Języki programowania
Nowoczesne techniki programowania

Wykład 4

Witold Dyrka
witold.dyrka@pwr.wroc.pl

26/10/2012

Kolokwium

- I termin
  - piątek, 30/11/2012 (w terminie wykładu)
- II termin
  - środa, 5/12/2012, godz. 7.30 s.314/A-1

Prawa autorskie

Slajdy do wykładu powstały
w oparciu o slajdy Bjarne Stroustrupa
do kursu Foundations of Engineering II (C++)
prowadzonego w Texas A&M University

http://www.stroustrup.com/Programming

Program wykładów

1. Pierwszy program 5/10
2. Obiekty, typy i wartości. Wykonywanie obliczeń 12/10
3. Błędy. 19/10
4. **Tworzenie oprogramowania** **26/10**
5. Techniki pisania funkcji i klas 9/11
6. Strumienie wejścia/wyjścia 16/11
7. Wskaźniki i tablice 23/11

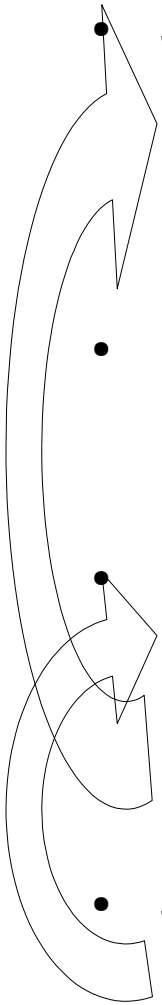
Plan na dziś

- O tworzeniu oprogramowania
- Pomysł na kalkulator
- Zastosowanie gramatyki
- Obliczanie wyrażeń
- Organizacja programu
- Tokeny i strumienie tokenów
- Czyszczenie kodu

Tworzenie programu

- Analiza
  - Lepsze rozumienie problemu
    - z myślą o docelowym zastosowaniu programu
- Projekt
  - Ogólna struktura programu
- Implementacja
  - Kodowanie
  - Debugowanie
  - Testowanie
- Powtarzanie etapów aż do osiągnięcia celu

Strategia pisania programu



- Jaki problem chcesz rozwiązać?
  - Czy problem został jasno postawiony?
  - Czy problem wydaje się do rozwiązania, biorąc pod uwagę czas, umiejętności i dostępne narzędzia?
- Podziel problem na dające się ogarnąć części
  - Czy znasz jakieś narzędzia, biblioteki itp., które mogą być przydatne? (np. iostream, vector)
- Stwórz niewielką, ograniczoną wersję programu, rozwiązującą najważniejszą część problemu
  - Zauważysz braki w rozumieniu, pomysłach, narzędziach
  - Prawdopodobnie zmodyfikujesz specyfikację problemu
- Jeśli taki prototyp nie działa lub jest bardzo brzydki
  - Wyrzuć go i stwórz nowy prototyp – powtarzaj do skutku
- Stwórz pełną wersję programu
  - Najlepiej wykorzystując części prototypu

Pisanie programu – przykład

- Będziemy popełniali typowe błędy
  - Nawet doświadczeni programiści robią mnóstwo błędów
- Projektowanie programu jest trudne
- Skupimy się na sensownym zaprojektowaniu
  - Wykrywanie większości błędów zostawimy kompilatorowi
- Zbudujemy pierwszą, niekompletną wersję
  - To pozwoli przetestować pomysł
  - Dobry program trzeba wyhodować od małego:-)

Prosty kalkulator

- Oblicza wyrażenia wprowadzane z klawiatury i wyświetla wyniki, np.
  - Wyrażenie: 2+2
    - Wynik: 4
  - Wyrażenie: 2+2*3
    - Wynik: 8
  - Wyrażenie: 2+3-25/5
    - Wynik: 0

- Ujmijmy to w pseudokodzie...

Pseudokod prostego kalkulatora

- Pierwszy pomysł:

```
int main()  {
        zmienne                         // pseudokod
        dopóki (dostajesz linię) {      // a co to jest linia?
                analizuj wyrażenie      // co to znaczy?
                oblicz wyrażenie
                wyświetl wynik
        }
}
```

- Jak przedstawić wyrażenie **45+5/7** jako dane programu?
- Jak znaleźć czynniki **45** + **5** / oraz **7** w łańcuchu wejściowym?
- Jak zapewnić, żeby **45+5/7** znaczyło **45+(5/7)**, a nie **(45+5)/7**?
- Czy należy dopuścić liczby zmiennoprzecinkowe? (jasne!)
- Czy należy dopuścić tworzenie zmiennych? **v=7; m=9; v*m** (nie od razu)

Prosty kalkulator

- Czekaj!
  - Czy nie próbujemy wynaleźć koła?
- Programy do wyliczania wyrażeń są tworzone od 50 lat
  - Istnieją gotowe rozwiązania!
- Co proponują specjaliści? Poczytaj, sprawdź!
  - Poczytać i popytać bardziej doświadczonych kolegów będzie przeważnie bardziej efektywne, przyjemne i szybsze niż mozolne kombinowanie po swojemu

Gramatyka wyrażeń arytmetycznych

- Typowym rozwiązaniem problemu kalkulatora wyrażeń jest **gramatyka:**

```
Expression :                    // wyrażenie
        Term
        Expression '+' Term     // np. 1+2,   (1-2)+3,   2*3+1
        Expression '-' Term

Term :                          // składnik
        Primary
        Term '*' Primary        // np. 1*2,   (1-2)*3.5
        Term '/' Primary
        Term '%' Primary

Primary :                       // czynnik
        Number                  // np. 1,   3.5
        '(' Expression ')'      // np.  (1+2*3)

Number :                        // liczba
        literał zmiennoprzecinkowy   // np. 3.14, 0.274e1 lub 42 – as defined for C++
```

Gramatyka języka polskiego

Zdanie:
- Grupa_podmiotu Grupa_orzeczenia
- Grupa_orzeczenia

Grupa_podmiotu:
- Podmiot
- Przydawka Podmiot

Grupa_orzeczenia:
- Orzeczenie
- Orzeczenie Dopełnienie

Przydawka:
- Przymiotnik
- Liczebnik
- Imiesłów_przymiotnikowy

Podmiot:
- Rzeczownik
- Zaimek

Orzeczenie:
- Czasownik

Dopełnienie:
- Rzeczownik

Przymiotnik:
- „skuteczny”, „wybitny”, ...

Rzeczownik:
- „polityk”, „poeta”, „wybory”
- „wiersze”, „sercem”, ...

Czasownik:
- „wygrywa”, „pisze”, …

Zdanie składa się z wyrazów (literałów łańcuchowych), np.

Skuteczny polityk wygrywa wybory

Wybitny poeta pisze wiersze

Wiersze wybory skuteczny - to nie jest poprawne zdanie!

Gramatyka wyrażeń arytmetycznych

- Typowym rozwiązaniem problemu kalkulatora wyrażeń jest **gramatyka:**

```
Expression :                    // wyrażenie
        Term
        Expression '+' Term     // np. 1+2,   (1-2)+3,   2*3+1
        Expression '-' Term

Term :                          // składnik
        Primary
        Term '*' Primary        // np. 1*2,   (1-2)*3.5
        Term '/' Primary
        Term '%' Primary

Primary :                       // czynnik
        Number                  // np. 1,   3.5
        '(' Expression ')'      // np.  (1+2*3)

Number :                        // liczba
        literał zmiennoprzecinkowy   // np. 3.14, 0.274e1 lub 42 – as defined for C++
```

- Wyrażenie składa się z **tokenów** (liczb i operatorów) – są one jakby słownikiem kalkulatora

Gramatyka wyrażeń arytmetycznych

- Reguły gramatyczne

  Expression : // *wyrażenie*
  Term
  Expression ‘+’ Term // *np. 1+2, (1-2)+3, 2*3+1*
  Expression ‘-’ Term

- Można czytać od góry:

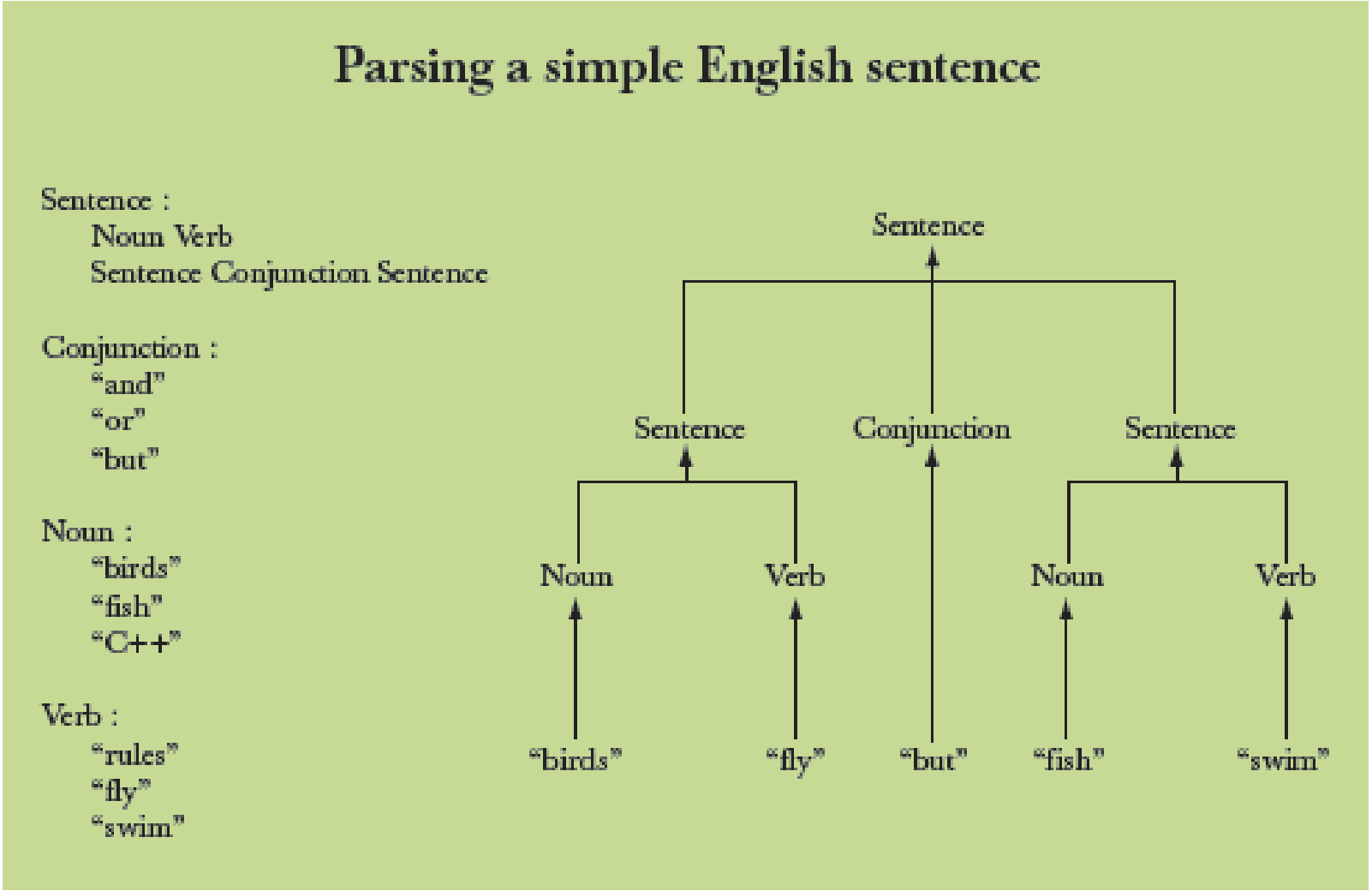
  Expression może składać się z:
  samego **Term**-u
  lub **Expression**, znaku **'+'** i **Term**-u
  lub **Expression**, znaku **'-'** i **Term**-u

- Lub od dołu:

  sam **Term** jest **Expression**
  lub **Expression**, znak **'+'** i **Term** jest **Expression**
  lub **Expression**, znak **'-'** i **Term** jest **Expression**

Gramatyka – język angielski

Parsowanie to przetwarzanie zdania lub wyrażenia zgodnie gramatyką



Parsowanie wyrażeń arytmetycznych

Wyrażenie

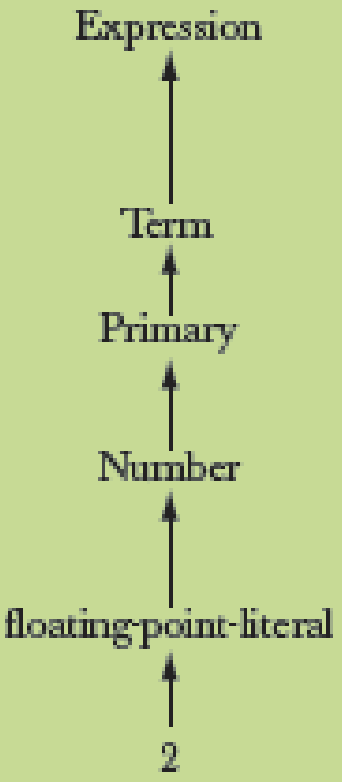
Składnik

Czynnik

Liczba

Parsing the number 2

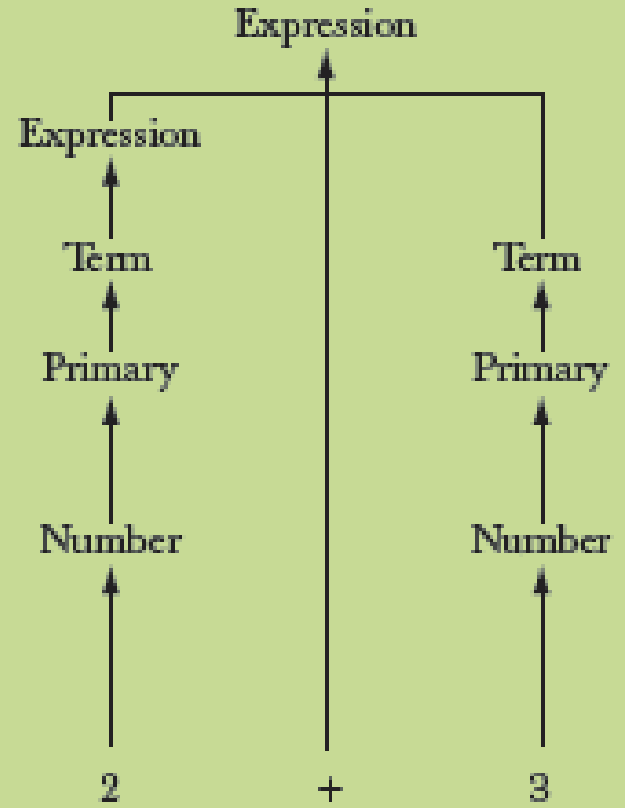
```
Expression:
    Term
    Expression "+" Term
    Expression "–" Term
Term:
    Primary
    Term "*" Primary
    Term "/" Primary
    Term "%" Primary
Primary:
    Number
    "(" Expression ")"
Number:
    floating-point-literal
```



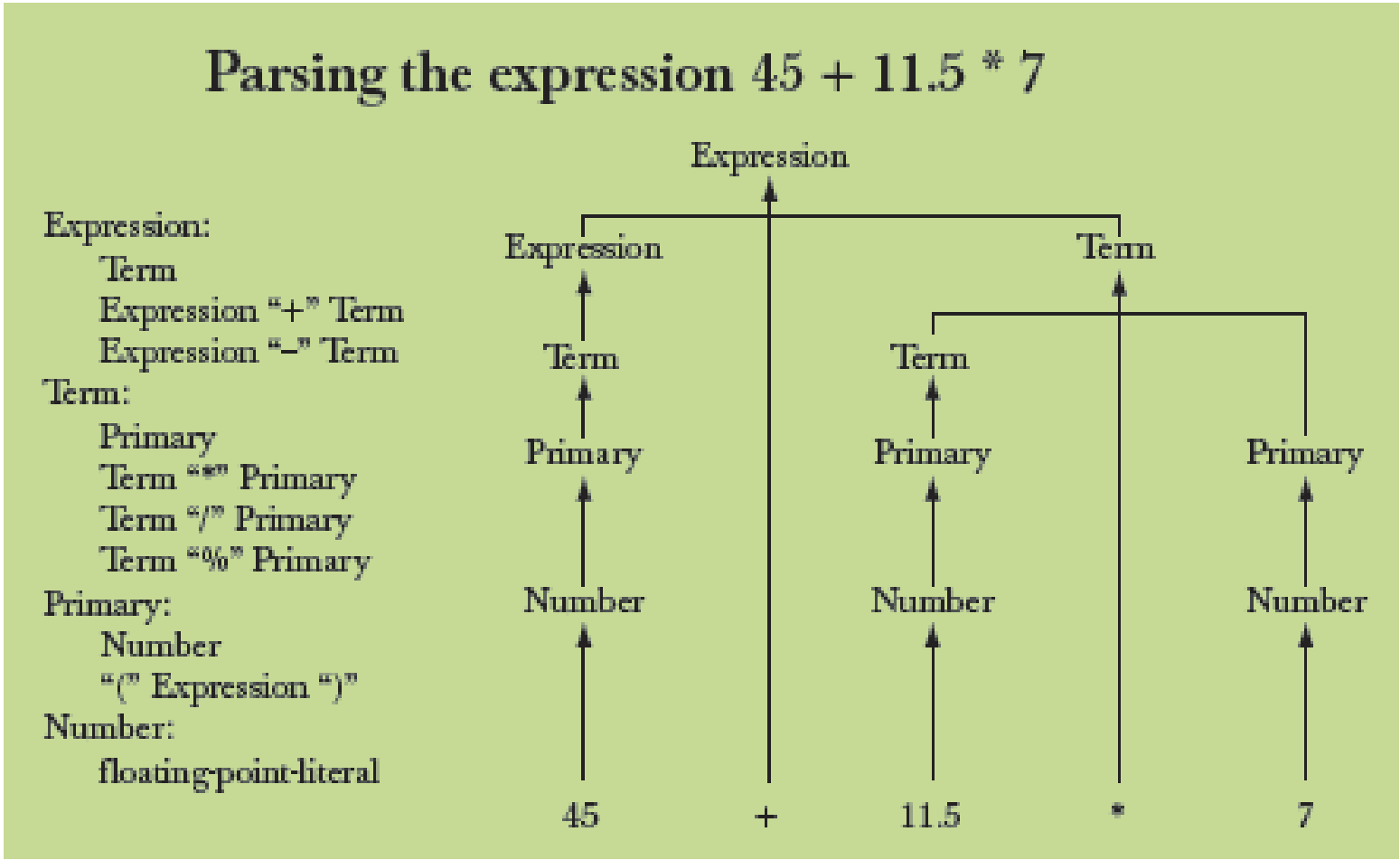
Gramatyka wyrażeń (2)

Parsing the expression 2 + 3

Expression:
    Term
    Expression "+" Term
    Expression "–" Term
Term:
    Primary
    Term "*" Primary
    Term "/" Primary
    Term "%" Primary
Primary:
    Number
    "(" Expression ")"
Number:
    floating-point-literal



Gramatyka wyrażeń (3)



O gramatykach

- Czym jest gramatyka?
  - Zbiorem **reguł**, jak analizować (parsować) wyrażenie (zdanie) składające się z
  - **tokenów** (symboli terminalnych), np. liczba, operator lub wyraz w jęz.naturalnym.
  - W toku analizy składni wprowadzamy **zmienne syntaktyczne** (symbole nieterminalne) wyrażające konstrukcje wyższego rzędu, np.: czynnik, wyrażenie lub rzeczownik, fraza czasownikowa.
- Co ciekawe, badania psycholingwistyczne wskazują, że ludzi rodzą się z pewnym gotowym zestawem reguł w umyśle, zwanym gramatyką generatywną:
  - Wiemy, że „*Ptaki latają, a ryby pływają”* to zdanie poprawne
  - A zdanie „*Latają ptaki ryby, ale pływają”* ma błędną składnię
  - Podobnie, wyrażenie arytmetyczne „2*3+4/2” jest poprawne
  - Natomiast: „2 * + 3 4/2” nie jest poprawne.
- Dlaczego?
- Skąd wiemy?
- Jak nauczyć komputer, tego co wiemy?

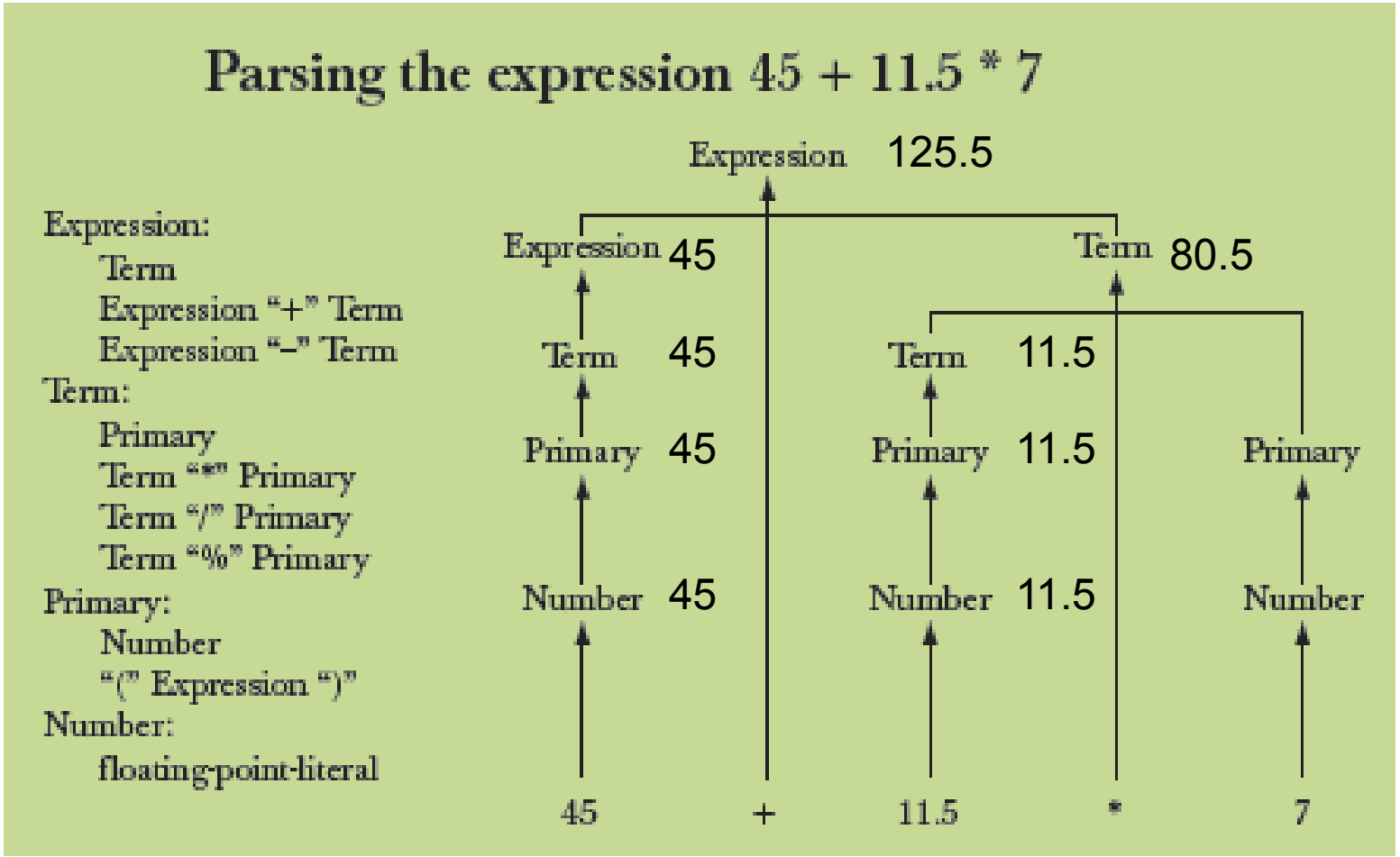
Zabieramy się za projektowanie: funkcje

- Stworzymy **parser** – program przetwarzający zadane wyrażenie przy użyciu gramatyki. Jak?
- Najprościej! Napiszemy osobną funkcję dla każdej zmiennej syntaktycznej oraz funkcję do pobierania tokenów:

```
get_token()     // wczytuje znaki i tworzy tokeny
                // wywołuje cin do czytania z wejścia

expression()    // obsługuje operacje + i –
                // wywołuje term() i get()

term()          // obsługuje operacje *, / i %
                // wywołuje primary() i get()

primary()       // obsługuje operacje liczby i nawiasy
                // wywołuje expression() i get()
```

- **Zauważmy:** Każda funkcja zajmuje się tylko określoną częścią wyrażenia i resztę pozostawia innym funkcjom. To radykalnie upraszcza kod funkcji
- **Analogia:** Zespół specjalistów, w którym każdy zajmuje się tylko tym, na czym się zna najlepiej i przekazuje wynik reszcie

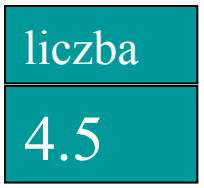
Przekazywanie wartości części wyrażenia



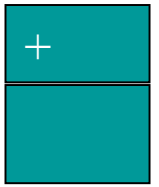
Projektowanie: typy zwracane

```
Token get_token()        // wczytuje znaki i tworzy tokeny
                         // zwraca token (obiekt typy Token)

double expression()      // obsługuje operacje + i –
                         // zwraca sumę lub różnicę – wartość typu zmiennoprzecinkowego

double term()            // obsługuje operacje *, / i %
                         // zwraca iloczyn, iloraz lub modulo

double primary()         // obsługuje operacje liczby i nawiasy
                         // zwraca wartość czynnika
```

- **Uwaga:** funkcje nie pobierają argumentów, ponieważ zakładamy, że wczytują kolejny token z wejścia
- Czym jest **Token**?



Czym jest Token?



- Na wejściu danych z klawiatury dostajemy strumień znaków
  - np. **1 + 4*(4.5-6)** składa się z 13 znaków, w tym 2 spacji
- Ale słownik naszej gramatyki wyrażeń nie składa się z liter i cyfr, tylko z liczb i operatorów, czyli tokenów:
  - mamy tutaj 9 tokenów: **1 + 4 * (4.5 - 6)**
  - w sumie 6 rodzajów tokenów: **liczba + * (-)**
- Token reprezentuje pewną całość, np. liczbę lub operator. W tym celu potrzebuje:
  - rodzaju (*kind*), np. liczba
  - wartości (*value*), np. 4
- Ideę tokena będziemy reprezentowali przez nowy typ **Token**
  - Opiszemy go później, na teraz ważne jest, że:
  - Funkcja **get_token()** daje nam kolejny token na wejściu
  - **t.kind** daje rodzaj tokena **t**, a **t.value** daje nam wartość tokena **t**

expression() – obsługa + i –

Expression:
    Term
    Expression ‘+’ Term   // *Zauważ: każde wyrażenie zaczyna się składnikiem (Term)*
    Expression ‘-’ Term

```
double expression()        // wczytuje i oblicza: 1   1+2.5   1+2+3.14  itp.
{
    double left = term();                    // wczytuje rozpoczynający składnik (term)
    while (true) {
        Token t = get_token();               // wczytuje kolejny token...
        switch (t.kind) {                    // ... i robi z nim co trzeba
        case '+':    left += term(); break;
        case '-':    left -= term(); break;
        default:     return left;            // zwraca wartość wyrażenia
        }
    }
}
```

term() – obsługa *, / i %

Term :
Primary
Term ‘*’ Primary
Term ‘/’ Primary
Term ‘%’ Primary

```
double term()     // wczytuje i oblicza wartość składnika (Term)
                  // – dokładnie jak dla wyrażenia
{
    double left = primary();                  // wczytuje czynnik (Primary)
    while (true) {
        Token t = get_token();                // wczytuje kolejny token...
        switch (t.kind) {
        case '*':    left *= primary(); break;
        case '/':    left /= primary(); break;
        case '%':    left %= primary(); break;
        default:     return left;             // zwraca wartość
        }
    }
}
```

- Ale... to się nie kompiluje! Operacja modulo nie jest zdefiniowana dla typu zmiennoprzecinkowego (patrz wykład 2)

term() – obsługa * i / – poprawione

Term :
    Primary
    Term ‘*’ Primary // *Zauważ: każdy składnik zaczyna się czynnikiem (Primary)*
    Term ‘/’ Primary

```
double term()     // wczytuje i oblicza wartość składnika (Term)
{
    double left = primary();                    // wczytuje czynnik (Primary)
    while (true) {
        Token t = get_token();                  // wczytuje kolejny token...
        switch (t.kind) {
        case '*':    left *= primary(); break;
        case '/':    left /= primary(); break;
        default:     return left;               // zwraca wartość
        }
    }
}
```

term() – obsługa * i /
obsługa dzielenia przez zero

Term :
Primary
Term ‘*’ Primary
Term ‘/’ Primary

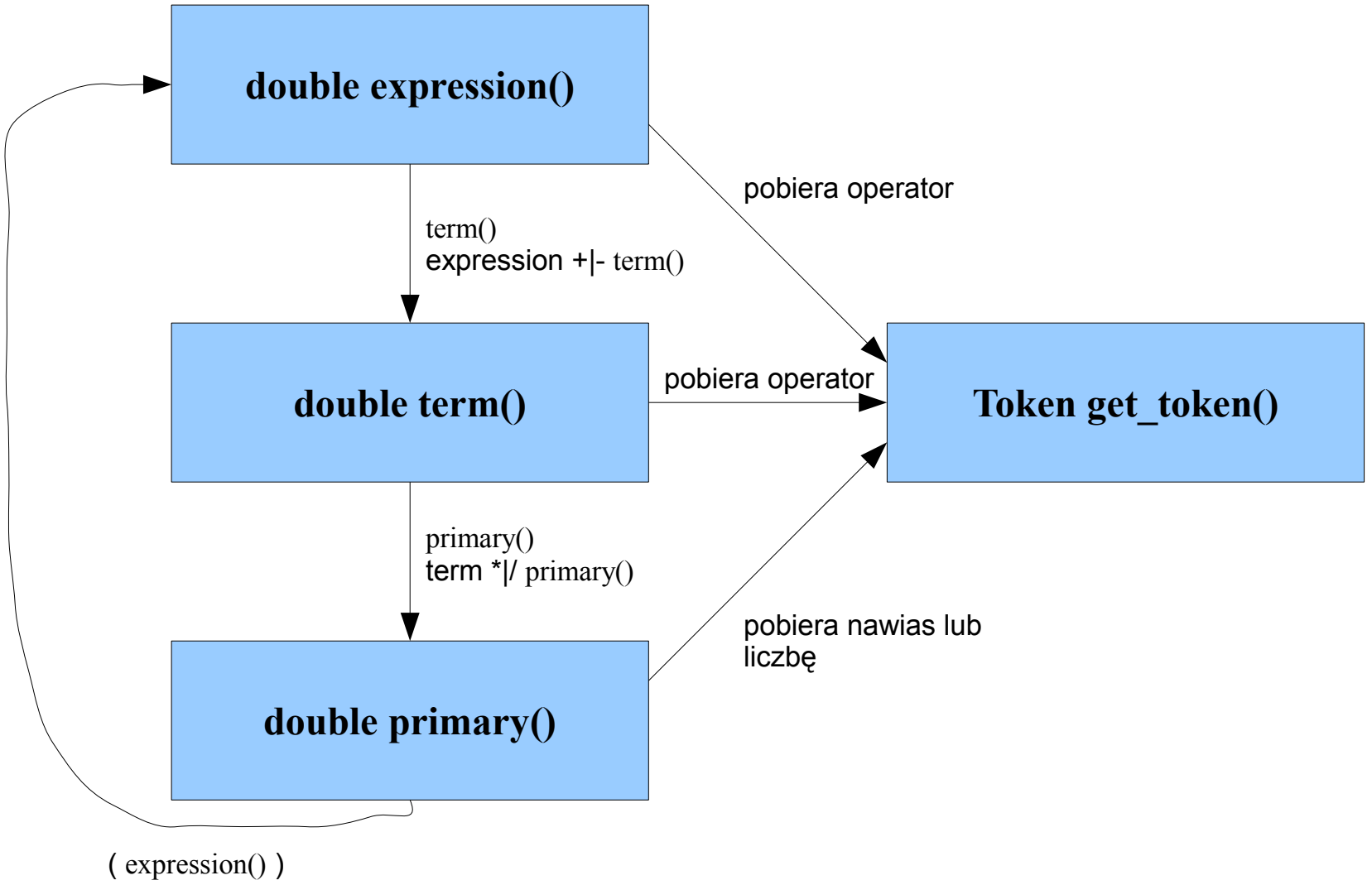
```
double term()       // wczytuje i oblicza wartość składnika (Term)
{
    double left = primary();            // wczytuje czynnik (Primary)
    while (true) {
        Token t = get_token();          // wczytuje kolejny token...
        switch (t.kind) {
        case '*':
                left *= primary();
                break;
        case '/':
        {       double d = primary();
                if  (d==0) throw runtime_error("dzielenie przez zero");
                left /= d;
                break;
        }
        default:
                return left;            // zwraca wartość
        }
    }
}
```

primary()
– obsługa liczb i nawiasów

Primary :
    Number
    ‘(‘ Expression ‘)’

```
double primary()          // liczba lub ‘(‘ wyrażenie ‘)’
{
    Token t = get_token();
    switch (t.kind) {
    case '(':                         // obsługuje ‘(’ wyrażenie ‘)’
    {       double d = expression();
            t = get_token();
            if (t.kind != ')') throw runtime_error("oczekiwano ')'");
            return d;
    }
    case '8':                         // rodzaj tokenu “liczba” oznaczamy znakiem ‘8’
            return t.value;           // zwraca wartośc liczby
    default:
            throw runtime_error("oczekiwano składnika");
    }
}
```

Zależności pomiędzy funkcjami



Program

```
#include<iostream>          // standardowa biblioteka strumieni wejścia-wyjścia(I/O stream)
using namespace std;        // standardowa przestrzeń nazw

// Tu umieścimy kod odpowiadający za obsługę tokenów

double expression();    // deklaracja potrzebna, aby primary() mogło wywołać expression()
double primary() { /* ... */ }      // obsługa liczb i nawiasów
double term() { /* ... */ }         // obsługa mnożenia i dzielenia (problem z modulo)
double expression() { /* ... */ }   // obsługa dodawania i odejmowania

int main() { /* ... */ }            // kod na następnym slajdzie
```

Program – funkcja **main()**

```
int main()
        try {
                while (cin)
                        cout << expression() << '\n';
                system("pause");                        // zatrzymuje okno po zakończeniu programu
                                                        // – potrzebne w niektórych wersjach Windows
        }

        catch (runtime_error& e) {                      // obsługa wyjątków typu runtime_error
                cerr << e.what() << endl;
                system("pause");
                return 1;                               // zwraca kod błędu 1
        }

        catch (…) {                                     // obsługa pozostałych wyjątków
                cerr << "wyjątek \n";
                system("pause");
                return 2;
        }
```

Pierwsza próba...

- Hmm...

```
2       wejście
        wejście
3       wejście
4       wejście
2       WYJŚCIE (zwrócił czynnik, ale z opóźnieniem)
5+6     wejście
5       WYJŚCIE (zwrócił tylko pierwszy czynnik)
X       wejście
zly token   WYJŚCIE (ok, tego się spodziewaliśmy)
Aby kontynuować, naciśnij dowolny klawisz . . .
```

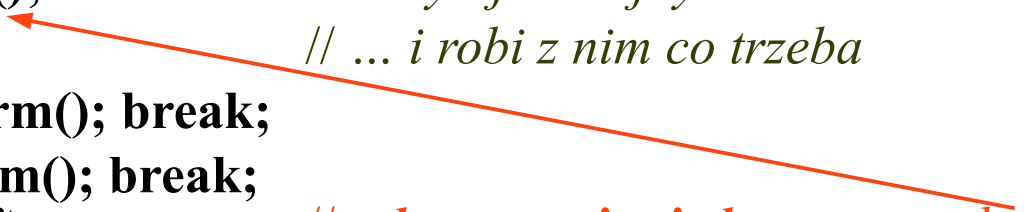
Testujemy dalej...

- Ha, wygląda na to, że program zjada 2 na 3 tokeny
  - Jak to możliwe?

expression() – obsługa + i –

Expression:
    Term
    Expression ‘+’ Term   *// Zauważ: każde wyrażenie zaczyna się składnikiem (Term)*
    Expression ‘-’ Term

```
double expression()        // wczytuje i oblicza: 1   1+2.5   1+2+3.14  itp.
{
    double left = term();                       // wczytuje rozpoczynający składnik (term)
    while (true) {
        Token t = get_token();                  // wczytuje kolejny token...
        switch (t.kind) {                       // ... i robi z nim co trzeba
        case '+':    left += term(); break;
        case '-':    left -= term(); break;
        default:     return left;               // token zostaje zjedzony: został pobrany,
                                                // nie został użyty i nie został zwrócony

        }
    }
}
```

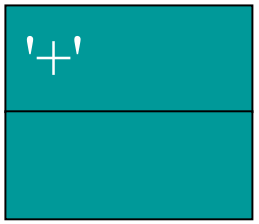


expression() – obsługa + i –

- Musimy odłożyć nieużyty token z powrotem

```
double expression()        // wczytuje i oblicza: 1   1+2.5   1+2+3.14  itp.
{
    double left = term();                    // wczytuje rozpoczynający składnik (term)
    while (true) {
        Token t = get_token();               // wczytuje kolejny token...
        switch (t.kind) {                    // ... i robi z nim co trzeba
        case '+':    left += term(); break;
        case '-':    left -= term(); break;
        default: putback_token(t);
                 return left;
        }
    }
}
```

- Ale gdzie odłożyć?
- Problem: do standardowego strumienia wejścia można odkładać tylko pojedyńcze znaki, to niewygodne. Przydałby się **strumień tokenów**



Typ użytkownika: Token


```
struct Token {                  // definiuje typ o nazwie Token
                                // Token składa się on z rodzaju oraz wartości:
    char kind;                  // rodzaj tokena
    double value;               // wartość tokena (używana tylko dla rodzaju liczba)

    // ...

};


Token t;                        // deklaracja zmiennej t typu Token

t.kind = '8';                   // . (kropka) jest używana do dostępu do składowych typu
                                // (używamy '8' jako oznaczenie typu "liczba")

t.value = 2.3;

Token u = t;                    // typ Token zachowuje się podobnie jak typy wbudowane,
                                // np. int. Tutaj u staje się kopią t

cout << u.value;                // ta instrukcja wydrukuje 2.3
```

Typy użytkownika: struktury i klasy

```
struct Token {      // typ użytkownika o nazwie Token
      // dane składowe
      // funkcje składowe
};
```

```
class Token {       // typ użytkownika o nazwie Token
      // dane składowe
      // funkcje składowe
};
```

- W C++ typ użytkownika nazywamy klasą (*class*)
- Klasy mogą mieć dwie bliźniacze formy: **struct** i **class**
  – o różnicy między nimi powiemy na następnym wykładzie.
- Definiowanie typów użytkownika jest kluczowym mechanizmem dostępnym w większości współczesnych języków programowania
- Klasy mają dwa typy składowych:
  - dane – przechowujące informacje
  - funkcje – zapewniające operacje na danych

Inicjowanie obiektu typu użytkownika – konstruktor Tokena

```
struct Token {
    char kind;          // rodzaj tokena
    double value;       // dla liczb: wartość

     Token(char ch) : kind(ch), value(0) { }                  // konstruktor
     Token(char ch, double val) : kind(ch), value(val) { }    // konstruktor

};
```

- Tworząc własny typ danych potrzebujemy określić sposób inicjowania obiektów tego typu: służy do tego **konstruktor**
- Konstruktor zawsze nazywa się tak samo jak jego klasa
- W przypadku **Tokena**, kind jest inicjowany przez **ch**, a **value** przez **val** lub **0**:

```
Token t1 = Token('+');       // tworzy Token rodzaju '+'
Token t2 = Token('8',4.5);   // tworzy Token rodzaju '8' i wartości 4.5
```

Strumień tokenów: klasa **Token_stream**

Czego potrzebujemy do obsługi tokenów?

- Funkcja 1: Wczytywanie znaków z wejścia i tworzenie tokenów
- Funkcja 2: Odkładanie tokenu na później
- W tym celu potrzebujemy bufora, w którym będziemy przechowywać tokeny

strumień WE: 1+2*3;

bufor: pusty

- Dla **1+2*3;**, **expression()** wywołuje **term()**, który wczyta **1**, a następnie '+', po czym zdecyduje, że '+' to robota dla „kogoś innego” i odłoży **Token('+')** do strumienia tokenów (tam znajdzie go **expression()**)

strumień WE: 2*3;

bufor: Token('+')

Projekt klasy **Token_stream**

- Funkcja 1: Wczytywanie znaków z wejścia i tworzenie tokenów
- Funkcja 2: Odkładanie tokenu na później
- Składowa: Bufor, w którym będziemy przechowywać odłożony token

```
class Token_stream {
    // reprezentacja danych nie jest bezpośrednio dostępna dla użytkownika
    Token buffer;     // bufor przechowujący Token odłożony przy użyciu putback()
    bool full;        // czy w buforze jest Token?
public:  // interfejs użytkownika klasy:
    Token get();              // pobierz Token (funkcja 1)
    void putback(Token);      // odłóż Token do strumienia Token_stream (fun.2)
    Token_stream();           // konstruktor: tworzy Token_stream
};
```

Implementacja **Token_stream**

- Konstruktor
  - definiuje sposób inicjacji obiektu klasy
  - ma taką samą nazwę jak klasa i nie ma typu zwrotnego

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

- Funkcja odkładania tokena

```
void Token_stream::putback(Token t)
{
    if (full) throw runtime_error("Odkładanie do pełnego bufora");
    buffer=t;
    full=true;
}
```

Implementacja **Token_stream** (2)

```
Token Token_stream::get()  // wczytuje Token ze strumienia tokenów
{
    if (full) { full=false; return buffer; }  // sprawdza, czy w buforze jest token

    char ch;
    cin >> ch;        // zauważ, że >> pomija białe znaki (spacja, nowa linia, tabulator itp.)

    switch (ch) {
    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' oznacza "liczbę"
    }
    default:
            throw runtime_error("Zly token");
    }
}
```

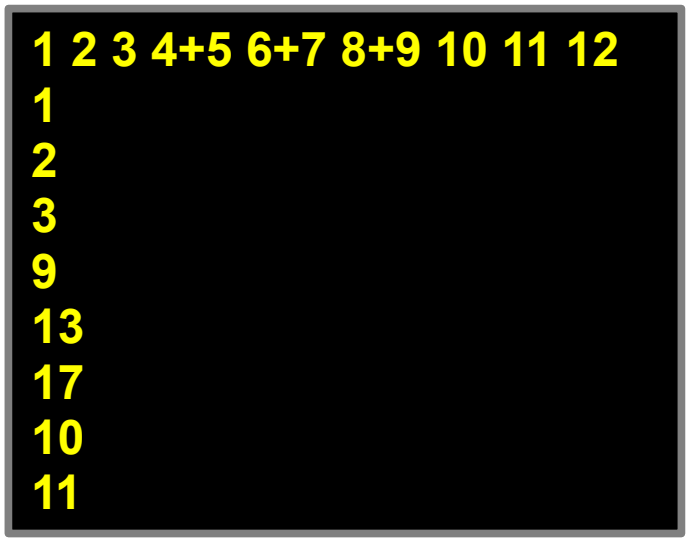
Teraz możemy poprawić program: **expression()** – obsługa + i –

- Musimy odłożyć nieużyty token z powrotem

```
double expression()        // wczytuje i oblicza: 1   1+2.5   1+2+3.14  itp.
{
    double left = term();              // wczytuje rozpoczynający składnik (term)
    while (true) {
        Token t = ts.get();            // wczytuje kolejny token ze strumienia tokenów ts...
        switch (t.kind) {              // ... i robi z nim co trzeba
        case '+':    left += term(); break;
        case '-':    left -= term(); break;
        default: ts.putback(t);    // odkłada token t do strumienia tokenów ts...
                 return left;
        }
    }
}
```

- Analogiczne zmiany wprowadzamy w funkcji **term().**
- Zmieniamy także funkcję **get_token()** na metodę **ts.get()** w **primary()**.

Uff... Jakoś działa:-)

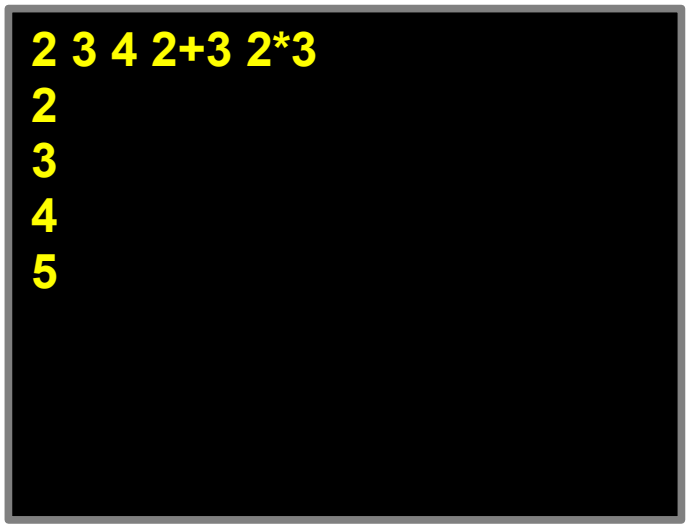


- Całkiem nieźle nam poszło. Ale „jakoś działa” to za mało!
- W zasadzie dopiero teraz zaczyna się praca

  i zabawa:-)

- Ale... Gdzie podziała się **12**?

Kolejny test potwierdza...



- Nasz program wczytuje jeden token do przodu
- Najłatwiej będzie rozwiązać problem dodając instrukcję drukowania na ekranie, np. **';'**
- Przy okazji dodamy instrukcję wyjścia z programu, np. **'q'**

Nowa wersja funkcji **main()**

```
int main()

try {

    double val = 0;

    while (cin) {
        Token t = ts.get();
        if (t.kind == 'q') break;              // ‘q’ jak “quit”
        if (t.kind == ';')                     // ‘;’ oznacza “drukuj teraz”
            cout <<  val << '\n';              // drukuj wynik
        else {
            ts. putback(t);     // odłóż token do strumienia tokenów
            val = expression();            // oblicz
        }

    }
     system("pause");

}

// ... obsługa wyjątków
```

Nowa wersja **Token_stream::get()**

```
Token Token_stream::get()  {
// ....
    switch (ch) {
    case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '/':
        return Token(ch);
// ...
}
```

Sukces!!

```
2; 3; 4; 2+3; 2*3; q
2
3
4
5
6
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Program jest wciąż prymitywny (1)

- Poprawimy go w kolejnych etapach rozwoju
  - Funkcjonalność
    - **Lepszy interfejs użytkownika kalkulatora**
    - Odzyskiwanie sprawności po błędach
    - Liczby ujemne
    - % (reszta z dzielenia/modulo)
    - Predefiniowane wartości symboliczne (np. pi, e itp.)
    - Zmienne

Interfejs użytkownika

- Początkowo chcieliśmy czegoś takiego:

  Wyrażenie: 2+3; 5*7; 2+9;
  Wynik : 5
  Wyrażenie: Wynik: 35
  Wyrażenie: Wynik: 11
  Wyrażenie:

- Ale zaimplementowaliśmy to tak:

  2+3; 5*7; 2+9;
  5
  35
  11

- Czego naprawdę potrzebujemy? Może tak?

  > 2+3;
  = 5
  > 5*7;
  = 35
  >

Wersja **main()** z interfejsem

```
int main()

try {
    double val = 0;
    cout << "> ";                          // drukuje zaproszenie
    while (cin) {
        Token t = ts.get();
        if (t.kind == 'q') break;          // ‘q’ jak “quit”
        if (t.kind == ';')                 // ‘;’ oznacza “drukuj teraz”
            cout << "= " << val << "\n> ";     // drukuje “= wynik” i zaproszenie
        else {
            ts. putback(t);                // odłóż token do strumienia tokenów
            val = expression();            // oblicz
        }
    }
     system("pause");
}
// ... obsługa wyjątków
```

Drobne poprawki

- Problem: wpisanie ”>1+2;;;” wyświetla trzy razy “= 3”. Sprawdź
- Poprawka i ulepszenie struktury kodu:

```
int main()
try {
    cout << "> ";
    while (cin) {
        Token t = ts.get();
        while (t.kind == ';') t=ts.get();    // zjedz wszystkie ';'
        if (t.kind == 'q') {
            system("pause");
            return 0;
        }
        ts.putback(t);                       // skończył jeść ';' na nie-';', niech go zwróci
        cout << "= " << expression() << "\n > ";
    }
}
// ... obsługa wyjątków
```

Program jest wciąż prymitywny (2)

- Poprawimy go w kolejnych etapach rozwoju
  - Styl – jasność kodu
    - Przeczytaj dokładnie i systematycznie cały kod
    - Popraw komentarze (czy są wciąż adekwatne, czy ktoś inny je zrozumie, czy są zwięzłe?)
    - Zmień niejasne nazwy klas, zmiennych i funkcji na znaczące
    - Ulepsz użycie funkcji (czy wciąż odzwierciedlają strukturę programu? Czy nie mieszamy logicznie rozdzielnych zadań – patrz nasza funkcja **main()**)
    - **Pozbądź się magicznych stałych (np. '8' w naszym kodzie)**
    - Daj sprawdzić kod koledze

O stałych magicznych

- Wbrew pozorom ludzie nie wszyscy muszą pamiętać, co znaczą np. 3.14159265358979323846264, 12, -1, 365, 24, 2.7182818284590, 299792458, 2.54, 1.61, -273.15, 6.6260693e-34, 0.5291772108e-10, 6.0221415e23 oraz 42!
- Ponoć stałe magiczne powodowały już katastrofy statków kosmicznych
  - na pewno powodują niewyspanie
- **Jeżeli** „stała” może ulec zmianie w czasie pielęgnacji programu lub ktoś może jej nie rozpoznać – użyj stałej symbolicznej
  - np. zmiana precyzji ma znaczenie: 3.14 != 3.14159265
  - 0 i 1 są z reguły dobre bez wyjaśnień, -1 i 2 tylko czasami (ale rzadko)
  - 12 może być w porządku jako liczba miesięcy, poza tym raczej nie
- **Jeżeli** stała jest używana co najmniej 2 razy – prawdopodobnie trzeba użyć stałej symbolicznej
  - w razie zmiany jej wartości – wystarczy zrobić je w jednym miejscu

Definiowanie stałych symbolicznych

```
// Rodzaje tokenów:
const char number = '8';        // liczba zmiennoprzecinkowa
const char quit = 'q';          // polecenie zakończenia
const char print = ';';         // polecenie drukownia

// Łańcuchy interfejsu użytkownika:
const string prompt = "> ";
const string result = "= ";
```

Usuwanie stałych magicznych (1)

// W ***Token_stream::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);
      double val;
      cin >> val;
      return Token(number,val); // zamiast Token('8',val)
    }
```

// W ***primary()****:*

```
case number:            // zamiast case '8':
      return t.value;
```

Usuwanie stałych magicznych (2)

*// W **main()**:*

```
while (cin) {
        cout << prompt;                     // zamiast "> "
        Token t = ts.get();
        while (t.kind ==print) t=ts.get();  // zamiast ==';'
        if (t.kind == quit) {               // zamiast =='q'
                system("pause");
                return 0;
        }
        ts.putback(t);
        cout << result << expression() << endl;
}
```

Dziś najważniejsze było to, że...

- Nie wymyślamy koła – szukamy istniejących rozwiązań problemów
- Dzielenie programu na logicznie odrębne części, czyli funkcje – bardzo ułatwia pracę
- Definiowanie typów użytkownika jest kluczową techniką nowoczesnego programowania
- Gdy program jakoś działa
  – to dopiero początek zabawy:-)

Ważne

- Język programowania – jak każdy język ma swoją gramatykę
- Parsowaniem naszych programów zajmuje się kompilator
- Dziś stworzyliśmy nie tylko prosty kalkulator, ale i prosty parser – interpreter języka wyrażeń
- W przyszłym semestrze będziecie Państwo pisać programy w nieco bardziej dla nieco bardziej rozwiniętego interpretera wyrażeń matematycznych – MATLABa :-)

A za 2 tygodnie...

- Techniki pisania funkcji i klas