do danych wyjściowych znaku `=`:

```
while (cin) cout << "= " << expression() << '\n'; // Wersja 1
```

Teraz po wprowadzeniu dokładnie takiej samej sekwencji znaków, jak wcześniej otrzymamy następujący wynik:

```
2

3
4
= 2
5+6
= 5
x
Nieprawidłowy token
```

Dziwne! Spróbuj odgadnąć, co zrobił program. My wypróbowaliśmy jeszcze kilka innych liczb, ale poprzestańmy na tych, które mamy. To jest zagadka:

Dlaczego program nie zareagował na pierwsze liczby `2` i `3` i znaki nowego wiersza?

Dlaczego program zwrócił wynik `2` zamiast `4`, gdy wpisaliśmy `4`?

Dlaczego program zwrócił wynik `5` zamiast `11`, gdy wpisaliśmy `5+6`?

Jest wiele procedur postępowania w takich tajemniczych sytuacjach. Opiszemy je szczegółowo w następnym rozdziale, a na razie tylko pomyślimy. Czy możliwe, żeby program wykonywał nieprawidłowe działania arytmetyczne? To bardzo mało prawdopodobne. Wartość `4` to nie `2`, a wyrażenia `5+6` to `11`, a nie `5`. Sprawdźmy, co się stanie, gdy wpiszemy `1 2 3 4+5 6+7 8+9 10 11 12` i nowy wiersz:

```
1 2 3 4+5 6+7 8+9 10 11 12
= 1
= 4
= 6
= 8
= 10
```

Co takiego? Nie `2` ani `3`? Czemu `4`, a nie `9` (bo `4+5`)? Czemu `6`, a nie `13` (bo `6+7`)? Przyjrzyj się uważnie — program zwraca co trzeci token! Może „pożera" część wprowadzonych danych bez obliczania ich wartości? Tak właśnie robi. Spójrz na funkcję `expression()`:

```
double expression()
{
    double left = term();  // Wczytuje składnik i oblicza jego wartość.
    Token t = get_token(); // następny token
    while(true) {
        switch(t.kind) {
        case '+':
            left += term();     // Oblicza wartość składnika i wykonuje dodawanie.
            t = get_token();
            break;
        case '–':
            left –= term();     // Oblicza wartość składnika i wykonuje odejmowanie.
            t = get_token();
            break;
        default:
            return left;        // Jeśli nie ma więcej znaków + lub –, zwraca odpowiedź.
        }
    }
}
```

Gdy funkcja `get_token()` zwróci inny `Token` niż `+` lub `-`, po prostu zwracamy wartość. Nie używamy tego tokenu i nie zapisujemy go do użytku przez inną funkcję. To nie było zbyt mądre. Odrzucanie danych wejściowych bez sprawdzenia nawet, co to dokładnie jest, to niezbyt dobry pomysł. Łatwo można się przekonać, że ten sam defekt ma funkcja `term()`. To wyjaśnia, dlaczego nasz kalkulator zjadał dwa tokeny na trzy.

Poprawimy funkcję `expression()`, aby nie zjadała tokenów. Gdzie możemy zapisać następny token (`t`), jeśli program go nie potrzebuje? Moglibyśmy opracować wiele różnych skomplikowanych rozwiązań, ale zastosujemy najbardziej oczywiste (wydaje się oczywiste, gdy się je już zobaczy) — ten token zostanie wykorzystany przez jakąś inną funkcję, a więc umieścimy go z powrotem w strumieniu wejściowym, aby mógł zostać stamtąd ponownie wczytany! W istocie można by było wstawiać znaki do strumienia `istream`, ale nie o to nam chodzi. My chcemy mieć tokeny, a nie bawić się ze znakami. Potrzebujemy strumienia wejściowego, w którym będzie można zapisywać tokeny.

Załóżmy więc, że mamy strumień tokenów — `Token_stream` — o nazwie `ts`. Przyjmijmy też, że ma on funkcję składową o nazwie `get()`, która zwraca następny token i funkcję składową `putback(t)`, ta zaś wstawia token `t` z powrotem do strumienia. Implementację strumienia `Token_stream` przedstawimy w podrozdziale 6.8, gdy będziemy już wiedzieli, jak go użyć.

Mając strumień `Token_stream`, możemy zmodyfikować funkcję `expression()` w taki sposób, żeby niepotrzebne tokeny zapisywała właśnie w nim:

```
double expression()
{
    double left = term(); // Wczytuje składnik i oblicza jego wartość.
    Token t = ts.get();   // Pobiera następny token ze strumienia tokenów.
    while(true) {
        switch(t.kind) {
        case '+':
            left += term();   // Oblicza wartość składnika i wykonuje dodawanie.
            t = ts.get();
            break;
        case '–':
            left –= term();   // Oblicza wartość składnika i wykonuje odejmowanie.
            t = ts.get();
            break;
        default:
            ts.putback(t);    // Wstawia token t z powrotem do strumienia tokenów.
            return left;      // Jeśli nie ma więcej znaków + lub –, zwraca odpowiedź.
        }
    }
}
```

Takie same zmiany muszą zostać wprowadzone w funkcji `term()`:

```
double term()
{
    double left = primary();
    Token t = ts.get(); // Pobiera następny token ze strumienia tokenów.

    while(true) {
        switch (t.kind) {
        case '*':
            left *= primary();
            t = ts.get();
            break;
        case '/':
            { double d = primary();
            if (d == 0) error("Dzielenie przez zero.");
            left /= d;
            t = ts.get();
            break;
            }
        default:
            ts.putback(t); // Wstawia token t z powrotem do strumienia tokenów.
            return left;
        }
    }
}
```

W ostatniej funkcji naszego parsera, `primary()`, musimy tylko zmienić `get_token()` na `ts.get()`. Funkcja ta używa każdego wczytanego przez siebie tokenu.

## 6.7. Wypróbowywanie drugiej wersji

Możemy przetestować naszą drugą wersję programu. Wpisz `2` i znak nowego wiersza. Brak reakcji. Wpisz jeszcze jeden znak nowego wiersza, aby sprawdzić, czy program nie zasnął. Nadal nic. Wpisz `3` i znak nowego wiersza. Jest odpowiedź — `2`. Spróbuj wyrażenia `2+2` ze znakiem nowego wiersza. Odpowiedź brzmi `3`. Teraz na ekranie znajdują się następujące dane:

```
2

3
=2
2+2
=3
```

Hmm. Może wprowadzenie funkcji `putback()` i użycie jej w funkcjach `expression()` i `term()` nie pomogło w rozwiązaniu problemu? Spróbujmy czegoś innego:

```
2 3 4 2+3 2*3
=2
=3
=4
=5
```

Tak, to są poprawne odpowiedzi! Ale brakuje ostatniej (`6`). Wciąż mamy problem z tokenami, ale tym razem nie chodzi o ich zjadanie, lecz o to, że wynik jest zwracany dopiero po wpisaniu kolejnego wyrażenia. Program nie drukuje od razu wyniku. Opóźnia to do momentu wczytania pierwszego tokenu następnego wyrażenia. Niestety nie widzi go, dopóki nie naciśniemy klawisza *Enter* po wpisaniu następnego wyrażenia. Wniosek taki, że program nie działa źle, tylko odpowiada z opóźnieniem.

Jak to poprawić? Jednym z oczywistych rozwiązań jest wprowadzenie „polecenia drukowania”. Rolę tę niech pełni średnik, którego pojawienie się będzie oznaczało koniec wyrażenia i wymuszało wydruk wyniku. Przy okazji dodamy polecenie zamknięcia programu. Do tego celu doskonale nada się znak *k* (od słowa koniec). Obecnie w funkcji `main()` mamy taki kod:

```
while (cin) cout << "=" << expression() << '\n'; // wersja 1
```

Możemy zamienić ten na bardziej zagmatwany, ale bardziej użyteczny kod:

```
double val = 0;
while (cin) {
    Token t = ts.get();

    if (t.kind == 'k') break; // 'k' = koniec
    if (t.kind == ';') // ';' = drukuj teraz
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

Teraz kalkulator nadaje się do użytku. Zwróci na przykład następujące wyniki:

```
2;
= 2
2+3;
= 5
3+4*5;
= 23
k
```

W tym momencie mamy już dobrą wstępną wersję programu. Nie jest to jeszcze to, czego chcieliśmy, ale stanowi dobrą bazę do rozszerzania. Co ważniejsze, możemy poprawiać błędy i dodawać nowe funkcje, pracując już z działającym programem.

## 6.8. Strumienie tokenów

Zanim przejdziemy do dalszego opisu kalkulatora, przedstawimy implementację strumienia `Token_stream`. Jeśli nie dostarczymy poprawnych danych do programu, to nic nam się nie uda. Strumień ten zaimplementowaliśmy już na samym początku, ale najpierw chcieliśmy skupić się na problemach obliczeń w minimalnym rozwiązaniu.

Nasz kalkulator przyjmuje na wejściu sekwencję tokenów, np. `(1.5+4)*11` z podrozdziału 6.3.3. Potrzebujemy czegoś takiego, co będzie wczytywać znaki ze standardowego strumienia wejściowego i podawać programowi kolejne tokeny, gdy ten o to poprosi. Poza tym zauważyliśmy, że program często wczytuje o jeden token za dużo, a więc musimy wygospodarować jakieś miejsce, aby przechować go do późniejszego użytku. Jest to całkowicie normalne. Jeśli wczytujemy wyrażenie `1.5+4` od lewej do prawej, skąd mamy wiedzieć, że liczba `1.5` to już cała liczba, jeśli nie wczytamy znajdującego się za nią znaku `+`? Dopóki go nie zobaczymy, równie dobrze możemy podejrzewać, że jesteśmy w trakcie wczytywania liczby `1.55555`. Dlatego potrzebujemy „strumienia", który zwraca token, gdy go zażądamy za pomocą funkcji `get()`, i w którym możemy przechowywać nadmiarowe tokeny, zapisując je tam za pomocą funkcji `putback()`. Wszystko w języku C++ ma określony typ, dlatego musimy zacząć od zdefiniowania typu `Token_stream`.

Pewnie zauważyłeś słowo `public` w definicji typu `Token`. Tam nie miało to żadnego widocznego znaczenia. Natomiast w typie `Token_stream`, o którym teraz mowa, słowo to będzie miało ważne zastosowanie. W języku C++ typy zdefiniowane przez użytkownika często składają się z dwóch części — interfejsu publicznego (oznaczonego etykietą `public:`) i prywatnego (oznaczonego etykietą `private:`). Chodzi o oddzielenie tego, co jest potrzebne użytkownikowi do wygodnego korzystania z typu, od implementacji typu, w której użytkownik nie powinien mieć możliwości grzebania:

```
class Token_stream {
public:
    // interfejs użytkownika
private:
    // szczegóły implementacyjne
    // (bezpośrednio niedostępne użytkownikom typu Token_stream)
};
```

Oczywiście użytkownik i implementator to często jedna i ta sama osoba odgrywająca różne role. Należy jednak zaznaczyć, że rozróżnienie między interfejsem publicznym przeznaczonym

dla użytkowników i prywatnym dla implementatorów jest doskonałym narzędziem wspomagającym strukturalizację kodu. Interfejs publiczny powinien zawierać wyłącznie to, co jest użytkownikowi potrzebne, a więc najczęściej zestaw funkcji, także konstruktorów służących do inicjowania obiektów. Implementację prywatną stanowi kod tych publicznych funkcji. Najczęściej są to dane i funkcje wykonujące skomplikowane działania, o których użytkownik nie musi wiedzieć i których nie musi bezpośrednio używać.

Rozszerzymy trochę nasz typ `Token_stream`. Jakie wymagania powinien spełniać? Bez wątpienia potrzebujemy funkcji `get()` i `putback()`, ponieważ to one były powodem, dla którego w ogóle wymyśliliśmy coś takiego jak strumień tokenów. Zadaniem typu `Token_stream` jest wytwarzanie tokenów ze znaków wczytywanych na wejściu. Zatem nasz strumień musi wczytywać dane ze strumienia `cin`. Oto najprostsza możliwa wersja typu `Token_stream`:

```
class Token_stream {
public:
    Token_stream();          // Tworzy obiekt typu Token_stream, który wczytuje dane ze strumienia
cin.
    Token get();             // Daje token (obiekt typu Token).
    void putback(Token t); // Wkłada token z powrotem.
private:
    // szczegóły implementacyjne
};
```

To wszystko, czego użytkownik potrzebuje do korzystania z typu `Token_stream`. Doświadczony programista mógłby się zastanawiać, dlaczego `cin` jest jedynym możliwym źródłem znaków — przypominamy, że zdecydowaliśmy się na razie pobierać dane tylko z klawiatury. Zrewidujemy tę decyzję w rozdziale 7.

Dlaczego zastosowaliśmy długawą nazwę `putback()` zamiast krótszej `put()`? Chcieliśmy podkreślić asymetrię między funkcjami `get()` i `putback()` — to jest strumień wejściowy, a nie coś, co można wykorzystać także do ogólnych celów związanych z wysyłaniem danych na wyjście. Poza tym w bibliotece `istream` także znajduje się funkcja `putback()`, a spójność nazw jest jedną z pożądanych cech każdego systemu. Dzięki temu łatwiej jest zapamiętywać te nazwy i unikać błędów.

Po tym wstępie możemy już utworzyć typ `Token_stream` i użyć go:

```
Token_stream ts;      // Obiekt typu Token_stream o nazwie ts.
Token t = ts.get(); // Daje następny token ze strumienia ts.
// …
ts.putback(t);        // Wkłada token t z powrotem do strumienia ts.
```

Mamy już wszystko, czego potrzebujemy do napisania pozostałej części kalkulatora.

## 6.8.1. Implementacja typu Token_stream

Zaimplementujemy trzy wymienione funkcje strumienia `Token_stream`. Jak będzie się ten strumień prezentował? Tzn., jakie dane muszą być w nim przechowywane, aby spełniał swoje zadanie? Potrzebujemy miejsca do przechowywania tokenów, które włożymy do niego. Dla uproszczenia załóżmy, że można w nim przechowywać tylko jeden token na raz. Na potrzeby naszego programu to wystarczy (i na potrzeby wielu innych też). W związku z tym potrzebujemy

miejsca do przechowania jednego tokenu i wskaźnika oznaczającego, czy to miejsce jest wolne czy zajęte:

```
class Token_stream {
public:
    Token_stream();          // Tworzy obiekt typu Token_stream, który wczytuje dane ze strumienia
cin.
    Token get();             // Daje token (funkcja get() została zdefiniowana w innym miejscu).
    void putback(Token t); // Wkłada token z powrotem.
private:
    bool full;      // Informuje, czy w buforze jest token.
    Token buffer; // Miejsce do przechowywania tokenu zapisanego w strumieniu za pomocą funkcji
                  //putback().
};
```

Teraz możemy zdefiniować (napisać) nasze trzy funkcje składowe. Konstruktor i funkcja `putback()` będą łatwe, ponieważ mają mało do zrobienia. Dlatego zajmiemy się nimi na początku.

Konstruktor ustawia tylko zmienną `full` na wartość oznaczającą, że bufor jest pusty:

```
Token_stream::Token_stream()
   :full(false), buffer(0) // Bufor jest pusty.
{
}
```

Jeśli definicja składowej znajduje się poza definicją klasy, należy wskazać, do której klasy ma należeć. Służy do tego następująca notacja:

```
nazwa_klasy::nazwa_składowej
```

Tutaj definiujemy konstruktor klasy `Token_stream`. Konstruktor to funkcja składowa o takiej samej nazwie, jak klasa, do której należy.

Po co definiować składową poza klasą? Przede wszystkim dla zachowania przejrzystości — w definicji klasy można znaleźć głównie informacje o tym, co klasa potrafi „robić". Definicje funkcji składowych to implementacje określające sposób, w jaki są robione różne rzeczy. Wolimy umieścić je gdzieś indziej, aby nie przeszkadzały. Ideałem, do którego dążymy, jest, aby każda jednostka logiczna programu mieściła się w całości na ekranie. Definicje klas zazwyczaj spełniają ten wymóg, ale tylko jeśli definicje ich funkcji składowych przeniesie się gdzieś indziej, poza klasę.

Składowe klasy inicjuje się w liście inicjatorów składowych (podrozdział 6.3.3). Instrukcja `full(false)` ustawia składową `full` na `false`, a `buffer(0)` inicjuje składową `buffer` „udawanym tokenem", który wymyśliliśmy na poczekaniu. Z definicji typu `Token` (podrozdział 6.3.3) wynika, że każdy token musi zostać zainicjowany, dlatego nie mogliśmy zignorować `Token_stream::buffer`.

Funkcja składowa `putback()` zapisuje swój argument w buforze strumienia `Token_stream`:

```
void Token_stream::putback(Token t)
{
    buffer = t;  // Kopiuje t do bufora.
    full = true; // Bufor jest pełny.
}
```

Słowo kluczowe `void` (oznaczające „nic”) wskazuje, że funkcja `putback()` nie zwraca żadnej wartości. Aby upewnić się, że funkcja ta nie zostanie wywołana dwa razy bez odczytania (za pomocą funkcji `get()`) tego, co zostało zapisane w strumieniu między tymi wywołaniami, można dodać specjalny test:

```
void Token_stream::putback(Token t)
{
    if (full) error("Wywołanie funkcji putback(), gdy bufor jest pełny.");
    buffer = t;  // Kopiuje t do bufora.
    full = true; // Bufor jest pełny.
}
```

Test składowej `full` polega na sprawdzeniu warunku wstępnego: „Nie ma żadnego tokenu w buforze”.

## 6.8.2. Wczytywanie tokenów

Całą prawdziwą pracę wykonuje funkcja `get()`. Jeśli w zmiennej `Token_stream::buffer` nie ma tokenu, funkcja ta musi wczytać znaki ze strumienia `cin` i złożyć z nich tokeny:

```
Token Token_stream::get()
{
    if (full) { // Sprawdzenie, czy jest gotowy token.
        // Usunięcie tokenu z bufora.
        full=false;
        return buffer;
    }

    char ch;
    cin >> ch; // Uwaga: >> pomija białe znaki (spacje, nowe wiersze, tabulatory itp.).

    switch (ch) {
    case ';':  // drukowanie
    case 'k':  // koniec
    case '(': case ')': case '+': case '-': case '*': case '/': case '%':
        return Token(ch); // Każdy znak reprezentuje sam siebie.
    case '.':
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        {   cin.putback(ch);        // Wstawia cyfrę z powrotem do strumienia wejściowego.
            double val;
            cin >> val;             // Wczytuje liczbę zmiennoprzecinkową.
            return Token('8',val); // '8' reprezentuje „liczbę”.
        }
    default:
        error("Nieprawidłowy token.");
    }
}
```

Przeanalizujemy szczegółowo funkcję `get()`. Najpierw sprawdza, czy w buforze jest token. Jeśli tak, zwraca go:

```
if (full) { // Sprawdzenie, czy jest gotowy token.
    // Usunięcie tokenu z bufora.
    full=false;
    return buffer;
}
```

Znakami musimy zajmować się tylko wówczas, gdy `full` ma wartość `false` (tzn. nie ma tokenu w buforze). W takim przypadku wczytujemy znak i postępujemy z nim odpowiednio do potrzeb. Szukamy nawiasów, operatorów i liczb. Jakikolwiek inny znak powoduje wywołanie funkcji `error()`, która zamyka program:

```
default:
    error("Nieprawidłowy token.");
}
```

Funkcja `error()` została opisana w rozdziale 5.6.3 i jest dostępna w pliku nagłówkowym *std_lib_facilities.h*.

Musieliśmy zdecydować się na jakiś sposób reprezentowania każdego rodzaju tokenu, tzn. trzeba było dobrać wartości dla składowej `kind`. Dla uproszczenia i ułatwienia debugowania zdecydowaliśmy się, że nawiasy i operatory same będą reprezentować swój rodzaj tokenu. Dzięki temu ich przetwarzanie jest bardzo łatwe:

```
case '(': case ')': case '+': case '–': case '*': case '/':
    return Token(ch); // Każdy znak reprezentuje sam siebie.
```

Mówiąc szczerze, w pierwszej wersji programu zapomnieliśmy o znakach: `';'` oznaczającym drukowanie i `'k'` oznaczającym koniec programu. Dodaliśmy je, dopiero gdy były potrzebne w drugiej wersji.

## 6.8.3. Wczytywanie liczb

Zostały nam jeszcze liczby, z którymi wcale tak łatwo nie pójdzie. Jak dowiedzieć się, jaka jest wartość `123`? To tyle samo, co `100+20+3`, ale co zrobić z `12.34`? Jest też pytanie, czy zezwalać na stosowanie notacji naukowej, np. `12.34e5`. Aby poradzić sobie z tym wszystkim, moglibyśmy potrzebować wielu godzin, a nawet dni. Na szczęście nie jest tak źle. Strumienie w C++ „wiedzą”, jak wyglądają literały, i potrafią zamieniać je na wartości typu `double`. Musimy tylko znaleźć sposób na zmuszenie strumienia `cin` do robienia tego w funkcji `get()`:

```
case '.':
case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7':
case '8':case '9':
    {   cin.putback(ch);        // Wstawia cyfrę z powrotem do strumienia wejściowego.
        double val;
        cin >> val;             // Wczytuje liczbę zmiennoprzecinkową.
        return Token('8',val); // '8' reprezentuje „liczbę”.
    }
```

Decyzja o wyborze znaku `'8'` do reprezentowania „liczb” w tokenach została podjęta arbitralnie.

Skąd wiadomo, że ma pojawić się liczba? Zgadując na podstawie własnych doświadczeń lub zaglądając do podręcznika języka C++ (np. w dodatku A), można stwierdzić, że literał liczbowy musi zaczynać się od cyfry lub znaku `.` (kropki dziesiętnej). Sprawdzamy więc, czy tak jest. Następnie chcemy, aby liczbę tę wczytał strumień `cin`, ale wczytaliśmy już pierwszy znak (cyfrę lub kropkę), a więc pozwalając temu strumieniowi zająć się resztą, uzyskalibyśmy nieprawidłowe wyniki. Moglibyśmy spróbować połączyć wartość pierwszego znaku z „resztą" wczytaną przez `cin`. To znaczy, jeśli wpisano by `123`, otrzymalibyśmy `1` i strumień `cin` wczytałby `23`. Wówczas trzeba by było dodać `100` do `23`. A fuj! A to tylko banalny przykład. Na szczęście (i nie przypadkowo) `cin` działa bardzo podobnie do `Token_stream` pod tym względem, że także pozwala wstawić odczytany znak z powrotem. Zamiast więc wykonywać zagmatwane obliczenia arytmetyczne, wstawiamy pierwszą cyfrę z powrotem do `cin` i odczytujemy z niego całą liczbę.

Zauważ, że cały czas unikamy skomplikowanej pracy i znajdujemy prostsze rozwiązania — często w tym celu wykorzystując zawartość biblioteki. To jest istota programowania — nieustanne poszukiwanie jak najprostszych rozwiązań. Niektórzy — nie zawsze słusznie — wyrażają to słowami: „Dobry programista to leniwy programista". W tym, i tylko w tym, sensie rzeczywiście powinniśmy być leniwi. Po co pisać dużo kodu, skoro można to samo zrobić, pisząc go mniej?

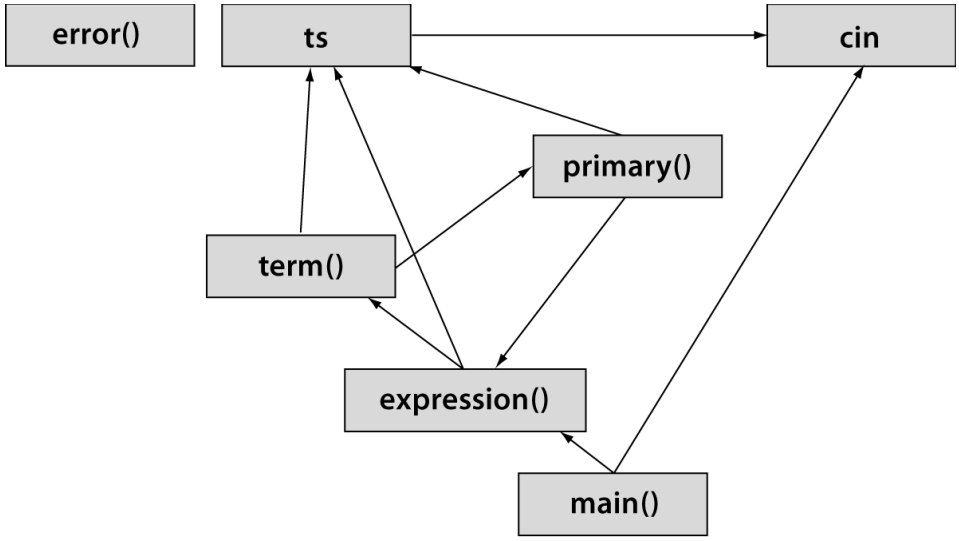
## 6.9. Struktura programu

Jest takie przysłowie, które głosi, że czasami trudno zauważyć las, ponieważ zasłaniają go drzewa. W analogiczny sposób można stracić z oczu program, patrząc na te jego wszystkie funkcje, klasy itp. Spójrzmy zatem na nasz program, pomijając chwilowo szczegóły:

```
#include "std_lib_facilities.h"

class Token { /*... */ };
class Token_stream { /*... */ };

Token_stream::Token_stream() :full(false), buffer(0) { /*... */ }
void Token_stream::putback(Token t) { /*... */ }
Token Token_stream::get() { /*... */ }

Token_stream ts;       // Udostępnia funkcje get() i putback().
double expression(); // Deklaracja umożliwiająca funkcji primary() wywoływanie funkcji expression().

double primary() { /*... */ }      // Obsługa nawiasów i liczb.
double term() { /*... */ }         // Obsługa operatorów *, / i %.
double expression() { /*... */ } // Obsługa operatorów + i –.

int main() { /*... */ } // Główna pętla i obsługa błędów.
```

Kolejność deklaracji ma znaczenie. Nie można użyć nazwy, zanim zostanie zadeklarowana. Dlatego `ts` musi zostać zadeklarowana, zanim zostanie użyta w `ts.get()`, a funkcja `error()` musi zostać zadeklarowana przed funkcjami parsera, ponieważ wszystkie z niej korzystają. Na schemacie wywołań można zauważyć ciekawe zapętlenie: funkcja `expression()` wywołuje `term()`, która wywołuje `primary()`, a ta z kolei wywołuje `expression()`.

Można to przedstawić graficznie (wywołania funkcji `error()` odkładamy na bok, ponieważ wszystkie funkcje ją wywołują):




To oznacza, że nie możemy po prostu zdefiniować tych trzech funkcji — nie da się ustalić takiej kolejności, w której każda z nich zostałaby zdefiniowana przed pierwszym użyciem. Potrzebujemy przynajmniej jednej deklaracji, która nie jest równocześnie definicją. Zdecydowaliśmy się zadeklarować z wyprzedzeniem funkcję `expression()`.

Czy to działa? Jeśli odpowiednio zdefiniuje się słowo „działa”, można powiedzieć, że tak. Przechodzi kompilację, da się uruchomić, poprawnie oblicza wyniki wyrażeń i zgłasza sensowne komunikaty o błędach. Ale czy działa tak, jak sobie tego życzymy? Nie będzie zaskoczeniem, gdy powiem „nie za bardzo”. Pierwszą wersję wypróbowaliśmy w podrozdziale 6.6. Wówczas usunęliśmy poważny błąd. Druga wersja (podrozdział 6.7) nie jest o wiele lepsza. Ale w porządku, tego się spodziewaliśmy. Program zadowalająco spełnia swoje główne zadanie, czyli pozwala zweryfikować nasze podstawowe pomysły i zorientować się, co robić dalej. Pod tym względem odnieśliśmy sukces, ale spróbuj z niego skorzystać — bez problemu doprowadzi Cię do szału!

### WYPRÓBUJ

Uruchom powyższą wersję kalkulatora i sprawdź, co robi. Spróbuj dojść, dlaczego tak działa.

## Ćwiczenia

Celem tego zestawu ćwiczeń jest poprawienie błędów w programie, aby zamienić go w coś użytecznego.

1. Weź kalkulator z pliku *calculator02buggy.cpp*. Spraw, żeby dał się skompilować. Musisz znaleźć i poprawić kilka błędów. Nie ma ich w tekście książki.
2. Zmień znak polecenia zamknięcia programu na *x*.

3. Zmień znak polecenia drukowania na =.
4. Dodaj do funkcji `main()` komunikat powitalny:

   *„Witaj w naszym prostym kalkulatorze.*
   *W wyrażeniach stosuj liczby zmiennoprzecinkowe."*
5. Dodaj do komunikatu powitalnego informację o tym, jakie operatory są obsługiwane oraz jak drukować wynik i zakończyć działanie programu.
6. Znajdź trzy błędy logiczne w programie, które zostały tam przemyślnie ukryte, i popraw je, aby kalkulator zwracał prawidłowe wyniki.

## Powtórzenie

1. Co rozumiemy pod pojęciem „Programować to zrozumieć"? Wymień trzy główne fazy produkcji oprogramowania.
2. W rozdziale tym został szczegółowo opisany proces tworzenia kalkulatora. Napisz krótką specyfikację wymagań dla takiego programu.
3. W jaki sposób dzieli się problem na mniejsze, łatwiejsze do ogarnięcia części?
4. Dlaczego utworzenie ograniczonej wersji programu jest dobrym pomysłem?
5. Co jest złego w mnożeniu wymagań dotyczących funkcjonalności na początku pracy nad programem?
6. Co to jest „przypadek użycia"?
7. Jaki jest cel przeprowadzania testów?
8. Posiłkując się informacjami zawartymi w rozdziale, opisz różnicę między składnikiem (`Term`), wyrażeniem (`Expression`), liczbą (`Number`) i czynnikiem (`Primary`).
9. Dane wejściowe kalkulatora rozkładaliśmy na następujące elementy: składnik, wyrażenie, czynnik i liczba. Rozłóż w ten sposób wyrażenie (17+4)/(5–1).
10. Dlaczego w programie nie ma funkcji o nazwie `number()`?
11. Co to jest token?
12. Co to jest gramatyka? Co to jest reguła gramatyki?
13. Co to jest klasa? Do czego służą klasy?
14. Co to jest konstruktor?
15. Dlaczego w funkcji `expression()` klauzula `default` instrukcji `switch` wstawia token z powrotem do strumienia?
16. Co to znaczy „wczytać z wyprzedzeniem"?
17. Co robi funkcja `putback()` i dlaczego jest przydatna?
18. Co nastręcza trudności w implementacji operatora `%` (modulo) w funkcji `term()`?
19. Do czego służą dwie zmienne składowe klasy `Token`?
20. Dlaczego czasami składowe klasy dzieli się na publiczne i prywatne?
21. Co dzieje się w klasie `Token_stream`, gdy w buforze jest token i zostanie wywołana funkcja `get()`?
22. Po co zostały dodane znaki `';'` i `'k'` do instrukcji `switch` w funkcji `get()` w klasie `Token_stream`?

23. Kiedy powinno się zacząć testowanie programu?
24. Co to jest „typ zdefiniowany przez użytkownika”? Do czego może się przydać?
25. Co to jest interfejs do „typu zdefiniowanego przez użytkownika” w języku C++?
26. Dlaczego powinno się używać kodu z bibliotek?

## Terminologia

| | | |
|---|---|---|
| analiza | implementacja | przypadek użycia |
| analizator składniowy | interfejs | pseudokod |
| `class` | parser | `public` |
| dzielenie przez zero | `private` | składowa klasy |
| funkcja składowa | projekt | token |
| gramatyka | prototyp | zmienna składowa klasy |

## Praca domowa

1. Jeśli jeszcze tego nie zrobiłeś, rozwiąż wszystkie ćwiczenia *Wypróbuj*.
2. Dodaj możliwość używania w programie zarówno nawiasów okrągłych `()`, jak i klamrowych `{}`, aby można było pisać wyrażenia typu `{(4+5)*6}/(3+4)`.
3. Dodaj operator silni `!` jako operator przyrostkowy. Na przykład `7!` oznacza `7*6*5*4*3*2*1`. Niech operator ten wiąże mocniej niż `*` i `/`. To znaczy, `7*8!` powinno oznaczać `7*(8!)`, a nie `(7*8)!`. Zacznij od dodania operatora wyższego poziomu do gramatyki. Aby pozostać w zgodzie ze standardową matematyczną definicją silni, niech `0!` wynosi `1`.
4. Zdefiniuj klasę `Name_value` przechowującą łańcuch i wartość. Utwórz konstruktor (podobny jak w klasie `Token`). Zmodyfikuj ćwiczenie 19. z rozdziału 4., używając `vector<Name_value>` zamiast dwóch wektorów.
5. Dodaj do gramatyki języka angielskiego z podrozdziału 6.4.1 przedimek **the**, aby można było za jej pomocą opisywać zdania typu „The birds fly but the Fish swim”.
6. Napisz program sprawdzający poprawność zdania zgodnie z gramatyką z podrozdziału 6.4.1. Przyjmij założenie, że każde zdanie kończy się kropką otoczoną białymi znakami, np. **birds fly but the fish swim .** jest zdaniem, a **birds fly but the fish swim** (brak kropki na końcu) i **birds fly but the fish swim.** (brak spacji przed kropką) nie. Dla każdego wpisanego zdania program niech zwraca tylko prostą odpowiedź `Dobrze` lub `Źle`. Wskazówka: nie zawracaj sobie głowy tokenami, wystarczy wczytać dane do łańcucha za pomocą operatora `>>`.
7. Napisz gramatykę dla wyrażeń logicznych. Są one podobne do arytmetycznych, tylko posługują się operatorami `!` (nie), `~` (uzupełnienie), `&` (i), `|` (lub) oraz `^` (lub wyłączające). Operatory `!` i `~` są jednoargumentowe i prefiksowe. Operator `^` ma pierwszeństwo przed `|` (podobnie jak `*` przed `+`), a więc `x|y^z` oznacza `x|(y^z)`, a nie `(x|y)^z`. Operator `&` ma pierwszeństwo przed `^`, a więc `x^y&z` oznacza `x^(y&z)`.
8. Przerób grę „Byki i krowy” z ćwiczenia 12. w rozdziale 5., używając liter zamiast cyfr.
9. Napisz program wczytujący cyfry i składający z nich liczby całkowite. Na przykład `123` zostanie wczytane jako znaki `1`, `2` i `3`. Odpowiedź programu powinna być następująca:

*Dekompozycja liczby 123: liczba setek: 1; liczba dziesiątek: 2; liczba jednostek: 3*. Liczba ma być wysyłana na wyjście jako typ `int`. Obsłuż liczby jedno-, dwu-, trzy- i czterocyfrowe. Wskazówka: aby uzyskać całkowitoliczbową wartość `5` znaku `'5'`, odejmij od niego `'0'`, tzn. `'5'-'0'==5`.

10. Permutacja to uporządkowany podzbiór pewnego zbioru. Wyobraź sobie na przykład, że chcesz zdobyć szyfr do sejfu. Jest 60 możliwych liczb, a kombinacja, której potrzebujesz, składa się z trzech różnych liczb. Jest P(60,3) permutacji dla tej kombinacji, gdzie P definiuje następujący wzór:

$$P(a, b) = \frac{a!}{(a-b)!}$$

We wzorze tym `!` oznacza przyrostkowy operator silni. Na przykład `4!` wynosi `4*3*2*1`. Kombinacje są podobne do permutacji, z tą różnicą, że nie jest w nich ważna kolejność elementów. Gdybyś na przykład robił sobie deser lodowy, chcąc użyć trzech różnych smaków lodów z pięciu dostępnych, nie zależałoby Ci, czy bananowy znajdzie się na wierzchu czy na samym dole, oby gdzieś był. Wzór kombinacji jest następujący:

$$C(a, b) = \frac{P(a, b)}{b!}$$

Zaprojektuj program proszący użytkownika o podanie dwóch liczb, pytający, czy ma obliczyć permutacje czy kombinacje i drukujący wynik. To będzie wymagało podzielenia go na kilka części. Wykonaj analizę opisanych wyżej wymagań. Napisz, co dokładnie program ma robić. Napisz pseudokod i podziel go na części. Ten program powinien mieć wbudowany mechanizm sprawdzania błędów. Spraw, aby dla każdego rodzaju błędnych danych były zwracane odpowiednie komunikaty o błędzie.

## Podsumowanie

Jedną z podstawowych czynności programistycznych jest odpowiednie rozpoznanie danych wejściowych. W taki czy inny sposób musi poradzić sobie z tym problemem każdy program. Do najtrudniejszych zadań należy rozszyfrowanie tego, co wytworzył bezpośrednio człowiek. Na przykład ciągłe problemy sprawia wiele aspektów technologii rozpoznawania głosu. Proste wersje tego problemu, jak nasz kalkulator, można rozwiązać za pomocą gramatyk definiujących wprowadzane dane.