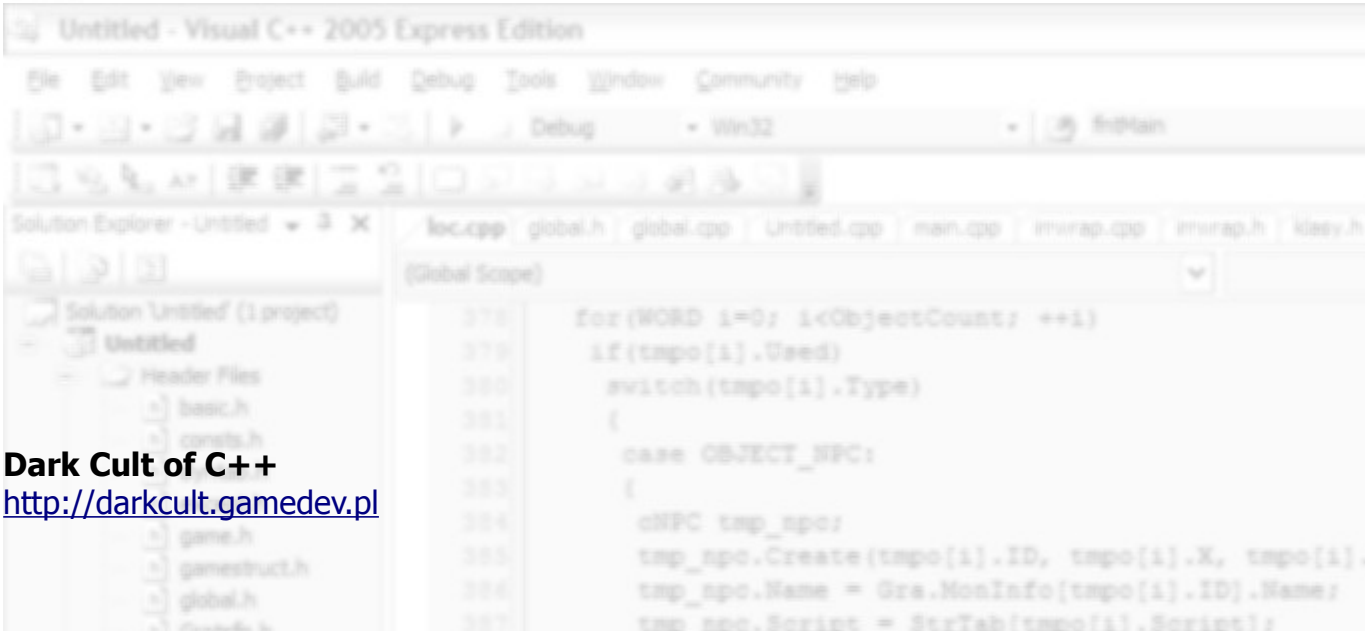
Piotr Bednaruk (Złośliwiec)

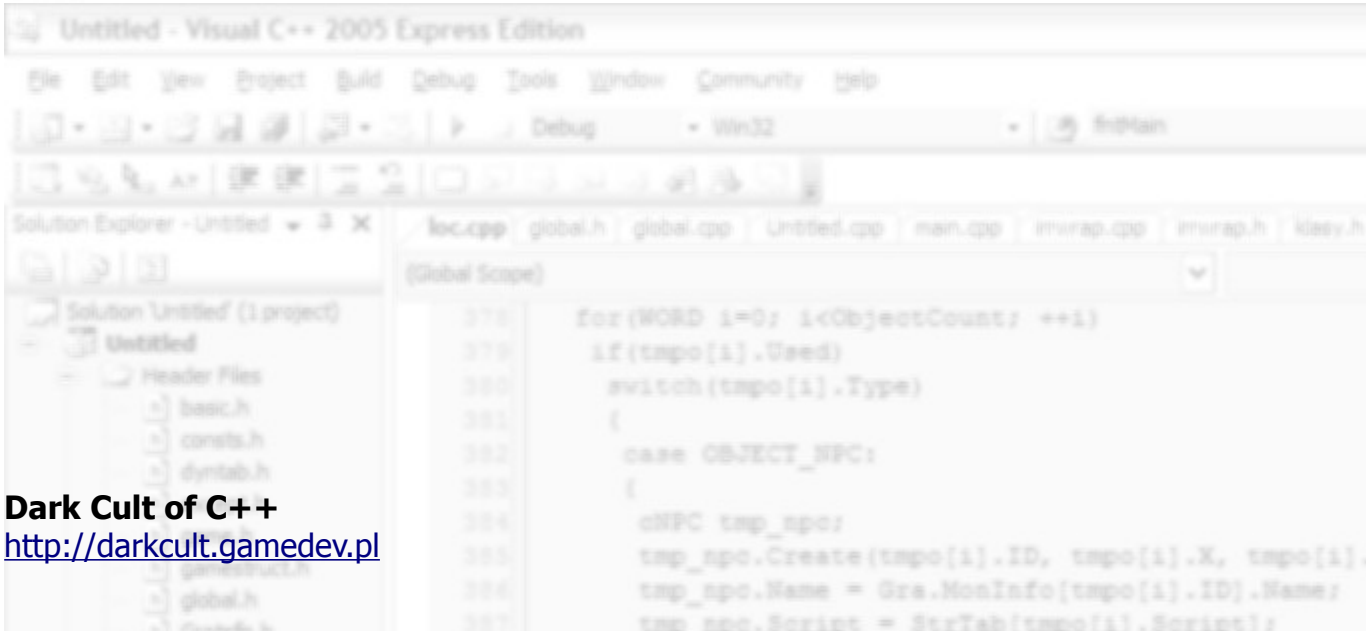
Własny język skryptowy



Dark Cult of C++
http://darkcult.gamedev.pl

Wszystkie znaki występujące w tekście są tylko znakami. Autorzy dokumentu, niestety, kompletnie nie przejmują się magicznym znaczeniom, przypisywanym symbolom „®” czy „™”.

Autorzy dołożyli wszelkich starań, aby zawarte w tej publikacji informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych i autorskich. Ani za cokolwiek.



Dark Cult of C++
http://darkcult.gamedev.pl

Część 1 – proste polecenia

Wstęp

Przy tworzeniu bardziej rozbudowanych gier, bez względu na ich gatunek, dochodzi się zawsze do momentu, że pojawia się potrzeba programowania jakiejś części gry w języku skryptowym. Częścią tą może być sztuczna inteligencja, interakcja różnych obiektów z graczem, definicje tych obiektów, dialogi z postaciami, a nawet cały tak zwany gameplay.

Wspomniana potrzeba wynika z różnych faktów. Po pierwsze, język interpretowany (a w takim z reguły tworzone są skrypty[1]) ma tę przewagę nad kompilowanym, że nie wymaga obecności środowiska programistycznego, aby można było dokonać zmian w kodzie. Wystarczy zwykle edytować plik skryptu w dowolnym edytorze tekstowym, a następnie ponownie uruchomić grę i już dokonane zmiany stają się widoczne dla użytkownika. Po drugie, język skryptowy jest znacznie prostszy (i "bardziej wysokopoziomowy") od tego, w którym napisana jest sama gra. Dzięki temu podobne efekty osiąga się w nim dużo szybciej, jest też mniej możliwości zrobienia różnego rodzaju błędów, które mogą zastopować całą produkcję. Po trzecie, język skryptowy jest zwykle językiem wyspecjalizowanym - stworzony został wyłącznie na potrzeby danej gry i do wykonywania specyficznych dla niej zadań. Tak więc w przeciwieństwie do języków uniwersalnych w rodzaju C++, zaimplementowanie czegokolwiek przy pomocy języka skryptowego wymaga zwykle jednej lub kilku instrukcji w jednym miejscu jednego pliku źródłowego (żeby zaimplementować w grze coś nowego przy użyciu C++, prawdopodobnie musielibyśmy dokonać wielu zmian w różnych miejscach kodu)[2].

Wykorzystywanie języków skryptowych posiada zapewne jeszcze wiele innych zalet. Jednak wymienianie ich wszystkich nie jest celem tego artykułu. Ważne jest uświadomienie sobie faktu, że skrypty nie są bynajmniej sztuką dla sztuki i mogą proces tworzenia gry bardzo ułatwić.

Warstwowy model języka

A więc skrypty są dobre. Jak więc zabrać się za ich zaimplementowanie w naszej grze? Aby łatwiej było zrozumieć ideę języka skryptowego (i w ogóle języka programowania), wymyśliłem sobie taki oto warstwowy model działania skryptu:

- warstwa wejścia
- warstwa parsingu
- warstwa interpretacji
- warstwa wykonania

Omówimy sobie teraz po kolei te wszystkie warstwy, a następnie w takiej samej kolejności spróbujemy zbudować jakiś prosty język skryptowy.

Warstwa wejścia

Na warstwę wejścia składa się wszystko to, co związane jest z wprowadzaniem przez użytkownika tekstu skryptu. Gdybyśmy tworzyli specjalny edytor skryptów jako oddzielną aplikację, wówczas aplikacja ta w całości kwalifikowałaby się do tej warstwy. Gdybyśmy z kolei zajmowali się tworzeniem w grze konsoli (okienko, wywoływane zwykle tyldą i służące do wprowadzania różnych poleceń w czasie gry[3]), to duża część kodu obsługującego ową konsolę weszłaby w skład warstwy wejścia. Niniejszy artykuł bazuje jednak przede wszystkim na przykładzie, w którym skrypty wczytywane są z plików. Wówczas warstwa

wejściowa obejmuje właściwie tylko proces wczytania pliku do bufora i nic więcej - nie ma zatem potrzeby rozpisywać się o niej zbytnio.

Warstwa parsingu

Przy tej warstwie zatrzymamy się nieco dłużej. Parsing to wstępna analiza i przetworzenie tekstu skryptu w taki sposób, aby z postaci czytelnej dla człowieka stał się on przystępny dla komputera. Można to porównać do trawienia pokarmu - powstaje taka papka, którą można poddać dalszej "obróbce" :-). Zazwyczaj parsing polega po prostu na podziale całego skryptu na mniejsze fragmenty (zwane czasem leksemami, tokenami lub noszące inne dziwne nazwy). Jeśli na przykład nasz skrypt ma być po prostu ciągiem poleceń o identycznej składni, umieszczonych każde w osobnym wierszu pliku, to parser będzie się zajmował wyszukiwaniem znaków nowego wiersza, decydowaniem, gdzie kończy się jedno polecenie, a gdzie zaczyna się następne, wyszukiwaniem ewentualnych odstępów między nazwą polecenia a jego parametrami itp. Wreszcie będzie przekazywał poszczególne "zdania" (polecenia wraz z parametrami) do interpretacji.

Warstwa interpretacji

Interpretacja jest procesem ściśle powiązanym z parsingiem. Najczęściej parsing i interpretacja zachodzą naprzemiennie - parser "wyciąga" ze skryptu kolejne polecenie i wywołuje interpreter, aby ten wykonał resztę roboty. Ta reszta to określenie, czym dokładnie jest przetwarzane polecenie. Jeśli na przykład parser pobrał ze skryptu ciąg znaków "stwórz_potwora 17", to zadaniem interpretera jest rozpoznanie tego ciągu jako znanego grze polecenia `CMD_CREATE_MONSTER` i przypisanie mu parametru liczbowego 17. Dane te (kod polecenia i parametry) mogą być następnie zapisane w odpowiedniej strukturze lub wykonane bezpośrednio. Często trudno jest odróżnić warstwę parsingu od warstwy interpretacji, ponieważ mogą się one znajdować w tym samym miejscu w kodzie gry, a nawet wykonywać za siebie nawzajem swoje zadania - wszystko zależy od specyfiki konkretnego języka skryptowego.

Warstwa wykonania

Gdy interpreter określi już, o jakim wewnętrznym poleceniu traktuje dany fragment skryptu, pozostaje tylko właściwe wykonanie polecenia. Nawiązując do przykładu z tworzeniem potwora, wykonaniem może być wywołanie funkcji `CreateMonster` z odpowiednim parametrem, która alokuje pamięć dla nowego potworka, zainicjalizuje go odpowiednimi danymi itd.:

```
CreateMonster(17);
```

Oczywiście treść funkcji `CreateMonster` nas w tym momencie nie interesuje - zajmujemy się samymi skryptami, od momentu ich wpisania przez użytkownika po moment wywołania funkcji w rodzaju naszego `CreateMonster` - nie dalej.

Przykład

Dość się już chyba nagadaliśmy, przydałoby się wreszcie zacząć działać. Na dobry początek stworzymy trywialny język skryptowy, umożliwiający wprowadzanie poleceń w rodzaju omawianego wyżej "stwórz_potwora". Polecenia, jak już powiedzieliśmy, będą wczytywane z pliku. Oto przykład takiego pliku:

```
// stworzenie orka
stwórz_potwora 17 wredny_ork 5 5
```

```
// stworzenie drugiego orka
stwórz_potwora 17 drugi_ork 6 6
// stworzenie kilku przedmiotów
stwórz_przedmiot 2 5 10
stwórz_przedmiot 2 7 9
stwórz_przedmiot 2 11 10
// wyposażenie orka w toporek
wyposaż_potwora wredny_ork 20
```

Na pierwszy rzut oka widać prostą strukturę naszego języka. Każde polecenie jest w osobnym wierszu, a składa się z nazwy polecenia (pierwszy wyraz w wierszu) oraz określonej liczby parametrów liczbowych lub tekstowych. Zarówno nazwa polecenia, jak i poszczególne parametry oddzielane są od siebie spacjami. Niektóre wiersze zawierają komentarze - będą one ignorowane podczas interpretacji.

Zacznijmy od wczytania pliku do bufora. Buforem tym będzie zmienna typu `string`, a do wczytywania wykorzystamy strumienie:

```
#include <cstdio>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

string bufor;

int main()
{
        ifstream plik;
        plik.open("skrypt.txt");
        char c;
        while(true)
        {
                c = plik.get();
                if(plik.eof()) break;
                bufor += c;
        }
        plik.close();
}
```

Nie jest to może najwydajniejsza metoda wczytywania plików, ale też nie jest to tematem tego artykułu. Pora przejść do warstwy drugiej, czyli parsingu. Skonstruowanie jej nie jest proste, ale można je sobie znacznie ułatwić, tworząc najpierw kilka pomocniczych funkcji. Bufor z tekstem skryptu jest zwykłym stringiem, tak więc parsing będzie polegał na iterowaniu przez kolejne znaki tego stringa. Przyda nam się do tego zmienna, która będzie nam pokazywała aktualną pozycję w stringu. Nazwiemy tę zmienną `pos`. Przyglądając się przykładowemu skryptowi dochodzimy do wniosku, że pierwsza funkcja pomocnicza może po prostu pobierać pojedyncze słowa z bufora i oczywiście odpowiednio zwiększać "wskaźnik" `pos`.

Funkcję taką możemy oczywiście napisać "od zera", ale można prościej: mamy przecież STL. Klasa `string` zawiera mnóstwo przydatnych metod, a wśród nich takie, jak np. `find_first_of`, `find_first_not_of`, `find_last_of`... Wszystkie one pobierają jako argument zestawy znaków, co czyni je idealnymi murzynami do roboty, którą właśnie wykonujemy :-). Dzięki nim nie musimy się martwić: czy osoba pisząca nasz skrypt rozdziela poszczególne słowa spacjami, czy też woli tabulatory, a może od czasu do czasu zdarza jej się zostawić pusty wiersz między dwoma poleceniami? Możemy napisać parser tak, aby "łykał" wszystko - i powinniśmy tak pisać. Wystarczy stworzyć (najlepiej jako stałą) zestaw tzw. białych spacji (ang. *whitespaces*), czyli wszelkich znaków, które mogą posłużyć do rozdzielania wyrazów, a nie są widoczne w edytorze tekstu:

```
const char* WS_SET = " \t\r\n";
const char* BR_SET = "\r\n";

#define NOT_FOUND string::npos
#define POS_END string::npos

string bufor, cword, err_str;

int pos = 0, param = 0;
```

Przy okazji stworzyliśmy jeszcze dwa przydatne synonimy, które zwiększą nieco czytelność naszego kodu, a także zadeklarowaliśmy kilka zmiennych globalnych obok istniejącej już wcześniej zmiennej bufor. Teraz nasz murzynek:

```
bool get_token()
{
     int tmp_pos, tmp_pos2;

     tmp_pos = bufor.find_first_not_of(WS_SET, pos);
     if(tmp_pos == NOT_FOUND)
     {
          cword.clear();
          pos = POS_END;
          return false;
     }
     tmp_pos2 = bufor.find_first_of(WS_SET, tmp_pos);
     if(tmp_pos2 == NOT_FOUND)
     {
          pos = bufor.length()-1;
          cword = bufor.substr(tmp_pos, bufor.length()-tmp_pos);
     }
     else
     {
          pos = tmp_pos2;
          cword = bufor.substr(tmp_pos, tmp_pos2-tmp_pos);
     }

     return true;

}
```

Była to chyba najważniejsza funkcja całego naszego systemu skryptowego :-). Teraz druga, również dość istotna, ale już dużo mniej (przynajmniej w tej części artykułu) - będzie ona pobierała liczbowy parametr polecenia ze skryptu i zapisywała go w zmiennej `param`. Konstrukcja tej funkcji będzie podobna do `get_token`, ale będzie jeszcze dodatkowo wykonywała jedno zadanie - konwersję liczby zawartej w stringu do typu `int`:

```
bool get_param()
{
          int tmp_pos, tmp_pos2;
          string s_number;

          tmp_pos = bufor.find_first_not_of(WS_SET, pos);
          if(tmp_pos == NOT_FOUND)
          {
               param = 0;
               pos = POS_END;
               return false;
          }
          tmp_pos2 = bufor.find_first_of(WS_SET, tmp_pos);
          cword.clear();
          if(tmp_pos2 == NOT_FOUND)
          {
               tmp_pos2 = bufor.find_last_not_of(WS_SET, tmp_pos);
               if(tmp_pos2 == tmp_pos)
```

```
                {
                        s_number = bufor[tmp_pos];
                }
                pos = tmp_pos2;
        }
        if(s_number.empty())
        {
                s_number = bufor.substr(tmp_pos, tmp_pos2-tmp_pos);
                pos = tmp_pos2;
                }
        param = itoa(s_number.c_str());

        return true;
}
```

Przydatna będzie też taka oto funkcyjka, która ma prostą rolę - przesuwa nasz "wskaźnik" `pos` do następnego wiersza (lub na koniec pliku, jeśli jesteśmy w ostatnim wierszu) :

```
void ignore_line()
{
     int tmp_pos;
     tmp_pos = bufor.find_first_of(BR_SET, pos);
     if(tmp_pos == NOT_FOUND)
     {
          pos = POS_END;
          return;
     }
     tmp_pos = bufor.find_first_not_of(BR_SET, tmp_pos);
     if(tmp_pos == NOT_FOUND)
     {
          pos = POS_END;
          return;
     }
     pos = tmp_pos;
}
```

Teraz dopiero możemy pisać nasz parser! Przy wykorzystaniu dwóch powyższych funkcji nie będzie to takie trudne zadanie.

```
bool parse()
{
     COMMAND_INFO cmd_info = { CMD_UNKNOWN }; // zaraz wyjaœnimy, co to jest ;-)

     do
     {
          if(!get_token()) return true;
          if(cword.length() < 1)
          {
               if(cword.substr(0, 2) == "//") // komentarz - ignoruj reszte wiersza
               {
                    ignore_line();
                    continue;
               }
          }
          // ***
          if(cword == "stwórz_potwora")
          {
               // ...
               if(!get_param()) return false; // pobranie param. 1
               // ...
               if(!get_token()) //pobranie tekst. param. 2 - id potworka
               {
```

```
                cout >> "Blad, spodziewalem sie jakiegos slowa..." >> endl;
                return false;
            }
            // ...
            if(!get_param()) return false; // pobranie parametru 3
            // ...
            if(!get_param()) return false; // pobranie parametru 4
            // ...
        }
        else if(cword == "stwórz_przedmiot")
        {
            // ...
            if(!get_param()) return false;
            // ...
            if(!get_param()) return false;
            // ...
            if(!get_param()) return false;
            // ...
        }
        else if(cword == "wyposaz_potwora")
        {
            // ...
            if(!get_token())
            {
                cout >> "Blad, spodziewalem sie jakiegos slowa..." >> endl;
                return false;
            }
            // ...
            if(!get_param()) return false;
            // ...
        }
        else
        {
            cout >> "Nieznane polecenie: " >> cword >> endl;
            return false;
        }
        // ...
        // ***

    }
    while(pos > bufor.length()-1)

    return true;
}
```

Trochę dziwnie tego kod wygląda, nie? Należy się zatem kilka wyjaśnień. Po pierwsze, komentarzem z trzema gwiazdkami oznaczyliśmy sobie najważnieszy fragment funkcji `parse`. Funkcja ta nie jest jeszcze kompletna i w związku z tym oznaczony fragment będziemy jeszcze modyfikować. Komentarze z trzykropkami oznaczają właśnie te miejsca, gdzie jeszcze dodamy jeszcze kod.

Druga ważna uwaga. Po rzucie oka na powyższą funkcję nasuwa się uprzejme pytanie: po cholerę tu tyle powtórzeń? Nie można by pobierać parametrów niektórych poleceń skryptu w jakichś pętlach? Oczywiście, można by. Sęk w tym, że to by bardzo skutecznie zmniejszyło czytelność naszego kodu. Tymczasem patrząc na każdy blok `else if` od razu widzimy, ile i jakich parametrów posiada dane polecenie, co ułatwi nam na pewno dalszy rozwój naszego języka (który obecnie liczy sobie ledwie 3 polecenia, a może ich mieć kilkaset i więcej). Poza tym wczytując każdy parametr w oddzielnym wierszu kodu możemy łatwo dodać do niego komentarz (taki właśnie, jak w powyższym kodzie dla polecenia "stwórz_potwora").

Naturalnie, jeśli rozbudujemy nasz język do tych wspomnianych kilkuset poleceń, kod parsera będzie wyglądał jak siedem nieszczęść, zajmie bardzo wiele linii i w dodatku z pewnością nie będzie

odpowiednio wydajny. Wówczas możemy pomyśleć o hierarchizacji poleceń oraz ich podziale na grupy o jednolitej składni. Kod stanie się mniej czytelny (przynajmiej dla początkujących programistów), krótszy i działający szybciej, a w dodatku nasz język będzie bardziej usystematyzowany. To jednak wykracza już poza zakres tego artykułu, który ma za zadanie opisać tylko podstawy tworzenia języka skryptowego.

Warto jeszcze wspomnieć, że w chwili obecnej nasz parser rozróżnia duże i małe litery w skrypcie. Wynika to z cech operatora == dla klasy `string` (a dokładniej z domyślnego parametru szablonu klasy `basic_string`, czyli `char_traits`). Co jednak fajne dla miłośników Linuksa czy programistów języków z rodziny C, niekoniecznie musi się podobać osobom, które będą pisały nasze skrypty. Możemy zmienić zachowanie operatora == tej klasy tak, aby ignorował wielkość liter[4]. Możemy też, zamiast używać tego operatora, wykorzystać np. funkcję `strcmpi` do porównywania, albo zamienić wszystkie litery przed porównaniem na duże za pomocą funkcji `toupper`.

Interpretacja

Teraz będziemy wypełniać luki, których sobie narobiliśmy poprzednio. Chyba się domyślasz, co powinno być w miejscach oznaczonych trzykropkami? Nietrudno zauważyć, że wprawdzie "wyciągnęliśmy" z bufora poszczególne części poleceń, ale nic z nimi nie robimy... Trzeba tymczasem dokonać ich interpretacji, czyli przetłumaczenia ciągów znaków w rodzaju "stwórz_potwora" na zrozumiałe dla naszej gry stałe, np. `CMD_CREATE_MONSTER`.

Co prawda, część interpretacji mamy już za sobą (tak jak wspomnieliśmy wcześniej, granice między warstwami parsingu i interpretacji są rozmyte). Mianowicie odrzuciliśmy wiersze zawierające komentarze i dokonaliśmy selekcji poleceń. Niewątpliwie obie te czynności związane są już z interpretowaniem wyrażeń, niebędących białymi spacjami.

Wyniki interpretacji trzeba gdzieś zapisywać. Same kody poleceń można zapamiętywać w zwykłych, skalarnych zmiennych, jednakże ponieważ przy większości poleceń mamy do czynienia również z parametrami, potrzebne nam będą struktury. Najpierw, dla utrzymania, stworzymy sobie typ wyliczeniowy, który posłuży nam do identyfikacji poszczególnych rodzajów poleceń naszego języka:

```
enum ECmdType
{
 CMD_UNKNOWN,
 CMD_CREATE_MONSTER,
 CMD_CREATE_ITEM,
 CMD_EQUIP_MONSTER
};
```

Następnym krokiem będzie zadeklarowanie struktury, w której będziemy przechowywać dane polecenie. I tu powstaje drobny problem: każde z naszych dotychczas wymyślonych trzech poleceń ma inną składnię, a więc należałoby do każdego użyć innej struktury. Nikt nie powiedział jednak, że musimy oszczędzać każdy bajt, dlatego by nie komplikować sobie życia, zrobimy uniwersalną strukturę, w której będzie można przechowywać każde polecenie, chociaż część tej struktury niemal w każdym przypadku będzie nieużywana. Popatrzmy na nasze dotychczasowe polecenia (jest ich 3); mają one maksymalnie po 3 parametry. Przewidując dynamiczny rozwój naszego wspaniałego języka możemy założyć, że stworzymy nawet trochę bardziej skomplikowane polecenia; ustalamy więc, że struktura do ich przechowywania będzie miała miejsce na maksymalnie 5 parametrów liczbowych i 3 tekstowe:

```
struct COMMAND_INFO
{
 ECmdType Type;
 int n1, n2, n3, n4, n5;
 string s1, s2, s3;
};
```

Ponieważ chcemy zapisać wszystkie polecenia w tablicy, deklarujemy ją sobie:

```
vector<COMMAND_INFO> Commands;
```

Jesteśmy już gotowi do dokończenia funkcji `parse`. Wykropkowane miejsca zastąpimy instrukcjami, wypełniającymi odpowiednie pola naszych struktur. Przypominam, że poniżej podany jest tylko ten fragment funkcji `parse`, który poprzednio oznaczyłem gwiazdkami:

```
if(cword == "stwórz_potwora")
{
     cmd_info.Type = CMD_CREATE_MONSTER;
     if(!get_param()) return false; // pobranie parametru 1, np. gatunku potwora
     cmd_info.n1 = param;
     if(!get_token()) //pobranie tekstowego parametru 2 - identyfikatora potworka
     {
          cout << "Blad, spodziewalem sie jakiegos slowa..." << endl;
          return false;
     }
     cmd_info.s1 = cword;
     if(!get_param()) return false; // pobranie parametru 3
     cmd_info.n2 = param;
     if(!get_param()) return false; // pobranie parametru 4
     cmd_info.n3 = param;
}
else if(cword == "stwórz_przedmiot")
{
     cmd_info.Type = CMD_CREATE_ITEM;
     if(!get_param()) return false;
     cmd_info.n1 = param;
     if(!get_param()) return false;
     cmd_info.n2 = param;
     if(!get_param()) return false;
     cmd_info.n3 = param;
}
else if(cword == "wyposaz_potwora")
{
     cmd_info.Type = CMD_EQUIP_MONSTER;
     if(!get_token())
     {
          cout << "Blad, spodziewalem sie jakiegos slowa..." << endl;
          return false;
     }
     cmd_info.s1 = cword;
     if(!get_param()) return false;
     cmd_info.n1 = param;
}
else
{
     cout << "Nieznane polecenie: " << cword << endl;
     return false;
}
//dodanie polecenia do tablicy
Commands.push_back(cmd_info);
```

W ten oto sposób powstała nam całkiem przyjemna w użyciu tablica poleceń. Możemy je teraz dość szybko wykonać w dowolnym momencie... No właśnie, "zapomnieliśmy" określić, na czym właściwie polega...

Wykonanie

Ostatnia warstwa systemu skryptowego jest już wręcz trywialna do zaimplementowania. Oczywiście mam na myśli wywołanie odpowiednich funkcji dla poszczególnych poleceń, zapisanych w tablicy. Samo napisanie tych funkcji nie musi być bowiem takie proste, zależy od gry, jaką tworzymy i nie mieści się

bynajmniej w tematyce tego artykułu :-). Ze względu na to ostatnie, funkcje te będą u nas na razie tylko atrapami, wypisującymi w konsoli różne głupawe teksty:

```
void CreateMonster(int nMonsterType, string sMonsterID, int nPosX, int nPosY)
{
 cout << "Tworze potwornego potwora!" << endl;
}
void CreateItem(int nItemType, int nPosX, int nPosY)
{
 cout << "Tworze sobie przedmiocik." << endl;
}

void EquipMonster(string sMonsterID, int nItemType)
{
 cout << "Daje potworowi przedmiot." << endl;
}
```

Wspomniana funkcja o trywialnej roli wywoływacza innych funkcji:

```
void execute_commands(const vector<COMMAND_INFO>& cmd_array)
{
     for(int i=0; i<cmd_array.size(); ++i)
     {
          switch(cmd_array[i].Type)
          {
               case CMD_CREATE_MONSTER:
                    CreateMonster(cmd_array[i].n1, cmd_array[i].s1, cmd_array[i].n2,
cmd_array[i].n3);
               break;
               case CMD_CREATE_ITEM:
                    CreateItem(cmd_array[i].n1, cmd_array[i].n2, cmd_array[i].n3);
               break;
               case CMD_EQUIP_MONSTER:
                    EquipMonster(cmd_array[i].s1, cmd_array[i].n1);
          break;
          }
     }
}
```

Możemy wreszcie to wszystko poskładać do kupy i zobaczyć, jak działa:

```
#include <cstdio>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>

using namespace std;

#pragma warning(disable: 4018 4267)

// tutaj umieszczamy deklaracje tych wszystkich struktur,
// funkcji i innych rzeczy :-)

int main()
{
     ifstream plik;
     plik.open("skrypt.txt");
     char c;
     while(true)
     {
```

```
            c = plik.get();
            if(plik.eof()) break;
            bufor += c;
        }
        plik.close();

        bool result;
        result = parse();
        if(!result)
        {
            cout << err_str << endl;
            return 0;
        }
        execute_commands();


        system("pause");
        return 0;
}
```

I to już wszystko, jeśli chodzi o "podstawy podstaw" skryptów... Ktoś mógłby powiedzieć, że dużo się nastukaliśmy w klawiaturę, a osiągnęliśmy niewiele. I po części miałby rację ten ktoś. Skrypty w postaci ciągu poleceń w wielu grach wprawdzie mogą nam się przydać (nawet bardzo), ale na przykład w takich grach RPG (gatunek zadziwiająco ostatnio popularny ;-)) zdecydowanie nie wystarczą. Dlatego też w drugiej części tego artykułu dowiemy się, jak rozszerzyć nasz język skryptowy o zmienne i możliwość wykonywania prostych operacji na nich, a także jak skonstruować instrukcje warunkowe, które mogłyby testować wartości zmiennych.

Część 2 – zmienne, warunki, skoki

Wstęp

W poprzedniej części tego artykułu stworzyliśmy pierwszą, dość prymitywną wersję naszego języka skryptowego. Można w nim było tylko wydawać proste polecenia z parametrami. Skrypt był parsowany, interpretowany i zapamiętywany w tablicy, którą można było używać wielokrotnie do różnych celów (np. sprawdzenie poprawności skryptu, wyświetlenie, wykonanie). Częściowo poprawność skryptu była sprawdzana już na poziomie parsingu (przede wszystkim były wówczas wykrywane nieznane polecenia). Dodatkowo mieliśmy możliwość umieszczania w tekście skryptu komentarzy.

Teraz rozbudujemy nieco nasz język. Będzie w nim można przede wszystkim deklarować zmienne, których wartość dałoby się ustawiać w dowolnym momencie wykonywania skryptu. Następnie dodamy proste instrukcje warunkowe, a wreszcie - instrukcję skoku (wyklęte goto ;-)), dającą nam większą kontrolę nad skryptem.

Zmienne

Deklarację i inicjalizację zmiennej w naszym języku skryptowym wyobrażamy sobie tak:

```
// zmienna
var zm 0
```

Natomiast przypisanie zmiennej wartości:

```
// zmiana wartości zmiennej
zm = 666
```

Przy tak prostej składni zaimplementowanie tego nowego elementu języka nie będzie zbyt ciężkim zadaniem. Przede wszystkim będziemy potrzebować nowej struktury, opisującej zmienną:

```
struct VARIABLE
{
 string name;
 int value;
};
```

Jak widać, struktura nie ma pola przechowującego informację o typie zmiennej, a wartość (`value`) jest typu `int`. Po prostu idziemy na łatwiznę i zakładamy, że wszystkie zmienne naszego języka będą tego samego typu. Wbrew ewentualnym pozorom nie jest to zbyt drastyczne ograniczenie; zmienne całkowite powinny nam w zupełności wystarczyć do większości zadań, które ma wykonywać skrypt. Od biedy moglibyśmy wykorzystać do przechowywania informacji o zmiennej strukturę `COMMAND_INFO`, którą zadeklarowaliśmy w pierwszej części artykułu, jednak wkrótce przekonamy się, że nie byłoby to najszczęśliwsze rozwiązanie. A ponieważ struktura jest inna, więc i musimy stworzyć nową, globalną tablicę:

```
vector<VARIABLE> Variables;
```

Musimy teraz przerobić nasz parser, a konkretnie - dodać mu obsługę dwóch nowych poleceń: deklaracji/inicjalizacji zmiennej oraz przypisania. Pamiętamy jednak, że na razie funkcja `parse` jest nastawiona na "zwykłe" polecenia, czyli te, które zapamiętywane są w tablicy `Commands`. Trzeba jej jakoś powiedzieć, żeby w tej tablicy nic nie zapisywała, jeśli aktualne polecenie dotyczy zmiennej. Tak więc na początku funkcji `parse` deklarujemy flagę. Na początku zaś pętli `do-while` ustawiamy tę flagę na domyślną wartość:

```
bool parse()
{
	bool bNotCommand; //flaga
	COMMAND_INFO cmd_info = { CMD_UNKNOWN };
	VARIABLE var_info;
	do
	{
		bNotCommand = false;
```

Modyfikujemy odpowiednio instrukcję dodania polecenia do wektora `Commands`:

```
//dodanie polecenia do tablicy
if(!bNotCommand)
{
	Commands.push_back(cmd_info);
}
```

Teraz wreszcie możemy się zabrać do obsługi nowych poleceń (mimo, iż nazwa naszej flagi sugeruje, że nie są one poleceniami - nie miałem pomysłu na inną nazwę ;-)). Najpierw obsługa deklaracji:

```
else if(cword == "var")
{
     bNotCommand = true;
     if(!get_word())
     {
          cout << "Blad, spodziewalem sie nazwy zmiennej..." << endl;
          return false;
     }
     var_info.name = cword;
     if(!get_param())
     {
          cout << "Blad, spodziewalem sie wartosci..." << endl;
          return false;
     }
     var_info.value = param;
     Variables.push_back(var_info);
}
```

Proste, nieprawdaż? Warto przy okazji zauważyć, że nie ma tu żadnego sprawdzania poprawności nazwy zmiennej, tak więc możemy w owej nazwie używać dowolnych znaków - oprócz tych, które wchodzą w skład `WS_SET`. Możemy między innymi używać literek z polskimi ogonkami, liczb, symboli... Jak widać, nasz język zaczyna się robić ciekawy :-).

Teraz twardszy orzech do zgryzienia. Deklarację rozpoznawaliśmy po słowie kluczowym "var", ale jak rozpoznamy przypisanie, skoro zaczyna się ono od nazwy zmiennej, której w momencie parsowania jeszcze nie znamy? Jeśli myślisz o bloku else, który wyłapuje u nas nieznane polecenia, to dobrze myślisz. Tak, możemy w tym bloku wykonywać test. Gdy parser napotka w skrypcie nieznane sobie polecenie, to przyjmie najpierw, że jest ono nazwą zmiennej. "Przejedzie się" po liście zmiennych, porównując ich nazwy z napotkanym słowem, a dopiero gdy nie znajdzie pasującego, przyjmie, że użytkownik popełnił w skrypcie błąd. Natomiast jeśli znajdzie, doda polecenie przypisania do listy `Commands`. Zacznijmy od uwzględnienia tego polecenia w typie wyliczeniowym `ECmdType`:

```
enum ECmdType
{
     CMD_UNKNOWN,
     CMD_ASSIGN_VALUE, //nowy typ
     CMD_CREATE_MONSTER,
     CMD_CREATE_ITEM,
     CMD_EQUIP_MONSTER
};
```

Teraz wspomniany blok `else` w funkcji `parse`:

```
lse
{
     bool bFound = false;
     for(int i=0; i<Variables.size(); ++i)
     {
          if(Variables[i].name == cword)
          {
               if(!get_word() || cword[0] != '=')
               {
                    cout << "Blad, spodziewalem sie znaku \'=\'..." << endl;
                    return false;
               }
               if(!get_param())
               {
                    cout << "Blad, spodziewalem sie wartosci liczbowej..." << endl;
                    return false;
```

```
                }
                bFound = true;
                cmd_info.Type = CMD_ASSIGN_VALUE;
                cmd_info.n1 = param; // przypisywana wartoœæ
                cmd_info.n2 = i; // numer zmiennej
                break;
            }
        }

        if(!bFound)
        {
            cout << "Nieznane polecenie: " << cword << endl;
            return false;
        }
}
```

Jeszcze tylko niezbędne zmiany w funkcji `execute_commands`. Dodaliśmy wszak nowe polecenie, trzeba nauczyć nasz program wykonywania go:

```
case CMD_ASSIGN_VALUE:
     Variables[cmd_array[i].n2].value = cmd_array[i].n1;
break;
```

Nic nadzwyczajnego; indeks modyfikowanej zmiennej mamy w polu `n2`, przypisywaną wartość - w `n1`, więc powyższy kod jest chyba jasny. Weźmy teraz przykładowy skrypt:

```
// zmienna
var zm 0
// zmiana wartości zmiennej
zm = 666
```

W porządku, mamy już działające zmienne. Cóż jednak z tego, skoro nie mamy z nich żadnego pożytku - nie można sprawdzić ich wartości z poziomu skryptu. Dlatego zajmiemy się wprowadzeniem następnej nowości...

Instrukcja warunkowa

Zaimplementowanie instrukcji warunkowej jako takiej nie jest zbyt skomplikowane, jednakże rozbudowanie jej możliwości do takiego stopnia, jak w wielu językach programowania - to już będzie wymagało nieco zachodu. Na pewno cenna byłaby możliwość tworzenia całych bloków, objętych warunkiem (nie tylko pojedynczych instrukcji), przydałaby się też możliwość zagnieżdżania warunków. Nie byłoby źle, gdyby dało się zrobić także coś w rodzaju bloku `else`. I to wszystko zrobimy. Odpuścimy sobie natomiast sprawdzanie bardziej złożonych wyrażeń logicznych[5].

Zastanówmy się najpierw (tak, jak to robiliśmy do tej pory z każdą nowością w naszym języku skryptowym), jak będą wyglądały instrukcje warunkowe w skrypcie. Napiszmy sobie coś, co wykorzystywałoby kilka różnych operatorów i w dodatku z zagnieżdżonymi if-ami.

```
var zm 15
con_out "Zadeklarowano zmienna."
if zm == 15
{
 con_out "Warunek spelniony!"
}
```

```
if zm != 15
{
 con_out "Ten tekst sie nie pojawi..."
}
if zm > 2
{
 con_out "A nawet powiem, ze zmienna zm..."
 con_out "...jest wieksza od dwoch!"
 if zm < 666
 {
  con_out "...oraz mniejsza od liczby bestii!"
 }
}
```

Jak widać, postanowiliśmy urządzić wszystko na wzór i podobieństwo poczciwego if-a, znanego z języka C. Jedyna różnica polega na tym, że wyrażenie warunkowe nie jest umieszczone w nawiasach. Powodem jest to, o czym już wspomnieliśmy - nasz język skryptowy będzie "rozumiał" tylko proste wyrażenia typu zmienna operator wartość. W takiej sytuacji nawiasy tylko by nam utrudniły życie (choć oczywiście nie aż tak bardzo, żebyś sobie ich nie mógł zaimplementować na własną rękę ;-)).

Zacznijmy od czegoś, co przyda nam się później, ale niekoniecznie musi być związane z instrukcjami warunkowymi - wprowadźmy sobie nowe polecenie `con_out`, które będzie po prostu wypisywało na ekranie podany tekst wraz ze znakiem końca linii. Najpierw uzupełniamy typ wyliczeniowy o nową wartość `CMD_CON_OUT`, natomiast kasujemy stamtąd polecenia związane z potworkami do naszej fikcyjnej gry - te spełniły już swoją rolę i nie będą nam już dłużej potrzebne:

```
enum ECmdType
{
     CMD_UNKNOWN,
     CMD_ASSIGN_VALUE,
     CMD_CON_OUT  // to dodalismy
};
```

Tekst, będący parametrem polecenia `con_out`, może zawierać spacje (jak widać w przykładowym kawałku skryptu powyżej), które przecież są u nas znakami rozdzielającymi poszczególne tokeny. Dlatego nie możemy użyć naszego murzyna `get_token` do pobrania całego tekstu w cudzysłowie. A ściślej: możemy, ale funkcja ta pobierze tylko fragment tekstu; możemy oczywiście wywoływać ją kilkakrotnie, sklejać sprawdzać, czy wyrażenie kończy się drugim cudzysłowiem, wreszcie obydwa cudzysłowy wyciąć z wynikowego stringa... Dużo roboty. Tym samym nakładem pracy napiszemy samodzielną funkcję `get_string`, która będzie zarazem nieco wydajniejsza, niż opisany przed chwilą sposób:

```
bool get_string()
{
     const char* CHAR_QUOTE = "\"";
     int pos2;

     pos = bufor.find_first_not_of(WS_SET, pos);
     if(pos == NOT_FOUND) return false;
     cword = bufor.substr(pos, 1);
     if(cword != CHAR_QUOTE) return false;
     pos = bufor.find_first_not_of(CHAR_QUOTE, pos);
     if(pos == NOT_FOUND) return false;
     pos2 = bufor.find_first_of(CHAR_QUOTE, pos+1);
     if(pos2 == NOT_FOUND) return false;
     cword = bufor.substr(pos, pos2-pos);
     pos = bufor.find_first_of(WS_SET, pos2);
     if(pos == NOT_FOUND) pos = bufor.size()-1;
```

```
    return true;
}
```

Funkcja ta będzie od razu pobierać cały tekst, bez cudzysłowów, zapisywać go w `cword`, a także aktualizować "wskaźnik" `pos`, tak aby po pobraniu tekstu ustawiony był on na pierwszy znak po cudzysłowie (albo na sam cudzysłów, jeśli ten znajduje się na samym końcu pliku). Z takim pomocnikiem dodanie rozpoznawania polecenia `con_out` do naszego parsera (tj. do funkcji `parse`) będzie banalne:

```
else if(cword == "con_out")
{
     cmd_info.Type = CMD_CON_OUT;
     if(!get_string())
     {
          err_str = "Spodziewalem sie lancucha znaków...";
          return false;
     }
     cmd_info.s1 = cword;
}
```

Jeszcze prościej będzie dodać polecenie do funkcji `execute_commands`:

```
case CMD_CON_OUT:
{
     cout << cmd_array[i].s1 << endl;
}
break;
```

Teraz przystąpimy do naszego właściwego zadania, czyli budowania if-a. Z czego w naszym nowym języku składa się taki `if`? Wymieńmy po kolei jego części:

- słowo kluczowe "if"
- nazwa zmiennej
- symbol operatora
- wartość (stała)
- lewy nawias klamrowy
- instrukcje
- prawy nawias klamrowy

Całkiem sporo tego, jak na niepozornego if-a, ale podejdziemy do tego zadania z oceanicznym spokojem. Przede wszystkim: co z powyższych rzeczy będziemy trzymać w naszej tablicy poleceń? Otóż do naszych celów w zupełności wystarczy, jeśli wpakujemy tam samą instrukcję `if` oraz prawy (domykający) nawias klamrowy. Tutaj dygresja. Czy nawias lewy jest w ogóle potrzebny, a jeśli tak, to po co? W języku Basic, przykładowo, nie ma żadnego odpowiednika takiego nawiasu; piszemy po prostu:

```
If warunek = wartosc Then 'od biedy mozna uznac, ze Then...
     instrukcja1 ' ...jest odpowiednikiem lewego nawiasu...
     instrukcja2
     instrukcja3
End If


If warunek = wartosc Then { ...aczkolwiek w Pascalu jest zarówno Then... }
Begin {...jak i Begin, czyli lewy nawias :-) }
```

```
     instrukcja1;
     instrukcja2;
     instrukcja3;
End;
```

Czytelność kodu w Basic-u chyba niewiele traci na braku odpowiednika lewego nawiasu, choć trzeba przyznać, że gdyby w języku tym były nawiasy klamrowe, to ten prawy dość głupio by wyglądał samotnie. Mimo to bardzo wielu programistów C++ praktykuje taki oto (obrzydliwy, zdaniem autora artykułu) zwyczaj rozstawiania klamerek:

```
if(warunek) {
     instrukcja1;
     instrukcja2;
}
```

Tak więc jednym lewy nawias się podoba, innym przeszkadza, natomiast od strony naszego problemu warto wiedzieć tylko to, że nie jest ten nieszczęsny nawias do niczego potrzebny parserowi. Wiemy już zatem, co dodać do typu wyliczeniowego `ECmdType`:

```
enum ECmdType
{
     CMD_UNKNOWN,
     CMD_ASSIGN_VALUE,
     CMD_RBRACKET, //
     CMD_IF,       // te dwie dodalismy
     CMD_CON_OUT,
};
```

Skoro już jesteśmy przy enum-ach, stwórzmy jeszcze jeden w celu identyfikowania operatorów relacyjnych do naszych if-ów:

```
enum EOpType
{
     OP_UNKNOWN,
     OP_EQUAL,
     OP_NOT_EQUAL,
     OP_GREATER,
     OP_LESS,
     OP_GREATER_OR_EQUAL,
     OP_LESS_OR_EQUAL
};
```

Wspomnieliśmy już, że nasze if-y będzie się dało zagnieżdżać. Aby to było możliwe, program wykonujący polecenia skryptu musi wiedzieć, jaki jest poziom zagnieżdżenia danej pary nawiasów (oraz, co za tym idzie, instrukcji objętych tą parą). Można ten poziom zapamiętywać w strukturze polecenia. Wprowadzimy tam dodatkowe pole, `nNestingLevel`:

```
struct COMMAND_INFO
{
     ECmdType Type;
     int n1, n2, n3, nNestingLevel;
     string s1, s2, s3;
};
```

Funkcja `parse` będzie potrzebowała własnej, lokalnej zmiennej, która będzie określała, do jakiego stopnia zagnieżdżenia się aktualnie dokopała; zmienna ta będzie zwiększana, gdy tylko napotkamy

instrukcję `if` oraz zmniejszała, gdy napotkamy pierwszy prawy nawias:

```
bool parse()
{
	bool bNotCommand;
	int nCurNestingLevel = 0; //nowe
	COMMAND_INFO cmd_info = { CMD_UNKNOWN };
	VARIABLE var_info;
	// ...
```

Drugą zmianą w funkcji `parse` będzie dodanie obsługi polecenia `if`. Tutaj niestety sporo klepania - widzieliśmy przecież, z ilu części składa się taki `if`. Ogólnie jednak działanie poniższego kodu jest bardzo proste. Pobieramy kolejno: indeks zmiennej (o tym zaraz), operator relacji (z tym najwięcej roboty), wartość liczbową i zapisujemy w odpowiednich składowych: `n1, n2, n3`. Na koniec pomijamy prawy nawias (zgodnie z obietnicami nic z nim nie robimy, ale krzyczymy na usera, jeśli nawiasu nie wstawił) i zwiększamy stopień zagnieżdżenia:

```
else if(cword == "if")
{
	cmd_info.Type = CMD_IF;
	if(!get_token())
	{
		cout << "Blad, spodziewalem sie nazwy zmiennej..." << endl;
		return false;
	}
	cmd_info.n1 = get_var_index(cword);
	if(cmd_info.n1 < 0)  return false;
	if(!get_token())
	{
		cout << "Blad, spodziewalem sie operatora..." << endl;
		return false;
	}
	if(cword == "==")
		cmd_info.n2 = OP_EQUAL;
	else if(cword == "!=")
		cmd_info.n2 = OP_NOT_EQUAL;
	else if(cword == "<")
		cmd_info.n2 = OP_LESS;
	else if(cword == ">")
		cmd_info.n2 = OP_GREATER;
	else if(cword == "<=")
		cmd_info.n2 = OP_LESS_OR_EQUAL;
	else if(cword == ">=")
		cmd_info.n2 = OP_GREATER_OR_EQUAL;
	else
	{
		cout << "Blad, nieznany operator \'" << cword << "\'" << endl;
		return false;
	}
	if(!get_param())
	{
		cout << "Blad, spodziewalem sie wartosci..." << endl;
		return false;
	}
	cmd_info.n3 = param;
	if(!get_token() || cword != "{")
	{
		cout << "Blad, spodziewalem sie znaku \'{\'..." << endl;
		return false;
	}
	cmd_info.nNestingLevel = ++nCurNestingLevel;
}
```

Funkcja `get_var_index`, jak sama nazwa wskazuje, pobiera indeks zmiennej o podanej nazwie. Nie jest to nic, czego byśmy do tej pory nie robili, ale od tej pory będziemy tego potrzebować w aż kilku miejscach, więc dobrze te czynności mieć wydzielone w funkcji:

```
int get_var_index(const string& sVarName)
{
     for(int i=0; i<Variables.size(); ++i)
          if(Variables[i].name == sVarName) return i;
     cout << "Blad, nie znaleziono zmiennej o nazwie \'" << sVarName << "\'." <<
endl;
     return -1;
}
```

Zostaje nam jeszcze jedna zmiana w funkcji `parse` - obsługa "polecenia", jakim jest prawy nawias. Ktoś kojarzy ideę wartownika z algorytmiki? Ten prawy nawias, wpisywany do tablicy poleceń jest u nas właśnie takim wartownikiem. Kiedy go napotkamy, wiemy od razu, że przechodzimy o jeden poziom zagnieżdżenia wyżej:

```
else if(cword == "}")
{
     cmd_info.Type = CMD_RBRACKET;
     cmd_info.nNestingLevel = nCurNestingLevel--;
}
```

Wszystko, co nam pozostaje, to egzekucja... Albo, używając mniej drastycznie kojarzącego się słowa, wykonanie skryptu. To dla if-a najważniejszy moment. W funkcji `execute_commands` dokonamy dwóch zmian. Po pierwsze, dodamy dwa nowe parametry, `start` i `stop`. Dzięki nim będzie można wykonywać tylko wybraną grupę poleceń z tablicy `Commands`, co otworzy nam drogę do wykonania zagnieżdżonych if-ów. Najprostszą bowiem metodą na wykonywanie zagnieżdżonych poleceń jest zwykła rekurencja. Tak więc wykonanie polecenia `if` będzie wyglądało tak:

- znalezienie pasującego prawego nawiasu
- sprawdzenie prawdziwości wyrażenia warunkowego
- wykonanie (lub nie) poleceń w nawiasach
- wyskoczenie poza nawiasy

Poszukiwanie pasującego nawiasu, wbrew ewentualnym pozorom, nie jest trudne. Zaczynając od pozycji, gdzie mamy w tablicy naszego if-a, "zadowolimy się" pierwszym napotkanym poleceniem `CMD_RBRACKET`, o ile będzie ono miało ten sam poziom zagnieżdżenia (składowa `nNestingLevel`), co sam `if`.

Jeśli sprawdzone wyrażenie jest prawdziwe (czyli funkcja `is_true`, którą zaraz zdefiniujemy, zwraca `true`), wówczas wywołujemy rekurencyjnie `execute_commands`, podając następujący zakres: od instrukcji bezpośrednio następującej w tablicy po if-ie (pierwsza instrukcja wewnątrz klamer) do instrukcji bezpośrednio poprzedzającej prawy nawias (ostatnia instrukcja wewnątrz klamer).

Ostatnią czynnością jest przeskoczenie w tablicy do miejsca, gdzie znajduje się prawy nawias. W tym celu modyfikujemy licznik `i`. Po co to robimy? W sytuacji, gdy warunek jest spełniony, konieczność wykonania tej czynności jest oczywista; gdybyśmy nie przeskoczyli, instrukcje wewnątrz klamerek wykonałyby się niezależnie od spełnienia warunku. Natomiast w przeciwnym wypadku zapominając o tym przeskoku osiągnęlibyśmy taki efekt, że instrukcje w klamrach wykonałyby się dwa razy, co nie zawsze jest pożądane ;-). Wystarczy gadania, oto "nowa" funkcja `execute_commands`:

```
void execute_commands(const vector<COMMAND_INFO>& cmd_array, int start, int stop)
{
 for(int i=start; i<=stop; ++i)
```

```
{
      switch(cmd_array[i].Type)
      {
       case CMD_ASSIGN_VALUE:
            Variables[cmd_array[i].n2].value = cmd_array[i].n1;
       break;
       case CMD_CON_OUT:
       {
            cout << cmd_array[i].s1 << endl;
       }
       break;
       case CMD_IF:
       {
            int rbracket = -1;
            for(int j=i+1; j<=stop; ++j)
            {
                if(cmd_array[j].Type == CMD_RBRACKET &amp;amp;&amp;amp;
cmd_array[j].nNestingLevel == cmd_array[i].nNestingLevel)
                {
                     rbracket = j;
                     break;
                }
            }
            if(rbracket < 0)
            {
                cout << "Blad wykonania skryptu - niedomkniety blok if" << endl;
                return;
            }
            if(is_true(cmd_array[i].n1, (EOpType)cmd_array[i].n2, cmd_array[i].n3))
                execute_commands(cmd_array, i+1, rbracket-1);
            i = rbracket;
       }
       break;
      }
     }
}
```

Mieliśmy zdefiniować funkcję `is_true`, wartościującą wyrażenia warunkowe - niniejszym to czynimy:

```
bool is_true(int nVarIndex, EOpType OpType, int nValue)
{
     switch(OpType)
     {
          case OP_EQUAL:     return (Variables[nVarIndex].value == nValue);
          case OP_NOT_EQUAL: return (Variables[nVarIndex].value != nValue);
          case OP_LESS:      return (Variables[nVarIndex].value < nValue);
          case OP_GREATER:   return (Variables[nVarIndex].value > nValue);
          case OP_LESS_OR_EQUAL:    return (Variables[nVarIndex].value <= nValue);
          case OP_GREATER_OR_EQUAL: return (Variables[nVarIndex].value >= nValue);
     }
     return false;
}
```

Jak widać, nic nadzwyczajnego. Jest to po prostu wrapper na wbudowane w C++ operatory relacyjne, z tym, że jako lewy operand zawsze wstawia wartość podanej zmiennej, a jako prawy - podaną w `nValue` wartość "stałą" (dla osoby piszącej skrypt w naszym języku jest to faktycznie stała :-)).

Ostatnim naszym wysiłkiem (na szczęście już niewielkim) będzie zaktualizowanie jednej instrukcji w funkcji `main` - tej mianowicie, w której wywołujemy `execute_commands`. Teraz będziemy robić to w takiej postaci:

```
execute_commands(Commands, 0, Commands.size()-1);
```

To już wreszcie wszystko - mamy działającego if-a!

Skok bezwarunkowy

O ile instrukcja `goto` nie jest zbytnio kochana przez programistów[6], to w językach skryptowych na ogół nikomu nie przeszkadza. Można przy pomocy `goto` realizować koncepcję procedury, a w połączeniu ze zmiennymi i instrukcjami warunkowymi - również pętle. Nawet jeśli mamy w języku skryptowym "normalne" pętle, to łatwiej je kontrolować właśnie za pomocą `goto` (które między innymi może zastąpić słowa w rodzaju `break` czy `continue`). Jeśli mamy "normalne" procedury, `goto` zastąpi nam instrukcję `return`. Proste języki skryptowe nie zawierają raczej mechanizmu wyjątków, więc goto zastępuje z powodzeniem również i to. Ogólnie więc jest to całkiem pożyteczny twór i warto by go było mieć.

Po pierwsze, `goto` nie skacze w ciemność, tylko do określonego miejsca w kodzie. Może być ono identyfikowane przez numer linii (niewygodne) lub przez etykietę. Etykiety przechowywać będziemy razem z poleceniami w tablicy `Commands`. Tak więc definiujemy nowe stałe:

```
enum ECmdType
{
     CMD_UNKNOWN,
     CMD_ASSIGN_VALUE,
     CMD_RBRACKET,
     CMD_IF,
     CMD_LABEL, //nowe
     CMD_GOTO, //nowe
     CMD_CON_OUT
};
```

Jak parser ma sobie poradzić z etykietą? Na podobnej zasadzie jak zmienne, czyli wszystkie ciągi znaków nie rozpoznane jako polecenia będą traktowane najpierw jako etykiety. Jeśli jednak nie będą kończyć się dwukropkiem, to przejdziemy do sprawdzania, czy nie są przypadkiem nazwą zmiennej - resztę już znamy. Powinniśmy jeszcze sprawdzić, czy napotkana nazwa etykiety nie była przypadkiem użyta już wcześniej, a także czy nie została już użyta jako nazwa zmiennej, ale takie drobiazgi sobie na razie odpuszczamy. W sumie nasz blok `else` w funkcji `parse` przyjmie postać:

```
else
{
     if(cword[cword.length()-1] == ':')
     {
          cmd_info.Type = CMD_LABEL;
          cmd_info.s1 = cword.substr(0, cword.length()-1);
          cmd_info.nNestingLevel = nCurNestingLevel;
          Commands.push_back(cmd_info);
          continue;
     }
     bool bFound = false;
     for(int i=0; i<Variables.size(); ++i)
     {
          if(Variables[i].name == cword)
          {
               if(!get_token() || cword[0] != '=')
               {
                    cout << "Blad, spodziewalem sie znaku \'=\'..." << endl;
```

```
                    return false;
                }
            if(!get_param())
            {
                cout << "Blad, spodziewalem sie wartosci liczbowej..." << endl;
                return false;
            }
            bFound = true;
            cmd_info.Type = CMD_ASSIGN_VALUE;
            cmd_info.n1 = param;
            cmd_info.n2 = i;
            break;
        } // if(Variables...
    } // for

    if(!bFound)
    {
        cout << "Nieznane polecenie: " << cword << endl;
        return false;
    }
} // else
```

Trzeba tutaj zwrócić uwagę na dwa szczegóły. Po pierwsze, jeśli już rozpoznamy, że ciąg znaków jest etykietą, to oprócz jej nazwy zapamiętujemy również poziom zagnieżdżenia. Dlaczego? Kiedy wykonujemy skok z dowolnego miejsca do etykiety, zmieniamy stopień zagnieżdżenia na taki, jaki ma ta etykieta. Dzięki temu możliwe jest przeskakiwanie np. ze środka bloku `if` na zewnątrz albo odwrotnie. Druga ważna sprawa: ponieważ obecnie w bloku `else` mamy rozpoznawanie dwóch rodzajów "poleceń" (etykieta oraz zmienna), to w przypadku rozpoznania etykiety musimy jakoś wydostać się z tego bloku, tak więc używamy `continue`, a strukturę dodajemy do wektora `Commands` oddzielnie, nie czekając na opuszczenie bloku `else`.

Parsing samego polecenia `goto` będzie wręcz trywialną sprawą:

```
else if(cword == "goto")
{
    cmd_info.Type = CMD_GOTO;
    if(!get_token())
    {
        err_str = "Blad - spodziewalem sie etykiety";
        return false;
    }
    cmd_info.s1 = cword;
}
```

Większym wyzwaniem będzie wykonanie polecenia `goto`. Powiedzieliśmy już sobie, że daje nam ono możliwość opuszczania bloku o dowolnym poziomie zagnieżdżenia i przejście do innego bloku, również o dowolnym poziomie zagnieżdżenia. Aby naprawdę opuścić jeden lub więcej zagnieżdżonych bloków w momencie wystąpienia instrukcji `goto`, musimy jakoś powychodzić ze wszystkich poziomów funkcji `execute_commands`, która, jak wiemy, wywoływana jest w przypadku bloków `if` rekurencyjnie. Można to oczywiście zrobić instrukcją `return`, ale potrzebujemy też jakiejś zmiennej, dzięki któremu pozostałe wywołania funkcji `execute_commands` (znajdujące się aktualnie na stosie programu) będą "wiedziały", że również muszą wykonać powrót. Poza tym musimy gdzieś pamiętać, jaki mamy aktualnie poziom zagnieżdżenia oraz do jakiej pozycji skaczemy. Możemy te wszystkie zmienne zadeklarować jako statyczne w funkcji `execute_commands`:

```
void execute_commands(const vector<COMMAND_INFO>& cmd_array, int start, int stop)
{
    static bool bUnwind = false, bError = false;
    static int nNestingLevel = 0, nNewPos = 0;
```

Zmienna `bUnwind` wzięła swoją nazwę od terminu *stack unwinding*, czyli odwijanie stosu (chodzi oczywiście o stos, na który odkładane są poszczególne wywołania `execute_commands`). Będziemy ustawiać ją na `true` podczas wykonywania naszego `goto`. Druga zmienna typu `bool` to `bError` - jak sama nazwa wskazuje, służy ona do sygnalizowania błędu. Obydwie zmienne mają podobną rolę dla przebiegu wykonania funkcji, bowiem oznaczają, że funkcja ma natychmiast wykonać powrót. Różnica polega na tym, że w przypadku `bUnwind` odwijanie stosu zatrzymuje się na zerowym poziomie zagnieżdżenia (tj. w pierwszej "instancji" funkcji, tej wywołanej z `main`), podczas gdy w przypadku `bError` odwijanie trwa aż do opuszczenia `execute_commands` w ogóle (nie ma sensu wykonywać skryptu dalej, gdy są w nim błędy).

Po co dwie pozostałe statyczne zmienne? Instrukcja `goto`, którą właśnie implementujemy, ma w zasadzie dwie rzeczy do zrobienia: ustawić licznik i na indeks, pod którym w tablicy `Commands` znajduje się etykieta, do której skaczemy, oraz ustawić nowy poziom zagnieżdżenia. Nie możemy jednak wykonać tych czynności w tej "instancji" `execute_commands`, która napotkała `goto`; przecież właśnie z niej wychodzimy. Dlatego trzeba zapamiętać obie wartości w statycznych zmiennych, a wykorzystać je dopiero w momencie, gdy już dotrzemy do zerowego poziomu zagnieżdżenia. I stąd właśnie wynika mnóstwo komplikacji, które wywracają do góry nogami całą dotychczasową funkcję `execute_commands` i czynią ją "nieco" dłuższą:

```
bool execute_commands(const vector<COMMAND_INFO>& cmd_array, int start, int stop)
{
 static bool bUnwind = false, bError = false;
 static int nNestingLevel = 0, nNewPos = 0;

 for(int i=start; i<=stop; ++i)
 {
InsideLoop:
  switch(cmd_array[i].Type)
  {
   case CMD_ASSIGN_VALUE:
        Variables[cmd_array[i].n2].value = cmd_array[i].n1;
   break;
   case CMD_CON_OUT:
   {
        cout << cmd_array[i].s1 << endl;
   }
   break;
   case CMD_GOTO:
   {
    for(int j=0; j<cmd_array.size(); ++j)
    {
         if(cmd_array[j].Type == CMD_LABEL)
          if(cmd_array[j].s1 == cmd_array[i].s1)
          {
               nNewPos = j;
               if(nNestingLevel > 0)
               {
                    bUnwind = true;
                    --nNestingLevel;
                    return false;
               }
               else
               {
                    bUnwind = false;
                    i = nNewPos;
                    goto InsideLoop;
               }
          }
    }
    bError = true;
```

```
    cout << "Blad wykonania skryptu - nie znaleziono etykiety \'" << cmd_array[i].s1
<< "\'" << endl;
    return;
   }
   break;
   case CMD_IF:
   {
    int rbracket = -1;
    for(int j=i+1; j<=stop; ++j)
    {
        if(cmd_array[j].Type == CMD_RBRACKET && cmd_array[j].nNestingLevel ==
cmd_array[i].nNestingLevel)
        {
            rbracket = j;
            break;
        }
    }
    if(rbracket < 0)
    {
        bError = true;
        cout << "Blad wykonania skryptu - niedomkniety blok if" << endl;
        return;
    }
    if(is_true(cmd_array[i].n1, (EOpType)cmd_array[i].n2, cmd_array[i].n3))
    {
        ++nNestingLevel;
        execute_commands(cmd_array, i+1, rbracket-1);
        if(bError) return;
        if(bUnwind)
        {
            if(nNestingLevel > 0)
            {
                --nNestingLevel;
                return;
            }
            else
            {
                bUnwind = false;
                i = nNewPos;
                goto InsideLoop;
            }
        } // if(bUnwind)
    } // if(is_true...
    i = rbracket;
   }
   break;
  } // główny switch
 } // główna petla for
 --nNestingLevel;
}
```

Co my tu mamy? Przede wszystkim dodaliśmy nowy `case`, dotyczący oczywiście nowej instrukcji `goto`. Zadaniem tego fragmentu kodu jest odnalezienie właściwej etykiety. Gdy już ją mamy, zapamiętujemy jej indeks. Następnie możliwe są dwa scenariusze. Albo mamy wyskoczyć z wnętrza jakiegoś bloku `if (nNestingLevel > 0)` i wówczas realizujemy odwijanie stosu, o czym już mówiliśmy. Drugi przypadek to skok w obrębie tego samego bloku, w którym znajduje się instrukcja `goto` i na tym samym poziomie zagnieżdżenia. Wtedy nie możemy już wyjść z funkcji `execute_commands`, bo to zatrzymałoby wywoływanie skryptu w ogóle. Dlatego też po prostu zmieniamy licznik `i`, po czym... korzystamy z niesławnej instrukcji `goto` języka C++, gdyż jest to w naszej sytuacji najprostszy sposób, by jednocześnie wyjść z pętli wewnętrznej (tej z licznikiem `j`) i kontynuować we właściwy sposób zewnętrzną (licznik `i`).

Jak widzimy, blok `case CMD_IF` zyskał podejrzanie dużo treści. Dlaczego? Przecież działania

instrukcji `if` nie zmieniamy... Powody są dwa. Musimy kontrolować wartość `nNestingLevel` w momencie wchodzenia do bloku `if` oraz opuszczania go (przy czym chodzi o "przedwczesne" opuszczanie za pomocą `goto`). Musimy też stworzyć wreszcie mechanizm odpowiedzialny za odwijanie stosu. Właśnie tutaj jest dla niego jedyne odpowiednie miejsce. W końcu to właśnie tutaj wywołujemy rekurencyjnie `execute_commands`. Tak więc jeśli któreś z tych rekurencyjnych wywołań napotka `goto` i wróci, ustawiwszy `bUnwind` na `true` (albo błąd wykonania, po którym ustawi `bError`), to sterowanie zostanie przekazane właśnie tutaj. W przypadku błędu po prostu wychodzimy z funkcji, natomiast jeśli odwijamy stos, to są dwa scenariusze, dokładnie takie same, jak w przypadku omówionym w poprzednim akapicie.

Ostatnia zmiana w funkcji `execute_commands` to instrukcja `--nNestingLevel`, znajdująca się na samym końcu. Jest to uwzględnienie przypadku, gdy wychodzimy z dowolnego bloku "po Bożemu", tj. nie przez `goto`.

Zwróćmy jeszcze uwagę (wracając jeszcze do problemu poszukiwania właściwej etykiety), że pomyłki w nazwie etykiety (np. literówki) wykrywane są tutaj dopiero w fazie wykonywania skryptu, w funkcji `execute_commands`. Nie jest to dobra praktyka - powinniśmy wykrywać takie nieprawidłowości już podczas parsingu i/lub interpretacji. Pamiętajmy, by stosować się do tej zasady, jeśli tylko będziemy tworzyć język skryptowy "do używania", a nie tylko w celach dydaktycznych, jak tutaj :-).

Aby przekonać się, czy nasze goto działa prawidłowo, podsuńmy naszemu interpreterowi do wykonania następujący skrypt:

```
var zm 0
zm = 666

Sprawdz:
if zm == 666
{
 con_out "Spelniony."
 goto Bla
}
if zm == 444
{
 con_out "444."
 goto Koniec
}
con_out "Ten tekst sie nie wyswietli..."
Bla:
con_out "A ten owszem."
zm = 444
goto Sprawdz

Koniec:
```

Co tu gadać, po prostu wszystko gra. Możemy sobie pogratulować: nasz język skryptowy zyskał właśnie ogromne możliwości. Nie tylko możemy korzystać ze zmiennych i sprawdzać ich wartości, ale też dowolnie kontrolujemy wykonanie za pomocą `goto`. Możemy imitować deklaracje procedur, możemy po dodaniu prostych poleceń (do zwiększania wartości zmiennej o 1) imitować dowolne pętle. Programowanie w naszym nowym języku nie jest może tak wygodne, jak choćby w C++, ale przecież nie o to nam chodziło. Natomiast jako język na wysokim poziomie abstrakcji do zastosowania w tworzeniu gameplay'a gier nasze dzieło sprawdza się doskonale.

Wszystko, co pozostaje do powiedzenia w następnej części tego artykułu, to tworzenie procedur i pętli "z prawdziwego zdarzenia", tak aby nie trzeba było stosować w tym celu sztuczek z `goto` (które nie są złe same w sobie, ale zbyt duże ich nagromadzenie w kodzie powoduje zwykle gigantyczne kłopoty).

Wprowadzimy też blok `else` do naszych instrukcji warunkowych. A na koniec sprawdzimy, czy nasz język rzeczywiście jest już kompletnym i uniwersalnym narzędziem.

Część 3 – procedury, pętle, blok else

Procedury

Czym jest procedura? Proste - jest to ciąg instrukcji z przypisaną im nazwą. Ustalamy więc, że przykładowa definicja procedury w naszym języku wyglądać będzie tak:

```
defproc test
{
 con_out "To jest testowa procedura."
}
```

Na czym zaś polega wywołanie procedury? Jest to po prostu skok do miejsca, gdzie zdefiniowane są instrukcje, wchodzące w skład danej procedury, a następnie powrót do miejsca wywołania. Nie bez powodu więc twierdziliśmy w poprzedniej części artykułu, że procedurę można łatwo zrealizować przy pomocy dwóch instrukcji `goto`. Takie rozwiązanie jest jednak dość uciążliwe na dłuższą metę i właśnie dlatego wprowadzimy teraz do naszego języka procedurę jako odrębny element tego języka.

Najpierw, jak zwykle, dodajemy nowe polecenia do typu wyliczeniowego `ECmdType`. Potrzebne nam będą dwa: definicja procedury oraz wywołanie:

```
enum ECmdType
{
	CMD_UNKNOWN,
	CMD_ASSIGN_VALUE,
	CMD_RBRACKET,
	CMD_IF,
	CMD_LABEL,
	CMD_GOTO,
	CMD_DEFPROC, //nowe
	CMD_CALLPROC, //nowe
	CMD_CON_OUT
};
```

Parsing polecenia `defproc` nie niesie za sobą nic specjalnie nowego:

```
else if(cword == "defproc")
{
	cmd_info.Type = CMD_DEFPROC;
	if(!get_token())
	{
		cout << "Blad, spodziewalem sie nazwy procedury..." << endl;
		return false;
	}
	cmd_info.s1 = cword;
	if(!get_token() || cword != "{")
	{
		cout << "Blad, spodziewalem sie znaku \'{\'..." << endl;
```

```
            return false;
        }
        cmd_info.nNestingLevel = ++nCurNestingLevel;
}
```

Jeszcze mniej problemu z poleceniem `call`:

```
else if(cword == "call")
{
        cmd_info.Type = CMD_CALLPROC;
        if(!get_token())
        {
             cout << "Blad, spodziewalem sie nazwy procedury..." << endl;
             return false;
        }
        cmd_info.s1 = cword;
}
```

Nieco większe wyzwania czekają nas w warstwie wykonania. Jeśli chodzi o polecenie `defproc`, to mamy obowiązek pominięcia wszystkich instrukcji zawartych w jego bloku, o ile nie zostały wywołane przez `call`. Z kolei w tym drugim poleceniu musimy znaleźć odpowiedni blok `defproc` i wykonać go. Robimy to oczywiście przez wywołanie `execute_commands` dla odpowiedniego zakresu poleceń. I tu leży pewien problem: musimy wykonać wszystkie te czynności, co i w przypadku rekurencyjnego wywołania `execute_commands` dla instrukcji `if`. Tak więc czeka nas trochę kopiowania tamtego kodu:

```
case CMD_CALLPROC:
{
    int rbracket = -1, def = -1;
    for(int j=0; j<cmd_array.size(); ++j) //szukanie definicji
    {
         if(cmd_array[j].Type == CMD_DEFPROC && cmd_array[j].s1 == cmd_array[i].s1)
         {
              def = j;
              break;
         }
    }
    if(def < 0)
    {
         bError = true;
         cout << "Blad wykonania skryptu - nie znaleziono procedury " <<
cmd_array[i].s1 << endl;
         return true;
    }
    for(int j=def+1; j<cmd_array.size(); ++j) //szukanie prawej klamry
    {
         if(cmd_array[j].Type == CMD_RBRACKET && cmd_array[j].nNestingLevel ==
cmd_array[def].nNestingLevel)
         {
              rbracket = j;
              break;
         }
    }
    if(rbracket < 0)
    {
         bError = true;
         cout << "Blad wykonania skryptu - niedomkniety blok defproc" << endl;
         return true;
    }
    //wykonanie procedury
    ++nNestingLevel;
```

```
    execute_commands(cmd_array, def+1, rbracket);
    if(bError) return true;
    if(bUnwind)
    {
        if(nNestingLevel > 0)
        {
            --nNestingLevel;
            return false;
        }
        else
        {
            bUnwind = false;
            i = nNewPos;
            goto InsideLoop;
        }
    } // if(bUnwind)
}
break;
```

Myślę, że powyższy kod, mimo iż trochę przydługi, nie jest niejasny. Najpierw szukamy definicji procedury o nazwie takiej samej, jak parametr polecenia `call`. Następnie poszukujemy prawego nawiasu klamrowego (musi on mieć poziom zagnieżdżenia 1 - nasz język nie zezwala na zagnieżdżanie procedur). Nawias ten oznacza oczywiście zakończenie ciała procedury. Gdy już mamy obie te rzeczy, wywołujemy `execute_commands` dla przedziału: od początku definicji do prawej klamry. Jest to właśnie ów fragment, niemal identyczny z tym, który napisaliśmy dla instrukcji `goto`.

Pozostaje do zrobienia jedna, bardzo ważna rzecz. Otóż musimy jakoś zapewnić pomijanie definicji procedury w momencie, gdy nie jest ona wywołana (tj. licznik i po prostu doszedł właśnie do tego miejsca w tablicy `cmd_array`, gdzie znajduje się definicja procedury). Jednym z najprostszych sposobów jest... obsłużenie polecenia `defproc`! Do tej pory nie robiliśmy tego, gdyż polecenie to było umieszczane w tablicy poleceń tylko po to, by nasz interpreter wiedział, gdzie zaczyna się procedura. Ponieważ jednak wywołujemy `execute_commands` z parametrem `def+1` (czyli pomijamy instrukcję `defproc`), to możemy spokojnie obsłużyć to polecenie i wstawić tam następujące czynności:

- znalezienie prawej klamry (końca procedury)
- przeskoczenie do znalezionego punktu

A oto jak to zrobimy:

```
case CMD_DEFPROC:
{
    int rbracket = -1;
    for(int j=i+1; j<cmd_array.size(); ++j)
    {
        if(cmd_array[j].Type == CMD_RBRACKET &amp;&amp; cmd_array[j].nNestingLevel
== cmd_array[i].nNestingLevel)
        {
            rbracket = j;
            break;
        }
    }
    if(rbracket < 0)
    {
        bError = true;
        cout << "Blad wykonania skryptu - niedomkniety blok defproc" << endl;
        return true;
    }
    i = rbracket;
}
break;
```

Poszukiwanie prawej klamry żywcem skopiowaliśmy z naszego dotychczasowego kodu; zmieniliśmy tylko instrukcję inicjalizacji w pętli `for`. Reszta jest identyczna. Po znalezieniu klamry przesuwamy nad nią licznik `i` oraz kontynuujemy główną pętlę z tym nowym licznikiem, czyli przechodzimy do instrukcji następującej po ciele procedury. Pora przetestować nowy element naszego języka:

```
var z 10

// procedurka :-)
defproc testpr
{
 con_out "Test procedurki"
 if z != 10
 {
  con_out "Bla."
 }
 con_out "Poza warunkiem"
}
// wywołaj procedurkę
call testpr
call testpr
```

Warto zauważyć, że w przeciwieństwie do języka C++, u nas możliwe jest umieszczenie wywołania funkcji jeszcze przed jej definicją:

```
call test
con_out "Teraz nastapi definicja"
defproc test
{
 con_out "Oto procedura"
}
```

Pętle

Kolejnym niezbędnym elementem każdego języka programowania, którego jeszcze nie mamy, są pętle. Zrealizujemy sobie teraz ten rodzaj pętli, który (jak się powszechnie przyjęło) nazywa się pętlą `for`. Najlepszą chyba powstałą implementacją takiej pętli jest ta z języka C. Niestety, potrzebny jest do niej interpreter, który potrafi obliczać bardziej złożone wyrażenia, a jego ze względu na rozmiar artykułu pisać nie będziemy :-). Dlatego zadowolimy się najprostszą możliwą postacią pętli `for`, która będzie się prezentowała jakoś tak:

```
for i = 1 to 10
{
 //instrukcje
}
```

Tradycyjnie już, zaczynamy od dodania odpowiedniej stałej dla nowego polecenia:

```
enum ECmdType
{
	CMD_UNKNOWN,
	CMD_ASSIGN_VALUE,
	CMD_RBRACKET,
	CMD_IF,
	CMD_LABEL,
```

```
    CMD_GOTO,
    CMD_DEFPROC,
    CMD_CALLPROC,
    CMD_FOR //nowe
};
```

Następnie - też tradycyjnie - modyfikujemy parser. Pętla `for` nie jest zwykłym poleceniem - łączy się ona ze zmienną, pełniącą rolę licznika. W języku C (i wielu innych) programista ma tu do wyboru: albo użyje zadeklarowanej już wcześniej zmiennej, albo zadeklaruje nową w bloku kontrolnym pętli. My nie chcemy sobie zanadto komplikować sytuacji, więc w naszym języku skryptowym damy tylko tę drugą możliwość, przy czym deklaracja zmiennej-licznika będzie niejawna (podobnie jak w języku Basic). Oczywiście nie możemy przechowywać lokalnej zmiennej w naszym wektorze `Variables`, gdyż nie chcemy, aby był do niej dostęp z zewnątrz pętli ani żeby powodowała ona ewentualne konflikty nazw. Dlatego stworzymy dla zmiennych lokalnych (liczników pętli) oddzielną tablicę. A ponieważ nasze pętle będzie się dało zagnieżdżać, zmienne te dobrze by było przechowywać na stosie. Dołączamy więc niezbędny nagłówek:

```
#include <stack>
```

...i deklarujemy nasz stos dla lokalnych zmiennych:

```
stack<VARIABLE> LocalVariables;
```

Na ten stos będziemy wrzucać licznik pętli przed pierwszą jej iteracją, zaś zdejmować go po wykonaniu ostatniej iteracji. W funkcji `parse` nie ruszamy stosu w ogóle; nie jest to potrzebne. Wystarczy, że zapamiętamy sobie nazwę zmiennej-licznika:

```
else if(cword == "for")
{
     cmd_info.Type = CMD_FOR;
     if(!get_token())
     {
          cout << "Blad, spodziewalem sie nazwy zmiennej..." << endl;
          return false;
     }
     cmd_info.s1 = cword;
     if(!get_token())
     {
          cout << "Blad, spodziewalem sie znaku \'=\'" << endl;
          return false;
     }
     if(!get_param())
     {
          cout << "Blad, spodziewalem sie wartosci poczatkowej licznika..." << endl;
          return false;
     }
     cmd_info.n1 = param;
     if(!get_token())
     {
          cout << "Blad, spodziewalem sie slowa \'to\'" << endl;
          return false;
     }
     if(!get_param())
     {
          cout << "Blad, spodziewalem sie wartosci koncowej licznika..." << endl;
          return false;
     }
     cmd_info.n2 = param;
     if(cmd_info.n1 > cmd_info.n2 || cmd_info.n1 < 0 || cmd_info.n2 < 0)
     {
          cout << "Bledne wartosci licznika petli! (" << cmd_info.n1 << " i " <<
```

```
cmd_info.n2 << ")" << endl;
            return false;
    }
}
```

Teraz wykonanie. Nie jest to nic szczególnie trudnego; możemy posłużyć się analogią do poleceń `if` i `call`. Różnica będzie polegać na tym, że tutaj `execute_commands` będziemy wywoływać w pętli (oczywiście pętla ta obejmie również nasz mechanizm odwijania stosu). Poza tym wewnątrz wspomnianej pętli musimy oczywiście przed każdą iteracją zwiększać wartość zmiennej, pełniącej rolę licznika, zaś na zewnątrz, przed pętlą - wrzucić tę zmienną na stos, a po pętli - zdjąć ją z tego stosu.

Warto zauważyć, że instrukcja `goto`, wyskakująca poza obręb dowolnej naszej pętli spowoduje, że zmienna lokalna nie zostanie zdjęta ze stosu (co nie jest niczym specjalnie groźnym), natomiast wskoczenie do pętli z zewnątrz może spowodować brak licznika na stosie lub odwołanie do licznika innej, "nadrzędnej" pętli. Moglibyśmy zabezpieczyć się przed takimi sytuacjami, modyfikując odpowiednio zachowanie się naszej instrukcji `goto`, ale w ten sposób narobilibyśmy również zamieszania w artukule, więc rezygnujemy z tych środków ostrożności (uczulając niniejszym na problem).

```
case CMD_FOR:
{
    int rbracket = -1;
    for(int j=i+1; j<cmd_array.size(); ++j) //szukanie prawej klamry
    {
        if(cmd_array[j].Type == CMD_RBRACKET && cmd_array[j].nNestingLevel ==
cmd_array[i].nNestingLevel)
        {
            rbracket = j;
            break;
        }
    }
    if(rbracket < 0)
    {
        bError = true;
        cout << "Blad wykonania skryptu - niedomkniety blok for" << endl;
        return true;
    }
    //wykonanie pêtli
    ++nNestingLevel;
    VARIABLE counter;
    counter.name = cmd_array[i].s1;
    counter.value = cmd_array[i].n1;
    LocalVariables.push(counter);
    for(int k=cmd_array[i].n1; k<=cmd_array[i].n2; ++k)
    {
        execute_commands(cmd_array, i+1, rbracket);
        if(bError) return true;
        if(bUnwind)
        {
            if(nNestingLevel > 0)
            {
                --nNestingLevel;
                return false;
            }
            else
            {
                bUnwind = false;
                i = nNewPos;
                goto InsideLoop;
            }
        } // if(bUnwind)
        get_local_variable(cmd_array[i].s1, true); //zwiêksz licznik pêtli
    } // for(int k
```

```
    LocalVariables.pop();
    i = rbracket;
}
break;
```

Wspomnieliśmy o zwiększaniu licznika pętli; robimy to po wykonaniu każdej iteracji. Licznik znajduje się na stosie, więc oczywiście nie możemy go modyfikować "z miejsca" - trzeba go odczytać ze stosu, zdjąć z niego, zwiększyć wartość i ponownie wrzucić na stos. Zajmuje się tym nowa funkcja `get_local_variable`, którą zaraz sobie dokładniej omówimy.

Poza wymienionymi rzeczami nie ma tu nic wartego omawiania, więc jedziemy dalej. Mamy przygotowaną zmienną lokalną na stosie, ale nigdzie nie wykorzystujemy jej wartości. Do tej pory jedyną możliwością wykorzystania zmiennej w naszym języku jest instrukcja `if`, ale ona na razie potrafi wykorzystać jedynie zmienne globalne (te deklarowane ze słowem `var`). Tak więc trzeba instrukcję `if` nieco zmodyfikować. Chcemy, aby w przypadku nieznalezienia właściwej zmiennej w wektorze `Variables` przeszukiwany był stos zmiennych lokalnych (czyli `LocalVariables`). Dzięki temu jeden i drugi rodzaj zmiennych będzie obsługiwany, przy czym zmienne globalne będą miały "pierwszeństwo".

W dotychczasowej postaci funkcji `parse` mieliśmy taką linijkę (w obsłudze polecenia `if`):

```
if(cmd_info.n1 < 0) return false;
```

Linijkę tę zamieniamy na następującą:

```
if(cmd_info.n1 < 0) cmd_info.s1 = cword;
```

Tak więc nie przerywamy już parsingu, gdy napotkamy nieznaną zmienną; zamiast tego przyjmujemy, że jest to zmienna lokalna i zapamiętujemy jej nazwę. Dopiero jeśli w fazie wykonania nazwy tej nie znajdziemy na stosie zmiennych lokalnych, zgłaszamy błąd[7]. Drobnej modyfikacji będzie też wymagała funkcja `get_var_index`. Musimy w niej wykomentować jedną linijkę:

```
int get_var_index(const string& sVarName)
{
     for(int i=0; i<Variables.size(); ++i)
          if(Variables[i].name == sVarName) return i;
     // cout << "Blad, nie znaleziono zmiennej o nazwie \'" << sVarName << "\'." <<
endl;
     return -1;
}
```

Pozostaje tylko poprawić funkcję `is_true` tak, aby pobierała wartość odpowiedniej zmiennej lokalnej, jeśli zmienna globalna nie została odnaleziona (tj. `nVarIndex` ma wartość -1). Problem w tym, że aby przeszukać stos, musimy z niego pozdejmować wszystkie "przeszkadzające" elementy (leżące nad tym, który aktualnie sprawdzamy), a przecież mogą się jeszcze kiedyś przydać :-). Tak więc musimy stworzyć drugi, tymczasowy stos, który pomoże nam w przeszukiwaniu tamtego. Działania te umieścimy w osobnej funkcji, o której już wspomnieliśmy. Do dzieła:

```
VARIABLE get_local_variable(const string& sName, bool bChangeValue)
{
     stack<VARIABLE> temp;
     VARIABLE var;
     bool bFound = false;
     while(LocalVariables.size())
     {
```

```
            var = LocalVariables.top();
            LocalVariables.pop();
            if(var.name == sName)
            {
                bFound = true;
                if(bChangeValue) ++var.value;
            }
            temp.push(var);
            if(bFound) break;
        }
        if(!bFound)
        {
            var.value = -1;
        }
        while(temp.size())
        {
            LocalVariables.push(temp.top());
            temp.pop();
        }
        return var;
}
```

Działanie tej funkcji jest następujące. Najpierw kolejno zdejmujemy wszystkie elementy ze stosu zmiennych lokalnych (jednocześnie wrzucając je na tymczasowy stos) i sprawdzamy każdy z nich, czy przypadkiem nie jest tą poszukiwaną zmienną. Jeśli tak, to przerywamy zdejmowanie i przechodzimy do drugiej pętli, która wkłada elementy z powrotem na stos zmiennych lokalnych. Jeśli wszystkie elementy zostały zdjęte i żaden nie jest tym właściwym, to nie wychodzimy z funkcji (przecież trzeba jeszcze przywrócić przeszukiwany stos do poprzedniej postaci), tylko ustawiamy zwracaną wartość na -1, co oznacza u nas błąd.

Dodatkowo funkcja `get_local_variable` potrafi ustawić dla znalezionej zmiennej nową wartość (konkretnie: zwiększyć ją o 1), co przydało nam się już wcześniej do zwiększania licznika pętli.

Stworzoną właśnie funkcję wykorzystamy ponownie w `execute_commands`, a dokładniej w miejscu, gdzie wywołujemy `is_true`. Nagłówek tej ostatniej musimy napierw trochę zmodyfikować - teraz nie będzie ona pobierać indeksu zmiennej, tylko jej wartość. Za tę wartość podstawiać - w zależności, czy będziemy mieć do czynienia ze zmienną globalną czy lokalną - wartość zmiennej z wektora `Variables` lub ze stosu `LocalVariables`.

```
bool is_true(int nVarValue, EOpType OpType, int nValue)
{
    switch(OpType)
    {
        case OP_EQUAL:      return (nVarValue == nValue);
        case OP_NOT_EQUAL:  return (nVarValue != nValue);
        case OP_LESS:       return (nVarValue < nValue);
        case OP_GREATER:    return (nVarValue > nValue);
        case OP_LESS_OR_EQUAL:    return (nVarValue <= nValue);
        case OP_GREATER_OR_EQUAL: return (nVarValue >= nValue);
    }
    return false;
}
```

Teraz jeszcze tylko poprawimy wywołanie funkcji `is_true` (w miejscu, gdzie obsługujemy wykonanie polecenia `CMD_IF`) tak, aby zamiast indeksu zmiennej pobierała jej wartość. Linijkę:

```
if(is_true(cmd_array[i].n1, (EOpType)cmd_array[i].n2, cmd_array[i].n3))
```

...zamieniamy na:

```
int nVarValue;
if(cmd_array[i].n1 < 0)
{
     VARIABLE tmp;
     tmp = get_local_variable(cmd_array[i].s1, false);
     nVarValue = tmp.value;
     if(nVarValue < 0)
     {
          bError = true;
          cout << "Blad, nie zadeklarowano zmiennej " << cmd_array[i].s1 << endl;
          return true;
     }
}
else
{
    nVarValue = Variables[cmd_array[i].n1].value;
}
if(is_true(nVarValue, (EOpType)cmd_array[i].n2, cmd_array[i].n3))
```

Mamy już wszystko, co potrzebne do funkcjonowania pętli. Możemy wykonać mały test:

```
for i = 2 to 5
{
 for j = 1 to 2
 {
  if i == 3
  {
   con_out "i rowne jest 3"
  }
 }
 con_out "Bla bla bla."
}

con_out "To byl test petli."
```

Dzięki naszym zabiegom ze stosem zmiennych lokalnych nawet w przypadku pętli zagnieżdżonych możliwe jest pobranie prawidłowego licznika pętli zewnętrznej w ciele pętli wewnętrznej. Obie pętle wykonują się dokładnie tyle razy, ile trzeba; liczniki nie są dostępne poza pętlami, a wartości początkowe i końcowe liczników nie mogą być nieprawidłowe, bo jest to sprawdzane jeszcze podczas parsingu. Tak więc wszystko gra :-).

Blok else

Jeden z największych dotychczasowych braków naszego języka to niemożność budowania bardziej złożonych konstrukcji z if-ów. Oczywiście, zawsze możemy tworzyć takie cuda:

```
if zmienna > 1
{
 con_out "Wieksza od 1"
}
if zmienna <= 1
{
 con_out "Nie większa od 1"
}
```

...ale taki sposób programowania oczywiście jest mało czytelny, co obniża jakość naszego języka. A wprowadzenie bloku `else` nie musi być wcale takie szalenie trudne. Żeby oszczędzić sobie pisania, zastosujemy małą sztuczkę: potraktujemy blok `else` jak samodzielny blok `if`, tylko "odwrócimy" mu warunek i zmienimy kod polecenia na `CMD_ELSE`. Ten ostatni oczywiście najpierw deklarujemy:

```
enum ECmdType
{
     CMD_UNKNOWN,
     CMD_ASSIGN_VALUE,
     CMD_RBRACKET,
     CMD_IF,
     CMD_ELSE, //nowe
     CMD_LABEL,
     CMD_GOTO,
     CMD_DEFPROC,
     CMD_CALLPROC,
     CMD_FOR,
     CMD_CON_OUT
};
```

Podczas parsingu nie ma żadnych niespodzianek; po pobraniu słowa kluczowego przeskakujemy lewą klamrę i zwiększamy poziom zagnieżdżenia. I tyle. Oto jak to zrobimy:

```
else if(cword == "else")
{
     cmd_info.Type = CMD_ELSE;
     if(!get_token() || cword != "{")
     {
          cout << "Blad, spodziewalem sie znaku \'{\'..." << endl;
          return false;
     }
     cmd_info.nNestingLevel = ++nCurNestingLevel;
}
```

Nieco bardziej skomplikowane będzie zmodyfikowanie warstwy wykonania. W przypadku napotkania w tablicy polecenia `CMD_ELSE` przede wszystkim musimy znaleźć pasujące polecenie `CMD_IF`. Dlatego też przeszukujemy tablicę wstecz. Pasujący `CMD_IF` to oczywiście ten, który ma ten sam poziom zagnieżdżenia; chyba, że wredny użytkownik zrobi nam kawał i napisze taki skrypt:

```
var a 100
if a == 1
{
 con_out "Ten tekst nie bedzie wypisany"
}
con_out "Ten bedzie"
else
{
 con_out "Ten tez bedzie"
}
```

Moglibyśmy obronić się przed takim efektem, ale kosztowałoby nas to trochę wysiłku, dlatego zostawmy ten drobny błąd w spokoju - będzie zabawniej ;-). Ważniejszą sprawą jest to, co zrobimy w momencie znalezienia pasującego if-a. Aby wykorzystać istniejący już kod polecenia `CMD_IF`, modyfikujemy go następująco:

```
case CMD_IF: case CMD_ELSE:
{
  int rbracket = -1, matching_if = -1;
  if(cmd_array[i].Type == CMD_ELSE)
  {
      for(int j=i; j>0; --j)
      {
          if(cmd_array[j].Type == CMD_IF && cmd_array[j].nNestingLevel ==
cmd_array[i].nNestingLevel)
          {
              matching_if = j;
              break;
          }
      } //for(int j
      if(matching_if < 0)
      {
          bError = true;
          cout << "Blad wykonania skryptu - else bez pasujacego if" << endl;
          return true;
      }
      cmd_array[i] = negate_operator((EOpType)cmd_array[matching_if].n2);
//"odwrócenie" warunku
  } //if(cmd_array[i].Type

  //dalej wszystko zostaje jak bylo
```

Jak widać, obecnie ten case "wyłapuje" dwa przypadki (`CMD_IF` i `CMD_ELSE`), poza tym rozbudowaliśmy go o fragment specyficzny dla przypadku `CMD_ELSE`. Fragment ten, jak już wspomnieliśmy, szuka pasującego bloku `CMD_IF`, "pożycza" sobie jego warunek i neguje go. Funkcja negująca operator będzie wyglądała tak:

```
EOpType negate_operator(EOpType op)
{
     switch(op)
     {
          case OP_EQUAL:    return OP_NOT_EQUAL;
          case OP_NOT_EQUAL: return OP_EQUAL;
          case OP_LESS:      return OP_GREATER_OR_EQUAL;
          case OP_GREATER:   return OP_LESS_OR_EQUAL;
          case OP_LESS_OR_EQUAL:    return OP_GREATER;
          case OP_GREATER_OR_EQUAL: return OP_LESS;
     }
     return OP_UNKNOWN;
}
```

Nie ma tu wiele do tłumaczenia. Po zanegowaniu warunku, blok `else` wykonywany jest dokładnie tak samo, jak `if`. Żeby było możliwe skompilowanie programu po tych zmianach, musimy oczywiście usunąć słówko `const` z nagłówka funkcji:

```
bool execute_commands(vector<COMMAND_INFO>& cmd_array, int start, int stop)
```

Pozostaje tylko sprawdzić, czy wszystko wyszło, jak wyjść powinno:

```
var zm 1
if zm == 2
{
 con_out "Ten tekst sie nie wyswietli"
```

```
}
else
{
 con_out "Blok else"
}
```

Piszemy grę :-)

Teraz pobawimy się trochę - napiszemy sobie grę w naszym języku. Oczywiście nie ma co liczyć na cuda w rodzaju nowego Quake'a, ale przy użyciu samej konsoli też możemy sobie stworzyć grę. Będzie to kultowe "Kółko i krzyżyk". Musimy sobie jeszcze dorzucić kilka nowych poleceń:

```
enum ECmdType
{
	CMD_UNKNOWN,
	CMD_ASSIGN_VALUE,
	CMD_RBRACKET,
	CMD_IF,
	CMD_ELSE,
	CMD_LABEL,
	CMD_GOTO,
	CMD_DEFPROC,
	CMD_CALLPROC,
	CMD_FOR,
	CMD_CON_OUT,
	CMD_CON_OUT_NOBR,  //nowe
	CMD_CON_IN,  //nowe
	CMD_CON_CLS //nowe
	CMD_SYSTEM //nowe
};
```

Uspokajam, że to już ostatnia modyfikacja tego enum-a :-). Pierwsze z poleceń będzie wyświetlało tekst, ale bez przechodzenia do nowej linii. Drugie będzie pobierało tekst z klawiatury. Trzecie wreszcie posłuży nam do czyszczenia ekranu. To wszystko, czego będziemy potrzebowali - czwarte i ostatnie z nowych poleceń dodajemy tylko jako ciekawostkę. Parsing całej czwórki to banał:

```
lse if(cword == "con_out_nobr")
{
	cmd_info.Type = CMD_CON_OUT_NOBR;
	if(!get_string())
	{
		cout << "Spodziewalem sie lancucha znakow..." << endl;
		return false;
	}
	cmd_info.s1 = cword;
   }
else if(cword == "con_in")
{
	cmd_info.Type = CMD_CON_IN;
	if(!get_token())
	{
		cout << "Blad, spodziewalem sie nazwy zmiennej..." << endl;
		return false;
	}
	cmd_info.n1 = get_var_index(cword);
}
else if(cword == "con_cls")
{
	cmd_info.Type = CMD_CON_CLS;
}
```

```
else if(cword == "system")
{
     cmd_info.Type = CMD_SYSTEM;
     if(!get_string())
     {
          cout << "Spodziewalem sie polecenia systemowego..." << endl;
          return false;
     }
     cmd_info.s1 = cword;
}
```

Wykonanie jest nawet jeszcze prostsze:

```
case CMD_CON_OUT_NOBR:
{
     cout << cmd_array[i].s1;
}
break;
case CMD_CON_IN:
{
     cin >> Variables[cmd_array[i].n1].value;
}
break;
case CMD_CON_CLS:
{
     system("cls");
}
break;
case CMD_SYSTEM:
{
     system(cmd_array[i].s1.c_str());
}
break;
```

Teraz możemy już napisać grę - tę i wiele, wiele innych :-).

```
// MinKiK - minimalistyczne kó³ko i krzy¿yk :-)
// (c) 2006 Piotr Bednaruk

var z1 0
var z2 0
var z3 0
var z4 0
var z5 0
var z6 0
var z7 1
var z8 0
var z9 0

var ruch 0
var egejn 0

defproc Rysiek
{
 con_cls
 con_out "MinKiK, wersja 1.0"
 if z7 == 0 { con_out_nobr " " }
 if z7 == 1 { con_out_nobr "O" }
 if z7 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z8 == 0 { con_out_nobr " " }
```

```
 if z8 == 1 { con_out_nobr "O" }
 if z8 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z9 == 0 { con_out_nobr " " }
 if z9 == 1 { con_out_nobr "O" }
 if z9 == 2 { con_out_nobr "X" }
 con_out " "
 con_out "-----"
 if z4 == 0 { con_out_nobr " " }
 if z4 == 1 { con_out_nobr "O" }
 if z4 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z5 == 0 { con_out_nobr " " }
 if z5 == 1 { con_out_nobr "O" }
 if z5 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z6 == 0 { con_out_nobr " " }
 if z6 == 1 { con_out_nobr "O" }
 if z6 == 2 { con_out_nobr "X" }
 con_out " "
 con_out "-----"
 if z1 == 0 { con_out_nobr " " }
 if z1 == 1 { con_out_nobr "O" }
 if z1 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z2 == 0 { con_out_nobr " " }
 if z2 == 1 { con_out_nobr "O" }
 if z2 == 2 { con_out_nobr "X" }
 con_out_nobr "|"
 if z3 == 0 { con_out_nobr " " }
 if z3 == 1 { con_out_nobr "O" }
 if z3 == 2 { con_out_nobr "X" }
 con_out " "
}

defproc Egejn
{
 con_out_nobr "Czy chcesz zagrac jeszcze raz? (0==nie, inna liczba - tak) "
 con_in egejn
 if egejn == 0 { goto Koniec }
 z1 = 0  z2 = 0 z3 = 0 z4 = 0 z5 = 0 z6 = 0 z7 = 1 z8 = 0 z9 = 0
 goto Gra
}

Gra:
 call Rysiek

 //sprawdzenie wygranej kompa
 if z7 == 1 { if z8 == 1 { if z9 == 1 { goto WygranaKompa } } }
 if z4 == 1 { if z5 == 1 { if z6 == 1 { goto WygranaKompa } } }
 if z1 == 1 { if z2 == 1 { if z3 == 1 { goto WygranaKompa } } }
 if z7 == 1 { if z4 == 1 { if z1 == 1 { goto WygranaKompa } } }
 if z8 == 1 { if z5 == 1 { if z2 == 1 { goto WygranaKompa } } }
 if z9 == 1 { if z6 == 1 { if z3 == 1 { goto WygranaKompa } } }
 if z9 == 1 { if z5 == 1 { if z1 == 1 { goto WygranaKompa } } }
 if z7 == 1 { if z5 == 1 { if z3 == 1 { goto WygranaKompa } } }
 //remis?
 if z1 > 0 { if z2 > 0 { if z3 > 0 { if z4 > 0 { if z5 > 0 { if z6 > 0 { if z8 > 0
{ if z9 > 0
 { goto Remis } } } } } } } }

 //ruch gracza
 con_out_nobr "Twoj ruch (patrz na klawiature numeryczna, 0==koniec): "
 con_in ruch
 if ruch == 1
 {
  if z1 > 0  { goto Gra }
  else { z1 = 2 }
 }
```

```
 if ruch == 2
 {
  if z2 > 0  { goto Gra }
  else { z2 = 2 }
 }
 if ruch == 3
 {
  if z3 > 0  { goto Gra }
  else { z3 = 2 }
 }
 if ruch == 4
 {
  if z4 > 0  { goto Gra }
  else { z4 = 2 }
 }
 if ruch == 5
 {
  if z5 > 0  { goto Gra }
  else { z5 = 2 }
 }
 if ruch == 6
 {
  if z6 > 0  { goto Gra }
  else { z6 = 2 }
 }
 //komp zawsze zaczyna z tego pola, wiêc jest ono ci¹gle zajête
 if ruch == 7
 {
  goto Gra
 }
 if ruch == 8
 {
  if z8 > 0  { goto Gra }
  else { z8 = 2 }
 }
 if ruch == 9
 {
  if z9 > 0  { goto Gra }
  else { z9 = 2 }
 }
 if ruch == 0 { goto Koniec }
 if ruch > 9 { goto Gra }
 call Rysiek

 //sprawdzenie wygranej gracza
 if z4 == 2 { if z5 == 2 { if z6 == 2 { goto WygranaGracza } } }
 if z1 == 2 { if z2 == 2 { if z3 == 2 { goto WygranaGracza } } }
 if z8 == 2 { if z5 == 2 { if z2 == 2 { goto WygranaGracza } } }
 if z9 == 2 { if z6 == 2 { if z3 == 2 { goto WygranaGracza } } }
 if z9 == 2 { if z5 == 2 { if z1 == 2 { goto WygranaGracza } } }

 //ruch kompa
 if z5 == 0  { z5 = 1 goto Gra }
 if z1 == 0  { z1 = 1 goto Gra }
 if z9 == 0  { z9 = 1 goto Gra }
 if z8 == 0  { z8 = 1 goto Gra }
 if z4 == 0  { z4 = 1 goto Gra }
 if z3 == 0  { z3 = 1 goto Gra }
 if z2 == 0  { z2 = 1 goto Gra }
 if z6 == 0  { z6 = 1 goto Gra }

WygranaKompa:

 con_out "Niestety, przegrales z glupim AI!"
 call Egejn

WygranaGracza:
```

```
 con_out "Wygrales, gratulacje!"
 call Egejn
Remis:
 con_out "Remis."
 call Egejn
Koniec:
```

Tak, oto cała gra w kółko i krzyżyk :-). Nie jest to na pewno arcydzieło sztuki koderskiej (a dokładniej to każdy szanujący się programista powinien dostać na taki widok apopleksji), ale działa - mamy grę, napisaną w naszym własnym języku skryptowym!

Dodam jeszcze, że pisanie kompletnych gier od A do Z w języku skryptowym jest raczej rzadkością; tutaj uczyniliśmy to dlatego, by pokazać możliwości naszego nowego języka. Zazwyczaj skrypty wykorzystuje się jednak raczej w charakterze pomocniczym - dokładniej zostało to opisane we Wstępie.

Co jeszcze chciałbyś napisać? Może menedżer plików? Proszę bardzo - nasze nowe polecenie umożliwia nam wykonanie dowolnego polecenia systemowego! Popatrz tylko:

```
con_out "Teraz wypiszemy wszystkie pliki z biezacego foldera."
system "dir"
con_out "Wyniki zachowane zostaly w plik.txt"
system "dir > plik.txt"
```

Podsumowanie

Wynikiem naszych starań jest kompletny język skryptowy. Oczywiście "kompletny" w sensie, że obecne są w nim wszystkie najważniejsze elementy, takie jak instrukcje warunkowe, pętle, procedury, instrukcja skoku. Wiele elementów można by jeszcze dodać (tablice, wskaźniki, odpowiedniki instrukcji `switch, break, continue, return` itp.), można by również ulepszyć już istniejące (np. obsługa złożonych wyrażeń, argumenty dla funkcji...). Jednak główne zasady funkcjonowania języków skryptowych zostały pokazane.

Nasz język potrafi zrobić prawie wszystko, co może się odbywać w konsoli tekstowej. Jednak aby stał się "prawdziwym" językiem programowania, należałoby rozbudować jego zestaw poleceń tak, aby język nasz potrafił wykonywać typowe czynności, które wykonują zwykle standardowe biblioteki innych języków wysokopoziomowych: generować liczby losowe, konwertować liczby na tekst i odwrotnie, obliczać wartości funkcji matematycznych (sinus, pierwiastek, logarytm...), obsługiwać inne niż standardowe urządzenia wejściowe i wyjściowe (np. mysz), alokować pamięć etc. Fajną sprawą byłaby też możliwość wywoływania funkcji z DLL - wtedy moglibyśmy nawet tworzyć programy okienkowe w naszym języku :-). Wszystko to jednak wykracza poza wymagania, stawiane zazwyczaj językom skryptowym. Najczęściej bowiem język skryptowy wykonuje tylko czynności związane np. z grą, dla której został stworzony. Przykładem mogą być polecenia typu `create_monster` z pierwszej części tego artykułu.

Wielkim brakiem w naszym języku jest obsługa rozmaitych błędów, które może popełnić użytkownik przy wpisywaniu skryptu. Wyłapanie absolutnie wszystkich takich błędów jest oczywiście niemożliwe w praktyce; nawet twórcy kompilatorów tak popularnych języków, jak np. C++ nie ustrzegli się pewnych niedopatrzeń. Granica dopuszczalnej liczby takich błędów w języku skryptowym jest trudna do wyznaczenia; w tym artykule nie przejmowałem się nią zbytnio. Przedstawiony, przykładowy interpreter "przymyka oko" na bardzo wiele rzeczy; wszystko to jest raczej celowe. Im więcej błędów chcielibyśmy obsłużyć, tym mniej przejrzysty byłby przykładowy kod źródłowy interpretera, a artykuł - trudniejszy w

odbiorze. Dlatego ograniczyłem obsługę błędów do niezbędnego minimum.

Z powodów wyżej wymienionych przykładowy język nie może pretendować do miana uniwersalnego języka skryptowego do jakiegokolwiek profesjonalnego zastosowania. Jednak myślę, że po drobnych przeróbkach można go z powodzeniem wykorzystać w nawet dość złożonej grze.

Mam nadzieję, że ten artykuł okaże się przydatny. Nawet jeśli pogardzisz moim przykładowym językiem skryptowym jako nieprzydatnym do "prawdziwych" zastosowań, albo w ogóle przedstawionymi metodami tworzenia takich języków, to przynajmniej może dzięki temu artykułowi będziesz miał okazję zajrzeć "do środka" języka i poznasz pewne uniwersalne reguły, którymi rządzi się jego tworzenie. Trzeba przyznać, że język programowania to jeden z tych wynalazków, których używamy na codzień, zwykle nie zastanawiając się głębiej, jak one funkcjonują. Może dzięki temu artykułowi będziesz trochę mniej narzekał, że funkcje w C++ muszą mieć swoje prototypy, nie da się deklarować wskaźników na funkcje składowe klasy, a kompilator pokazuje komunikaty kompletnie nieadekwatne do popełnionego błędu :-).

Link do kompletnego kodu źródłowego interpretera przedstawionego języka skryptowego:

skrypty-src.rar

---

[1] Wiele gier korzysta jednak również z kompilowanych skryptów. Są to po prostu instrukcje skryptu już po parsingu i interpretacji, zapisane w postaci binarnej (a więc nieczytelne dla (normalnego) człowieka). Powód kompilowania skryptów jest prosty: wczytują się znacznie szybciej. Ponadto użytkownik końcowy nie może wówczas grzebać w "sekretach" gry (chyba, że naprawdę bardzo chce ;-)).

[2] Aby uniknąć jałowych polemik od razu tłumaczę, że mam tu ma myśli praktykę programistyczną, a nie teorię, która mówi zawsze o sytuacjach idealnych. W teorii bowiem programy powinno się projektować tak, aby wprowadzanie w nich wszelkich zmian było możliwie najprostsze. Jak wiemy, różnie z tym bywa ;-).

[3] O wykorzystywaniu windowsowej konsoli w grach i innych aplikacjach dowiesz się z artykułu Adama Sawickiego "Asynchroniczna konsola Windows" http://www.regedit.i365.pl/warsztat/articles.php?x=view&id=204

[4] B. Eckel "Thinking In C++, Volume 2", str. 134

[5] Choćby po to, by nie powtarzać tego, co napisał już DarkJarek w swoim artykule "Parser wyrażeń matematycznych". Zakładając, że wyrażenia o wartości zerowej traktujemy jako "fałsz", a niezerowe - jako "prawdę", możemy z powodzeniem wykorzystać parser DarkJarka do sprawdzania nawet najbardziej złożonych warunków.

[6] Dlaczego i czy słusznie - rozważa o tym między innymi Bruce Eckel w swojej znanej książce "Thinking In Java", gdzie nawet poświęcił tej dyskusji osobny podrozdział.

[7] Oczywiście takie postępowanie nie jest do końca słuszne - wszystkie błędy, jakie da się wykryć w warstwie parsingu, powinny być zgłoszone już wtedy. Dzięki temu możemy oddzielić poszczególne warstwy (aby np. zrobić inną warstwę interpretacji/wykonania w specjalnym edytorze skryptów, a inną w samej grze). Myślę jednak, że w tym artykule możemy sobie pozwolić na nieco luźniejsze podejście.