Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Paweł Aleksander Fedoryński
Nr albumu: 2097

Silnik polskojęzycznych tekstowych gier przygodowych

Praca magisterska
na kierunku INFORMATYKA
w zakresie PRZETWARZANIA JĘZYKA NATURALNEGO

Praca wykonana pod kierunkiem
dra Krzysztofa Szafrana
Instytut Informatyki

Wrzesień 2005

Oświadczenie kierującego pracą

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiono algorytmy, służące do analizy wprowadzanych przez użytkownika zdań w języku naturalnym. Polecenia są poddawane rozbiorowi gramatycznemu, a następnie interpretacji, uwzględniającej kontekst, w jakim zostały wypowiedziane. Przedstawiony mechanizm umożliwia stworzenie gry, przyjmującej bardziej złożone polecenia niż wszystkie dotąd istniejące polskojęzyczne gry tekstowe i MUDy. W ramach pracy przedstawiona została przykładowa implementacja opisanego silnika w języku Common Lisp, która następnie została wykorzystana do stworzenia polskojęzycznego *remake*'u klasycznej gry "Hobbit".

Słowa kluczowe

interactive fiction, gry przygodowe, rozbiór gramatyczny, przetwarzenie języka naturalnego, Common Lisp

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

I. Computing Methodologies
I.2 Artificial Intelligence
I.2.7 Natural Language Processing

Spis treści

Rozdział 1

Wstęp

W tekstowych grach przygodowych (ang. *interactive fiction*) gracz steruje zachowaniem postaci, w którą się wciela, poprzez wydawanie tekstowych poleceń w języku naturalnym. Informacje o skutkach podejmowanych działań otrzymuje również w postaci tekstowej. Tematem niniejszej pracy są algorytmy, umożliwiające interpretację wprowadzanych poleceń w grach w języku polskim.

1.1. Wprowadzenie historyczne

Anglojęzyczne tekstowe gry przygodowe to bardzo obszerny temat; przedstawimy krótkie wprowadzenie, ze szczególnym uwzględnieniem aspektów związanych z tematem pracy. Następnie przejdziemy do omówienia gier polskojęzycznych.

1.1.1. *Interactive fiction*

Pierwsza w historii *interactive fiction* to stworzona przez Willa Crowthera w 1975 roku "Adventure" (znana później pod nazwą "Colossal Cave Adventure"). Fabuła polegała na eksploracji połączonych jaskiń (Crowther wykorzystał swoje doświadczenie grotołaza). Gra była napisana w języku FORTRAN, na komputerze PDP-10[1]. Autor stworzył "Adventure" z myślą o swoich dwóch córkach, jednak program szybko zdobył popularność wśród ówczesnych użytkowników Internetu. W 1976 roku Don Woods, hacker ze Stanford Artificial Intelligence Lab, znacznie rozbudował grę, dodając do niej zapożyczone od Tolkiena elementy fantastyczne (elfy, trolla, przypominający Górę Przeznaczenia wulkan). W tym samym roku Jim Gillogly z Rand Corporation stworzył wersję uniksową, przepisując kod z języka FORTRAN do C. W 1981 roku firma The Software Toolworks (przy współpracy oryginalnych autorów, Crowthera i Woodsa) wypuściła na rynek komercyjną wersję na IBM PC, pod nazwą "The Original Adventure".

Przełom lat siedemdziesiątych i osiemdziesiątych to okres świetności gier tekstowych. W Stanach Zjednoczonych największe znaczenie miała na tym rynku firma Infocom, wydawca m.in. klasycznej gry "Zork". Aby poradzić sobie z problemem przenośności, programiści z Infocomu opracowali wirtualną maszynę, *Z-machine*; poszczególne gry były plikami wejściowymi do *Z-machine*, w formacie nazywanym *Z-code*[2]. Pliki te były przygotowywane

[1] Kod źródłowy oryginalnej gry "Adventure" jest dostępny, jednak wykorzystuje on niskopoziomowe szczegóły architektury PDP-10 i nie daje się w prosty sposób skompilować na innych komputerach.

[2] Możliwe, że *Z-machine* jest najbardziej przenośną maszyną wirtualną w historii informatyki; lista obsługiwanych platform rozciąga się od komputerów ośmiobitowych po współczesne systemy operacyjne Windows, Linux i Mac OS X.

Rysunek 1.1: *Screenshot* z gry "Hobbit"

przez kompilowanie źródeł w języku ZIL (*Zork Implementation Language*). Używany przez Infocom kompilator ZILa nie został nigdy udostępniony, jednak, dzięki rozpracowaniu formatu *Z-code*'u przez grupę entuzjastów pod nazwą InfoTaskForce, sama *Z-machine* nadal żyje i jest wykorzystywana do tworzenia nowych gier. Używanym obecnie sposobem generowania *Z-code*'u jest kompilowanie plików źródłowych w języku Inform, stworzonym w 1993 roku przez Grahama Nelsona [1].

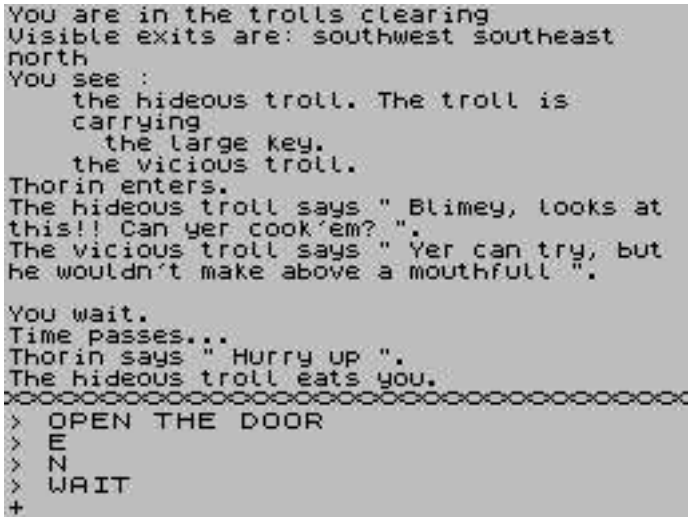
Infocom wydał dziesiątki gier tekstowych (w tym szereg sequeli do "Zorka"), które sprzedawały się znakomicie. Jednak od połowy lat osiemdziesiątych, między innymi w związku z niezbyt udaną próbą rozszerzenia działalności na aplikacje biznesowe, firma zaczęła przynosić straty i ostatecznie została przejęta przez Activision, i w 1989 roku zlikwidowana [2]. Niektóre tytuły Infocomu (m.in. "Zork") zostały udostępnione i można je legalnie ściągnąć z Internetu, jednak większość nadal jest chroniona copyrightem, i sprzedawana przez Activision w różnych składankach.

Poza Ameryką najważniejszym ośrodkiem rozwoju gier tekstowych była australijska firma Beam Software. Jej pierwszym i największym przebojem okazała się opublikowana w 1982 roku gra "Hobbit" [3], oparta na powieści J.R.R. Tolkiena. Stworzony przez pracujących na pół etatu studentów[3] program wzbudził niezwykłe uwielbienie[4]. "Hobbit" jest — w każdym razie w Europie — najbardziej klasyczną tekstową grą przygodową wszechczasów; większość graczy właśnie dzięki niemu po raz pierwszy zetknęła się z tym gatunkiem.

Źródeł sukcesu można dociekać różnie. Po pierwsze, "Hobbit" działał na ośmiobitowych domowych komputerach ZX Spectrum i Commodore 64, i był pierwszą podobnie rozbudowaną grą dla tego kręgu odbiorców. Dodatkowym czynnikiem przyciągającym klientów było włączenie grafiki, ilustrującej niektóre lokacje. Niezwykła, jak na tamte czasy, była samodzielność innych postaci (NPC — *Non-Player Characters*) — mogły się przemieszczać, otwierać i zamykać drzwi, walczyć pomiędzy sobą, i wykonywać inne działania, według podobnych reguł jakie dotyczyły działań gracza, ale zupełnie niezależnie od niego. Zdarzało się na przykład, że roz-

[3]Głównym programistą silnika był Philip Mitchell; Veronika Megler projektowała treść gry, programowała zachowanie postaci itd.; oprócz tego nad projektem pracowali Stuart Ritchie i Alfred Milgrom. Praca nad "Hobbitem" trwała 18 miesięcy.

[4]Różne źródła podają liczby między pół miliona a milionem łącznie sprzedanych egzemplarzy. Jest to wynik jak na tamte czasy rekordowy.



Rysunek 1.2: *Screenshot* z gry "Hobbit"

stawaliśmy się z Gandalfem, by po pewnym czasie nieoczekiwanie znów go spotkać w Górach Mglistych lub w lochu goblinów. Albo natrafialiśmy w drodze na zwłoki postaci, które zadarły z kimś silniejszym od siebie. Zwiększało to bardzo poczucie realności świata gry. Postacie reagowały na działania gracza (i swoje wzajemnie) — zaatakowane broniły się i atakowały odwetowo w następnych turach. Można było też rozmawiać z innymi postaciami, czasem nawet okazywało się to konieczne ('Say to Bard "Shoot the dragon"'). Fabuła była dość złożona, nieliniowa (tzn. większość zagadek miała co najmniej dwa różne rozwiązania) i raczej wierna książce. Ciekawą cechą było też wprowadzenie namiastki czasu rzeczywistego — jeśli przez dłuższą chwilę gracz nie dotykał klawiatury, to program automatycznie wprowadzał polecenie "Wait"; taka chwila nieuwagi mogła się źle skończyć (jak pokazuje ilustracja 1.2).

Jednak chyba najbardziej przełomowym aspektem był sposób wprowadzania poleceń. Standardem dla gier tekstowych w tamtych czasach — przynajmniej jeśli chodzi o komputery domowe — było ubogie słownictwo i jedno- lub dwuwyrazowe polecenia typu "Look" czy "Open door". "Hobbit" znał ponad 500 słów i rozumiał złożone zdania w rodzaju "Open everything except the wooden chest", "Go east then north", "Take the rope and the sword" czy "Throw the barrel through the large trap door and jump onto it". Autorzy byli bardzo dumni[5] z możliwości językowych "Hobbita". Określili obsługiwany przez niego podzbiór języka angielskiego mianem "Inglish" i w dołączanej do gry książeczce zamieścili krótkie wprowadzenie do języka *Inglish*. Dozwolone były czasowniki i rzeczowniki ("Open the door"), przymiotniki ("Kill the hideous troll"), ale także przysłówki ("Carefully attack the goblin"), i różnego rodzaju przyimki ("Climb into the boat", "Kill the goblin with the sword", ...). *Inglish* zawierał też (co pokazują poprzednie przykłady) różne znaczenia słowa "and". Program pozwalał na wydawanie poleceń w formie skróconej, o ile tylko były one jednoznaczne. Na przykład, jeśli jedynym przedmiotem w polu widzenia gracza był klucz, można było go wziąć pisząc po prostu "Take". Jeśli w zasięgu był klucz i lina, konieczna była większa precyzja: "Take the key"[6]. Gdyby akurat gracz mógł wziąć dwa klucze, należałoby dodatkowo użyć przymiotnika: "Take the golden key".

[5]Dołączana do gry instrukcja zaczyna się od słów: "Congratulations! You are about to play the most sophisticated game program yet devised for any microcomputer."

[6]Tak naprawdę, "the" we wszystkich przykładach jest całkowicie niekonieczne. Interpreter po prostu je pomijał.

Inne stworzone przez tę samą firmę tytuły to "Sherlock" (gra kryminalna o Sherlocku Holmesie) i "Lord of the Rings". Były one udane, jednak nie powtórzyły w pełni sukcesu "Hobbita".

Firma Beam Software zezwoliła na swobodne kopiowanie wyprodukowanych przez nią gier; można je ściągnąć z Internetu i uruchomić na emulatorze. Wersja "Hobbita" na ZX Spectrum została zdeasemblowana i opatrzona komentarzem przez Chrisa Wilda i Seana Irvine'a; zatem kod źródłowy (w asemblerze Z80) dostępny jest w Internecie [4].

Mniej więcej od początku lat 90-tych gry tekstowe straciły atrakcyjność dla masowego odbiorcy i komercyjne firmy przerzuciły się na inne rodzaje rozrywki. Jednak istnieje społeczność hobbystów, którzy dalej tworzą gry tekstowe i w nie grają. Autorzy najlepszych są nagradzani branżową nagrodą *XYZZY Award*, przyznawaną w kilku kategoriach (za najlepszą grę, najlepszą postać, największą wartość literacką itd.). Generalnie uważa się, że współczesne, tworzone amatorsko gry przewyższyły komercyjne produkty z lat osiemdziesiątych.

1.1.2. Polskojęzyczne tekstowe gry przygodowe

Gry tekstowe po polsku pojawiły się znacznie później niż angielskie i nigdy nie osiągnęły podobnej popularności. Pierwsze poważne próby zaatakowania problemu podjęto w drugiej połowie lat 80-tych. Ówczesne podejście do zagadnienia opisuje artykuł Andrzeja Kadlofa [5]. Autor przedstawił w nim budowę silnika polskojęzycznych gier przygodowych, z kompletnym omówieniem algorytmu, formatów danych, a nawet wskazówkami dotyczącymi implementacji; artykuł jest zilustrowany przykładami i fragmentami kodu w języku BASIC. Większość tego materiału ma dziś znaczenie czysto historyczne, warto jednak przyjrzeć się fragmentowi, omawiającemu analizę poleceń gracza.

Pomysł polegał na pozostawieniu z każdego słowa rdzenia, otrzymywanego przez odrzucenie ewentualnego prefiksu ("ZA", "NA", "PO", ...) i wzięcie pierwszych czterech (lub mniej, jeśli wyraz jest krótszy) liter z reszty; np. "Zamknij bramę" → "Mkni bram". Przetworzone w ten sposób wyrazy były następnie, od lewej do prawej, sprawdzane w słowniku. Wyrazy nierozpoznane algorytm ignorował. Pętla kończyła się w momencie natrafienia na drugi znany programowi wyraz. W ten sposób np. każde ze zdań: "Zamknij bramę", "Zamknij żelazną bramę", "Przymknij bramę", "Domknij ostrożnie żelazną bramę", dawało ten sam wynik "mkni bram"; podobnie np. każde ze zdań "Biegnij na wschód", "Pobiegnij szybko w kierunku wschodnim", "Biegnij w stronę wschodu" sprowadzało się do "bieg wsch" (wielkość liter była ignorowana). Tak uproszczona instrukcja była dalej sprawdzana w słowniku instrukcji i zależnie od stanu gry, wykonywana lub nie (każde akceptowane przez grę polecenie musiało się składać z dokładnie dwóch wyrazów).

Pierwszą opublikowaną grą tekstową po polsku był "Smok wawelski" [6] na komputer ZX Spectrum. Wydaje się, że autorzy zastosowali algorytm podobny, co do idei, do opisanego w zacytowanym artykule. Nie został wykorzystany pomysł odrzucania przedrostków — widocznie w praktyce nie dawało to istotnych oszczędności. Pewnym usprawnieniem jest też zmienna długość wyrazu w słowniku — zamiast brać zawsze czteroliterowe prefiksy, program porównuje słowo ze słownika z prefiksem wprowadzonego wyrazu, o długości równej długości słowa ze słownika. Np. "otwórz" jest rozpoznane jako "otw", ale "drzwi" jako pełne "drzwi". Wydaje się to rozsądne — zauważmy np. że przy prefiksach czteroliterowych, "wodę", w zdaniu "wypij wodę z butelki", oraz "wody", w zdaniu "napij się wody", musielibyśmy traktować jako różne wyrazy (za to "wypij" i "napij" przy odrzucaniu przedrostków zostałyby utożsamione).

Pod względem fabuły i złożoności świata gry "Smok Wawelski" ustępował zachodnim tytułom, jednak niewątpliwie stanowi jedno ze szczytowych osiągnięć gatunku. Najwybit-

Rysunek 1.3: *Screenshot* z gry "Smok Wawelski"

niejszym dziełem tego typu była prawdopodobnie późniejsza gra "Mózgprocesor". Miała ona rozleglejszą mapę oraz bardziej rozbudowaną fabułę i była dobrze przyjęta przez graczy (ukazała się wersja na ZX Spectrum oraz na Atari). Wydaje się jednak, że "Mózgprocesor" był stworzony w oparciu o ten sam, lub podobny, silnik co "Smok Wawelski" i od strony możliwości językowych te gry różnią się niewiele.

Niestety, polskojęzyczna *interactive fiction* właściwie zupełnie przestała istnieć wraz ze zmierzchem gier komercyjnych. Przez pewien czas pojawiały się niekomercyjne produkcje hobbystów (można tu wymienić komediową grę "Pięć gówien Eepcha" na komputer Atari) lecz z czasem i one wymarły[7].

1.1.3. Współczesne polskojęzyczne MUDy

MUDy (ang. *Multi-User Dungeon*) są to tekstowe gry sieciowe, które oryginalnie wyewoluowały z *interactive fiction*. Obecnie stanowią zupełnie inny rodzaj rozrywki, jednak nadal prowadzenie gry polega na wpisywaniu poleceń i czytaniu pojawiających się komunikatów.

Polskojęzyczne MUDy, oparte na przerobionych silnikach angielskich MUDów, pojawiły się dość dawno. Początkowo tłumaczenia były dość toporne, nie uwzględniały odmiany przez przypadki, polskich znaków diakrytycznych, dotyczyły jedynie opisów itp[8]. Przełomem był, otwarty dla graczy w 1997 roku, MUD Arkadia [7], oparty na mocno przerobionym silniku anglojęzycznego MUDa Genesis. Arkadia była pierwszym MUDem w pełni polskojęzycznym i do dziś jest najpopularniejszym z polskich MUDów. Po udostępnieniu silnika Arkadii w 2001 roku powstała pewna liczba projektów na nim opartych, z których najistotniejszym jest działający do dziś MUD Barsawia.

W obu MUDach można wydawać polecenia w języku polskim, jednak z pewnymi ograniczeniami. Na przykład, istotna jest kolejność wyrażeń — jeśli chcemy coś wyjąć ze skrzyni, musimy najpierw napisać co, a potem z czego, itp.

[7] Jedyne, co udało mi się znaleźć, to projekt "Otchłań", jednak — jak określa to sam autor — nie jest to *interactive fiction* a raczej jednoosobowy MUD, co przekłada się na trochę inne rozłożenie priorytetów. W szczególności strona językowa jest zupełnie zlekceważona, włącznie z pominięciem odmiany rzeczowników, możemy np. zobaczyć komunikat "Kupujesz maczuga".

[8] "Glowa czarny rycerz rolls over trawnik."

1.2. Cel i zakres pracy

Opisana w poprzednim podrozdziale heurystyczna analiza polskojęzycznych poleceń, opierająca się na obcinaniu końcówek wyrazów i wybieraniu dwóch znaczących słów z każdego zdania, sprawdzała się znakomicie. Niemniej jest to rozwiązanie pod pewnymi względami niesatysfakcjonujące. Akceptowanie poleceń w rodzaju "Otwabcxyz qwerty drzwiuiop aaa" na równi z "Otwórz drewniane drzwi" to nie jest zdrowa sytuacja — wyraźnie widać, że przedstawione rozwiązanie (choć sprytne i skuteczne) stanowi zaledwie pierwszy krok. Nie bardzo wiadomo, jak można by je rozbudowywać w kierunku rozumienia bardziej skomplikowanych zdań, gdyż w takich zdaniach większość informacji o funkcji wyrazu pochodzi właśnie z jego końcówki.

Wiąże się to ze wspólnym dla wszystkich wymienionych w poprzednim podrozdziale silników — także silników MUDów — nierozumieniem szyku przestawnego. Nie jest ono może wielkim problemem samo w sobie — raczej rzadko ktoś, w dobrej wierze, napisze "Drzwi otwórz" — jednak jest symptomem czegoś poważniejszego: oznacza, iż algorytm nie odzwierciedla podstawowej cechy języka polskiego, w którym — inaczej niż np. w języku angielskim — rola wyrazu zależy nie od jego położenia w zdaniu, lecz od formy gramatycznej. Co więcej, problem nabiera na znaczeniu wraz ze wzrostem komplikacji wypowiedzi: zdanie "Drzwi otwórz" jest bardzo nienaturalne, ale już "Zabij mieczem trolla" brzmi dużo lepiej, a zdania "Przełóż pieniądze ze skrzyni do kuferka" i "Przełóż pieniądze do kuferka ze skrzyni" są niemal równie dobre.

Celem tej pracy jest zaprezentowanie teoretyczne, oraz zaimplementowanie, innego, nowego rozwiązania, które umożliwiałoby analizę bardziej skomplikowanych zdań niż dotychczasowe silniki, a przy tym było łatwe do dalszej rozbudowy. Przedstawiony algorytm doprowadza pobłażliwość dla szyku przestawnego do skrajności — czerpie informacje niemal wyłącznie z form gramatycznych wyrazów, zwracając uwagę na ich położenie w zdaniu tylko wtedy, gdy jest to konieczne. Zaakceptuje na przykład na równi zdania "Otwórz zielone drzwi", "Otwórz drzwi zielone", "Drzwi zielone otwórz", "Zielone otwórz drzwi" itd. Za wzór siły wyrazu i semantyki implementowanego podzbioru języka polskiego przyjąłem *Inglish*, czyli podzbiór języka angielskiego obsługiwany przez grę "Hobbit". Udało się odtworzyć wiele jego cech; przedstawiony program rozumie polecenia takie jak "Zabij ohydnego trolla mieczem", "Wejdź do beczki", "Otwórz wszystko oprócz drewnianej skrzyni" czy "Wyjmij ze skrzyni wszystkie klucze oprócz ogromnego". Możliwa jest też, jak w "Hobbicie", interakcja między postaciami ('Powiedz do Gandalfa "Daj mi mapę"', 'Powiedz Thorinowi "Zabij wszystkie trolle"'). Również na wzór "Hobbita" program umożliwia wydawanie poleceń skróconych, w sytuacji gdy nie prowadzi to do niejednoznaczności: można napisać "Otwórz okrągłe, zielone drzwi", ale jeśli te drzwi są jedyną rzeczą, jaką postać może otworzyć, wystarczy polecenie "Otwórz". Jeśli można otworzyć drzwi lub skrzynię, to gracz musi wpisać "Otwórz drzwi", itd. Jednak sukces nie jest całkowity — głównym brakiem, w porównaniu z wersją angielską, jest brak obsługi spójnika "i".

Rozdział 2 opisuje zaproponowany silnik w sposób oderwany od implementacji, koncentrując się na koncepcyjnym modelu danych i algorytmach, nie wnikając, jak konkretnie te dane będą przechowywane, ani jaka będzie struktura kodu czy użyty język programowania. W rozdziale 3 omawiamy przygotowaną w ramach pracy przykładową implementację. Wreszcie rozdział 4 prezentuje, działającą w oparciu o zaimplementowany silnik, grę wzorowaną na oryginalnym "Hobbicie".

Cały stworzony w ramach pracy kod objęty jest licencją GNU GPL.

Rozdział 2

Działanie programu

2.1. Uwaga na temat terminologii

Przedstawiony opis nie wymaga od czytelnika żadnej specjalistycznej wiedzy na temat języka polskiego. Stosowana terminologia nie wykracza poza wiedzę ze szkoły średniej (części mowy, przypadki, rodzaje). Wszystkie występujące pojęcia są omówione np. w [8].

Często pojawiający się termin "miejsce składniowe" został wprowadzony na potrzeby tej pracy; jego definicja znajduje się w rozdziale 2.3.

2.2. Model danych

Rysunek przedstawia używane przez program zbiorniki z danymi i związki pomiędzy nimi. Na górze każdej prostokątnej ramki znajduje się nazwa zbiornika (np. `postacie`), a poniżej wymienione są nazwy pól w należących do tego zbiornika rekordach (np. `gdzie_lok`, `gdzie_prz`, `gdzie_osoba`, `nazwa`, ...). Linie ze strzałką oznaczają, że rekord z jednego zbiornika jest skojarzony z co najwyżej jednym rekordem[1] z innego zbiornika. Linie bez strzałek oznaczają, że rekordy z połączonych zbiorników mogą być skojarzone dowolnie.

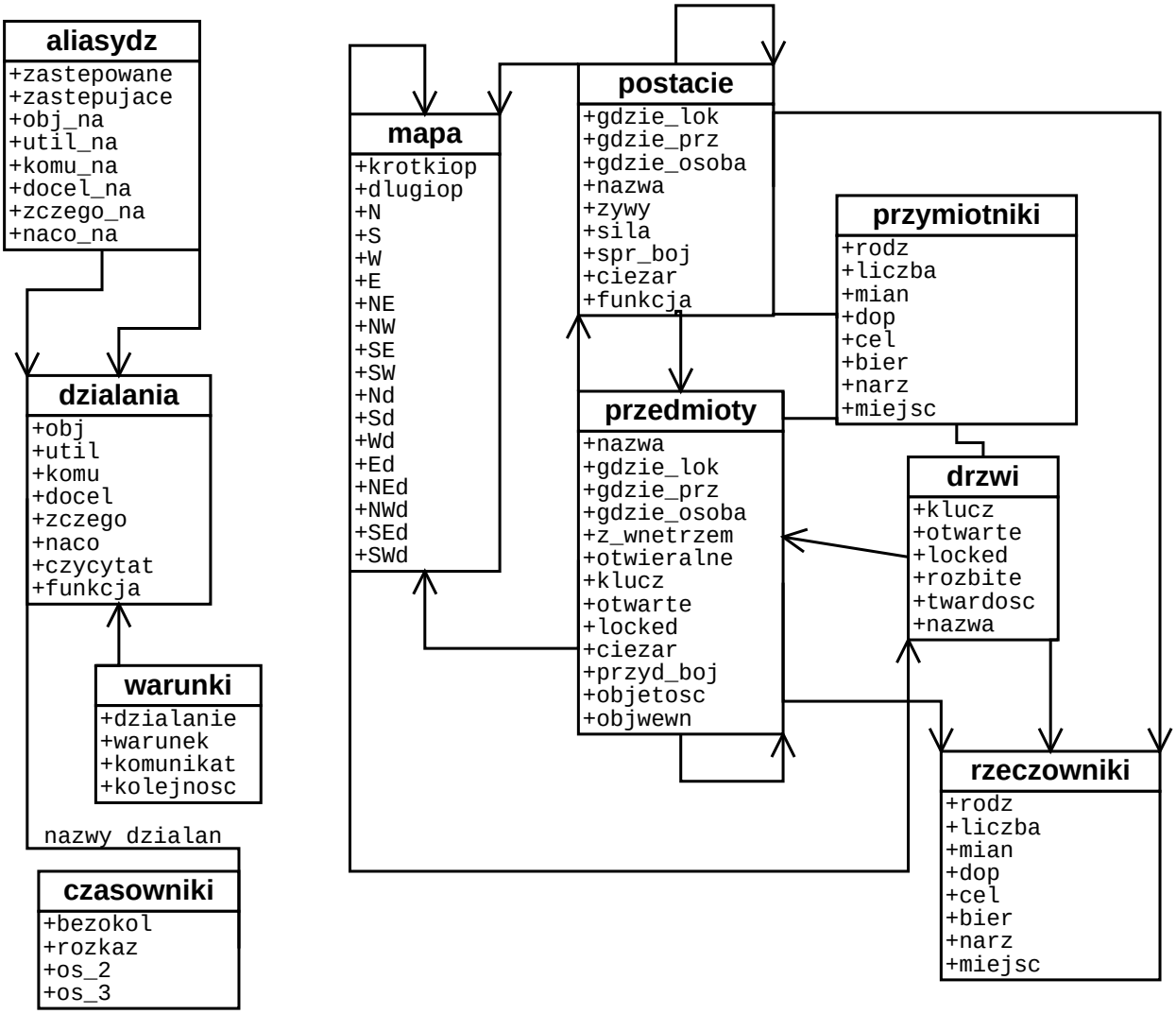
Omówione w tym rozdziale dane są trwałe, tzn. są wczytywane na początku gry i istnieją przez cały czas jej trwania (oczywiście, mogą być modyfikowane). Tymczasowe struktury danych, powoływane do istnienia podczas obróbki pojedynczego polecenia gracza, zostaną omówione w rozdziałach 2.3 i 2.4.

Przyjrzyjmy się teraz każdej z występujących na rysunku kolekcji.

mapa

`mapa` zawiera lokacje (takie jak np. ścieżka, polana, jaskinia, ...), w których mogą znajdować się postacie i przedmioty. `dlugiop` to pole tekstowe, zawierające opis który gracz ujrzy gdy po raz pierwszy odwiedzi daną lokację. Przy następnych razach zobaczy skrócony opis znajdujący się w polu `krotkiop`. Pola `N`, `S`, `W`, `E`, ... wskazują na lokacje do których trafimy idąc w danym kierunku (jeśli nie da się iść w jakimś kierunku, to odpowiednie pole będzie puste). Pola `Nd`, `Sd`, ... wskazują na drzwi, znajdujące się w danym kierunku. Co najwyżej jedno pole z każdej pary `K`, `Kd`, gdzie `K` jest dowolnym kierunkiem, może być niepuste.

[1] Lub — jak za chwilę się przekonamy — z co najwyżej trzema rekordami, w przypadku skojarzenia z rzeczownikiem bez określenia rodzaju.



Rysunek 2.1: Model danych

postacie

Rekordy z tego zbiornika reprezentują występujące w grze postacie. Pola `gdzie_lok`, `gdzie_prz` i `gdzie_osoba` mówią, w jakiej lokacji, lub w jakim przedmiocie (np. w skrzyni, w beczce) znajduje się dana postać, lub przez kogo jest niesiona. Dokładnie jedno z pól `gdzie_lok`, `gdzie_prz` i `gdzie_osoba` powinno być niepuste. `nazwa` wskazuje na rzeczownik opisujący postać; może to być imię własne ("Gandalf", "Thorin"), lub zwykły rzeczownik ("troll", "goblin"). Pole to — podobnie jak pola `nazwa` we wszystkich omawianych typach danych — wskazuje na rzeczownik bez określenia jego rodzaju. Zatem strzałki wskazujące na zbiornik `rzeczowniki` mogą, wyjątkowo, oznaczać skojarzenie z jednym, dwoma lub trzema rekordami z tego zbiornika. Przykładowo, w zbiorniku `rzeczowniki` mogłyby znajdować się takie trzy różne rekordy:

```
1. { miejsc => "trollu", narz => "trollem",
     bier => "trolla", cel => "trollowi", dop => "trolla",
     mian => "troll", liczba => poj, rodz => m }
2. { miejsc => "trollach", narz => "trollami",
     bier => "trollów", cel => "trollom", dop => "trolli",
     mian => "trolle", liczba => mn, rodz => mos }
3. { miejsc => "trollach", narz => "trollami",
     bier => "trolle", cel => "trollom", dop => "trolli",
     mian => "trolle", liczba => mn, rodz => nmos }
```

(zauważmy, że pole `rodz` opisuje nie rodzaj samego rzeczownika — gdyż wtedy we wszystkich trzech rekordach byłby to rodzaj męski — lecz raczej rodzaj przymiotników w formie zgodnej z formą rzeczownika). Otóż opisujący jakiegoś konkretnego trolla rekord w zbiorniku `postacie` zawierałby odwołanie do wszystkich trzech wypisanych rekordów. Jeśli np. zbiorniki byłyby tabelami w relacyjnej bazie danych, to tabela `rzeczowniki` mogłaby mieć klucz główny złożony z dwóch pól, liczbowego `id` oraz określającego rodzaj `rodz`, a w polu `nazwa` rekordów z tabeli `postacie` (i tak samo `drzwi` i `przedmioty`) byłaby jedynie wartość `id` (to tylko przykład — w przedstawionej implementacji dane nie są trzymane w relacyjnej bazie; więcej szczegółów podaje rozdział 3.1.2).

Pole `zywy` mówi, czy dana postać żyje. Pole `sila` mówi, ile dana postać może naraz unieść. `spr_boj` to sprawność bojowa — im wyższa, tym trudniej zabić daną postać i tym łatwiej ona może zabijać innych. `ciezar` mówi, jak trudno daną postać unieść. Siła, sprawność bojowa i ciężar nie są wyrażone w żadnych realnych jednostkach w rodzaju kilogram; są to umowne, bezwymiarowe liczby.

Wreszcie `funkcja` zawiera odniesienie do fragmentu kodu odpowiedzialnego za zachowanie postaci.

Postać może być określana przez dowolną liczbę przymiotników (co ilustruje linia bez strzałek na rysunku 2.1).

drzwi

Rekordy z tego zbiornika reprezentują drzwi, prowadzące z jednych lokacji do innych. Każde drzwi powinny być wskazywane przez dokładnie dwie lokacje. Pole `klucz` zawiera odniesienie do przedmiotu, który jest kluczem do danych drzwi; może ono być puste, jeśli drzwi nie da się zamknąć na klucz. Pola `otwarte`, `locked` i `rozbite` są typu logicznego i mówią, czy drzwi są odpowiednio otwarte, zamknięte na klucz lub rozbite. `twardosc` to liczba związana z tym, jak trudno jest rozbić (wyłamać) dane drzwi. Pole `nazwa` wskazuje na rzeczownik, który jest nazwą danych drzwi (najczęściej, ale nie obowiązkowo, będzie to rzeczownik "drzwi").

Ponadto każde drzwi mogą być określane przez dowolną liczbę przymiotników.

przedmioty

Występujące w grze przedmioty. Pola `nazwa`, `gdzie_lok`, `gdzie_prz` i `gdzie_osoba` mają takie samo znaczenie, jak w przypadku postaci. Pole `z_wnetrzem` mówi, czy coś może znajdować się w środku danego przedmiotu. Pole `otwieralne` mówi, czy przedmiot można otwierać i zamykać (np. miecz będzie miał zarówno `z_wnetrzem` jak i `otwieralne` ustawione na `false`; łódka będzie mieć `z_wnetrzem` równe `true`, zaś `otwieralne` równe `false`; skrzynia będzie mieć oba pola równe `true`; kombinacja `z_wnetrzem = false`, `otwieralne = true` nie jest możliwa). `klucz` wskazuje na przedmiot będący kluczem do danego przedmiotu; może być niepuste tylko gdy `otwieralne = true`. `otwarte` i `locked` mówi, czy przedmiot jest otwarty/zamknięty na klucz. Pole `ciezar` mówi, jak trudno unieść dany przedmiot. `przyd_boj` mówi, jak bardzo użycie przedmiotu do zaatakowania kogoś zwiększa szanse zwycięstwa w walce. `objetosc` to objętość przedmiotu, `objwewn` to objętość wnętrza. Nie można włożyć przedmiotu do czegoś, co ma objętość wnętrza mniejszą niż objętość przedmiotu.

Przedmiot może być określany przez dowolną liczbę przymiotników.

rzeczowniki

Kolekcja rzeczowników. `rodz` i `liczba` mówią, jaki jest rodzaj (męski, żeński, nijaki, męskoosobowy, niemęskoosobowy) i liczba (pojedyncza, mnoga) danego rzeczownika (znaczy to, że będą osobne rekordy dla tego samego rzeczownika w liczbie pojedynczej i mnogiej). Pozostałe pola to rzeczownik odmieniony przez przypadki.

przymiotniki

Kolekcja przymiotników. Znaczenia pól takie same, jak dla kolekcji `rzeczowniki`. Będą osobne rekordy dla różnych rodzajów — "ładny", "ładna", "ładne" (dziecko), "ładne" (kobiety), ...

dzialania

`dzialania` to zbiornik możliwych do wykonywania przez postacie działań, takich jak podnoszenie i wyrzucanie przedmiotów, otwieranie drzwi, atakowanie innych postaci, itp. Pole `funkcja` zawiera odniesienie do fragmentu kodu, wołanego aby wykonać działanie. Pozostałe pola zawierają informacje o tym, jakich argumentów oczekuje ów kod. `czycytat` może być równe `true` lub `false` i mówi, czy działanie wymaga przekazania fragmentu tekstu, wprowadzonego przez użytkownika w cudzysłowach — ściśle mówiąc, jest dokładnie jedno działanie z `czycytat` ustawionym na `true`: rozmawianie z innymi postaciami ("Powiedz do Gandalfa 'Daj mi mapę' "). Pozostałe pola są to miejsca składniowe — więcej o nich powiemy w rozdziale 2.3. Na razie ograniczmy się do stwierdzenia, że nazwy pól to nazwy argumentów kodu z pola `funkcja`; jeśli pole jest puste, to znaczy, że funkcja nie oczekuje takiego argumentu; jeśli jest niepuste, to mówi, jakiego typu powinien być argument — typ może to być przedmiot, postać lub drzwi. Każde działanie może być nazwane wieloma różnymi czasownikami ("Zaatakuj trolla", "Zabij trolla", "Uderz trolla").

warunki

Zbiornik warunków, które muszą być spełnione, aby działania dało się wykonywać (np. aby wyrzucić jakiś przedmiot trzeba go nieść, aby dało się otworzyć drzwi nie mogą one być już

otwarte, ani też zamknięte na klucz, ani wyłamane, itp.). Pole `dzialanie` wskazuje na działanie, którego dotyczy warunek. `warunek` to odniesienie to fragmentu kodu, który sprawdza, czy warunek jest spełniony. `komunikat` to odniesienie do fragmentu kodu, generującego, w razie niespełnienia warunku, odpowiedni komunikat o błędzie. `kolejnosc` to kolejność sprawdzania.

aliasydz

Jeśli dwa działania są identyczne, a różnią się jedynie przyjmowanymi argumentami, to w zbiorniku `aliasydz` znajduje się rekord mówiący, które działanie należy zastąpić którym. Jest to przydatne aby identyfikować, że np. "Powiedz Thorinowi '...' " i "Powiedz do Thorina '...' " znaczy jedno i to samo. `zastepowane` i `zastepujace` to odniesienia do rekordów ze zbiornika `dzialania`, pozostałe pola mówią, w jaki sposób podczas takiej zamiany należy przemapować argumenty.

czasowniki

Kolekcja czasowników. Każdy czasownik może nazywać wiele działań. Pola `bezokol`, `rozkaz`, `os_2` i `os_3` to forma danego czasownika w bezokoliczniku, w trybie rozkazującym, oraz w drugiej i trzeciej osobie w czasie teraźniejszym. Brak pola `os_1` wynika z faktu, że nigdzie podczas gry czasownik w takiej formie się nie pojawia (`rozkaz` występuje w poleceniach gracza, `os_2` np. w zdaniu "Otwierasz drewnianą skrzynię", `os_3` np. w zdaniu "Gandalf idzie na północ").

2.3. Rozbiór gramatyczny zdania

W tym rozdziale opisana jest wstępna obróbka, jakiej podlega pojedyncze zdanie z wprowadzonego przez użytkownika polecenia. Operacje te są całkowicie niezależne od świata gry, w szczególności możliwość lub sensowność wykonania pewnych działań w żaden sposób nie wpływa na przebieg tej części algorytmu — czyli np. zdanie "Zabij miecz do klucza" będzie zaakceptowane jako poprawne (z punktu widzenia funkcji dokonującej rozbioru) i zostanie przeanalizowane. Wynikiem działania jest lista możliwych rozbiorów badanego zdania. Na tym etapie nie stanie się jeszcze nic złego, jeśli wynik będzie niejednoznaczny — pewne warianty mogą zostać wyeliminowane w następnym kroku.

Rozbiór ogranicza się do zdań rozkazujących.

Pojedynczy wariant rozbioru polega w uproszczeniu na

- Przypisaniu tworzącym zdanie wyrazom obiektów reprezentujących znane programowi słowa, wraz z ustaleniem formy, w jakiej słowo wystąpiło (już na tym etapie może pojawić się niejednoznaczność, np. napotkany w danych wejściowych wyraz "śmieć" może być formą rozkazującą czasownika "śmiecić", lub mianownikiem rzeczownika "śmieć").

- Zidentyfikowaniu orzeczenia. Musi to być czasownik w formie rozkazującej. W zdaniu powinno być dokładnie jedno orzeczenie (czyli np. zdanie "Podnieś jabłko i zjedz" niestety nie przechodzi).

- Przypisanie pozostałym słowom miejsc składniowych, które w danym wariancie wypełniają. Lista dopuszczalnych miejsc składniowych jest konfigurowalna, jednak generalnie będą to różnego rodzaju dopełnienia i okoliczniki. Definicja miejsca składniowego będzie polegać na podaniu odpowiadającego mu

przypadku i ewentualnie przyimka. Przykładowo, gdybyśmy mieli zdefiniowane miejsca składniowe `obj = { przypadek => biernik, brak przyimka }` oraz `docel = { przypadek => dopełniacz, przyimek => 'do' }`, to w zdaniu "Włóż pieniądze do skrzyni" słowo "pieniądze" mogłoby być przypisane do miejsca `obj`, zaś "do skrzyni" do miejsca `docel`.

Dodatkowo chcielibyśmy, aby system umożliwiał komunikację między postaciami. Czyli np. powinien rozumieć zdania w rodzaju "Powiedz do Gandalfa 'Daj mi mapę' ". W tym celu wprowadzamy coś w rodzaju dodatkowego miejsca składniowego `cytat`. Nie jest ono konfigurowalne i jest rozpoznawane dzięki umieszczeniu w cudzysłowach. W zdaniu może występować co najwyżej jeden cytat. Zawartość cytatu nie jest na tym etapie w żaden sposób analizowana.

Chcielibyśmy też, aby możliwe było wydawanie poleceń w rodzaju "Weź wszystko", "Zaatakuj wszystkich", "Weź wszystko oprócz miecza", "Wyrzuć wszystkie klucze oprócz srebrnego". W związku z tym potrzebne jest jeszcze gromadzenie dodatkowych informacji: czy dla danego miejsca składniowego pojawił się wyraz "wszystko" lub podobny, a jeśli tak to czy jego zakres jest ograniczony przez jakieś "oprócz". Kwalifikator "wszystko" może być jawny lub domyślny, np. zdanie "Zabij trolle" ma takie samo znaczenie jak "Zabij wszystkie trolle".

2.3.1. Postać wyniku

Operacja rozbioru gramatycznego zwraca strukturę — będziemy ją nazywać `gramtab` — składającą się z:

listy wariantów Każdy wariant przechowuje jeden ze sposobów rozbicia zdania na słowa z przypisanymi im miejscami składniowymi. Struktura pojedynczego elementu listy będzie omówiona poniżej.

cytatu Jeśli w zdaniu był fragment otoczony cudzysłowami to znajdzie się on w tym polu wyniku. Nie ma możliwości wziąć cytatu za coś innego i dlatego — w przeciwieństwie do zwykłych miejsc składniowych — umieszczamy informację o nim raz dla całego wyniku, zamiast osobno w każdym elemencie listy wariantów.

błędu Obecność tego pola sygnalizuje, że rozbiór zdania się nie powiódł (może to być spowodowane wystąpieniem nieznanego wyrazu, brakiem orzeczenia, występowaniem słowa o bezsensownym przypadku lub rodzaju itd.). Zawiera ono tekstowy komunikat o błędzie. Gdy pole to jest ustawione, wtedy program ignoruje zawartość pozostałych pól.

Element listy wariantów zawiera pole `rozkaz`, będące odniesieniem do pojedynczego rekordu ze zbiornika `czasowniki`, oraz kolekcję struktur – po jednej dla każdego zdefiniowanego miejsca składniowego. Każda taka struktura ma następujące pola:

rodz Rodzaj (męski, żeński, nijaki, męskoosobowy, niemęskoosobowy) w jakim wystąpiły słowa, wypełniające bieżące miejsce składniowe (jak za chwilę zobaczymy, pole to jest nadmiarowe — gdyż informację o rodzaju można by wydobyć z któregoś z pól `rz`, `prz` lub `all`).

prz Lista odniesień do rekordów ze zbiornika `przymiotniki`, opisujących słowa, które w tej wersji rozbioru przypisano do danego miejsca składniowego. Przypomnijmy, że wybór rekordu ze zbiornika `przymiotniki` ustala też rodzaj.

rz Lista odniesień do rekordów ze zbiornika `rzeczowniki`. Ponieważ zakładamy, że do każdego miejsca składniowego może być przypisany tylko jeden rzeczownik, więc lista ta będzie miała co najwyżej jeden element. Uwaga o rodzaju tu również obowiązuje, można jednak zauważyć, że w przypadku rzeczowników ustalenie rodzaju jest tym samym, co ustalenie liczby (pojedynczej lub mnogiej).

mowc To pole jest niepuste, jeśli miejsce składniowe zostało wypełnione słowami, jakich osoba mówiąca użyła na określenie siebie samej. Jest to użyteczne w sytuacji, gdy rozmawiamy z innymi postaciami w grze (postacie nie będące graczami używają oczywiście tej samej funkcji do analizy gramatycznej zdania). Np. jeśli wydamy polecenie "Powiedz do Gandalfa 'Daj mi mapę' ", to Gandalf wskutek rozbioru zdania "Daj mi mapę" dostanie strukturę z niepustym polem `mowc` dla miejsca składniowego `komu = { przypadek => celownik, brak przyimka }`.

all Jeśli w danym miejscu składniowym został, niekoniecznie jawnie, umieszczony kwalifikator "wszystko", "wszyscy" lub "wszystkie", to w tym polu znajdzie się jego rodzaj (nijaki, męskoosobowy lub niemęskoosobowy).

opr Jeżeli pole `all` jest obecne, to pole `opr` może ograniczyć jego zasięg działania. Pole `opr` będzie niepuste, jeśli w zdaniu wystąpił wyraz "oprócz", który w tym wariancie rozbioru skojarzyliśmy z rzeczownikiem i/lub przymiotnikami, w formie odpowiadającej bieżącemu miejscu składniowemu. `opr` jest samo strukturą, mającą pola `rodz`, `rz`, `prz` i `mowc`, takiego samego typu, jak pola o tych nazwach w strukturze opisującej miejsce składniowe. Różna jest interpretacja — większa struktura określa, kogo lub czego zdanie dotyczy, zaś `opr` określa kogo lub czego zdanie nie dotyczy.

Każde z wymienionych pól może być puste. Jeśli dane miejsce składniowe w zdaniu nie jest wypełnione, to wszystkie pola opisującej je struktury będą puste — będzie to zresztą najczęściej występująca sytuacja.

2.3.2. Przykłady

Tu i w dalszej części pracy będziemy zakładać, że mamy zdefiniowane następujące miejsca składniowe:

- `obj = { przypadek => biernik, brak przyimka }`
- `util = { przypadek => narzędnik, brak przyimka }`
- `komu = { przypadek => celownik, brak przyimka }`
- `docel = { przypadek => dopełniacz, przyimek => 'do' }`
- `zczego = { przypadek => dopełniacz, przyimek => 'z' }`
- `naco = { przypadek => biernik, przyimek => 'na' }`

W poniższych przykładach będę wypisywał tylko te elementy otrzymanej struktury, które są niepuste. Elementy list będą otoczone nawiasami kwadratowymi: `[ elem1, elem2, ... ]`, zaś elementy struktur nawiasami klamrowymi: `{ pole1 => wartość1, pole2 => wartość2, ... }`.

I tak na przykład, w wyniku analizy zdania "Otwórz okrągłe, zielone drzwi" dostaniemy:

```
gramtab = {
  [
    {
      obj => {
        rodz => nmos,
        prz => [ ‘okrągłe’, ‘zielone’ ],
        rz => [ ‘drzwi’ ]
      },
      rozkaz => ‘otworzyć’
    }
  ]
}
```

‘okrągłe’, ‘zielone’, ‘drzwi’ i ‘otworzyć’ w tym przykładzie reprezentują pojedyncze rekordy ze zbiorników `przymiotniki`, `rzeczowniki` i `czasowniki`. W rzeczywistości mogłyby to być — zależnie od konkretnego sposobu przechowywania danych — adresy w pamięci, wartości indeksów w tablicy, wartości kluczy głównych, itp.

Jak widać, w tym przypadku rozbiór okazuje się jednoznaczny. Dokładnie identycznie wyglądałby też dla zdań “Okrągłe, zielone otwórz drzwi” albo “Otwórz okrągłe drzwi zielone”, itp. (zdania takie są zapewne na pograniczu poprawności, lecz wolimy mylić się na korzyść wygody gracza — zwłaszcza, że nic nas to nie kosztuje). Weźmy dla odmiany zdanie “Otwórz wszystko, oprócz drewnianej skrzyni”:

```
gramtab = {
  [
    {
      obj => {
        rodz => n,
        all => n,
        opr => {
          rodz => ż,
          prz => [ ‘drewniana’ ],
          rz => [ ‘skrzynia’ ]
        }
      },
      rozkaz => ‘otworzyć’
    },
    {
      obj => {
        rodz => n,
        all => n,
        opr => {
          rodz => ż,
          prz => [ ‘drewniana’ ]
        }
      },
      komu => {
        rodz => ż,
        rz => [ ‘skrzynia’ ]
      },
```

```
      rozkaz => ‘otworzyć’
    }
  ]
}
```

Okazuje się, że tu możliwe były dwa warianty: słowo “skrzyni” mogło zostać uznane za dopełniacz i przypisane, razem ze słowem “drewnianej”, do “oprócz” ograniczającego zasięg słowa “wszystko”, jednak równie dobrze może to być celownik — w takim wypadku polecenie oznaczałoby rozkaz otwarcia (*komu? czemu?*) skrzyni wszystkiego, oprócz jakiejś bliżej nieokreślonej drewnianej rzeczy (oczywiście, na ogół, tylko pierwszy wariant miałby sens, jednak to jest ustalane na późniejszym etapie działania programu). Gdybyśmy zamiast “oprócz drewnianej skrzyni” napisali “oprócz skrzyni drewnianej”, wtedy w drugim wariancie do miejsca składniowego `komu` zostałby przypisany przymiotnik “drewnianej”, zaś “skrzyni” w obu wariantach byłoby w sekcji `opr` miejsca `obj` — bo wyraz bezpośrednio sąsiadujący ze słowem “oprócz” musi być z nim powiązany.

2.3.3. Algorytm rozbioru

Przejdźmy teraz do omówienia szczegółów działania funkcji, dokonującej rozbioru składniowego.

Na początku konstruowana wartość inicjalizowana jest strukturą bez ustawionego cytatu ani błędu, z listą wariantów rozbioru składającą się z jednego elementu, którego pole `rozkaz` jest puste, tak samo jak pola każdej ze struktur opisujących miejsca składniowe.

Oprócz pól mających znaczenie dla końcowego wyniku, każdy element listy ma pomocnicze pola: `oprtmp` — takiego samego typu jak pole `opr` w każdej ze struktur opisujących wypełnienie miejsc składniowych, `lastall` — które może być puste, lub zawierać nazwę miejsca składniowego, oraz `jakiopr` — które może być puste, zawierać nazwę miejsca składniowego, lub zawierać specjalną wartość `’tmp’`. Do `oprtmp` trafia zawartość tych “oprócz”, które znajdują się w zdaniu wcześniej, niż “wszystko” którego dotyczą (“Oprócz klucza weź wszystko”). `jakiopr` mówi, do którego `opr` mogą trafić napotkane słowa, czyli którego miejsca składniowego dotyczyło ostatnie “oprócz” — jeśli jeszcze nie wiadomo, to `jakiopr` jest równe “tmp” i napotkane słowa mogą trafić do `oprtmp` (oczywiście tylko te, które są w odpowiednim przypadku, tzn. w dopełniaczu); jeśli natomiast nie było jeszcze żadnego “oprócz”, to `jakiopr` jest puste. `lastall` mówi, którego miejsca składniowego dotyczyło ostatnie “wszystko”. Początkowo te pomocnicze pola także są puste. Struktury opisujące zawartość miejsc składniowych mają pomocnicze pole `niejawne`, typu logicznego, które służy do odróżniania kwalifikatorów “wszystko”, występujących jawnie w tekście polecenia (“Weź wszystko”), od domyślnych (“Weź klucze”) i początkowo jest ustawione na `false`.

Dodatkowo stosowane są następujące zmienne pomocnicze: `spseek`, która może być pusta, lub zawierać nazwę miejsca składniowego, `seekopr` i `spseekopr`, zawierające wartość logiczną, oraz dla każdego miejsca składniowego `ms` zmienna `seek[ms]`, również zawierająca wartość logiczną. `seek[ms]` mówi, czy napotkane słowa mogą być przypisane do miejsca `ms` — co zależy od tego, czy został dotychczas napotkany odpowiedni przyimek. Początkowo dla miejsc składniowych bez przyimka `seek[ms]` jest ustawione na `true`, zaś dla pozostałych na `false`. `seekopr` ma podobną rolę jak `seek[ms]` — mówi, czy napotkane słowa mogą zostać przypisane do jakiegoś `opr`, i jego wartość zależy od tego czy w dotychczasowej części zdania było “oprócz”; początkowo jest ustawione na `false`. Niepuste `spseek` komunikuje, że następny wyraz *musi* zostać przypisany do danego miejsca składniowego. `spseek` jest ustawiane po napotkaniu związanego z miejscem składniowym przyimka. Jego działanie obejmuje tylko

jedno najbliższe słowo — potem jest z powrotem czyszczone. `spseekopr` działa podobnie: gdy jest równe `true`, to następny wyraz musi zostać wstawiony do odpowiedniego (zależnego od wartości pola `jakiopr` w danym wariancie rozbioru) `opr`; jest ustawiane na `true` po napotkaniu wyrazu "oprócz" i jego działanie rozciąga się na jedno następne słowo, potem jest z powrotem przestawiane na `false`. Początkowo `spseek` jest puste, zaś `spseekopr` jest równe `false`. `seek[]`, `spseek`, `seekopr` i `spseekopr` są — w przeciwieństwie do `lastall` i `jakiopr` — wspólne dla całej listy wariantów, co wynika z upraszczającego założenia, iż przyimki i słowo "oprócz" są jednoznaczne (tzn. nie mogą być pomylone z żadnym innym słowem — rzeczownikiem, czasownikiem, itp.).

Funkcja w pętli wczytuje kolejne tokeny z przekazanego jako argument zdania. Tokenem może być pojedyncze słowo lub cytat, czyli ciąg słów otoczony znakami cudzysłowów. Jeśli jest to cytat, to funkcja

- Sprawdza, czy w konstruowanym wyniku jest już cytat, jeśli tak to zgłasza błąd i kończy działanie.
- Ustawia pole z cytatem.
- Przechodzi do następnego tokenu.

W przeciwnym razie:

Jeśli `spseek` jest pusty, a `spseekopr` równy `false`, to funkcja:

- Dla każdego miejsca składniowego `ms` sprawdza, czy wczytane słowo jest przyimkiem odpowiadającym `ms` (tzn. iterujemy po wszystkich zdefiniowanych miejscach składniowych; jeśli np. zdefiniowane będą takie jak w rozdziale 2.3.2, to `ms` będzie przebiegać wartości `obj`, `util`, `komu`, `docel`, ...; jeśli np. wczytane słowo to "do", to sprawdzenie zwróci wartość `true` przy `ms` równym `docel`). Jeśli tak, to
  - jeżeli `seek[ms]` jest równe `false`, to ustawia `seek[ms]` na `true`, `spseek` na `ms` i przechodzi do następnego tokenu;
  - w przeciwnym razie (tzn. `seek[ms]` równe `true`; zauważmy, że może tak się zdarzyć dopiero przy którymś kolejnym wczytanym słowie; wartość początkowa wszystkich `seek[ms]` to `false`) zgłasza błąd i kończy pracę.
- Sprawdza, czy wczytane słowo to "oprócz". Jeśli tak, to dla każdego elementu `elem` listy wariantów:
  - jeśli `elem.jakiopr` jest puste: ustawia `elem.jakiopr` na wartość `elem.lastall` o ile `elem.lastall` nie jest puste, lub na `'tmp'` w przeciwnym przypadku;
  - w przeciwnym razie:
    - jeśli `elem.jakiopr` jest równe `'tmp'`, to usuwa `elem` z listy wariantów
    - w przeciwnym razie: jeśli `elem.jakiopr` jest równe `elem.lastall`, to ustawia `elem.jakiopr` na `'tmp'`, w przeciwnym razie ustawia `elem.jakiopr` na `elem.lastall`.
- Jeśli po wykonaniu tych operacji lista wariantów jest pusta, to zgłasza błąd i kończy pracę.
- Ustawia `seekopr` i `spseekopr` na `true` i przechodzi do następnego tokenu.

Następnie wczytane słowo porównywane jest z zawartością zbiorników `czasowniki`, `rzeczowniki` i `przymiotniki`, a także z odmienionymi przez wszystkie przypadki słowami "wszystko", "wszyscy", "wszystkie" oraz "ja". W wyniku tych porównań wypełniane są następujące listy (`ms` przebiega wszystkie zdefiniowane miejsca składniowe — czyli, trzymając się przykładowej listy z rozdziału 2.3.2: `obj`, `util`, `komu`, `docel`, ...):

rozk lista odniesień do rekordów ze zbiornika `czasowniki`, których forma rozkazująca jest identyczna z wczytanym słowem;

oprrz lista odniesień do rekordów ze zbiornika `rzeczowniki`, których dopełniacz jest identyczny z wczytanym słowem;

oprprz lista odniesień do rekordów ze zbiornika `przymiotniki`, których dopełniacz jest identyczny z wczytanym słowem;

oprmowc zawiera jeden element jeśli wczytane słowo to "mnie", w przeciwnym razie jest pusta;

rz[ms] lista odniesień do takich rekordów ze zbiornika `rzeczowniki`, których przypadek odpowiadający miejscu `ms` jest identyczny z wczytanym słowem; jeśli na przykład w zbiorniku `rzeczowniki` będą znajdować się takie trzy rekordy, jak podane w rozdziale 2.2, a wczytane słowo to "trollami", wtedy lista `rz[util]` będzie mieć dwa elementy, wskazujące na rekordy oznaczone w rozdziale 2.2 numerami 2 i 3, zaś wszystkie pozostałe listy (`rz[obj]`, `rz[komu]`, ...) będą puste; inaczej niż w przypadku odniesień w polu `nazwa` rekordów omawianych w rozdziale 2.2, elementy list `rz[ms]` odnoszą się do pojedynczych rekordów ze zbiornika `rzeczowniki`;

prz[ms] lista odniesień do takich rekordów ze zbiornika `przymiotniki`, których przypadek odpowiadający miejscu `ms` jest identyczny z wczytanym słowem;

all[ms] jest podciągiem listy: `['n', 'nmos', 'mos']`; `'n'` występuje, jeśli wczytane słowo to "wszystko" w przypadku odpowiadającym miejscu składniowemu `ms`; `'nmos'` występuje, jeśli wczytane słowo to "wszystkie" w takim przypadku; `'mos'`, jeśli "wszyscy";

mowc[ms] zawiera jeden element jeśli wczytane słowo to "ja" w przypadku odpowiadającym miejscu `ms`, w przeciwnym razie jest pusta.

Jeśli `spseekopr` jest równe `true`, to wypełniane są jedynie listy `oprrz`, `oprprz` i `oprmowc` (ponieważ znaczy to, że poprzednio wczytanym słowem było "oprócz"; zatem szukamy tylko wśród słów, które mają sens bezpośrednio po "oprócz"). Jeśli `spseek` jest niepuste, to wypełniane są jedynie listy `rz[spseek]`, `prz[spseek]`, `all[spseek]` i `mowc[spseek]` (niepuste `spseek` oznacza, że poprzednio wczytanym słowem był przyimek; zatem szukamy jedynie wśród słów, mających sens po tym przyimku). Listy `rz[ms]`, `prz[ms]`, `all[ms]` i `mowc[ms]` będą puste dla tych `ms`, dla których `seek[ms]` jest ustawione na `false` (bo znaczy to, że nie było jeszcze odpowiadającego danemu miejscu składniowemu przyimka). List `oprrz`, `oprprz` i `oprmowc` nie wypełniamy, jeśli zmienna `seekopr` jest ustawiona na `false` (bo znaczy to, że nie było jeszcze żadnego "oprócz").

Po wypełnieniu powyższych list funkcja ustawia `spseek` na wartość pustą a `spseekopr` na `false`.

Jeśli wszystkie otrzymane listy są puste, to w tym momencie funkcja zgłasza błąd i kończy działanie.

Następnie lista wariantów rozbioru jest powielana tylokrotnie, ile wynosi suma długości list otrzymanych w poprzednim kroku. W wyniku tej operacji powstaje nowa lista wariantów z powtarzającymi się elementami. Np. jeśli dotychczasowa lista wariantów miała trzy elementy, zaś suma długości otrzymanych list wynosi 4, to nowa lista wariantów będzie miała dwanaście elementów, z których pierwszy, czwarty, siódmy i dziesiąty będą identyczne dotychczasowemu pierwszemu, drugi, piąty, ósmy i jedenasty dotychczasowemu drugiemu itd. Dzięki temu można powiedzieć — i w ten sposób będziemy od tej pory na to patrzyli — że każdemu elementowi którejś z otrzymanych w poprzednim kroku list jest przypisany wycinek nowej listy wariantów identyczny ze starą listą wariantów. Sens tej operacji jest taki, że, mówiąc niezbyt ściśle, będziemy robić coś w rodzaju iloczynu kartezjańskiego dotychczasowego zbioru możliwych wariantów rozbioru gramatycznego, przez zbiór sposobów zidentyfikowania wczytanego słowa — przy czym zbiór sposobów zidentyfikowania słowa jest reprezentowany przez wszystkie listy `rozk`, `oprrz`, `oprprz`, ... (bardzo często słowo będzie zidentyfikowane jednoznacznie, tzn. dokładnie jedna z tych list będzie niepusta i będzie ona mieć tylko jeden element).

Musimy w tym momencie wspomnieć o pewnym szczególe technicznym, którego znajomość jest konieczna do prześledzenia dalszej części algorytmu. Otóż opisane w podrozdziale 2.3.1 pola `mowc` i `all`, oraz pole `mowc` struktury znajdującej się w `opr`, ze względu na wygodę implementacji również są listami. W podrozdziale 2.3.1 listy, które mogą mieć co najwyżej jeden element zostały dla czytelności zamienione na zwykłe wartości — wyjątkiem jest tu pole `rz`, gdzie zostawiliśmy listę, aby podkreślić arbitralność założenia o co najwyżej jednym rzeczowniku w każdym miejscu składniowym.

Następnie funkcja przechodzi do właściwego wypełniania miejsc składniowych. Każdy z elementów otrzymanych w poprzednim kroku list jest nanoszony na swój wycinek nowej listy wariantów wg. następującego schematu:

- Elementy listy `rozk` są dodawane do listy w polu `rozkaz`, w każdym z elementów odpowiedniego wycinka listy wariantów.

- Elementy listy `oprmowc` są dodawane do listy w polu `mowc` struktury `opr`, odpowiadającej miejscu składniowemu ustalonemu przez zawartość pola `jakiopr` (lub, jeśli pole `jakiopr` ma wartość `'tmp'`, w polu `mowc` struktury `oprtmp`), w każdym z elementów odpowiedniego wycinka listy wariantów.

- Elementy listy `oprrz` są dodawane do listy w polu `opr.rz` w strukturze, odpowiadającej miejscu składniowemu, ustalonemu przez zawartość pola `jakiopr` (lub, jeśli pole `jakiopr` ma wartość `'tmp'`, w polu `oprtmp.rz`), w każdym z elementów odpowiedniego wycinka listy wariantów.

- Elementy listy `oprprz` analogicznie.

- Elementy list `mowc[ms]` są dodawane do listy w polu `mowc` struktury, odpowiadającej miejscu `ms`, w każdym z elementów wycinka.

- Elementy list `all[ms]` są dodawane do listy w polu `all` struktury, odpowiadającej miejscu `ms`, w każdym z elementów odpowiedniego wycinka. Również w każdym z elementów wycinka, odpowiadającego elementowi `all[ms]`, pole `lastall` jest ustawiane na `ms`. Jeśli pole `jakiopr` w danym elemencie wycinka było równe `'tmp'`, to jest ono przestawiane na `ms`, zaś zawartość `oprtmp` jest przenoszona do `opr` w strukturze, odpowiadającej miejscu `ms`.

- Elementy list `rz[ms]` są dodawane do listy w polu `rz` struktury, odpowiadającej miejscu `ms` w każdym z elementów odpowiedniego wycinka. Jeśli element listy `rz[ms]` ma rodzaj `mos` lub `nmos`, a w uzupełnianym elemencie wycinka pole `all` struktury odpowiadającej miejscu `ms` zawiera listę pustą, to:
  - funkcja sprawdza w zbiorniku `rzeczowniki`, czy istnieje wersja danego rzeczownika z liczbą pojedynczą (służy to odróżnieniu zdań w rodzaju "Zabij trolle" od zdań w rodzaju "Otwórz drzwi"); jeśli istnieje, to do listy w polu `all` struktury, odpowiadającej miejscu `ms` w uzupełnianym elemencie wycinka, dodawany jest element równy rodzajowi rzeczownika, zaś pole `niejawne` jest ustawiane na `true`;
  - pole `lastall` jest ustawiane na `ms`;
  - jeśli pole `jakiopr` jest równe `'tmp'`, to zostaje zmienione na `ms`, zaś zawartość `oprtmp` jest przenoszona do `opr` w strukturze, odpowiadającej miejscu `ms`.

  Zauważmy, że dwa ostatnie punkty są wykonywane nawet, jeśli rzeczownik nie ma liczby pojedynczej — gdyż mimo, że np. zdanie "Otwórz drzwi" nie implikuje, że chodzi o wszystkie dostępne drzwi, to jednak zdanie "Otwórz drzwi oprócz zielonych" jest jak najbardziej poprawne (w przeciwieństwie do np. "Weź klucz oprócz zielonego").
- Elementy list `prz[ms]` są dodawane do listy w polu `prz` struktury, odpowiadającej miejscu `ms`, w każdym z elementów odpowiedniego wycinka.

Po wykonaniu opisanych kroków dla każdego wczytanego tokenu mamy listę wariantów, zawierającą wszystkie możliwe przyporządkowania słów miejscom składniowym. Jednak niektóre warianty nie mają sensu. Ostatnim etapem algorytmu rozbioru składniowego jest więc czyszczenie listy wariantów. Usuwane są te elementy listy wariantów, dla których po wczytaniu ostatniego tokenu zachodzi którykolwiek z warunków:

- Pole `oprtmp` jest niepuste.
- Dla któregokolwiek z miejsc składniowych lista `all` ma więcej niż dwa elementy, lub ma więcej niż jeden element, a `niejawne` jest ustawione na `false`.
- Dla któregokolwiek z miejsc składniowych równocześnie niepusta jest lista `mowc` i któraś z list `all`, `rz`, `prz`.
- Dla któregokolwiek z miejsc składniowych nie wszystkie elementy list `all`, `rz` i `prz` mają ten sam rodzaj.
- Dla któregokolwiek z miejsc składniowych nie wszystkie elementy list należących do `opr` mają ten sam rodzaj.
- Lista w polu `rozkaz` jest pusta lub ma więcej niż jeden element.
- Dla któregokolwiek z miejsc składniowych lista `rz` ma więcej niż jeden element.

Jeśli po oczyszczeniu lista wariantów została pusta, to funkcja zgłasza błąd i kończy pracę.

Na koniec we wszystkich elementach listy wariantów uzupełniane jest pole `rodz` dla każdej ze struktur opisujących miejsca składniowe oraz dla każdej ze struktur w polach `opr` (po wyczyszczeniu listy wariantów można to zrobić jednoznacznie). Jeśli `rodz` dla jakiegoś miejsca składniowego jest równe `mos` lub `nmos` a lista `all` jest pusta, zaś lista `rz` dla tego miejsca jest również pusta, to do `all` jest wstawiany jeden element równy `rodz` (aby umożliwić generowanie niejawnych "wszystko" także przez przymiotniki).

Na tym kończy się działanie algorytmu rozbioru gramatycznego — skonstruowany wynik jest przekazywany do dalszych części aplikacji.

2.3.4. Przykład

Obejrzyjmy dla przykładu kolejne kroki rozbioru zdania "Włóż do drewnianej skrzyni wszystko oprócz srebrnego klucza". Podobnie jak w rozdziale 2.3.2, będę wypisywał tylko niepuste elementy struktur. Również podobnie jak w rozdziale 2.3.2, odwołania do konkretnych rekordów ze zbiorników `czasowniki`, `przymiotniki` i `rzeczowniki` będą reprezentowane przez napisy w rodzaju `'włożyć'`, `'drewniana'`. Listy będą oznaczane nawiasami kwadratowymi, a struktury klamrami.

Na początku, przed wczytaniem pierwszego wyrazu, lista wariantów ma jeden element, o wszystkich polach pustych; możemy ją zapisać tak: `[{}]`. `seekopr` i `spseekopr` są ustawione na `false`, zaś `spseek` jest pusta. `seek[ms]` jest równe `true` dla `ms` równego `obj`, `util` i `komu` (gdyż te miejsca składniowe nie są związane z żadnym przyimkiem), zaś `false` w pozostałych przypadkach.

Wczytujemy słowo "włóż". Zostaje ono jednoznacznie rozpoznane jako forma rozkazująca czasownika `'włożyć'` (czyli spośród list `rozk`, `oprrz`, `oprprz`, ... niepusta będzie tylko lista `rozk` równa `[ 'włożyć' ]`). Zatem w wyniku naniesienia go na dotychczasową, pustą listę wariantów, dostaniemy listę `[ { rozkaz => 'włożyć' } ]`. Żadna ze zmiennych pomocniczych nie ulega zmianie.

Wczytujemy słowo "do". Jest ono przyimkiem, odpowiadającym miejscu składniowemu `docel`. Zatem ustawiamy `spseek` na `docel`, przestawiamy `seek[docel]` na `true` i przechodzimy do następnego słowa.

Wczytujemy "drewnianej". W ogólności przy takim słowie każda z list `oprprz`, `prz[docel]`, `prz[zczego]` (które reagują na dopełniacz) i `prz[komu]` (która reaguje na celownik) mogłaby być równa `[ 'drewniana' ]`, jednak `oprprz` musi być pusta, bo `seekopr` wynosi `false`, a `prz[zczego]` musi być pusta, bo `seek[zczego]` też jest `false`; zresztą nawet, gdyby obie te zmienne były równe `true`, to i tak `spseek` jest niepuste i równe `docel`, zatem jedyna niepusta lista to `prz[docel]`. Dostajemy więc nadal jednoelementową listę wariantów

```
[
  {
    docel => { prz => [ 'drewniana' ] },
    rozkaz => 'włożyć'
  }
]
```

Przed przejściem do następnego wyrazu zerujemy wartość zmiennej `spseek`.

Następnie wczytujemy słowo "skrzyni". Listy `oprrz` i `rz[zczego]` muszą być puste, bo `seekopr` i `seek[zczego]` są równe `false`. Nic jednak nie stoi na przeszkodzie, aby niepusta była lista `rz[komu]`, podobnie jak `rz[docel]`: obie są równe `[ 'skrzynia' ]`. Następuje więc rozmnożenie wariantów:

```
[
  {
    docel => { prz => [ 'drewniana' ], rz => [ 'skrzynia' ] },
    rozkaz => 'włożyć'
  },
  {
    docel => { prz => [ 'drewniana' ] },
    komu => { rz => [ 'skrzynia' ] },
```

```
    rozkaz => ‘włożyć’
  }
]
```

(Wycinek rozmnożonej listy wariantów, przypisany do jedynego elementu `rz[docel]` był to jej pierwszy element, a wycinek, przypisany do jedynego elementu `rz[komu]` — element drugi).

Następne słowo to “wszystko”. Niepusta będzie jedynie lista `all[obj]` (reagująca na biernik) i będzie ona równa `[ n ]` (rodzaj nijaki). Nanosząc listę `all[obj]` dodatkowo ustawiamy pole `lastall`:

```
[
  {
    lastall => obj,
    obj => { all => [ n ] },
    docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    lastall => obj,
    obj => { all => [ n ] },
    docel => { prz => [ ‘drewniana’ ] },
    komu => { rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  }
]
```

Kolejnym słowem jest “oprócz”. Powoduje ono przestawienie `seekopr` oraz `spseekopr` na `true` i ustawienie, w każdym elemencie listy wariantów, pola `jakiopr` na wartość wziętą z pola `lastall` (czyli `obj`).

Wczytujemy “srebrnego”. Ponieważ `spseekopr` jest równe `true`, więc niepusta jest tylko lista `oprprz`. Za to ma ona aż dwa elementy: `[ ‘srebrny’, ‘srebrne’ ]`, bowiem w kolekcji `przymiotniki` istnieją dwa rekordy o dopełniaczu równym “srebrnego”: jeden reprezentujący przymiotnik w rodzaju męskim, drugi w rodzaju nijakim. Znów dochodzi więc do rozmnożenia listy wariantów; pierwsze dwa elementy rozmnożonej listy stanowią wycinek, przypisany do pierwszego elementu `oprprz`, kolejne dwa, to wycinek przypisany do drugiego elementu `oprprz`. Po naniesieniu elementów na odpowiednie wycinki nowa lista wariantów wygląda następująco:

```
[
  {
    jakiopr => obj,
    lastall => obj,
    obj => { opr => { prz => [ ‘srebrny’ ] }, all => [ n ] },
    docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    jakiopr => obj,
    lastall => obj,
```

```
    obj => { opr => { prz => [ ‘srebrny’ ] }, all => [ n ] },
    docel => { prz => [ ‘drewniana’ ] },
    komu => { rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    jakiopr => obj,
    lastall => obj,
    obj => { opr => { prz => [ ‘srebrne’ ] }, all => [ n ] },
    docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    jakiopr => obj,
    lastall => obj,
    obj => { opr => { prz => [ ‘srebrne’ ] }, all => [ n ] },
    docel => { prz => [ ‘drewniana’ ] },
    komu => { rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  }
]
```

Wreszcie wczytujemy ostatnie słowo, “klucza”. Potencjalnie niepuste mogłyby być, reagujące na dopełniacz, listy `oprrz`, `rz[docel]` i `rz[zczego]`, ale ostatnia odpada, bo `seek[zczego]` jest równe `false`. Pozostałe będą równe `[ ‘klucz’ ]`. Element `oprprz` nanosimy na pierwszą połowę rozmnożonej listy, element `rz[docel]` na drugą:

```
[
  {
    jakiopr => obj,
    lastall => obj,
    obj => {
      opr => { prz => [ ‘srebrny’ ], rz => [ ‘klucz’ ] },
      all => [ n ]
    },
    docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    jakiopr => obj,
    lastall => obj,
    obj => {
      opr => { prz => [ ‘srebrny’ ], rz => [ ‘klucz’ ] },
      all => [ n ]
    },
    docel => { prz => [ ‘drewniana’ ] },
    komu => { rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
```

```
  jakiopr => obj,
  lastall => obj,
  obj => {
    opr => { prz => [ ‘srebrne’ ], rz => [ ‘klucz’ ] },
    all => [ n ]
  },
  docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
  rozkaz => ‘włożyć’
},
{
  jakiopr => obj,
  lastall => obj,
  obj => {
    opr => { prz => [ ‘srebrne’ ], rz => [ ‘klucz’ ] },
    all => [ n ]
  },
  docel => { prz => [ ‘drewniana’ ] },
  komu => { rz => [ ‘skrzynia’ ] },
  rozkaz => ‘włożyć’
},
{
  jakiopr => obj,
  lastall => obj,
  obj => {
    opr => { prz => [ ‘srebrny’ ] },
    all => [ n ]
  },
  docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’, ‘klucz’ ] },
  rozkaz => ‘włożyć’
},
{
  jakiopr => obj,
  lastall => obj,
  obj => { opr => { prz => [ ‘srebrny’ ] }, all => [ n ] },
  docel => { prz => [ ‘drewniana’ ], rz => [ ‘klucz’ ] },
  komu => { rz => [ ‘skrzynia’ ] },
  rozkaz => ‘włożyć’
},
{
  jakiopr => obj,
  lastall => obj,
  obj => { opr => { prz => [ ‘srebrne’ ] }, all => [ n ] },
  docel => { prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’, ‘klucz’ ] },
  rozkaz => ‘włożyć’
},
{
  jakiopr => obj,
  lastall => obj,
  obj => { opr => { prz => [ ‘srebrne’ ] }, all => [ n ] },
```

```
    docel => { prz => [ ‘drewniana’ ], rz => [ ‘klucz’ ] },
    komu => { rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  }
]
```

Następnie przechodzimy do czyszczenia listy wariantów. Zauważamy, że

- w wariancie trzecim i czwartym rodzaj rzeczownika w polu `opr` miejsca składniowego `obj` (`‘klucz’`) nie zgadza się z rodzajem przymiotnika (`‘srebrne’`)
- w wariancie piątym i siódmym lista `rz` w miejscu `docel` ma dwa elementy
- w wariancie szóstym i ósmym nie zgadza się rodzaj rzeczownika (`‘klucz’`) i przymiotnika (`‘drewniana’`) w miejscu `docel`

Zostają zatem dwa pierwsze elementy. Po uzupełnieniu rodzajów mamy końcową listę wariantów:

```
[
  {
    jakiopr => obj,
    lastall => obj,
    obj => {
      rodz => n,
      opr => { rodz => m, prz => [ ‘srebrny’ ], rz => [ ‘klucz’ ] },
      all => [ n ]
    },
    docel => { rodz => z, prz => [ ‘drewniana’ ], rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  },
  {
    jakiopr => obj,
    lastall => obj,
    obj => {
      rodz => n,
      opr => { rodz => m, prz => [ ‘srebrny’ ], rz => [ ‘klucz’ ] },
      all => [ n ]
    },
    docel => { rodz => z, prz => [ ‘drewniana’ ] },
    komu => { rodz => z, rz => [ ‘skrzynia’ ] },
    rozkaz => ‘włożyć’
  }
]
```

Wariant pierwszy odzwierciedla faktyczny sens zdania. Wariant drugi reprezentuje polecenie, aby włożyć *komu? czemu?* skrzyni, wszystko, oprócz srebrnego klucza, do jakiejś niesprecyzowanej drewnianej rzeczy. Z punktu widzenia rozbioru gramatycznego oba warianty są jednak równie dobre.

2.3.5. Uwagi

Przytoczony algorytm jest bardzo ogólny. W praktyce rozbiór będzie na ogół jednoznaczny i większość list będzie się degenerowała do list pustych lub jednoelementowych. Jednak nie musi tak być — dobrym przykładem jest rozbiór przedstawiony w poprzednim podrozdziale, gdzie na pewnym etapie mieliśmy aż osiem wariantów.

Co więcej, można przypuszczać — choć nie zostało to przetestowane — że dzięki takiej ogólności algorytm może być, bez żadnych zmian, wykorzystany dla innych podobnych języków (np. dla języka rosyjskiego). Elementy specyficzne dla konkretnego języka zostały wydzielone i mogą być modyfikowane bez ingerowania w kod funkcji dokonującej rozbioru składniowego. Są to:

- lista przypadków, przez które odmieniają się rzeczowniki i przymiotniki;
- słowa "wszystko", "wszyscy" i "wszystkie", odmienione przez przypadki;
- słowo "ja", odmienione przez przypadki;
- przypadki i przyimki, odpowiadające miejscom składniowym;
- słowo "oprócz" i przypadek, w jakim występują wyrazy kojarzone z "oprócz" (w języku polskim dopełniacz);

Oczywiście oprócz tych danych konfiguracyjnych należałoby zmienić zawartość plików z danymi, zasilających zbiorniki `przymiotniki`, `rzeczowniki` i `czasowniki`.

2.4. Interpretacja polecenia

Wynik opisanej w poprzednim rozdziale procedury rozbioru gramatycznego jest następnie poddawany interpretacji, która ma na celu ustalenie możliwych znaczeń analizowanego polecenia. Polega to na odszukaniu, dla każdego wariantu rozbioru, wszystkich działań nazwanych czasownikiem, wypełniającym w danym wariancie pole `rozkaz`, a następnie, dla każdego z tych działań, znalezieniu wszystkich sposobów przypisania wymaganym przez nie argumentom obiektów, znajdujących się w polu widzenia postaci, której dotyczy polecenie, opisywanych rzeczownikami i czasownikami, wypełniającymi miejsce składniowe, odpowiadające argumentowi, i należących do odpowiedniego typu (`postacie`/`przedmioty`/`drzwi`). W ten sposób z jednego wariantu rozbioru gramatycznego może zostać wyprodukowanych kilka wariantów interpretacji — np. rozbiór polecenia "Weź klucz" jest jednoznaczny, jednak jeśli w zasięgu wzroku postaci będą dwa klucze, to powstaną dwa warianty interpretacji. Wciąż jeszcze nie musi to oznaczać błędu: jeśli tylko jeden z kluczy jest możliwy do wzięcia (bo np. drugi jest już niesiony przez postać), to wszystko w porządku; usuwanie takich niejednoznaczności (tzn. eliminacja wariantów, które mają określone znaczenie, lecz są z jakiegoś powodu niewykonalne) nie jest jednak częścią procesu interpretacji — odbywa się ono dopiero w części algorytmu opisanej w rozdziale 2.5.

Jeśli w jakimś wariancie rozkładu gramatycznego jest wypełnione miejsce składniowe, którego działanie nie oczekuje, to zostanie on pominięty (w ten właśnie sposób odrzucony byłby błędny wariant w drugim z przykładów w rozdziale 2.3.2). Przeciwne stwierdzenie nie jest jednak prawdziwe — jeśli któreś wymaganych przez działanie miejsc składniowych jest w pewnym wariancie rozbioru gramatycznego puste, to program próbuje zgadnąć, co powinno tam trafić. Na poziomie interpretacji polecenia sprowadza się to po prostu do stworzenia wariantów interpretacji, w których puste miejsce jest wypełnione przez każdy widziany przez

postać obiekt odpowiedniego typu. Warianty niewykonalne są eliminowane w późniejszej części algorytmu. Zatem np. polecenie "Otwórz" jest jak najbardziej poprawne, jeśli postać może otworzyć drzwi i nie ma nic innego, co mogłaby otworzyć. Takie podejście naśladuje zachowanie oryginalnej gry "Hobbit" (i oszczędza graczowi pisania). Pojawia się tu jednak pewna trudność. Z czasownikiem może być skojarzone więcej niż jedno działanie, a zbiór miejsc składniowych, wymaganych przez jedno z działań, może zawierać się w zbiorze miejsc składniowych wymaganych przez inne działanie (np. można zamknąć drzwi lub zamknąć drzwi kluczem). W takiej sytuacji gracz nie ma możliwości doprecyzowania, że chodzi mu o działanie z mniejszą liczbą miejsc składniowych — bo nadmiernie domyślny program będzie wprowadzał niejednoznaczność, dodając przy interpretacji polecenia "Zamknij drzwi" wariant z zamykaniem drzwi na klucz. Dlatego przyjmujemy zasadę, że warianty z jawnie podanymi wszystkimi argumentami mają pierwszeństwo przed wariantami, będącymi wynikiem domyślności programu. Aby to umożliwić, funkcja dokonująca interpretacji zaznacza, które argumenty działań zostały wygenerowane z pustych miejsc składniowych.

Funkcja zaznacza też, które argumenty działań zostały wygenerowane z miejsc składniowych z niepustym polem `all`. Poza tym te miejsca składniowe są traktowane niemal identycznie jak miejsca bez `all` — funkcja tworzy po jednym wariancie interpretacji dla każdego obiektu odpowiedniego typu, opisywanego rzeczownikami i przymiotnikami wypełniającymi miejsce, lub w ogóle wszystkimi obiektami właściwego typu jeśli w miejscu nie ma żadnych rzeczowników ani przymiotników. Jedyne dodatkowe ograniczenie jest takie, że rzeczownik, nazywający obiekt, musi mieć formę o rodzaju zgodnym z rodzajem, znajdującym się w polu `all` — aby polecenie "Podnieś wszystkich" nie oznaczało nakazu podniesienia np. klucza i skrzynki (jeśli chcemy, aby jakieś obiekty były rozpoznawane zarówno jako męskoosobowe jak i niemęskoosobowe — aby np. gracz mógł napisać "Zabij wszystkie trolle" równie dobrze jak "Zabij wszystkich trollów" — to nic nie stoi na przeszkodzie, aby umieścić w zbiorniku `rzeczowniki` dwa rekordy liczby mnogiej rzeczownika "troll", w wersji męskoosobowej i niemęskoosobowej).

Kolejną rzeczą, jaką musimy wziąć pod uwagę, jest zawartość pola `opr`. Aby to uwzględnić funkcja dokonująca interpretacji zwraca w wyniku dwie listy — pierwsza z nich to omawiana do tej pory lista wariantów interpretacji. Druga to lista działań zakazanych, zbudowana w podobny sposób jak ta pierwsza, ale w oparciu o zawartość pól `opr`. Będziemy je nazywać `acttab` i `donttab`, odpowiednio.

Pole `cytat` w ogóle nie jest dotykane podczas interpretacji.

2.4.1. Postać wyniku

Podobnie, jak opisana w rozdziale 2.3.1 struktura `gramtab`, `acttab` składa się z opcjonalnego pola błędu, opcjonalnego pola cytatu, oraz listy wariantów. Element listy wariantów ma pole `dzial`, zawierające odniesienie do pewnego działania (tzn. rekordu ze zbiornika `dzialania`), kolekcję struktur — po jednej strukturze dla każdego miejsca składniowego, wymaganego przez to działanie — oraz pole `szemrane`, typu logicznego, które mówi, czy zawartość którejś z tych struktur została wygenerowana z pustego miejsca składniowego w wariancie rozbioru gramatycznego. Struktura opisująca miejsce składniowe ma pola:

obiekt Odniesienie do elementu zbiornika `postacie`, `przedmioty` lub `drzwi` (zależnie od tego, jakiego typu obiektu oczekuje w tym miejscu składniowym działanie wskazywane przez `dzial`).

all Wartość logiczna; równe `true`, jeśli odpowiednie miejsce składniowe w wariancie rozbioru

gramatycznego, z którego został wyprodukowany ten wariant interpretacji, miało niepuste pole `all`.

szemr Wartość logiczna; równe `true`, jeśli odpowiednie miejsce składniowe w wariancie rozbioru gramatycznego, z którego został wyprodukowany ten wariant interpretacji, było puste.

Dodatkowo `acttab` ma pole `jestall`, ustawione na `true` jeśli któreś z pól `all` w strukturach opisujących miejsca składniowe na liście wariantów jest równe `true`. Ponieważ słów będących przyczyną niepustego `all` ("wszystko", "wszystkie", ...) nie można zinterpretować inaczej, więc jeśli którykolwiek wariant zawiera strukturę, mającą `all` równe `true`, to każdy wariant zawiera taką strukturę.

`donttab` nie ma innych pól poza listą wariantów. Elementy listy wariantów mają identyczną postać jak w `acttab`.

2.4.2. Przykłady

Przyjmijmy, że gracz przebywa w pomieszczeniu, w którym znajdują się okrągłe zielone drzwi oraz drewniana skrzynia. Przyjmijmy dodatkowo, że mamy w zbiorniku `dzialania` cztery działania: `otwdrzwi`, wymagające w miejscu `obj` obiektu typu `drzwi`, `otwdrzkl`, wymagające w miejscu `obj` obiektu typu `drzwi` a w miejscu `util` obiektu typu `przedmioty`, `otwprz`, wymagające w miejscu `obj` obiektu typu `przedmioty`, oraz `otwprzkl`, wymagające w `obj` oraz w `util` obiektów typu przedmioty. Żadne z działań nie oczekuje obecności cytatu.

Wtedy pierwszy z rozbiorów w rozdziale 2.3.2 przełoży się na następującą interpretację:

```
acttab = {
  [
    {
      dzial => otwdrzwi,
      obj => { obiekt => drzwi, all => false, szemr => false },
      szemrane => false
    },
    {
      dzial => otwdrzwikl,
      obj => { obiekt => drzwi, all => false, szemr => false },
      util => { obiekt => skrzynia, all => false, szemr => true },
      szemrane => true
    }
  ],
  jestall => false
}
donttab = []
```

Jak widzimy, zdanie może zostać potraktowane zarówno jako polecenie wykonania działania `otwdrz` jak i `otwdrzkl`, jednak drugi wariant został oznaczony jako wątpliwy gdyż jego zbudowanie wymagało wypełnienia miejsca składniowego, które w źródłowym `gramtab` było puste. Spójrzmy teraz na interpretację drugiego z przykładów w rozdziale 2.3.2:

```
acttab = {
  [
    {
```

```
      dzial => otwdrzwi,
      obj => { obiekt => drzwi, all => true, szemr => false },
      szemrane => false
    },
    {
      dzial => otwdrzkl,
      obj => { obiekt => drzwi, all => true, szemr => false },
      util => { obiekt => skrzynia, all => false, szemr => true },
      szemrane => true
    },
    {
      dzial => otwprz,
      obj => { obiekt => skrzynia, all => true, szemr => false },
      szemrane => false
    },
    {
      dzial => otwprzkl,
      obj => { obiekt => skrzynia, all => true, szemr => false },
      util => { obiekt => skrzynia, all => false, szemr => true },
      szemrane => true
    }
  ],
  jestall => true
}
donttab = [
  {
    dzial => otwprz,
    obj => { obiekt => skrzynia, all => true, szemr => false },
    szemrane => false
  },
  {
    dzial => otwprzkl,
    obj => { obiekt => skrzynia, all => true, szemr => false },
    util => { obiekt => skrzynia, all => false, szemr => true },
    szemrane => true
  }
]
```

Tu funkcja odnalazła aż cztery warianty: otwarcie drzwi, otwarcie drzwi skrzynią, otwarcie skrzyni i otwarcie skrzyni skrzynią (jak stąd widzimy, proces interpretacji odnajduje możliwe znaczenia, lecz nie wypowiada się na temat ich sensowności). Wszystkie zostały wyprodukowane z pierwszego wariantu źródłowego `gramtab`. Drugi wariant rozbioru zginął bezpotomnie, gdyż żadne z działań, skojarzonych z czasownikiem "otworzyć", nie oczekuje niczego w miejscu składniowym `komu`. Drugi i czwarty wariant interpretacji zostały oznaczone jako wątpliwe, gdyż mają niepuste `util`, mimo że w wariancie rozbioru gramatycznego, z którego pochodzą, miejsce składniowe `util` było puste. Warianty trzeci i czwarty zostały zabronione przez zawartość listy `donttab`.

2.4.3. Algorytm interpretacji

Oprócz wyprodukowanej w poprzednim kroku struktury `gramtab`, zawierającej listę wariantów rozbioru gramatycznego, wejściem dla algorytmu interpretacji polecenia są jeszcze: `os` — odnośnik do osoby, której polecenie dotyczy (może to być gracz lub dowolny element zbiornika `postacie`), oraz `mowca` — odnośnik do postaci, która wypowiedziała polecenie, z którego wyprodukowany został interpretowany `gramtab`; jeśli polecenie było wpisane przez gracza, to `mowca` zawiera wartość pustą.

Początkowo obie konstruowane wartości — `acttab` i `donttab` — są inicjalizowane listami pustymi. Jeśli pole błędu w `gramtab` jest niepuste, to jest ono przepisywane bez zmian do `acttab` i na tym kończy się działanie funkcji.

Interpretacja zaczyna się od ustalenia miejsca pobytu postaci `os`, oraz listy obiektów widocznych dla `os`. Przez miejsce pobytu postaci lub przedmiotu będziemy tu rozumieć wynik przechodzenia po obiektach, wskazywanych przez pole `gdzie_lok/gdzie_prz/gdzie_osoba`, aż do trafienia na lokację lub na zamknięty przedmiot. Czyli np. jeśli na polanie znajduje się troll trzymający klucz, to miejscem pobytu klucza jest polana. Z kolei, jeśli w norce znajduje się skrzynia, w której siedzi Gandalf, trzymający mapę, to miejscem pobytu mapy oraz Gandalfa jest norka lub skrzynia, zależnie od tego, czy skrzynia jest otwarta czy zamknięta. Lista obiektów, widocznych dla `os`, będzie to po prostu lista przedmiotów/postaci, mających takie samo jak `os` miejsce pobytu, oraz, jeśli miejscem pobytu jest lokacja, drzwi wychodzących z tej lokacji.

Następnie, dla każdego wariantu rozbioru gramatycznego, funkcja wybiera wszystkie rekordy ze zbiornika `dzialania`, które są skojarzone z czasownikiem, wypełniającym pole `rozkaz` w danym wariancie, oraz wymagają/nie wymagają cytatu, zależnie od tego czy w `gramtab` jest cytat czy nie. Dla każdego działania funkcja konstruuje osobną, lokalną wersję list `acttab` i `donttab`, wg. algorytmu opisanego poniżej.

Konstrukcja przy ustalonym wariancie rozbioru i działaniu

Ściśle mówiąc, będziemy konstruowali listy `acttab` i `botab` — ta druga będzie listą z pominiętymi wariantami, zakazanymi przez zawartość pól `opr`. Lista `donttab` będzie to po prostu różnica `acttab` i `botab`.

Zarówno `acttab`, jak i `botab` inicjalizujemy listami jednoelementowymi, zawierającymi element z polem `dzial` równym bieżącemu działaniu i z pustą kolekcją struktur, opisujących miejsca składniowe. Następnie funkcja iteruje po wszystkich miejscach składniowych `ms`. Jeśli dla jakiegoś `ms` bieżący wariant rozbioru ma niepustą zawartość, zaś bieżące działanie nie wymaga takiego miejsca, to koniec — lista `acttab` dla tego wariantu i działania będzie pusta. Jeśli wariant rozbioru ma puste `ms` i działanie nie wymaga `ms`, to w porządku — funkcja nic nie robi dla tego `ms`. Jeśli natomiast działanie wymaga `ms`, to funkcja

1. Inicjalizuje dwie zmienne pomocnicze, `szemrane` i `all`. Zmienna `szemrane` zdecyduje o ustawieniu pola `szemr`, zaś zmienna `all` o ustawieniu pola `all`, w strukturach, odpowiadających miejscu `ms`, w elementach konstruowanej lokalnej wersji `acttab`. Jeśli w bieżącym wariancie rozbioru gramatycznego pole `all` dla miejsca `ms` jest niepuste, to zmienna `all` jest ustawiana na `true`, zaś zmienna `szemrane` na `false`. W przeciwnym wypadku zmienna `all` jest ustawiana na `false`, zaś zmienna `szemrane` na `true`.

2. Konstruuje listę `lob` (*lista obiektów*), zawierającą odnośniki do wszystkich obiektów, zgodnych z zawartością miejsca `ms` w bieżącym wariancie rozbioru, nie biorąc pod uwagę pola `opr`. W tym celu:

- Jeśli pole `mowc` w miejscu `ms` w bieżącym wariancie rozbioru jest niepuste, to
  - jeśli działanie oczekuje w miejscu `ms` innego typu niż postacie, to koniec — lista `lob` będzie pusta;
  - funkcja sprawdza, czy `mowca` należy do obiektów widocznych dla `os`; jeśli należy, to `lob` będzie mieć jeden element równy `mowca`, zaś `szemrane` zostanie ustawione na `false`; jeśli nie należy, to `lob` będzie pusta.
- (Tu funkcja dochodzi, jeśli `mowc` było puste). Funkcja wstawia do `lob` te spośród obiektów widocznych dla `os`, które mają taki typ, jakiego oczekuje bieżące działanie w miejscu `ms` i są opisywane wszystkimi rzeczownikami i przymiotnikami znajdującymi się w miejscu `ms` w bieżącym wariancie rozbioru. W szczególności, jeśli pola `rz` i `prz` dla miejsca `ms` w bieżącym wariancie rozbioru są puste, oznacza to po prostu wszystkie widoczne dla `os` obiekty właściwego typu. Jeśli `rz` lub `prz` są niepuste, to zmienna `szemrane` jest ustawiana na `false`.
- Jeśli pole `rodz` w miejscu `ms` dla bieżącego wariantu rozbioru jest niepuste i różne od `n` (rodzaj nijaki), to funkcja wyrzuca z listy `lob` elementy, niezgodne z zawartością pola `rodz`, tzn. takie, że opisujący je rzeczownik nie ma formy w rodzaju `rodz`. Rodzaj nijaki w polu `rodz` nie powoduje usuwania elementów z listy `lob`, gdyż chcemy, aby polecenia typu "Otwórz wszystko" odnosiły się do wszystkich przedmiotów, a nie tylko tych opisywanych rzeczownikami rodzaju nijakiego.

Jeśli lista `lob` jest pusta to koniec — `acttab` dla bieżącego wariantu rozbioru i działania będzie pusta.

3. Konstruuje listę `oprlob`, zawierającą odnośniki do wszystkich obiektów, zgodnych z zawartością pola `opr` dla bieżącego wariantu rozbioru. To znaczy:
   - Jeśli `opr.rz` lub `opr.prz` są dla miejsca `ms` w bieżącym wariancie rozbioru niepuste, to funkcja umieszcza w `oprlob` te elementy `lob`, które są opisywane wszystkimi rzeczownikami i przymiotnikami w `opr.rz` i `opr.prz`, i są zgodne z rodzajem `opr.rodz`.
   - Jeśli `opr.mowc` jest niepuste dla miejsca `ms` w bieżącym wariancie rozbioru, to funkcja umieszcza w `oprlob` jeden element równy `mowca`.
4. Konstruuje listę `lobbo`, będącą różnicą list `lob` i `oprlob`.
5. Powiela zawartość lokalnej wersji `acttab` w podobny sposób, jak w rozdziale 2.3.3 powielana była lista wariantów rozbioru, po jednej kopii na każdy element listy `lob`. Tzn. jeśli przez n oznaczymy liczbę elementów dotychczasowej listy `acttab`, zaś przez m liczbę elementów listy `lob`, to nowa `acttab` ma mn elementów, przy czym wycinki $0 \ldots n-1$, $n \ldots 2n-1$, $\ldots$ są identyczne z dotychczasową `acttab`.
6. Robi to samo z `botab`, biorąc `lobbo` zamiast `lob`.
7. Nanosi każdy element listy `lob` na odpowiadający mu wycinek listy `acttab`, tzn. w każdym elemencie wycinka pole `obiekt` struktury, opisującej `ms`, jest ustawiane na wartość tego elementu `lob`, do którego należy wycinek.
8. Nanosi każdy element listy `lobbo` na odpowiadający mu wycinek listy `botab`.

9. We wszystkich elementach tworzonego `acttab` ustawia pola `all` i `szemr` dla miejsca `ms` na wartości równe, odpowiednio, zmiennej `all` i `szemrane`. Jeśli zmienna `szemrane` jest równa `true`, to ustawia we wszystkich elementach tworzonego `acttab` pole `szemrane` na `true`.

Po wykonaniu tych kroków dla każdego miejsca składniowego, listy `acttab` i `botab` są skonstruowane. Na koniec funkcja odejmuje `botab` od `acttab`, aby otrzymać lokalną wersję listy `donttab`.

Konstrukcja pełnego wyniku

Po opisanym w poprzednim podrozdziale wyznaczeniu małych `acttab` i `donttab` dla każdego wariantu rozbioru i działania, funkcja konkatenuje je, otrzymując wynikowe `acttab` i `donttab`. Na koniec jeszcze:

- Z listy `acttab` usuwane są te elementy, które mają w którymś miejscu składniowym pole `obiekt` równe `os`.
- Wykonywane są podstawienia działań, tzn. dla każdego elementu `acttab` i `gramtab`, którego pole `dzial` jest równe polu `zastepowane` w którymś rekordzie ze zbiornika `aliasydz`, `dzial` jest zmieniane na wartość pola `zastepujace` w tym rekordzie, zaś wszystkie miejsca składniowe są przenoszone, zgodnie z zawartością tego rekordu.
- Wartość pola `cytat` w źródłowym `gramtab` jest bez zmian przenoszona do pola `cytat` w wynikowym `acttab`.

2.5. Wykonanie polecenia

Zbierzemy teraz w całość przedstawione dotąd informacje, aby prześledzić pracę programu od momentu wprowadzenia przez użytkownika polecenia, aż do całkowitego jego wykonania. Najpierw w rozdziale 2.5.1 omówimy, w jaki sposób program wybiera z listy `acttab` jedno działanie do wykonania. Opisane w rozdziale 2.5.2 pełne wykonanie wielozdaniowej instrukcji polega, w uproszczeniu, na odrywaniu z niej po jednym zdaniu, dokonywaniu rozbioru gramatycznego tego zdania, a następnie wykonywaniu, jeden lub więcej razy, czynności opisanych w rozdziale 2.5.1. Powtórzenie ich więcej niż raz może być potrzebne, jeśli w obrabianym zdaniu występuje słowo "wszystko" lub podobne. Np. dla zdania "Weź wszystko" lista `acttab` będzie zawierać po jednym wariancie działania dla każdego przedmiotu w polu widzenia postaci. Z tej listy, w kolejnych turach, program będzie wybierał po jednym z możliwych do wykonania wariantów, na osobnej liście zapamiętując, które warianty zostały już wykorzystane, aby nie wybrać ich ponownie.

Algorytm wyboru działania może się nie powieść, jeśli żadnego z działań nie da się wykonać. W takiej sytuacji procedura dokonująca wyboru zwraca, nadający się do wyświetlenia użytkownikowi, komunikat o błędzie. Jednak komunikat ten należy wyświetlić tylko w sytuacji, gdy błąd pojawił się za pierwszą próbą wyboru działania — np. gracz wpisał "Otwórz wszystko", procedura zwraca komunikat "Drzwi są otwarte", program informuje użytkownika o błędzie i na tym kończy się wykonywanie polecenia. Jeśli natomiast procedura zwróci błąd przy kolejnym zawołaniu, to wszystko w porządku — wyczerpane zostały po prostu możliwe do wykonania warianty. Gdy np. gracz wpisuje "Otwórz wszystko", postać szczęśliwie otwiera skrzynkę i kuferek, a trzecia próba wyboru działania kończy się komunikatem "Drzwi są otwarte", to program nie zgłasza błędu lecz przechodzi do następnego zdania (jeśli takie jest).

Inną przyczyną porażki algorytmu wyboru działania może być niejednoznaczność. Jeśli w rozpatrywanym zdaniu nie ma żadnego "wszystko", to żądanie jednoznaczności sprowadza się do wymagania, aby z obecnych w `acttab` wariantów dokładnie jeden był możliwy do wykonania — z dokładnością jednak do, opisanego w rozdziale 2.4, podziału wariantów na gorsze (wątpliwe) i lepsze: gdy w `acttab` jest dokładnie jeden wariant bez flagi `szemrane`, to wybór jest jednoznaczny niezależnie od liczby pozostałych wariantów. Jeśli w zdaniu było jakieś "wszystko", to, jak widzieliśmy na powyższych przykładach, lista wygenerowanych wariantów może mieć więcej niż jeden element. Nie znaczy to, że rezygnujemy z jednoznaczności — np. lista wariantów, wygenerowanych ze zdania "Włóż wszystko do skrzyni", może być poprawna, o ile w polu widzenia znajduje się dokładnie jedna otwarta skrzynia; ale jeśli będą dwie, to funkcja dokonująca wyboru zgłosi błąd z powodu niejednoznaczności.

Opisane operacje odwołują się do następujących zmiennych globalnych:

uzyteat Lista działań, które już zostały wykonane; jej elementy mają taką samą postać, jak elementy `acttab`.

krzyczec Wartość logiczna. Mówi, czy w razie niepowodzenia funkcji wybierającej działanie należy zgłaszać błąd.

powtorzyc Wartość logiczna. Funkcja wybierająca działanie ustawia ją na `true`, jeśli życzy sobie zostać zawołana ponownie — czyli jeśli otrzymana w wyniku interpretacji struktura `acttab` ma pole `jestall` równe `true`.

blad Wartość logiczna. Ustawiana na `true` m.in. w przypadku niepowodzenia funkcji wybierającej działanie.

2.5.1. Wybór i wykonanie pojedynczego działania

Omawiana funkcja przyjmuje takie same argumenty jak opisana w rozdziale 2.4 funkcja dokonująca interpretacji — `gramtab`, `os` i `mowca`. Jak łatwo stąd wywnioskować, może być używana także do ustalania działań postaci niegrających. Dla uproszczenia będziemy się koncentrować na przypadku, gdy polecenie zostało wpisane przez użytkownika i dotyczy postaci gracza, i pomijać niektóre szczegóły, ważne w innych przypadkach (w rzeczywistości np. `uzyteat` jest dodatkowo indeksowana odniesieniem do postaci, której dotyczy — `uzyteat[os]`).

Pierwszym krokiem jest właśnie interpretacja, w wyniku której tworzone są `acttab` i `donttab`. Jeśli otrzymana `acttab` ma ustawione pole sygnalizujące błąd, lub jeśli jej lista wariantów jest pusta, to funkcja kończy działanie zgłaszając błąd. Od listy wariantów `acttab` funkcja odejmuje listy `donttab` i `uzyteat`. Jeśli `acttab` ma pole `jestall` równe `true`, to zmienna `powtorzyc` jest ustawiana na `true`.

Następnie, dla każdego przedmiotu i postaci znajdujących się w polu widoczności postaci `os` (pole widoczności rozumiemy tak jak w rozdziale 2.4), funkcja znajduje właściciela — tzn. przechodzi po obiektach, wskazywanych przez pola `gdzie_*`, tak długo, aż natrafi na postać lub na lokację; w tym drugim przypadku obiekt nie ma właściciela. Z listy `acttab` usuwane są wszystkie warianty, dla których któryś argument działania ma właściciela i tym właścicielem nie jest `os`. Jeśli po tej operacji lista `acttab` jest pusta, to funkcja kończy działanie zgłaszając błąd.

Następnie dla każdego wariantu funkcja wykonuje fragmenty kodu, znajdujące się w polach `warunek` tych rekordów ze zbiornika `warunki`, których pole `dzialanie` jest identyczne z polem `dzial` danego wariantu, w kolejności wyznaczonej przez pole `kolejnosc`. Jeśli którykolwiek fragment kodu zwróci negatywną odpowiedź (czyli warunek nie jest spełniony),

funkcja usuwa wariant z listy `acttab`. Jeśli po tej operacji lista `acttab` jest pusta, to funkcja kończy działanie zgłaszając błąd.

Konkretna postać zwróconego komunikatu o błędzie w obu operacjach zależy od tego, czy usunięty wariant był unikatowy. Jeśli test spowodował usunięcie jedynego wariantu, lub jedynego wariantu bez flagi `szemrane`, to komunikat mówi precyzyjnie, dlaczego zażądane działanie jest niemożliwe (w pierwszym przypadku będzie to informacja w rodzaju "Troll trzyma ogromny klucz"; w drugim zdanie wyprodukowane na podstawie zawartości pola `komunikat` w niespełnionym warunku). W przeciwnym razie funkcja zwraca ogólnikowy komunikat, taki jak "Nie rozumiem" lub "Nie możesz tego zrobić".

Następnie funkcja bada jednoznaczność otrzymanej listy możliwych do wykonania działań. Jeśli pole `jestall` w strukturze `acttab` jest ustawione na `false`, to:

- funkcja sprawdza, czy istnieje dokładnie jeden wariant, jeśli tak to wybiera go do wykonania;
- jeśli lista wariantów ma więcej niż jeden element, funkcja sprawdza, czy istnieje dokładnie jeden wariant z polem `szemrane` ustawionym na `false` — jeśli tak, wybiera go do wykonania;
- jeśli jest więcej niż jeden wariant, a liczba wariantów z polem `szemrane` równym `false` jest różna niż jeden, funkcja kończy pracę z komunikatem o błędzie.

W przypadku gdy pole `jestall` jest równe `true`, funkcja wykonuje poniższe czynności, najpierw dla wszystkich wariantów, a jeśli zakończy się to błędem, to tylko dla wariantów z polem `szemrane` ustawionym na `false`:

- sprawdza, czy w każdym wariancie `all` jest w tych samych miejscach składniowych — jeśli nie, to błąd;
- sprawdza, czy dla każdej możliwej kombinacji wartości pól z `all` istnieje dokładnie jeden wariant realizujący tę kombinację — jeśli nie, to błąd;
- wybiera do wykonania dowolny wariant.

Zauważmy, że funkcja nie sprawdza, czy wszystkie warianty mają identyczne pole `dzial` — dzięki temu np. polecenie "Otwórz wszystko" zostanie wykonane, nawet jeśli jeden z otwieranych przedmiotów jest zamknięty na klucz.

Wybrany wariant jest następnie dodawany do listy `uzyteat` i wykonywany. Wykonanie działania polega na zawołaniu kodu, do którego odniesienie znajduje się w polu `funkcja` rekordu, wskazywanego przez pole `dzial` wybranego wariantu. Kod ten może robić zupełnie cokolwiek, w szczególności modyfikować zawartość zbiorników, opisanych w rozdziale 2.2, oraz komunikować się z fragmentami kodu, obsługującymi zachowanie innych postaci. Jest też odpowiedzialny za wypisanie odpowiedniego komunikatu, np. "Otwierasz okrągłe zielone drzwi".

Przed zwróceniem kodu informującego o sukcesie funkcja ustawia zmienną `krzyczec` na `false`.

2.5.2. Wykonanie pełnego polecenia

Przechodzimy wreszcie do opisu całości procesu wykonywania wprowadzonych przez użytkownika instrukcji. Omawiana funkcja będzie w pętli odrywać z wprowadzonego polecenia po jednym zdaniu. Przyjmujemy, że zdania są oddzielone kropkami. Na początku każdego

obiegu pętli ustawiane są zmienne globalne: `uzyteat` na listę pustą, `krzyczec` na wartość `true`, `powtorzyc` i `blad` na wartość `false`.

Pierwszym etapem obróbki zdania jest wykonanie pewnych zdefiniowanych zastąpień, które ułatwią jego dalszą analizę. Na przykład słowo "ze" jest zastępowane przez "z" — gdyż, z punktu widzenia omówionej w poprzednich rozdziałach analizy, oba te przyimki grają dokładnie identyczną rolę.

Kolejny krok wiąże się z tym, że — śladem oryginalnego "Hobbita" — dla wygody gracza poruszanie po mapie może odbywać się za pomocą krótkich poleceń z nazwami kierunków, np. "Pn", "Wsch" itp. Jeśli zdanie jest równe nazwie jakiegoś kierunku, to w tym momencie funkcja wykonuje próbę przemieszczenia w tym kierunku. Jeśli próba się powiedzie, to funkcja wykonuje resztę tury (czyli m.in. pozwala wykonać jakieś działanie pozostałym postaciom) i przechodzi do następnego obiegu pętli, czyli następnego zdania. Jeśli próba przemieszczenia się nie powiedzie, to funkcja wypisuje komunikat i kończy działanie (cała reszta polecenia — dalsze zdania — jest ignorowana).

Następnie funkcja, stosując opisany w rozdziale 2.3.3 algorytm, oblicza `gramtab` dla obrabianego zdania, po czym wchodzi do wewnętrznej pętli. Wewnętrzna pętla obejmuje następujące czynności:

- Zawołanie opisanej w rozdziale 2.5.1 procedury, zawierającej interpretację `gramtab`, próbę wyboru spośród uzyskanych wariantów działania do wykonania i w razie sukcesu, wykonanie wybranego działania. Jeżeli procedura zwróci komunikat o błędzie, wtedy
  - zmienna `blad` jest ustawiana na `true`;
  - jeśli `krzyczec` jest równa `true`, to funkcja wypisuje otrzymany komunikat.
- Jeśli `blad` jest równy `false`, ruch reszty świata. Oznacza to wykonanie działań przez pozostałe postacie, a także zajście różnych zdarzeń które uaktywniają się losowo, lub w sytuacji gdy stan świata spełnia jakiś warunek.
- Sprawdzenie, czy gracz podczas któregoś z poprzednich punktów nie stracił życia. Jeśli stracił, to gra się kończy.
- Sprawdzenie warunku pętli: czy `powtorzyc` jest równe `true` i `blad` jest równy `false`. Zależnie od wyniku, przejście do nowego obiegu lub wyjście z pętli.

Po wyjściu z wewnętrznej pętli funkcja sprawdza warunek dużej pętli: jeśli wprowadzone polecenie zostało już całkowicie skonsumowane (nie ma juz więcej zdań do odrywania), lub jeśli obie zmienne — `blad` i `krzyczec` — są ustawione na `true`, to funkcja wychodzi z pętli i kończy działanie. W przeciwnym razie rozpoczyna się kolejny obieg zewnętrznej pętli, obsługujący kolejne zdanie.

2.5.3. Uwagi

Zastosowany sposób traktowania pól `all` — polegający na spłaszczającym wstawieniu listy powstałych z takiego pola wariantów do listy wszystkich wariantów, a następnie badaniu jednoznaczności uzyskanej listy przy ustalonych wartościach obiektów w polach gdzie było `all` — może początkowo wydawać się dziwny. Można pomyśleć, że prościej byłoby dla takich pól umieszczać zamiast obiektu, w elemencie `acttab`, całą listę wygenerowanych z pola `all` obiektów. Warunek jednoznaczności byłby wtedy taki sam, niezależnie czy były jakieś `all` czy nie. Jednakże nie jest to dobry sposób. Rozpatrzmy zdanie "Zamknij wszystkie drzwi

na klucz". Powiedzmy, że postać trzyma dwa klucze. Powstałyby zatem dwa warianty interpretacji, oba z listą wszystkich widocznych drzwi w miejscu `obj`, natomiast różniące się wybranym kluczem. Jeśli teraz każdy klucz pasowałby do jednych drzwi, to nie da się jednoznacznie wybrać wariantu, ponadto żaden wariant nie odpowiadałby intencjom użytkownika. Natomiast, jak łatwo sprawdzić, faktycznie użyty algorytm zachowałby się właściwie.

Można też zauważyć, że interpretacja zdania może być wykonywana wielokrotnie, za każdym obiegiem wewnętrznej pętli opisanej w rozdziale 2.5.2. Dzięki temu, gdy użytkownik wpisze np. polecenie "Otwórz wszystko", a po otwarciu skrzynki okaże się, że było w niej schowane pudełko, to postać otworzy też pudełko. Wybór takiej semantyki jest znów spowodowany naśladownictwem gry "Hobbit". Oczywiście możliwy był też inny wybór — gdybyśmy chcieli, aby polecenie dotyczyło wyłącznie obiektów widocznych w momencie jego wydawania, wtedy należałoby interpretację przenieść przed wejście do wewnętrznej pętli, bezpośrednio za rozbiór gramatyczny. Należałoby wtedy jeszcze rozstrzygnąć, czy polecenie obejmuje obiekty, które w momencie jego wydawania były widoczne ale niedostępne (obecnie oczywiście obejmuje, o ile tylko w którymś momencie podczas wykonywania polecenia staną się dostępne).

2.6. Ograniczenia i perspektywy rozwoju

Przedstawiony algorytm rozpoznaje podzbiór języka polskiego mający trochę mniejszą siłę wyrazu niż, służący za wzór, *Inglish*. Jednak został tak pomyślany, aby dodawanie rozszerzeń było jak najłatwiejsze.

Na przykład, angielska wersja rozumie przysłówki: "Quickly open the door". Nic nie stoi na przeszkodzie, aby wprowadzić nową kolekcję `przyslowki` na modelu danych i dodać do elementu listy wariantów w strukturze `gramtab` pole `przyslowki`, zawierające listę odwołań do rekordów z tej kolekcji. Wiązałoby się to oczywiście z dodaniem nowego sposobu identyfikacji słowa, tzn. oprócz list `rozk`, `oprrz`, `oprprz`, ... w rozdziale 2.3.3 mielibyśmy jeszcze listę `przysl`. Wszystkie te zmiany są nieskomplikowane. Oczywiście, pozostaje pytanie, jak traktowany byłby taki przysłówek na etapie interpretacji. Prawdopodobnie chcielibyśmy, aby modyfikował w jakiś sposób skojarzone z czasownikiem z pola `rozkaz` działanie (np. "Ostrożnie wyważ drzwi mieczem" wiązałoby się z mniejszym ryzykiem złamania miecza niż samo "Wyważ drzwi mieczem").

Inne możliwe rozszerzenie to rzeczowniki w funkcji przydawki. W oryginalnym "Hobbicie" występowały np. "Goblins back door", co można przetłumaczyć jako "Tylne drzwi goblinów". Opisany algorytm nie daje możliwości wprowadzenia do gry takiego obiektu. Jednak można bez problemu dodać taką funkcjonalność. Należałoby w tym celu wprowadzić na modelu danych skojarzenie między zbiornikiem `drzwi` i `rzeczowniki` (a także między `postacie` a `rzeczowniki` i `przedmioty` a `rzeczowniki`), działające tak, jak obecnie istniejące skojarzenie między zbiornikiem `drzwi` a zbiornikiem `przymiotniki`. Na przykład, obiekt opisywany jako "Tylne drzwi goblinów" byłby skojarzony z rekordem w zbiorniku `rzeczowniki`, zawierającym rzeczownik "goblin" (oraz z rekordem w zbiorniku `przymiotniki`, zawierającym przymiotnik "tylne"). Następnie należałoby dodać do struktur, odpowiadających miejscom składniowym, w elemencie listy wariantów rozbioru, dodatkowe pole np. `rzprzyd`, zawierające listę odwołań do rekordów ze zbiornika `rzeczowniki`, oraz nowy sposób identyfikacji słowa — wszystko to analogicznie do tego, jak obecnie działa opisywanie obiektów przymiotnikami. Jedyna różnica byłaby taka, że podczas czyszczenia listy wariantów musielibyśmy usuwać te warianty, które dla jakiegoś miejsca składniowego miałyby puste pole `rz` a niepuste `rzprzyd` — bo nie można napisać np. "Otwórz goblinów" zamiast "Otwórz drzwi goblinów" (podczas gdy może być poprawne — o ile nie prowadzi do niejednoznaczności — "Otwórz zielone"

zamiast “Otwórz zielone drzwi”).

W podobny sposób można by dodawać inne rozszerzenia, np. umożliwić wprowadzenie obiektu “drzwi w podłodze” (ang. “trap door”).

Istotnie trudniejsze wydaje się dodanie obsługi spójnika “i”. Tutaj prawdopodobnie dobrym punktem wyjścia byłoby zauważenie, że “i” zawsze łączy dwa konkretne wyrazy znajdujące się po różnych jego stronach (np. w zdaniu “Weź jabłko i zjedz” wyrazy “weź” i “zjedz”; w zdaniu “Weź miecz i linę” wyrazy “miecz” i “linę”), oraz, że brakujące części zdania można uzupełniać wyrazami z przeciwnej strony “i”, przetwarzając jedno zdanie z “i” na dwa zdania połączone kropką (np. “Weź jabłko i zjedz” → “Weź jabłko. Zjedz jabłko”; “Weź miecz i linę” → “Weź miecz. Weź linę”). Oczywiście, należałoby radzić sobie z niejednoznacznościami (budując zapewne jakieś listy wariantów).

Kolejnym krokiem po dodaniu obsługi “i” mogłoby być dodanie zaimków (“Przeczytaj mapę i daj *ją* Gandalfowi”). Zapewne w tym celu należałoby pamiętać, jakie obiekty wypełniały miejsca składniowe w poprzednich poleceniach.

Rozdział 3

Szczegóły implementacji

Opisany w poprzednim rozdziale algorytm został w ramach pracy magisterskiej zaimplementowany w języku Common Lisp [9]. Użyta implementacja Common Lispa to CLISP [10]. Program był rozwijany i działa w systemie operacyjnym GNU/Linux.

Większość użytych struktur danych jest budowana w oparciu o *property lists*. W szczególności dotyczy to wszystkich, z wyjątkiem elementów zbiornika `warunki`, obiektów opisanych w rozdziale 2.2, a także opisanych w rozdziałach 2.3 i 2.4 struktur `gramtab` i `acttab`. Lispowe makra są używane bardzo sporadycznie. Do sygnalizacji błędów podczas rozbioru gramatycznego wykorzystano mechanizm lispowych *conditions*.

Manipulacje symboliczne są wykonywane na ogół na symbolach z pakietu `KEYWORD`. Dotyczy to m.in. przypadków (`:mian`, `:dop`, `:cel`, ...) i miejsc składniowych (`:obj`, `:util`, `:komu`, ...).

Program operuje na obiektach, odzwierciedlających struktury gramatyczne języka polskiego, które często nie mają nazw w języku angielskim. Zresztą, ze względu na jego przeznaczenie, raczej nie byłby interesujący dla osób niepolskojęzycznych. Z tych powodów nie uznałem za celowe narzucania sobie dyscypliny stosowania wyłącznie angielskich nazw zmiennych i komentarzy.

3.1. Pliki źródłowe

Omówimy teraz pokrótce pliki źródłowe.

3.1.1. constants.lisp

W tym pliku zdefiniowane są różne parametry konfiguracyjne, niezależne od konkretnej gry. Na ogół są to różnego rodzaju listy, np. *property list* `*kierunki*` z nazwami kierunków, `*you*` i `*mowca*` z odmianą przez przypadki słów "ty" i "ja", odpowiednio, czy *association list* `*kierskr*`, zawierająca skrócone nazwy kierunków ("PN", "WSCH", "PD-ZACH" itd.). Tu także zdefiniowane są, za pomocą zmiennych `*ms-przyp*` i `*ms-przim*`, miejsca składniowe. `*ms-przyp*` to *property list*, przypisująca każdemu miejscu składniowemu odpowiedni przypadek. `*ms-przim*` przypisuje odpowiednie przyimki tym miejscom składniowym, które tego wymagają. Oto przykładowa zawartość tych zmiennych:

```
(defparameter *ms-przyp*
  '(:obj :bier
    :util :narz
    :komu :cel
```

```
    :docel :dop
    :zczego :dop
    :naco :bier
    :przezco :bier))

(defparameter *ms-przim*
  '(:docel "do"
    :zczego "z"
    :naco "na"
    :przezco "przez"))
```

Ponadto plik `constants.lisp` udostępnia funkcje, ułatwiające korzystanie ze zdefiniowanych w nim parametrów. Przede wszystkim jest to funkcja `get-const`, przyjmująca dwa argumenty, z których pierwszy to *property list*, a drugi to klucz (*keyword*) i zwracająca wartość, przechowywaną we wskazanej liście pod wskazanym kluczem.

3.1.2. persistent-data.lisp

Plik odpowiedzialny za wczytanie, przechowywanie i udostępnianie danych omówionych w rozdziale 2.2. Wczytanie wykonywane jest przez funkcję `read-data`. Funkcja przyjmuje jeden argument — jest to lista nazw plików do wczytania (bez sufiksów, np. jeśli plik nazywa się `dzialania.dane`, to elementem listy będzie napis "dzialania". Lista ta przygotowana jest w zmiennej `*types*`. Wszystkie pliki z danymi mają ten sam format (zostanie on omówiony później), jednak można je podzielić na dwie kategorie: zwyczajne dane i skojarzenia. Zwyczajne dane odpowiadają omówionym w rozdziale 2.2 zbiornikom (prostokąty na rysunku), zaś skojarzenia związkom z gatunku wiele-do-wielu między tymi zbiornikami (na rysunku linie bez strzałek). Każdy rekord zwyczajnych danych posiada, identyfikującą go, etykietę (odczytaną z pliku), oraz liczbowy identyfikator `id` (nadany przez funkcję `read-data`). `read-data` tworzy tablicę haszującą `*labels*`, przypisującą etykietom odpowiadające im identyfikatory. Zgodnie z uwagą z rozdziału 2.2, w przypadku rzeczowników i przymiotników etykieta i identyfikator są wspólne dla rekordów, różniących się jedynie rodzajem.

`read-data` umieszcza zwyczajne dane w tablicy `*index-id*`. Każdy element tej tablicy to *cons cell*, której `cdr` zawiera listę rekordów o `id` równym indeksowi elementu w tablicy, zaś `car` zawiera *keyword*, określający zbiornik, z którego te rekordy pochodzą (np. `:przedmioty`, `:dzialania`, `:rzeczowniki`, ...). Pojedynczy rekord to *property list*. `id` również należy do rekordu, czyli może być odczytany przez

```
(getf rekord :id)
```

Związki, zaznaczone na rysunku w rozdziale 2.2 liniami zakończonymi strzałką, są zapisane w ten sposób, że odpowiednie pole ma wartość równą `id` wskazywanego rekordu.

Zatem przykładowe elementy tablicy `*index-id*` mogłyby wyglądać tak:

```
;; element piąty
  (:PRZYMIOTNIKI
   (:ID 5 :MIEJSC "ciężkim" :NARZ "ciężkim"
    :BIER "ciężki" :CEL "ciężkiemu" :DOP "ciężkiego"
    :MIAN "ciężki" :LICZBA :POJ :RODZ :M)
   (:ID 5 :MIEJSC "ciężkiej" :NARZ "ciężka"
    :BIER "ciężka" :CEL "ciężkiej" :DOP "ciężkiej"
    :MIAN "ciężka" :LICZBA :POJ :RODZ :Z)
```

```
   (:ID 5 :MIEJSC "ciężkim" :NARZ "ciężkim"
    :BIER "ciężkie" :CEL "ciężkiemu" :DOP "ciężkiego"
    :MIAN "ciężkie" :LICZBA :POJ :RODZ :N)
   (:MIEJSC "ciężkich" :NARZ "ciężkimi"
    :BIER "ciężkie" :CEL "ciężkim" :DOP "ciężkich"
    :MIAN "ciężkie" :LICZBA :MN :RODZ :NMOS :ID 5))
;; element 66-ty
  (:POSTACIE
   (:FUNKCJA "ftroll" :CIEZAR 300 :SPR_BOJ 1000
    :SILA 1000 :NAZWA 166
    :GDZIE_LOK 83 :ZYWY T :ID 66))
;; element 166-ty
  (:RZECZOWNIKI (:ID 166 :BIER "trollów" :RODZ :MOS)
   (:ID 166 :MIEJSC "trollach" :NARZ "trollami"
    :BIER "trolle" :CEL "trollom" :DOP "trolli"
    :MIAN "trolle" :LICZBA :MN :RODZ :NMOS)
   (:MIEJSC "trollu" :NARZ "trollem" :BIER "trolla"
    :CEL "trollowi" :DOP "trolla" :MIAN "troll"
    :LICZBA :POJ :RODZ :M :ID 166))
```

Wersja męskoosobowa rekordu, opisującego rzeczownik "troll" ma wypełnione jedynie pole `:bier` — można prześledzić, że opisany w rozdziałach 2.3 i 2.4 algorytm rozbioru i interpretacji polecenia będzie działał dobrze również z takimi niekompletnymi rekordami.

Aby dodatkowo ułatwić korzystanie z danych w zbiornikach `rzeczowniki`, `przymiotniki` i `czasowniki`, funkcja `read-data` buduje tablicę haszującą `*dictionary*`; klucze to słowa, a wartości to wszystkie możliwe interpretacje danego klucza. Ściślej, wartość to *property list*, przypisująca zbiornikom (`:przymiotniki`, `:rzeczowniki` lub `:czasowniki`) listy sposobów rozpoznania wyrazu jako elementu tego zbiornika. Element takiej listy sposobów jest to *cons cell*, w której `cdr` to *property list*, zawierająca pola, identyfikujące element zbiornika (`:id` i `:rodz` w przypadku rzeczowników i przymiotników, samo `:id` w przypadku czasowników), zaś `car` zawiera listę pól wyznaczanego przez `cdr` rekordu, identycznych ze słowem, które jest kluczem w haszu `*dictionary*`.

Przykładowo, dla zawartości `*index-id*` jak w poprzednim przykładzie, wartość przechowywana w haszu `*dictionary*` pod kluczem "troll" mogłaby wyglądać tak:

```
(:RZECZOWNIKI (((:MIAN) :ID 166 :RODZ :M)))
```

Pod kluczem "ciężkiej":

```
(:PRZYMIOTNIKI (((:DOP :CEL :MIEJSC) :ID 5 :RODZ :Z)))
```

Pod kluczem "goblinów" (zakładając, że `id` odpowiednich rekordów to 189):

```
(:RZECZOWNIKI (((:DOP :BIER) :ID 189 :RODZ :MOS)
               ((:DOP) :ID 189 :RODZ :NMOS)))
```

Pod kluczem "śmieć" (zakładając, ze odpowiednie `id` to 209 i 173):

```
(:CZASOWNIKI (((:ROZKAZ) :ID 209))
 :RZECZOWNIKI (((:MIAN :BIER) :ID 173 :RODZ :M)))
```

Hasz `*dictionary*` jest tworzony za pomocą `(make-hash-table :test #'equalp)` — dzięki temu nie ma znaczenia wielkość liter we wpisywanych przez gracza poleceniach.

Skojarzenia są dwóch rodzajów — skojarzenia między obiektami (tzn. drzwiami, przedmiotami lub postaciami) a przymiotnikami, oraz skojarzenia między czasownikami a działaniami. Te pierwsze funkcja `read-data` umieszcza w tablicy `*obiekty->przymiotniki*`, drugie w tablicy `*czasowniki->dzialania*`. Obie tablice zbudowane są tak samo: w komórce o indeksie równym `id` obiektu/czasownika znajduje się lista `id`-ów przymiotników/działań z danym obiektem/czasownikiem skojarzonych.

Oprócz funkcji `read-data`, wczytującej dane, plik `persistent-data.lisp` definiuje szereg funkcji/makr dostępowych. Oto najważniejsze z nich:

find-object (id &optional rodz) Zwraca obiekt (czyli *property list*) wyznaczony przez argumenty; argument `rodz` należy przekazać tylko gdy szukamy obiektu typu `rzeczowniki` lub `przymiotniki`.

zob (pole id &optional rodz) Zwraca wartość pola `pole` w obiekcie, wyznaczonym przez argumenty `id` i (w przypadku rzeczowników i przymiotników) `rodz`, lub `nil` jeśli takiego pola w obiekcie nie ma. `zob` jest `setf`-owalne, tzn. można przez `(setf (zob ...) ...)` zmieniać wartość istniejącego pola lub dodać nowe.

przymiotniki-opisujace (id) Zwraca element tablicy `*obiekty->przymiotniki*` o indeksie równym przekazanemu argumentowi. Jest `setf`-owalna; można `setf`-ować tylko elementy tablicy, które istnieją (inaczej mówiąc, odpowiadające istniejącym obiektom).

dzialania-opisywane (idcz &optional czycytat) Zwraca element tablicy `*czasowniki->dzialania*` o indeksie równym przekazanemu argumentowi; jeśli przekazany został argument `czycytat`, to wcześniej odrzuca ze zwracanej listy `id`-y tych działań, których pole `:czycytat` jest niezgodne z wartością argumentu.

dictionary (slowo) Zwraca element hasza `*dictionary*`, znajdujący się pod przekazanym kluczem.

get-label-id (label) Zwraca `id`, odpowiadający przekazanej etykiecie.

przyp (przypadek id &key bez-przymiotnikow poczatek-zdania) Zwraca opis obiektu o identyfikatorze `id` w odpowiednim przypadku (argument `przypadek` musi to być *keyword*, odpowiadający jednemu ze zdefiniowanych w pliku `constants.lisp` przypadków, tzn. `:mian`, `:dop`, `:cel`, `:bier`, ...). Jeśli przekazany zostanie argument `bez-przymiotnikow` o wartości różnej od `nil`, to opis składa się z samego rzeczownika. Jeśli przekazany zostanie argument `poczatek-zdania` o wartości różnej od `nil`, to opis będzie zaczynać się od wielkiej litery. Np.:
`(przyp :mian 48)` → "krótki solidny miecz"
`(przyp :miejsc 48)` → "krótkim solidnym mieczu"
`(przyp :miejsc 48 :poczatek-zdania t)` → "Krótkim solidnym mieczu"
`(przyp :miejsc 48 :poczatek-zdania t :bez-przymiotnikow t)` → "Mieczu"
Dla wygody plik definiuje też osobną funkcję dla każdego przypadku:
`(mian 48)` → "krótki solidny miecz"
`(miejsc 48 :poczatek-zdania t)` → "Krótkim solidnym mieczu"

gdzie (id &key stop) Znajduje miejsce pobytu danego obiektu (`id` musi być identyfikatorem postaci, przedmiotu lub gracza) w sensie opisanym w rozdziale 2.4, tzn. przechodzi zgodnie ze wskazaniami pól `gdzie_lok`/`gdzie_prz`/`gdzie_osoba` i zatrzymuje się, gdy

dojdzie do lokacji lub do zamkniętego przedmiotu. Jeśli przekazany jest argument `stop` to może też zatrzymać się wcześniej; `stop` może być

- identyfikatorem osoby lub przedmiotu — wtedy funkcja zatrzyma się, jeśli natrafi na obiekt o takim identyfikatorze i zwróci ten identyfikator;
- równy `:prz` lub `:przedmiot` — wtedy funkcja zatrzyma się, jeśli natrafi na jakikolwiek przedmiot i zwróci `id` tego przedmiotu;
- równy `:os` lub `:osoba` — wtedy funkcja zatrzyma się, jeśli natrafi na osobę, czyli postać lub gracza i zwróci `id` tej postaci lub gracza.

Jeśli nie było argumentu `stop` lub określony przez niego obiekt/rodzaj obiektu nie został napotkany, to funkcja zwraca `id` lokacji lub zamkniętego przedmiotu.

`gdzie` jest `setf`-owalne.

3.1.3. describe.lisp

Plik dostarczający mechanizm generowania — na podstawie reprezentacji świata gry opisanej w poprzednim rozdziale (oraz w rozdziale 2.2) — opisów w języku naturalnym, pokazywanych graczowi. Dzieje się to w dwóch etapach: najpierw tworzona jest zawierająca opis lista, złożona z napisów i zagnieżdżonych list, a następnie trafia ona do funkcji `print-description`, która przerabia ją na gotowy do wypisania tekst.

Zaczniemy od omówienia funkcji `print-description`, a następnie przejdziemy do właściwego generowania opisu (czyli skąd się bierze dla niej argument).

`print-description` przyjmuje jako argument listę, której elementami mogą być napisy lub listy; przy czym taka wewnętrzna lista jest zbudowana według dokładnie tych samych zasad co zewnętrzna. Na przykład:

```
("Jesteś w komfortowym hallu, przypominającym tunel."
 "Na wschodzie znajdują się okrągłe zielone drzwi."
 "Widzisz:"
 ("drewnianą skrzynię"
  "Gandalfa. Gandalf trzyma"
  ("interesującą mapę")
  "Thorina"))
```

Funkcja wypisuje elementy będące napisami, oddzielając je znakiem nowej linii. Zagnieżdżone listy wypisywane są rekurencyjnie, z większym wcięciem przed każdą linijką wyniku. Szerokość wcięcia zależy od globalnego parametru `*indent-step*`. Jest to dwuelementowa tablica; pierwszy element ustala wcięcie list na pierwszym poziomie zagnieżdżenia, drugi element ustala wcięcie na wszystkich kolejnych poziomach. Najbardziej zewnętrzna lista jest wypisywana bez wcięć. Drugi parametr wpływający na wygląd wyniku to `*line-width*`. Decyduje on, w którym momencie linie będą łamane. Jeśli (wzorując się na oryginalnym "Hobbicie") ustalimy parametry na

```
(defparameter *line-width* 42)
(defparameter *indent-step* #(4 2))
```

wtedy przytoczony wcześniej przykładowy argument da w wyniku:

```
Jesteś w komfortowym hallu,
przypominającym tunel.
Na wschodzie znajdują się okrągłe zielone
drzwi.
Widzisz:
    drewnianą skrzynię
    Gandalfa. Gandalf trzyma
      interesującą mapę
    Thorina
```

Każdy wypisywany tekst, należący do normalnego przebiegu gry (komunikatów o błędach to nie dotyczy), przechodzi przez funkcję `print-description`. Może to być opis wyprodukowany przez jedną z funkcji w pliku `describe.lisp`, ale równie dobrze może pochodzić z funkcji odpowiedzialnych za wykonanie działania, zachowanie postaci niegrających itp. Często wejściem do `print-description` będzie po prostu jednoelementowa lista zawierająca napis w rodzaju "Otwierasz skrzynię" albo "Thorin idzie na północ".

Oprócz `print-description` w pliku `describe.lisp` zdefiniowane są funkcje: `desc-location`, tworząca opis lokacji, w której znajduje się gracz, `opisz-przedmiot`, tworząca opis przedmiotu, w którym znajduje się gracz, oraz kilka funkcji pomocniczych dla tych dwóch. Spośród pomocniczych funkcji warto wymienić `przedmiot-info` i `postac-info`, które są tak skonstruowane, aby dało się ich używać też w innych okolicznościach (na przykład `przedmiot-info` jest wykorzystana w funkcji odpowiedzialnej za wykonanie działania otwarcia przedmiotu).

Omówmy pokrótce działanie tych funkcji.

desc-location (lokid &key short) Funkcja produkuje opis lokacji, taki mniej więcej jak w przytoczonym przykładzie. Składniki wyniku to:

- Pole `dlugiop` (jeśli nie przekazano różnego od `nil` argumentu `short`), lub `krotkiop` (w przeciwnym razie), reprezentującego lokację obiektu — czyli, używając omówionej w poprzednim podrozdziale funkcji dostępowej, `(zob :dlugiop lokid)` lub `(zob :krotkiop lokid)`.
- Jeśli nie było różnego od `nil` argumentu `short`, po jednej linijce na każde widoczne drzwi (np. "Na północy znajdują się ciężkie kamienne drzwi").
- Jedna linijka wyliczająca kierunki, w które można się udać, nie licząc przechodzenia przez drzwi. Występuje tylko, jeśli są takie kierunki. Np. "Widoczne wyjścia: północ, wschód, północny wschód."
- Linijka "Widzisz:", a następnie zagnieżdżona lista, zawierająca opisy przedmiotów i postaci, wyprodukowane przez funkcje `przedmiot-info` i `postac-info`; jeśli w danej lokacji nie ma żadnych przedmiotów ani postaci, to jedynym elementem zagnieżdżonej listy jest napis "Nic interesującego."

opisz-przedmiot (przid) Produkuje opis przedmiotu; funkcja podobna do poprzedniej, tylko prostsza, gdyż odpadają drzwi oraz widoczne wyjścia, a także krótka/długa wersja. Składniki wyniku to:

- Jedna linijka "Jesteś w ...", np. "Jesteś w skrzyni.", "Jesteś w beczce."
- Linijka "Widzisz:", a po niej zagnieżdżona lista, taka jak w funkcji `desc-location`.

przedmiot-info (przid &key przyp) Funkcja produkująca opis pojedynczego przedmiotu. Jeśli przekazany został argument `przyp` — który musi być jednym ze zdefiniowanych w pliku `constants.lisp` przypadków (`:mian`, `:dop`, `:cel`, ...) — wtedy opis zaczyna się od pełnej nazwy przedmiotu w takim przypadku. Jeśli przedmiot nie jest pusty, to bezpośrednio po nazwie, w tej samej linijce (ściślej: w tym samym elemencie zwracanej listy), następuje zdanie "W ... znajduje się:" i potem zagnieżdżona lista, zawierająca opisy wszystkich przedmiotów i postaci, przebywających we wnętrzu przedmiotu o identyfikatorze `przid`; opisy są wygenerowane przez rekurencyjne wołanie funkcji `przedmiot-info` oraz `postac-info`, z argumentem `przyp` równym `:mian` ("W skrzyni znajduje się ..." *kto? co?*).

postac-info (osid &key przyp) Funkcja podobna do poprzedniej; produkuje opis postaci. Opcjonalny argument `przyp` ma ten sam skutek co w `przedmiot-info`. Zasadniczy opis to zdanie "... niesie" i potem zagnieżdżona lista opisów przedmiotów i postaci, niesionych przez postać o identyfikatorze `osid`; opisy generowane są przez rekurencyjne wołanie funkcji `przedmiot-info` i `postac-info`, z argumentem `przyp` równym `:bier`.

Jeśli `osid` jest równy identyfikatorowi gracza, to funkcja wypisuje tylko wyraz "ty" w przypadku `przyp`, pomija natomiast listę niesionych przez gracza przedmiotów i postaci.

3.1.4. dzialania.lisp

Plik definiuje funkcje odpowiedzialne za wykonywanie działań, przedsiębranych zarówno przez gracza, jak i przez inne postacie, i za przygotowywanie komunikatów o tych działaniach informujących (w formie oczekiwanej przez, opisaną w poprzednim podrozdziale, funkcję `print-description`). Dla każdego rekordu z opisanego w rozdziale 2.2 zbiornika `dzialania` w pliku `dzialania.lisp` muszą istnieć dwie funkcje: `funkcja-ok` i `funkcja`, gdzie za `funkcja` należy postawić zawartość pola `funkcja` w rekordzie. Jeśli na przykład wyrażenie `(zob :funkcja 222)` daje w wyniku napis "otwdrzwi", to w pliku `dzialania.lisp` muszą istnieć funkcje `otwdrzwi-ok` oraz `otwdrzwi` (nic nie stoi natomiast na przeszkodzie, aby kilka działań miało taką samą wartość pola `funkcja`, wtedy te działania będą obsługiwane przez te same funkcje z pliku `dzialania.lisp`).

`funkcja-ok` sprawdza, czy działanie może być wykonane przez daną postać. Jeśli tak, zwraca `nil`. W przeciwnym razie zwraca komunikat o błędzie. Funkcje te są wołane przez fragment algorytmu opisanego w rozdziale 2.5.1. Tak naprawdę, cała zawartość zbiornika `warunki` to fragmenty kodu takich funkcji — jest to jedyny z przedstawionych w rozdziale 2.2 zbiorników, któremu nie odpowiadają pewne *property lists* osiągalne za pomocą funkcji `find-object`.

`funkcja` (bez `-ok`) jest odpowiedzialna za faktyczne wykonanie działania (co sprowadza się do pewnych manipulacji na danych obsługiwanych przez funkcje z pliku `persistent-data.lisp`), oraz za wyprodukowanie opisu do zaprezentowania graczowi. Treść opisu zależy od tego, czy działanie jest wykonywane przez gracza, czy przez inną postać, a także od tego, czy gracz i postać wykonująca działanie znajdują się w tym samym miejscu. Jeśli ich miejsca pobytu są różne, to na ogół opis będzie pusty, ale nie zawsze — np. jeśli gracz znajduje się po przeciwnej stronie otwieranych drzwi, to zobaczy komunikat w rodzaju "Ktoś otwiera okrągłe zielone drzwi."

Obie funkcje, z i bez `-ok`, przyjmują takie same argumenty. Pierwszy to `id` osoby, która chce wykonać działanie. Potem następują argumenty typu *keyword*, których liczba i nazwy zależą od zawartości rekordu ze zbiornika `dzialania`. Jak to zostało opisane w rozdziale

2.5.1, gdy działanie oczekuje argumentu, pochodzącego od miejsca składniowego `ms`, wtedy ma w polu o nazwie `ms` typ tego argumentu (`:przedmioty`, `:postacie` lub `:drzwi`). Otóż w takiej sytuacji musi mieć jeszcze pole o nazwie `ms->`, zawierające nazwę argumentu obsługujących to działanie funkcji. Na przykład otwieranie drzwi kluczem mogłoby być reprezentowane przez następujący rekord:

```
(:id 225 :funkcja "otwdrzkl"
 :util :przedmioty :util-> :klucz
 :obj :drzwi :obj-> :drzwi)
```

Oznaczałoby to, że w pliku `dzialania.lisp` muszą istnieć funkcje `otwdrzkl` oraz `otwdrzkl-ok`, których definicje wyglądają tak:

```
(defun otwdrzkl-ok (osid &key drzwi klucz)
  ...)

(defun otwdrzkl (osid &key drzwi klucz)
  ...)
```

Argumenty `drzwi` i `klucz` muszą to być `id`-y obiektów typu, odpowiednio, `drzwi` i `przedmioty`. Funkcja `otwdrzkl-ok` mogłaby sprawdzać np. czy drzwi są rzeczywiście zamknięte na klucz, czy ten klucz do nich pasuje itd. i zwracać odpowiedni komunikat o błędzie (np. "Drzwi nie są zamknięte na klucz"), lub `nil` jeśli wszystko OK. Funkcja `otwdrzkl` musi zmodyfikować stan świata gry — w tym wypadku zapewne przez instrukcję w rodzaju

```
(setf (zob :otwarte drzwi) t (zob :locked drzwi) nil)
```

— a następnie zwrócić odpowiedni opis. Jeśli `osid` był równy `id` gracza, wtedy może to być np. `("Otwierasz ciężkie kamienne drzwi ogromnym kluczem.")`. Jeśli nie, może to być np. `("Thorin otwiera ciężkie kamienne drzwi ogromnym kluczem.")` lub `("Ktoś otwiera ciężkie kamienne drzwi.")`, lub lista pusta.

Pola ze strzałką (`:obj->`, `:util->`, ...) nie zostały umieszczone na rysunku w rozdziale 2.2, gdyż użyty mechanizm jest wybitnie zależny od implementacji — w szczególności, nie da się tego zrobić w opisany sposób w języku bez parametrów typu *keyword*.

Oprócz funkcji wymaganych przez rekordy ze zbiornika `dzialania`, plik `dzialania.lisp` dostarcza dwie funkcje:

```
go-direction-ok (osid &key direction)
go-direction (osid &key direction)
```

które obsługują przemieszczanie się i działają podobnie jak inne funkcje z tego pliku. `osid` to `id` osoby która chce iść w kierunku `direction`. `direction` to `:N`, `:S`, `:W`, `:E`, ...

3.1.5. gramtab.lisp

Ten plik udostępnia funkcję `rozbior`, która dość dokładnie realizuje algorytm opisany w rozdziale 2.3.3. `gramtab` to w istocie *cons cell*, w `car` której znajduje się lista wariantów rozbioru, zaś w `cdr` *property list*, która jest dość zdegenerowana i ma tylko jedno pole `:cytat`. Pole błędu zniknęło, gdyż błędy są sygnalizowane za pomocą zdefiniowanego w tym celu *condition*:

```
(define-condition nie-rozumiem (error)
  ((opis :initarg :opis :reader opis)))
```

Pojedynczy element listy wariantów jest to *property list*, wiernie odzwierciedlająca opisaną w rozdziale 2.3.1 strukturę.

Zatem np. `gramtab` powstała w wyniku rozbioru zdania "Otwórz okrągłe, zielone drzwi" mogłaby mieć postać:

```
(((:OBJ (:RODZ :NMOS
         :RZ ((:ID 163 :RODZ :NMOS))
         :PRZ ((:ID 2 :RODZ :NMOS) (:ID 3 :RODZ :NMOS)))
   :ROZKAZ (198)))
 :CYTAT NIL)
```

"Otwórz wszystko oprócz drewnianej skrzyni":

```
(((:OBJ (:RODZ :N
         :OPR (:RODZ :Z
               :RZ ((:ID 168 :RODZ :Z))
               :PRZ ((:ID 7 :RODZ :Z)))
         :ALL (:N))
   :ROZKAZ (198))
  (:KOMU (:RODZ :Z
          :RZ ((:ID 168 :RODZ :Z)))
          :OBJ (:RODZ :N
                :OPR (:RODZ :Z
                      :PRZ ((:ID 7 :RODZ :Z)))
                :ALL (:N)) :ROZKAZ (198)))
 :CYTAT NIL)
```

(Zakładając, że występujące liczby odpowiadają identyfikatorom odpowiednich rekordów ze zbiorników `czasowniki`, `rzeczowniki` i `przymiotniki`).

3.1.6. acttab.lisp

Plik udostępnia funkcję `interp`, która, według opisanego w rozdziale 2.4.3 algorytmu, wykonuje interpretację struktury `gramtab`. Pewne uproszczenie polega na tym, że `interp` zwraca pojedynczą strukturę `acttab`, która uwzględnia już wyeliminowanie pewnych wariantów przez zawartość pól `opr` (podział wyniku w rozdziale 2.4 na `acttab` i `donttab` służył głównie lepszemu wyeksponowaniu podstawowych koncepcji, oraz większej ilustracyjności przykładów; w praktyce nie jest on przydatny — choć, oczywiście, można by niewiele większym nakładem sił zaimplementować algorytm dokładnie taki jak opisany i działałby równie dobrze).

`acttab` jest zbudowana podobnie jak `gramtab`, tzn. jest to *cons cell*, której `car` to lista wariantów zaś `cdr` to *property list* z polami `:cytat` i `:jestall`. Pojedynczy element listy wariantów to *property list*, wiernie odzwierciedlająca strukturę opisaną w rozdziale 2.4.1.

Konstrukcja przy ustalonym wariancie rozbioru i działaniu jest wykonywana przez funkcję pomocniczą `interp-wr-dzial`.

3.1.7. game.lisp

Plik zawiera implementację algorytmu opisanego w podrozdziale 2.5. Część odpowiedzialna za wybór i wykonanie pojedynczego działania realizowana jest przez funkcję `wykonaj-gramtab`. Wykonanie pełnego polecenia realizuje funkcja `wykinstr`.

Ponadto `game.lisp` dostarcza funkcję `game-loop`, która zawiera główną pętlę gry.

3.1.8. npc.lisp

W tym pliku znajdują się funkcje, odpowiedzialne za akcje postaci innych niż gracz. Dla każdego rekordu w zbiorniku `postacie` musi w `npc.lisp` istnieć funkcja, o nazwie identycznej z zawartością pola `funkcja` tego rekordu. Funkcje te powinny dokonywać modyfikacji na danych, reprezentujących świat gry (najczęściej, ale nie obowiązkowo, przez odwołanie się do odpowiednich funkcji z pliku `dzialania.lisp`), oraz, jeśli działania miały skutek widoczny dla gracza, wypisać odpowiednią informację (poprzez zawołanie funkcji `print-description`, z argumentem otrzymanym jako wartość zwrócona przez funkcję z pliku `dzialania.lisp` lub wyprodukowanym ręcznie). Podobnie jak w przypadku działań, jedna funkcja może obsługiwać wiele rekordów.

Nic oczywiście nie stoi na przeszkodzie, aby funkcje z pliku `npc.lisp` posługiwały się dowolnie rozbudowanymi, wewnętrznymi trwałymi strukturami danych. Jednak na ogół trwałe dane są trzymane w globalnej zmiennej `*npc-data*`, co ułatwia komunikację między fragmentami kodu, odpowiedzialnymi za działania różnych postaci. `*npc-data*` jest tablicą haszującą, w której klucze to `id`-y postaci, a wartości to *property lists*. Wygodnym sposobem odwoływania się do tych danych jest `setf`-owalne makro dostępowe `npc-data`, które przyjmuje dwa argumenty — `id` postaci i *keyword*, będący kluczem w *property list*, zawierającej dane tej postaci.

Przykładem użycia `*npc-data*` jest obsługa rozmów pomiędzy postaciami. Jeśli gracz, lub dowolna inna postać, zwróci się do kogoś z prośbą/poleceniem (np. wykonując instrukcję ‘Powiedz Thorinowi “Otwórz okno”’), to na ogół — jeśli nie ma konkretnych przeciwwskazań — chcemy, aby polecenie to zostało wykonane. Funkcja `powiedz` (z pliku `dzialania.lisp`) ma działanie zależne od adresata: jeśli adresatem jest gracz, wtedy po prostu powoduje wypisanie komunikatu w rodzaju ‘Elrond mówi “Witaj”’, zawierającego pełną treść argumentu `cytat`; jeśli jednak adresatem jest dowolna inna postać, to funkcja, poza wypisaniem (jeśli rozmowa odbywa się w polu widzenia gracza) komunikatu (np. “Rozmawiasz z Gandalfem.”), wstawia treść i autora wypowiedzi do `*npc-data*` — ściśle mówiąc, umieszcza *cons cell*, której `car` to `id` mówcy, zaś `cdr` to treść wypowiedzi, na stosie, znajdującym się pod kluczem `:powiedziane` w *property list*, będącej wartością przechowywaną w `*npc-data*` pod kluczem równym `id` adresata wypowiedzi. Zdefiniowana w `npc.lisp` funkcja odpowiedzialna za zachowanie adresata wypowiedzi może teraz zrobić z tymi danymi cokolwiek — zignorować, bezwarunkowo wykonać polecenie, wykonać lub nie zależnie od treści, od autora wypowiedzi, itp.

Innym przykładem użycia `*npc-data*` może być zachowanie funkcji `attack`, która, oprócz ustalenia skutków walki i ewentualnie przedstawienia opisu jej przebiegu, wstawia do `*npc-data*`, do danych postaci zaatakowanej, informację, kto atakował. Dzięki temu zaatakowana postać może np. szukać odwetu.

3.1.9. zdarzenia.lisp

Ten plik odpowiedzialny jest za te zmiany świata gry, które nie wynikają bezpośrednio z działań żadnej postaci i dlatego nie mieszczą się w zakresie obowiązków pliku `npc.lisp` (przykładowo, nastanie poranka, które prowadzi do skamienienia trolli). Plik dostarcza funkcję `zdarzenia`, wołaną po zakończeniu każdej tury gry. Funkcja ta z kolei woła funkcje, odpowiedzialne za pojedyncze zdarzenia (ich nazwy są przechowywane w zmiennej `*zdarzenia*`) i wypisuje wyniki ich działania (które mają postać odpowiednią dla funkcji `print-description`).

3.2. Format plików z danymi

Wczytywane przez — opisaną w rozdziale 3.1.2 — funkcję `read-data` pliki zbudowane są z elementów, połączonych znakami `@`. Element może reprezentować jednostkę danych (pojedynczy rekord w przypadku plików ze zwykłymi danymi lub pojedyncze skojarzenie w przypadku plików ze skojarzeniami), lub dostarczać wartości domyślne dla dalszych elementów. Element reprezentujący pojedynczy rekord ma postać ciągu par *nazwa pola*: *wartość*, poprzedzonego etykietą. *wartość* może być

- tekstem w apostrofach; jeśli częścią tekstu jest znak \ (*backslash*) lub apostrof, to muszą one być poprzedzone znakiem \; tekst może rozciągać się na wiele linii
- tekstem bez apostrofów — jeśli tekst nie zawiera spacji ani znaków niealfabetycznych, apostrofy można pominąć
- liczbą
- specjalną wartością `T` lub `N`, oznaczającą odpowiednio prawdę i fałsz (jeśli chcemy umieścić wartość będącą tekstem o treści "T" lub "N", to musimy użyć formy z apostrofami: `'T'`, `'N'`; analogicznie z innymi specjalnymi wartościami)
- specjalną wartością `m`, `z`, `n`, `nmos`, `mos`, oznaczającą odpowiednio rodzaj męski, żeński, nijaki, niemęskoosobowy i męskoosobowy
- specjalną wartością `poj` lub `mn`, oznaczającą liczbę pojedynczą lub mnogą
- specjalną wartością `przedmioty`, `postacie` lub `drzwi`, oznaczającą typ (nazwę zbiornika)
- referencją do innego rekordu; wtedy ma postać `*`*etykieta*

Na przykład w zasilającym zbiornik `mapa` pliku `mapa.dane` mógłby się znaleźć element:

```
sciezka

krotkiop: 'Ścieżka trolli.'
dlugiop:
'Jesteś na ukrytej między drzewami ścieżce, na której widzisz ślady
 stóp trolli.'
Nd: *heavyrock
S: *polanka
```

(Przy założeniu, że pewien element w pliku `drzwi.dane` ma etykietę `heavyrock` zaś pewien inny element w pliku `mapa.dane` etykietę `polanka`). Między nazwę pola i dwukropek nie wolno wstawiać spacji.

W przypadku rekordów ze zbiornika `rzeczowniki` lub `przymiotniki`, opisujących ten sam wyraz w różnych rodzajach, tylko jeden z elementów powinien być poprzedzony etykietą; pozostałe muszą mieć dodatkowe pole `id`, zawierające referencję z tą etykietą. Podany w rozdziale 3.1.2 przykładowy fragment zawartości zbiornika `rzeczowniki` mógłby pochodzić z fragmentu pliku:

```
troll
```

```
rodz: m
liczba: poj
mian: troll
dop: trolla
cel: trollowi
bier: trolla
narz: trollem
miejsc: trollu

@

rodz: nmos
liczba: mn
mian: trolle
dop: trolli
cel: trollom
bier: trolle
narz: trollami
miejsc: trollach
id: *troll

@

rodz: mos
bier: trollów
id: *troll
```

Element dostarczający wartości domyślne zaczyna się od słowa `default`, po którym następuje ciąg par *nazwa pola*: *wartość*, interpretowanych tak samo jak w elementach reprezentujących rekord, zakończony słowem `enddefault`. Wartości podane jawnie nadpisują wartości domyślne. Na przykład:

```
@

default
nazwa: *goblin
spr_boj: 140
ciezar: 30
sila: 100
funkcja: fgoblin
goblin: T
enddefault

@

nastygoblin
gdzie_lok: *dsp1

@
```

```
hideousgoblin
gdzie_lok: *dsp2
sila: 200

@

horriblegoblin
gdzie_lok: *dsp3

@

...
```

Elementy reprezentujące skojarzenie wyglądają podobnie, jednak nie mają etykiet i muszą zawierać dokładnie dwie pary *nazwa pola*: *wartość*, przy czym *wartość* to referencje to kojarzonych rekordów, zaś *nazwa pola* zależy od zbiornika, do którego odnosi się referencja. Przykładowo, w pliku `sk_prz_os.dane` mógłby znajdować się fragment:

```
przym: *ohydny
osoba: *hideousgoblin

@

przym: *nikczemny
osoba: *meangoblin

@

przym: *obrzydliwy
osoba: *disgustinggoblin
```

(Zakładając, że w pliku `przymiotniki.dane` byłyby elementy o etykietach `ohydny`, `nikczemny` i `obrzydliwy`, a w pliku `postacie.dane` o etykietach `hideousgoblin`, `meangoblin` i `disgustinggoblin`).

Rozdział 4

Przykładowe zastosowanie: remake gry "Hobbit"

Aby zaprezentować możliwości stworzonego silnika, przygotowana została korzystająca z niego kompletna gra, o treści opartej na treści klasycznej gry "Hobbit". Założenie było takie, aby osoba znająca "Hobbita" z czasów ZX Spectrum mogła grać w *remake* odczuwając, poza zmianą języka, tak mało różnic w stosunku do oryginału jak to możliwe. Dla osób, które nie miały kontaktu z oryginalną grą, w dodatku A znajduje się krótka instrukcja.

Występujące w grze fragmenty dialogów, o ile były zaczerpnięte z książki, zostały przełożone w oparciu o tłumaczenie Marii Skibniewskiej [11].

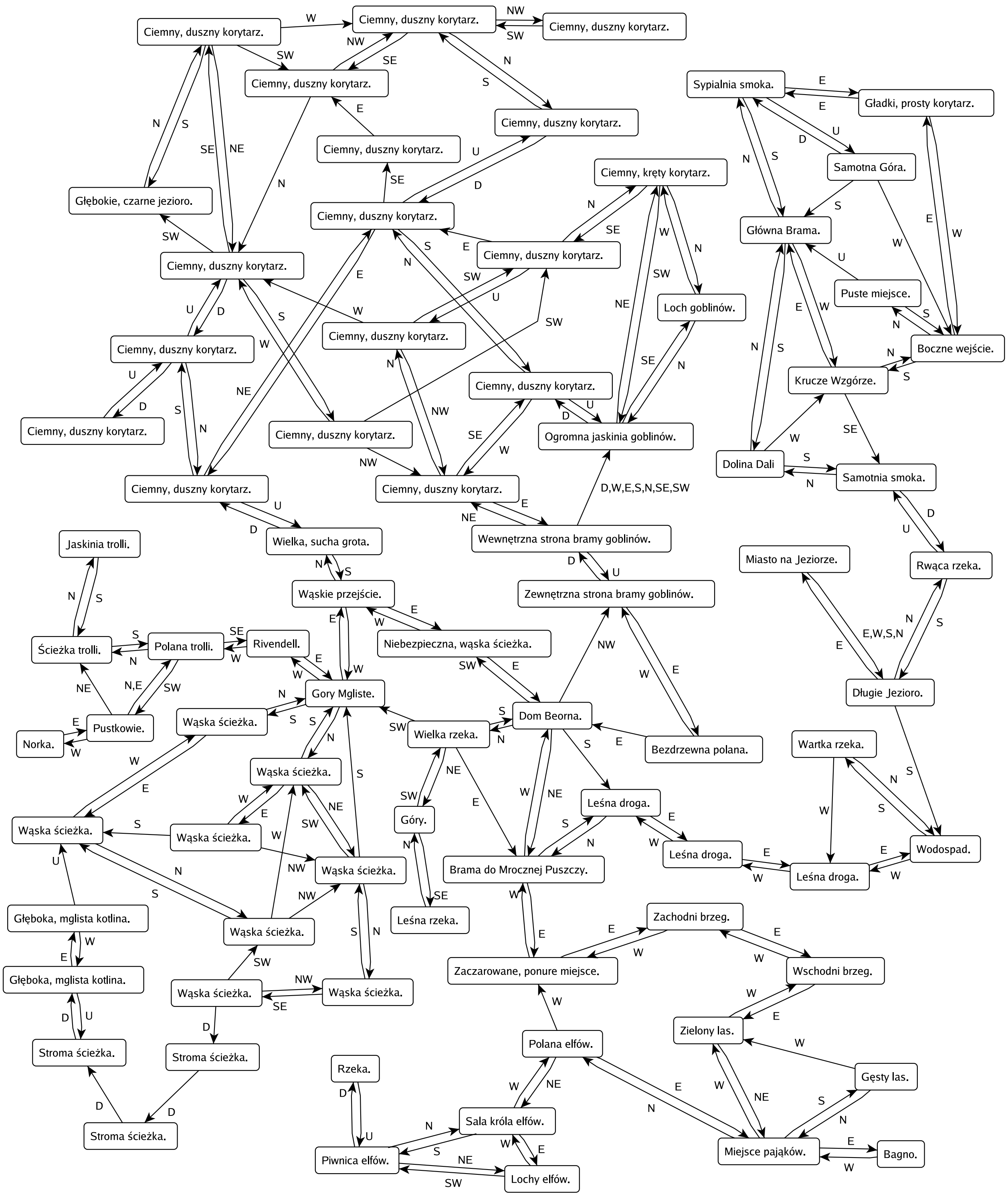
Od strony technicznej przygotowanie gry wymagało dwóch rzeczy: po pierwsze, dostarczenia odpowiednich plików z danymi; po drugie, napisania kodu, odpowiedzialnego za specyficzne dla gry działania, w pliku `dzialania.lisp`, za zachowanie postaci, w pliku `npc.lisp`, oraz za zdarzenia, w pliku `zdarzenia.lisp`. Pliki `npc.lisp` i `zdarzenia.lisp` są całkowicie zależne od konkretnej gry (choć najprostsze, generyczne zachowania z pliku `npc.lisp`, jak np. losowe błąkanie się po mapie i atakowanie kogo popadnie, w zasadzie mogłyby być ponownie wykorzystane w innej grze). Plik `dzialania.lisp` zawiera wszystkie działania; część z nich (poruszanie się, otwieranie i zamykanie drzwi i przedmiotów, podnoszenie i wyrzucanie przedmiotów, itd.) jest uniwersalna; inne są typowe dla konkretnej gry. Przykładem działania, mającego sens w jednej tylko grze, jest występujące w "Hobbicie" branie do niewoli. Gdy goblin bierze do niewoli (ang. *captures*) jakąś postać, wtedy jest ona przeniesiona do lochu goblinów. Może to zrobić także elf, wtedy postać zostaje przeniesiona do lochu elfów. Inne postacie nie mogą tego robić — w ich przypadku polecenie "Weź do niewoli ..." będzie równoważne "Zaatakuj ...". Kod odpowiedzialnej za to działanie funkcji `capture` zależy od istnienia odpowiednich typów postaci i odpowiednich lokacji (rozpoznawanych po etykietach), i raczej nie uda się go wykorzystać w jakiejkolwiek innej grze.

Lokacje i połączenia między nimi są dokładnie takie same jak w oryginale (rysunek 4.1). Występujące postacie odpowiadają postaciom z oryginalnej gry, a ich zachowanie zostało tak zaprogramowane, aby przypominać zachowanie wzorów z angielskiej wersji. Są to Thorin, Gandalf, Elrond, Bard, sześć goblinów, dwa trolle, leśny elf (*wood elf*), podczaszy (*butler*), Gollum, wredny warg (*vicious warg*), czerwonozłoty smok (*red golden dragon*).

Występujące przedmioty również odpowiadają przedmiotom występującym w oryginale. Z istotniejszych można wymienić skarb, którego zdobycie jest celem gry, oraz drewnianą skrzynię, do której trzeba włożyć skarb po powrocie do domu, aby gra została uznana za wygraną.

Z innych ciekawych faktów można dodać, że występujące w grze rzeki z punktu widzenia

silnika są nietypowym rodzajem drzwi.



Rysunek 4.1: Mapa gry "Hobbit"

Dodatek A

Instrukcja do gry "Hobbit"

W grze "Hobbit" wcielasz się w rolę hobbita, Bilbo Bagginsa.

Będziesz swobodnie poruszać się po Śródziemiu, odkrywać i badać ten wspaniały, zaczarowany świat. Spotkasz wiele różnych stworzeń; niektóre z nich będą przyjazne, inne znacznie mniej. Czeka Cię niebezpieczna i fascynująca przygoda.

Być może nie orientujesz się w sprawach związanych z hobbitami — są to małe istoty, wysokie mniej więcej na metr, mniejsze niż na przykład krasnoludy. Nie umieją też czarować. Bardziej kompletny opis znajdziesz w książce "Hobbit czyli tam i z powrotem", lecz to co powiedzieliśmy powinno wystarczyć do zrozumienia, że na ogół istoty które napotkasz podczas tej przygody, włącznie z krasnoludami, będą większe i silniejsze od Ciebie. Musisz więc wykorzystać cały swój spryt.

Gandalf, czarodziej, namówił Cię do wzięcia udziału w nowej i ekscytującej przygodzie; zamierzasz pomóc Thorinowi (który jest krasnoludem). Zadanie polega na odnalezieniu nikczemnego smoka, odebraniu mu zagrabionego skarbu i umieszczeniu go w Twojej skrzyni (gdzie będzie bezpieczny). Gra rozpoczyna się w momencie, gdy opuszczacie bezpieczną norkę w Shire i wyruszacie w drogę.

Grasz po prostu wydając komputerowi polecenia. Aby na przykład otworzyć skrzynię, napisz "Otwórz skrzynię". Komputer przeanalizuje polecenie i — jeśli to, czego żądasz, da się wykonać — zobaczysz komunikat w rodzaju "Otwierasz drewnianą skrzynię". Jeśli jednak coś stoi na przeszkodzie, zobaczysz odpowiednie wyjaśnienie — np. "Skrzynia jest zamknięta na klucz". Zobaczysz też opisy różnych zdarzeń i działań innych osób:

```
Otwierasz drewnianą skrzynię.
Thorin ogląda interesującą mapę.
```

Poruszasz się wydając polecenia takie jak "Idź na północ". Dla wygody możesz też użyć skrótu — "Pn". Gdy po raz pierwszy odwiedzisz jakieś miejsce, komputer przedstawi Ci jego dłuższy opis, za następnym razem opis skrócony. Aby znów zobaczyć dłuższy opis użyj polecenia "Patrz".

Inne ważne komendy to "Weź ...", "Wyrzuć ...", "Obejrzyj ...", "Daj ...", "Otwórz ...", "Zamknij ...", "Zabij ...", "Wyłam ...", "Wejdź do ...", "Wyjdź z ...".

Aby zobaczyć listę przedmiotów, które niesiesz, napisz "kieszeń". Aby zakończyć grę, napisz "koniec".

Podczas drogi może tak się zdarzyć, że nie będziesz w stanie poradzić sobie bez pomocy innych osób. Nic prostszego niż o nią poprosić: możesz napisać na przykład "Powiedz do Gandalfa "Przeczytaj mapę"". Oczywiście, może się zdarzyć i tak, że ktoś nie będzie chciał (lub nie będzie w stanie) zrobić tego, o co poprosisz.

Powodzenia!

Bibliografia

[1] http://www.inform-fiction.org/

[2] Hector Briceno, Wesley Chao, Andrew Glenn, Stanley Hu, Ashwin Krishnamurthy, Bruce Tsuchida *Down From the Top of Its Game — The Story of Infocom, Inc.*, http://mit.edu/6.933/www/Fall2000/infocom/

[3] Beam Software (Philip Mitchell, Veronika Megler et al) *The Hobbit* Melbourne House (1982)

[4] http://www.icemark.com/downloads/files/hobbitsrc.zip

[5] Andrzej Kadlof *Gry przygodowe po polsku* Komputer nr 1,2 (kwiecień, maj) 1986

[6] P. Kucharski, K. Piwowarczyk *Smok Wawelski* Komputer — Program komputerowy, nr 16, 1987

[7] http://akson.sgh.waw.pl/ jtryzn/arkadia/

[8] *Encyklopedia języka polskiego* Pod red. S. Urbańczyka i M. Kucały, Ossolineum 1999

[9] Guy Steele *Common LISP : The Language* Digital Press; 2 edition (June 15, 1984)

[10] http://clisp.cons.org/

[11] J.R.R. Tolkien *Hobbit czyli tam i z powrotem* tłumaczyła Maria Skibniewska, Państwowe Wydawnictwo "Iskry", Warszawa, 1960