Uniwersytet w Białymstoku

Wydział Matematyczno-Fizyczny

Instytut Matematyki

Aneta Staskielunas

Drzewo zależności referencji w tekście matematycznym napisanym w LaTeX-u

Praca dyplomowa napisana
pod kierunkiem
dr. Mariusza Żynela

Białystok 2006

Składam serdeczne podziękowania
dr. Mariuszowi Żynelowi oraz rodzicom
za pomoc i wsparcie w pisaniu pracy.

Aneta Staskielunas

Spis treści

Wstęp

W większych publikacjach naukowych, na przykład w monografiach, spotyka się graficznie przedstawione zależności pomiędzy poszczególnymi rozdziałami, co pomaga zorientować się w układzie treści i organizacji książki. Formalnie rzecz biorąc, mamy tu zazwyczaj do czynienia z grafem skierowanym.

W przypadku mniejszych prac matematycznych, interesujące są związki pomiędzy dowodzonymi twierdzeniami. Drzewo użytych referencji może mieć też znaczenie bardzo praktyczne w trakcie pracy nad publikacją. Mianowicie może ono pozwolić wyłapać błędne powiązania lub uzupełnić powiązania brakujące.

Tematej mojej pracy jest właśnie opracowanie programu, który na podstawie tekstu źródłowego pracy w LaTeX-u, rekonstruuje drzewo powołań w takiej formie, aby możliwa była prezentacja graficzna wyników. Samo zadanie mechanicznego wyrysowania drzewa referencji jest bardzo trudne i wykracza poza zakres tej pracy. Są jednak dostępne gotowe narzędzia, które to zadanie doskonale realizują, jak na przykład system Graphviz (por. [3]).

Jako język programowania wybrałam *Perl* z uwagi na jego moc przy przetwarzaniu różnego rodzaju tekstów. Wprawdzie nie ma w *Perlu* zbyt wielu narzędzi do obróbki tekstów w LaTeX-u, ale moduł LaTeX::TOM okazał się wystarczającym parserem, aby zrealizować postawione zadanie w dość efektywny sposób. Założenia do programu znajdują się w podrozdziale 3.1.1, natomiast kod skryptu, który nazwałam `ref2dot.pl`, zamieszczony jest w Dodatku B na stronie 26.

Rozdział 1

Referencje w LaTeX-u

1.1 LaTeX – system składu dokumentów

TeX jest systemem składu komputerowego, opracowanym w latach 1977-1982 przez wybitnego matematyka i informatyka, profesora Donalda E. Knutha. Pierwsze wydanie systemu pojawiło się w roku 1982 i, co ciekawe, niemalże nie zmieniło się po dzień dzisiejszy.

TeX (wym. tech) jest systemem profesjonalnego składu drukarskiego. Wśród systemów służących do podobnych celów wyróżnia się dbałością o jakość wyników. Nie ma sobie równych przy składaniu trudnych tekstów naukowych (szczególnie matematycznych), słowników, itp. Inną zaletą tego programu, istotną w środowisku akademickim, jest jego status oprogramowania *public domain*, co oznacza, że każdy może zostać jego legalnym użytkownikiem bez żadnych opłat licencyjnych.

System TeX, odpowiednio przystosowany do pracy w różnych językach, jest używany przez setki tysięcy użytkowników na całym świecie. TeX został zaprojektowany w taki sposób, że może być przystosowany do składania tekstów w dowolnych językach, nawet tak egzotycznych dla nas, użytkowników alfabetu łacińskiego, systemach, jak drukowanie z góry na dół (chińska albo japońska wersja językowa) czy z prawa na lewo (hebrajska i arabska). Na większości wyższych uczelni na świecie jest używany do składania tekstów naukowych. Wykorzystywany jest np. przez dwóch największych na świecie wydawców tekstów naukowych, tj. American Mathematical Society oraz Springer Verlag, a ponadto przez inne cenione wydawnictwa, takie jak: Oxford University Press, Addison-Wesley Publishing Group itd.

TeX umożliwia efektywne składanie tekstów o dowolnej trudności. Unikalny algorytm, którym posługuje się TeX przy składaniu akapitów, powoduje, że nie ma programu oferującego w tym względzie lepsze możliwości. TeX działa tak samo na wszystkich platformach. W rezultacie użytkownicy TeX-a na całym świecie mogą się porozumiewać (np. wymieniać dokumenty poprzez pocztę elektroniczną) bez względu na to, na jakim sprzęcie pracują. Wreszcie

TEX jest oprogramowaniem otwartym, przez co rozumieć należy jego zdolność do współpracy z innymi programami.

Bieżąca wersja TEX-a ma numer 3.141592, zaś najnowsza dystrybucja LATEX-a jest oznaczana jako LaTeX2E (LaTeX2epsilon). Jego twórca to Leslie Lamport. LATEX jest bardzo rozbudowanym zestawem makr, zawierającym wiele mechanizmów opisu struktury logicznej dokumentów (por. [1]). Zestaw taki, wraz z wzorcami dzielenia wyrazów, jest wstępnie prekompilowany i używany jako tzw. format. LATEX to obecnie najbardziej rozpowszechniony format i dostarczany w każdej dystrybucji stanowi tym samym ważny składnik systemu TEX. LATEX to system składu posiadający ogromne możliwości. Między innymi są to: tworzenie wszelkiego rodzaju dokumentów technicznych, naukowych, listów, reportaży czy nawet książek. Praktycznie nadaję się on do wszystkiego. Argumenty, które przemawiają za używaniem LATEX-a, to:

- dostępność gotowych, przygotowanych przez zawodowców układów graficznych, dzięki zastosowaniu których dokumenty wyglądają "jak z drukarni",
- wygodne składanie wzorów matematycznych,
- do rozpoczęcia pracy wystarczy znajomość zaledwie kilkunastu łatwych do zrozumienia instrukcji, określających strukturę logiczną dokumentu,
- nawet elementy takie jak przypisy, odnośniki, spisy treści, spisy tabel, skorowidze oraz spisy bibliograficzne przygotowuje się bardzo łatwo,
- istnieje wiele bezpłatnych pakietów poszerzających możliwości typograficzne LATEX-a,
- LATEX uczy tworzyć dokumenty o określonej strukturze. LATEX ułatwia skład tekstu, pozwalając autorowi skupić się na treści i strukturze tekstu.

W sposób automatyczny tworzone są:

- spis treści,
- numerowanie rozdziałów i podrozdziałów,
- spisy ilustracji i tabel,
- skorowidze,
- bibliografia,
- i wiele innych.

Jak wspomnieliśmy wcześniej LaTeX jest bardzo rozbudowanym zestawem instrukcji (poleceń, makrodefinicji, makr) zawierającym wiele mechanizmów opisu struktury logicznej dokumentów umożliwiającym autorowi złożenie oraz wydrukowanie ich prac na najwyższym poziomie typograficznym. Dokumenty LaTeX-a są plikami tekstowymi zawierającymi oprócz treści instrukcje formatujące. Pod tym względem jest to format bardzo przypominający dokumenty HTML. Możemy traktować LaTeX jako język programowania. Słowa poprzedzone znakiem `\ (backslash)`, mają specjalne znaczenie, są interpretowane przez program LaTeX-a, mają zatem charakter funkcji czy jak kto woli procedur. W kontekście LaTeX-a mówi się, że są to *makra*.

Istnieje wiele otoczeń pomagających autorowi w pisaniu pracy. Tak na przykład otoczenia *math*, *displaymath*, *equation* wprowadzają TeX-a w tryb matematyczny. W tym trybie TeX ignoruje spacje w pliku źródłowym (lecz znaki spacji mogą być potrzebne, by zaznaczyć koniec nazwy polecenia).

Wzór umieszczony w wierszu otaczającego tekstu, zwany *wzorem w tekście*, jest tworzony przez otoczenie *math*. Otoczenie to, oprócz zwykłej konstrukcji `\begin...\end`, można przywołać dwoma krótszymi sposobami`\(...\)` lub `$...$`. Otoczenie *displaymath*, które ma również krótką postać `\[...\]`, tworzy nienumerowany wzór wystawiony. Wersje `$...$` , `\(...\)`, `\[...\]` działają jako pełnoprawne otoczenia ograniczające zasięg deklaracji zawartych w ich wnętrzu. Numerowany wzór wystawiony można uzyskać za pomocą otoczenia *equation*.

Otoczenia *displaymath* i *equation* tworzą wzory jednowierszowe. Licznik *equation* generuje numer równania. Numer równania jest dosuwany do prawego marginesu, jeśli nie użyto opcji klasy dokumentu *leqno*. Układ oddzielonych wzorów lub wzór za długi wymagają podzielenia na więcej wierszy. Do składu wyrażeń wielowierszowych można zamiast środowiska *equation* użyć środowisk *eqnarray* lub *eqnarray**. W środowisku *eqnarray* każdy wiersz zawartego w nim wyrażenia posiada osobny numer. Otoczenie *eqnarray* przypomina bardziej *array*[1] dla trzech kolumn. Jest jednak pewna różnica. W każdym wierszu jest wstawiany numer równania, chyba, że umieszczono w nim polecenie `\nonumber`.

TeX zna około 300 instrukcji podstawowych (wbudowanych), tworzących jądro języka. Użytkownik posługuje się instrukcjami (makrodefinicjami, makrami) zdefiniowanymi za pomocą instrukcji wbudowanych. Zbiór takich instrukcji oraz wzorce dzielenia wyrazów dla różnych języków są czytane podczas uruchamiania TeX-a z parametrem *–ini* i, po przetworzeniu, zapisywane do pliku, który nazywamy `formatem`. Plik formatu ma zwyczajowe rozszerzenie *fmt*. Powszechnie używane formaty to: Plain, LaTeX, AMSTeX i ConTeXt. Istnieje oczywiście więcej formatów i użytkownik ma pełną swobodę tworzenia własnego formatu, przeznaczonego do specyficznych zadań.

[1] *array* tworzy układy tabelaryczne; ma jeden argument określający liczbę kolumn i sposób wyrównywania elementów w kolumnach.

Po przeczytaniu formatu, TEX rozpoczyna przetwarzanie dokumentu źródłowego. Plikiem źródłowym LATEX-a jest zwykły plik tekstowy (plik ASCII). Taki plik można utworzyć za pomocą dowolnego edytora tekstowego. Zawiera on tekst dokumentu oraz instrukcje, dzięki którym LATEX wie, jak złożyć tekst. Dokument źródłowy, najczęściej ma rozszerzenie *tex* lub *ltx*. Plik źródłowy zawiera tekst oraz polecenia języka TEX. Jeżeli polecenia opisują wygląd dokumentu, to mówimy o *formatowaniu wizualnym*[2], a jeżeli dotyczą jego logicznej struktury, to mówimy o *formatowaniu logicznym*[3]. Posługując się TEX-em możemy wykorzystywać oba sposoby formatowania. Na początku dokumentu źródłowego znajdują się instrukcje, które powodują, że zanim LATEX rozpocznie składanie jakiegokolwiek tekstu, musi zapoznać się z minimum pliku tzw. klasą dokumentu. Plik źródłowy LATEX-a kompilujemy stosując program TEX oraz makrodefinicje LATEX-a. W wyniku kompilacji powstaje dokument przeznaczony do wydruku. Współczesne dystrybucje LATEX-a umożliwiają automatyczne generowanie dokumentów w wersji PS (pliki *ps*) czy PDF (pliki *pdf*).

1.2 Pliki pomocnicze

W trakcie przetwarzania dokumentów LATEX tworzy pewną liczbę plików pomocniczych. Wszystkie mają taki sam rdzeń nazwy jak plik nadrzędny. Dalej będę mówić o tych plikach, powołując się na rozszerzenia ich nazw.

aux Plik używany do umieszczania w tekście powołań na różne obiekty, do układania spisu treści, ilustracji i tablic. Oprócz głównego pliku *aux* jest tworzony także oddzielny plik *aux* dla każdego pliku włączonego poleceniem `\include`, mający ten sam rdzeń nazwy co plik włączony. Wszystkie pliki *aux* są czytane przez polecenie `\begin{document}`. Polecenie `\begin{document}` zaczyna również zapisywanie w głónym pliku *aux* . Zapisywanie pliku *aux* przypisanego plikowi włączanemu poleceniem `\include` jest inicjowane przez polecenie `\include`, a kończone, kiedy plik włączony tym poleceniem zostanie do końca przetworzony. Polecenie `\nofiles` powstrzymuje tworzenie wszystkich plików *aux* .
Spis treści i informacja związana z powołaniami zawarta w pliku *aux* może być wydrukowana przez uruchomienie LATEX-a na pliku `lablst.tex`.

bib Plik ten zawiera zestaw pozycji bibliograficznych. Do przygotowywania spisów bibliograficznych służy program BibTEX. Jest to narzędzie przydatne dla osób piszących np. prace naukowe, które zawierają wiele odwołań do innych dokumentów. Polecenie `\bibliography` robi dwie rzeczy:

[2]Formatowanie wizualne to odstępy, stopień i krój pisma, kolory itp.

[3]Formatowanie logiczne operuje takimi pojęciami, jak tytuł rozdziału, tytuł punktu, tabela, tytuł tabeli, odsyłacz itp.

tworzy wpis w pliku *aux* wymieniający `pliki_bib`[4], które będą czytane przez BibTEX-a, oraz czyta plik *bbl* utworzony w celu wydrukowania spisu literatury.
Każda pozycja jest oznaczona etykietą. W dokumencie LATEX-owym zamiast pełnej treści odwołania wstawia się etykietę, którą podczas kompilacji LATEX zapisuje do pliku *aux*. Na podstawie pliku *aux* program BibTEX tworzy spis bibliograficzny, obejmujący oczywiście tylko te pozycje z pliku *bib*, które były cytowane w dokumencie. Spis jest formatowany według specyfikacji zawartej w pliku *bst* i zapisywany do pliku *bbl*. Komunikaty programu BibTeX są zapisywane do pliku *blg*. Dwa kolejne przetworzenia dokumentu powodują poprawne sformatowanie bibliografii i odwołań.

toc Plik czytany przez polecenie `\tableofcontents` w celu utworzenia spisu treści. LATEX przetwarza dokument strona po stronie, dlatego w pojedyńczym przebiegu niemożliwe jest wstawienie spisu treści na początku dokumentu, ponieważ jego treść nie jest jeszcze znana. Plik zawiera pozycje spisu utworzone przez wszystkie polecenia podziału (z wyjątkiem ich wariantów z gwiazdką). Plik *toc* jest generowany przez polecenie `\end{document}`. Jest tworzony tylko wtedy, gdy występuje polecenie `\tableofcontents` i nie ma polecenia `\nofiles`.

lot Czytany przez polecenie `\listoftables` w celu utworzenia spisu tablic. Zawiera pozycje spisu utworzone przez wszystkie polecenia `\caption` w otoczeniach `table`. Plik *lot* jest generowany w momencie wykonywania `\end{document}`. Zostaje utworzony tylko wtedy, gdy występuje polecenie `\listoftables` i nie ma polecenia `\nofiles`.

log Plik *log* zawiera komunikaty i ewentualne ostrzeżenia TEX-a, wygenerowane podczas kompilacji dokumentu. Przeglądanie tego pliku może być często pomocne w diagnostyce błędów.

idx ind i ist Plik z rozszerzeniem *idx* zawiera hasła skorowidza (indeksu). Hasła te powinny być następnie posortowane przez program `makeindex` albo `plmindex`[5]. Drugi z tych programów jest zdolny do tworzenia skorowidza zarówno według reguł angielskich, jak i polskich. Wynikiem działania programu `makeindex`\`plmindex` jest plik z rozszerzeniem *ind*, który zawiera gotowy do przetworzenia przez LATEX-a skorowidz. Postacią tworzonego skorowidza można sterować w ograniczony sposób za pomocą specjalnych instrukcji zapisanych w pliku *ist* (index style). Dodatkowo powstający plik *ilg* zawiera komunikaty pracy programu `makeindex`\`plmindex`.

[4] Argument pliki_bib jest spisem rdzeni nazw plików (bib) bibliograficznej bazy danych, odzielonych przecinkami; nazwy tych plików muszą mieć roszerzenie *bib*.

[5] `Plmindex` jest zmodyfikowaną wersją `makeindex`.

1.3 Mechanizm etykiet i powołań

W tekście często napotyka się na zdania typu: „Więcej szczegółów można znaleźć na rysunku nr 5". Jednym z powodów numerowania takich obiektów, jak ilustracje, tabele czy równania, jest właśnie powoływanie się na nie. Pewnie lepiej byłoby, gdyby w pliku źródłowym nie figurowała liczba „5", bowiem w razie dodania kolejnego rysunku ten piąty może stać się szóstym. Tu właśnie pojawia się problem, zaś LaTeX ułatwia pracę piszącemu.

Zamiast wprost wpisać numer, można przypisać danemu obiektowi wybrany *identyfikator*[6] i powoływać się na ten obiekt za pomocą tego właśnie *identyfikatora*, a LaTeX-owi pozostawić zamianę powołania na numer obiektu. Numer identyfikatorowi przypisuje polecenie `\label`, a drukuje ten numer polecenie `\ref`. Polecenie `\label` występujące w zwykłym tekście przypisuje identyfikatorowi numer bieżącej jednostki podziału. Jeśli zaś występuje w pewnym numerowanym otoczeniu, przypisuje numer tego otoczenia.

LaTeX udostępnia w każdym miejscu dokumentu *bieżącą wartość* `\ref`, która jest ustawiana za pomocą deklaracji `\refstepcounter`[7]. Polecenie `\label` umieszcza w pliku *aux*, o którym była mowa w poprzednim podrozdziale, zapis zawierający *identyfikator*, bieżącą wartość `\ref` oraz numer bieżącej strony. Kiedy odpowiednia pozycja pliku *aux* jest czytana przez polecenie `\begin{document}` (przy następnym uruchomieniu LaTeX-a na tym samym pliku źródłowym), wartość `\ref` oraz numer strony są kojarzone z *identyfikatorem*. Powoduje to, że polecenie

```
\ref{identyfikator}
```

lub

```
\pageref{identyfikator}
```

wstawi odpowiednią wartość `\ref` lub numer strony. Polecenie `\label` jest kruche, ale można go używać w argumencie polecenia podziału lub polecenia `\caption`, ale nie w innych argumentach ruchomych. Polecenie to należy umieszczać bezpośrednio za instrukcją `\caption`. Dobrym pomysłem jest też umieszczenie jej wewnątrz argumentu instrukcji `\caption` (na przykład na końcu tytułu rysunku czy tabeli). Niektórzy użytkownicy błędnie sądzą, że wystarczy umieścić instrukcję `\label` wewnątrz środowiska `figure` czy `table`, gdy tymczasem umieszczenie jej przed poleceniem `\caption` spowoduje błędy w numerach odsyłaczy.

W środowisku matematycznym instrukcje `\label` oraz `\ref` służą do tworzenia odsyłaczy do równań.

[6] Identyfikator jest dowolnym ciągiem liter, cyfr i znaków interpunkcyjnych; litery wielkie i małe są traktowane jako różne znaki.

[7] Ta deklaracja jest wydawana przez polecenia podziału, otoczenia numerowane, takie jak *equation*, oraz przez polecenie \item w otoczeniu *enumerate*.

Rozdział 2

Język programowania *Perl*

2.1 Wady i zalety

Przy wyborze narzędzi programistycznych, którymi będziemy posługiwać się, aby zrealizować zadanie postawione w temacie tej pracy, decyduje łatwość skanowania tesktu LaTeX-owego. Zdecydowanie najlepszym *parserem* do LaTeX-a jest sam program LaTeX, jednak nie będziemy go mogli zastosować do rozwiązania naszego zadania. Zdecydowaliśmy się używać w tym celu interpreter *Perl* ze względu na jego duże możliwości w przetwarzaniu wszelkich tekstów.

Perl od ang. *Practical Extraction and Report Language*, praktyczny język ekstrakcji i raportowania lub, jak twierdzą niektórzy *Pathologically Eclectic Rubbish Lister*, patologicznie eklektyczny roztrząsacz śmieci. *Perl* to interpretowany język programowania autorstwa Larry'ego Walla przeznaczony głównie do pracy z danymi tekstowymi, ale doskonale radzący sobie z wszystkimi potrzebami programisty, czy administratora systemu. Wzorowany na takich językach jak *C*, skryptowe: *sed*, *awk* i *sh* oraz na wielu innych. *Perl* jest dostępny dla wielu systemów operacyjnych, lecz jego naturalne środowisko to Unix i jego pochodne.

Perl został zaprojektowany jako praktyczne narzędzie do analizy plików tekstowych i tworzenia raportów. Jednym z naczelnych haseł jest :" *There is more than one way to do it" - TIMTOWTDI* - wymawiane jak *Tim Toady*. Co tłumaczy się jako „*Można to zrobić na różne sposoby*". Jednym z podstawowych zamysłów projektu było uczynienie łatwych zadań łatwymi do wykonania, zaś trudnych - wykonalnymi. Wszechstronność *Perla* pozwala na programowanie w różnych modelach: proceduralnym, funkcyjnym czy obiektowym, chociaż purystom przeszkadza podejście polegające na przedkładaniu wygody programisty nad czystość projektu. Obecnie rozwijany jest *Perl 6*, który będzie działał używając maszyny wirtualnej Parrot.

Chociaż *Perl* posiada większość cech języka interpretowanego, nie wykonuje ściśle każdego wiersza kodu źródłowego po kolei. Program jest najpierw kompilowany do kodu pośredniego (podobnie jak Java); jednocześnie dokony-

wana jest jego optymalizacja. Możliwe jest skompilowanie programu do kodu pośredniego i używanie go zamiast postaci źródłowej, jednak nadal konieczny jest interpreter – program wykonujący.

Istotną zaletą *Perla* jest jego bogata biblioteka modułów. Jest ich obecnie ponad 10 tys. (por. [2]). Wprowadzają one dodatkowe funkcjonalności, jak np.:

- manipulacje na datach,
- przetwarzanie plików o specyficznym formacie (XLS, CSV, PNG, JPG itp.),
- wspólpraca z bazami danych (MySQL, InterBase, Firebird, Oracle, PostgreSQL itd.),
- obsługa protokołów Internetowych (SMTP, POP, IMAP, HTTP itp.),

oraz wiele innych.

2.2 Wyrażenia regularne

Siłą *Perla* jest wygoda przetwarzania tekstów. Najprostsze sytuacje trzeba by opisywać wieloma warunkami i wieloma funkcjami. Wyrażenie regularne (ang. skrót *regexp* lub *regex*) to sposób na opisanie jednym, krótkim wzorcem zarówno prostych ciągów znaków, jak i złożonych układów. Wyrażenia regularne są niezwykle potężnym mechanizmem służącym do dopasowywania i manipulowania tekstami, wywodzą się z teorii automatów oraz języka formalnego. Te dziedziny zajmują się modelami i sposobami na klasyfikowanie języków formalnych.

Poniższy przykład ma za zadanie czytać dane ze standardowego wejścia (*STDIN*), wiersz po wierszu, szukać w nich słowa Pingwin i wypisywać wiersze, w których znajdzie to słowo, na standardowe wyjście (*STDOUT*):

Przykład 2.1.

```
#!/usr/bin/perl

while ( $wiersz = <STDIN> ) {
   print $wiersz if $wiersz =~ /Pingwin/;
}
```

Kluczowym fragmentem jest tutaj warunek `$wiersz = ~/Pingwin/`. Zawiera on dwa istotne elementy. Po pierwsze, samo wyrażenie regularne zostało zapisane w ukośnikach, po drugie zostało porównane ze zmienną `$wiersz` nie przy pomocy tradycyjnego porównania, ale z użyciem operatora `=~`, który powoduje, że jego prawa strona traktowana jest jako wyrażenie regularne.

Umiejętne stosowanie wyrażeń regularnych pozwala radykalnie uprościć przetwarzanie wszelkiego rodzaju informacji, poczynając od poczty elektronicznej, poprzez pliki dzienników aż do dokumentów tekstowych. Mechanizm ten odgrywa niezwykle ważną rolę w programowaniu skryptów *CGI* (wykorzystywanych na stronach WWW), często przetwarzających rozmaite dane tekstowe.

Wyrażenia regularne nie funkcjonują samodzielnie. Oprócz doskonale wszystkim znanego programu *grep*, wchodzą one w skład takich narzędzi programisty, jak:

- translatory języków skryptowych (m.in. *Perl, Tcl, awk* i *Python*),
- edytory tekstów (*Emacs, vi, Nisus Writer* i inne),
- środowiska programowania (m.in.*Delphi* i *Visual C++*)
- inne wyspecjalizowane narzędzia (np. *lex, Expert* czy *sed*).

Korzystanie z wyrażeń regularnych wymaga nie tylko wiedzy teoretycznej, ale również znajomości pewnych niuansów. Z wyrażeń regularnych korzystamy w skryptach pisanych w ramach tej pracy.

Rozdział 3

Generowanie drzewa referencji

3.1 Rekonstrukcja powołań

3.1.1 Dane wejściowe i algorytm

Tak jak pisalismy wcześniej, w tekście matematycznym napisanym przy pomocy LaTeX-a powołania są tworzone za pomocą poleceń `\label` i `\ref`. Zastosowanie tych makr w tekście powoduje, że powstają logiczne zależności pomiędzy faktami matematycznymi. Zależności te można wizualizować w postaci drzewa, którego korzenie to elementarne stwierdzenia lub założenia, natomiast na wyższych gałęziach są bardziej złożone twierdzenia.

Zanim zaczniemy pisać program, który będzie realizował postawione w pracy zadanie, musimy dobrze określić jakimi danymi wejściowymi dysponujemy i naszkicować algorytm programu.

Praca matematyczna, której powołania chcemy wizualizować w postaci drzewa, musi spełniać następujące warunki:

1. cała praca umieszczona jest w jednym pliku,
2. tekst musi być napisany zgodnie ze składnią języka LaTeX,
3. powołania realizowane są z wykorzystaniem mechanizmu `\label` - `\ref`,
4. dopuszczalne środowiska zawierające treść twierdzeń to: `thm`, `prop`, `lem`, `cor`, `fact`, `axiom`, `conj` oraz `assum` ,
5. do wszystkich twierdzeń dodano etykiety za pomocą makra `\label`,
6. dowody twierdzeń umieszczono w środowisku `proof`.

Wybór nazw środowisk służących do formułowania twierdzeń nie jest przypadkowy. Nie ma wprawdzie powszechnego standardu jak takie środowiska powinny się nazywać, ale organizacje takie jak AMS (American Mathematical Society), czy poważne wydawnictwa jak Elsvier Science, stosują właśnie te, które przyjęliśmy.

W pracach matematycznych mamy dwa rodzaje powołań: jawne i niejawne. Jawne to te, gdzie autor wprost podaje z jakich założeń i twierdzeń korzysta. Powszechną konwencją jest pisanie wniosków do twierdzeń bezpośrednio za twierdzeniem. Jeśli wniosek jest bezpośrednią konsekwencją twierdzenia to dowód jest wtedy pomijany i nie ma jawnie podanego powołania na to twierdzenie. Wówczas mówimy o niejawnym powołaniu.

Zdarza się również, że przesłanki do twierdzenia podane są jako zapowiedź poprzedzająca treść samego twierdzenia i dowód jest pomijany. Mogą być wówczas podane jawnie powołania poprzez `\ref`, ale nie są one umieszczone w środowisku `proof`. Takie powołania trudno jest zidentyfikować w sposób mechaniczny. Będą one niestety ignorowane.

W dużym uproszczeniu można powiedzieć, że nasz program będzie działał dwuprzebiegowo. W pierwszym przebiegu będą wyszukiwane powołania poprzez `\ref` w dowodach, w drugim, jako powołania traktować będziemy umieszczenie wniosku `cor` bezpośrednio za twierdzeniem, czyli za środowiskiem `thm`, `prop`, `lem`, `fact`, `axiom`, `conj` lub `assum`.

3.1.2 Opis powołań w języku *dot*

Zanim przejdziemy do realizacji naszego algorytmu przy pomocy konkretnych narzędzi programistycznych musimy zdecydować w jaki sposób chcemy wizualizować wynikowe drzewo powołań. Pomijamy tutaj oczywiście wszelkie edytory do rysowania diagramów itp. Jedynym narzędziem jakie udało nam się znaleźć, które w sposób automatyczny generuje obrazy grafów na podstawie ich opisu w formie pliku tekstowego, jest zestaw *Graphviz* [3].

Graphviz to zestaw programów wraz z biblioteką funkcji do wizualizacji wszelkiego rodzaju grafów. Dwa z nich `dot` oraz `neato` czytają opis grafu z pliku tekstowego i generują bitmapę przedstawiającą graf. Różnica polega na zastosowanym algorytmie, poza tym `neato` rysuje wyłącznie grafy nieskierowane. Program `lefty` pozwala w sposób interaktywny rysować nie tylko grafy, ale dowolne rysunki techniczne, natomiast program `dotty` to interaktywne narzędzie, którym można ręcznie stworzyć graf od podstaw lub zmodyfikować obraz grafu wygenerowany za pomocą algorytmu wizualizacji grafów.

Opis grafu umieszcza się w pliku tekstowym w języku *dot*. Jest to prosty język, którego składnię przedstawiamy w Dodatku A na stronie 25. Na nasze potrzeby wystarczy wiedzieć, że opis grafu zaczyna się od słowa `graph` oznaczającego graf nieskierowany lub `digraph` oznaczającego graf skierowany. Po nim następuje identyfikator grafu, czyli wymyślona przez nas dowolna nazwa i w nawiasach klamrowych zebrane są opisy wierzchołków (ang. node) i krawędzi (ang. edge). Wierzchołki (czasem zwane węzłami) to zwykłe napisy, które odwzorowane zostaną na rysynku grafu, natomiast krawędzie tworzymy przy pomocy operacji (*edgeop*) `--` w przypadku grafu nieskierowanego i za pomocą `->` w przypadku grafu skierowanego. Opis krawędzi kończy się średnikiem.

Poniżej, jako przykład, przedstawiamy opis grafu stanu procesów w jądrze systemu operacyjnego.

Przykład 3.1.

```
graph G {
    run -- intr;
    intr -- runbl;
    runbl -- run;
    run -- kernel;
    kernel -- zombie;
    kernel -- sleep;
    kernel -- runmem;
    sleep -- swap;
    swap -- runswap;
    runswap -- new;
    runswap -- runmem;
    new -- runmem;
    sleep -- runmem;
}
```

Na rysunku 3.1 przedstawiony jest wynik działania programu *dot* na danych z przykładu 3.1 powyżej. Aby go uzysać wystarczy zawołać:

```
dot -T ps -o process.eps process.dot
```

gdzie zakładamy, że plik `process.dot` zawiera opis grafu, a wynik w formacie Postscript zostanie zapisany w pliku `process.eps`.

Tak więc, to co nasz algorytm wyszukujący powołania powinien wygenerować to opis grafu skierowanego w postaci:

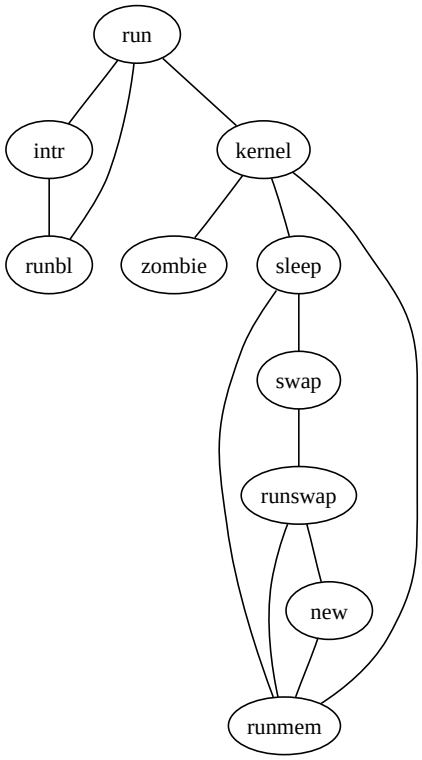
```
etykieta_przesłanki -> etykieta_twierdzenia;
```

Sam obraz drzewa powołań zostanie utworzony za pomocą programu `dot` na podstawie naszego opisu.

3.1.3 Skanowanie pracy matematycznej

Tak jak uzasadnialiśmy wcześniej nasz algorytm będziemy realizować w *Perlu*. W zasadzie jedynym gotowym do użycia *parserem* do LaTeX-a w *Perlu* jest moduł LaTeX::TOM [4]. Nazwa TOM jest skrótem od TeX Object Model. Moduł ten został zaprojektowany do przetwarzania plików LaTeX-owych z nastawieniem na ekstrakcję zwykłego tekstu i jego modyfikację.

Moduł LaTeX::TOM dostarcza parser, który skanuje i interpretuje, chociaż nie w pełni, dokumenty LaTeX-owe. Parser ten jako wynik skanowania zwraca reprezentacje dokumentu w postaci drzewa. To drzewo jest obiektem o nazwie



Rysunek 3.1: Wizualizacja grafu przedstawionego w przykładzie 3.1.

LaTeX::TOM::Tree. Węzłami (ang. nodes) tego drzewa są obiekty o nazwie LaTeX::TOM::Node. Rozpoznawanych jest pięć konstrukcji logicznych LaTeX-a i stąd pięć możliwych typów węzłów. Są to:

TEXT – Ten typ węzła reprezentuje fragmenty zwykłego tekstu zawartego w dokumencie. Zawiera on wzory matematyczne oraz wszystko, co nie zostanie rozpoznane jako makro.

```
\label{etykieta}
```

Zostanie ono zidentyfikowane jako węzeł COMMAND, który będzie posiadał jednoelementowe poddrzewo posiadające węzeł typu TEXT, zawierający napis `etykieta`.

ENVIRONMENT – Otoczenia (środowiska) reprezentowane są jako węzły typu ENVIRONMENT. Zawierają one metadane na temat danego otoczenia oraz poddrzewo reprezentujące to, co zawarte jest w tym otoczeniu. Na przykład środowisko:

```
\begin{equation}
   r \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\end{equation}
```

zostanie zeskanowane jako węzeł ENVIRONMENT z nazwą `equation`. Związane z nim poddrzewo zawierać będzie rezultat skanowania powyższej formuły matematycznej.

GROUP – Obiekt tego typu jest poddrzewem przedstawiającym efekt skanowania tego, co zostało umieszczone w nawiasach klamrowych ({}). W nawiasach tych umieszcza się fragmenty kodu LaTeX-owego, aby semantycznie odizolować je od reszty kodu. Jest to zachowywane przez parser LaTeX::TOM.

COMMENT – Węzeł COMMENT jest bardzo podobny do węzła typu TEXT. Różnica polega na tym, że zawiera on fragmenty tekstu poprzedzone znakiem %.

Moduł LaTeX::TOM dostarcza trzy obiekty, które teraz opiszemy. Skupimy się wyłącznie na metodach i własnościach, które wykorzystujemy w pracy.

LaTeX::TOM::Parser

Poza konstruktorem tego obiektu dostępne są dwie metody:

`parseFile(filename)` – Metoda ta czyta plik o nazwie `filename` i skanuje go zwracając obiekt LaTeX::TOM:Tree.

`parse(string)` – Ta metoda skanuje napis `string` i zwraca jako wynik obiekt LaTeX::TOM::Tree.

LaTeX::TOM::Tree

Opisywany tu obiekt reprezentuje strukturę dokumentu w postaci drzewa. Między innymi posiada następujące metody:

`print` – Wypisuje pełną strukturę drzewa w postaci tesktowej.

`toLaTeX` – Zwraca napis w języku LaTeX odpowiadający dla danego drzewa. Szczególnie przydatna metoda, gdy chcemy zapisać zmiany dokonane w dokumencie wejściowym.

`getCommandNodesByName(name)` – Zwraca tablicę adresów wszystkich węzłów typu COMMAND o nazwie `name`.

`getEnvironmentsByName(name)` – Zwraca tablicę adresów wszystkich węzłów typu ENVIRONMENT o nazwie `name`.

`getNodesByCondition(expression)` – Zwraca tablicę adresów wszystkich węzłów, które spełniają podane wyrażenie `expression`. Jest to bardzo wygodna metoda do przeszukania całego drzewa dokumentu i wyłapania

interesujących nas fragmentów. Wyrażenie `expression` jest dowolnym, logicznym wyrażeniem w Perlu, co świadczy o dużych możliwościach tej metody. W wyrażeniu to odwołuje się do przetwarzanego węzła za pomocą zmiennej `$node`.

LaTeX::TOM::Node

Obiekt ten odzwierciedla poszczególne konstrukcje dokumentu LaTeX-owego zwane węzłami.

`getNodeType` – Zwraca typ węzła, czyli jedno z TEXT, COMMAND, ENVIRONMENT, GROUP, albo COMMENT.

`getNodeText` – Ta metoda może być użyta w kontekście TEXT albo COMMENT i zwraca tekst zawarty w odpowiednim węźle,

`setNodeText(text)` – Ustawia wartość węzłów tekstowych TEXT i COMMENT na podaną wartość `text`.

`getEnvironmentClass` – Dla otoczeń, czyli węzłów typu ENVIRONMENT zwraca nazwę otoczenia.

`getCommandName` – Zwraca nazwę makra. Ma oczywiście sens wyłącznie dla węzłów typu COMMAND.

`getChildTree` – Zwraca obiekt LaTeX::TOM::Tree będący drzewem węzłów „poniżej" danego węzła. Stosuje się wyłącznie do węzłów typu COMMAND, ENVIRONMENT lub GROUP.

`getFirstChild` – Zwraca obiekt LaTeX::TOM::Node będący pierwszym węzłem poddrzewa obiektów poniżej danego węzła. Metoda ta może być użyta dla węzłów typu COMMAND, ENVIRONMENT lub GROUP.

`getPreviousSibling` – Zwraca adres węzła znajdującego się bezpośrednio przed danym węzłem, na tym samym poziomie drzewa.

`getNextSibling` – Zwraca adres węzła znajdującego się bezpośrednio za danym węzłem, na tym samym poziomie drzewa.

`getParent` – Zwraca adres węzła bezpośrednio nad danym węzłem.

3.2 Implementacja

Zastosowanie modułu LaTeX::TOM pozwoliło zrealizować nasz algorytm w języku *Perl* w bardzo efektywny sposób. Przedstawimy teraz implementację tego algorytmu. Kod naszego skryptu `ref2dot.pl` w całości umieszczony jest w Dodatku B na stronie 26.

Zacznijmy od opisania istotnych dla naszego programu podprocedur.

```
sub get_label {
    my ($theorem) = @_;
    my $labels = $theorem->getChildTree->getCommandNodesByName("label");
    if (@$labels[0]) {
        return @$labels[0]->getFirstChild->getNodeText;
    } else {
        return "";
    }
}
```

Procedura `get_label` jako argument pobiera obiekt typu LaTeX::TOM::Node odpowiadający otoczeniu twierdzenia. Jako wynik zwraca treść etykiety zastosowanej dla tego środowiska. Jest to wykonywane w ten sposób, że z listy wszystkich makr `\label` występujących w środowisku wybierane jest pierwsze z nich.

```
sub get_proof {
    my ($theorem) = @_;
    my $next = $theorem->getPreviousSibling;
    while ($next->getNodeType eq "COMMENT" ||
     ($next->getNodeType eq "TEXT" && $next->getNodeText =~ /^\s*$/)) {
        $next = $next->getPreviousSibling;
    }
    if ($next->getNodeType eq "ENVIRONMENT" &&
            $next->getEnvironmentClass eq "proof") {
        return $next;
    } else {
        return undef;
    }
}
```

Procedura `get_proof` podobnie jak `get_label` jako argument otrzymuje adres środowiska twierdzenia. Zadaniem tej procedury jest wyszukanie w drzewie reprezentującym tekst pracy matematycznej dowodu dla danego twierdzenia. Dowód jest to środowisko `proof`. Zakłada się tutaj, że środowisko dowodu następuje bezpośrednio za środowiskiem twierdzenia. To, co może wystąpić pomiędzy nimi to jedynie komentarze lub pusty tekst. Są one przewijane w pętli `while` tej procedury.

```
sub get_auxlabel {
    my ($label) = @_;
    foreach my $aux (@aux) {
        if (@$aux[0] eq $label) {
            return @$aux[1];
        }
    }
}
```

Argumentem powyższej procedury jest nazwa etykiety twierdzenia, a wynikiem wartość etykiety nadana przez LaTeX-a, pobrana z pliku **aux**. Plik **aux**

jest wczytany jako lista `@aux` tak, że każdy jej wyraz jest dwuelementową tablicą. Na jej pierwszym miejscu znajduje się nazwa etykiety, a na drugiej wartość.

Dalej zaczyna się właściwa część programu. Zaczynamy od pobrania parametru, jakim jest nazwa przetwarzanego pliku LaTeX-owego.

```
my $latexfile = @ARGV[0];
(my $auxfile = $latexfile) =~ s/\.tex/.aux/;
(my $dotfile = $latexfile) =~ s/\.tex/.dot/;
```

Lista parametrów do skryptu perlowego znajduje się na liście `@ARGV`. Pobieramy jej pierwszy wyraz i zapamiętujemy w zmiennej `$latexfile`. Ponieważ potrzebować będziemy tę samą nazwę pliku tylko z rozszerzeniem **.aux** oraz **.dot**, tworzymy dwie dodatkowe zmienne `$auxfile` oraz `$dotfile` zamieniając odpowiednio rozszerzenie **.tex**.

Jako pierwsze większe zadanie czytamy plik **.aux** przetwarzanej pracy matematycznej, aby móc używać etykiet nadanych przez LaTeX-a.

```
open(AUXFILE, $auxfile)
    || error("Cannot open file for reading: $auxfile: $!");

while (<AUXFILE>) {
    chomp;
    if (/^\\newlabel/) {
        m/\\newlabel{([^}]+)}{{([0-9.]+)}{.*}}/;
        push(@aux, [ $1 => $2 ]);
    }
}

close(AUXFILE);
```

Otwierany jest odpowiedni plik, ewentualnie zgłaszany jest błąd, gdy nie ma pliku **.aux**. Następnie czytany jest plik **.aux** wierszami. Ignorowane są te wiersze, które nie zaczynają się od napisu `\newlabel`. Makro `\newlabel` jak wiemy kojarzy nazwę etykiety z wartością odpowiedniego licznika. Nas będą interesować właśnie te wartości, a nie same nazwy etykiet, gdyż to one będą węzłami na obrazie drzewa powołań. Wiersz pliku poddawany jest analizie poprzez regularne wyrażenie i w efekcie otrzymujemy dwie zmienne: `$1` i `$2`, które zawierają odpowiednio nazwę etykiety i skojarzoną z nią wartość licznika. Dane te są wstawiane jako tablica dwuelementowa do listy o nazwie `@aux`.

Teraz zaczynamy skanowanie pracy.

```
my $parser = new Parser;
my $article = $parser->parseFile($latexfile);
```

Pierwsze polecenie powyżej to zainicjowanie obiektu parsera LaTeX::TOM i zapamiętanie go w zmiennej `$parser`. W drugim kroku czytany jest plik wejściowy i na jego podstawie parser LaTeX::TOM tworzy reprezentację w postaci drzewa obiektów.

```
my $theorems = $article->getNodesByCondition(
  '$node->getNodeType eq \'ENVIRONMENT\' &&
   $node->getEnvironmentClass =~ /^(thm|prop|lem|cor|fact|conj|axiom|assum)$/');
```

Tutaj wyszukujemy wystąpienia wszystkich środowisk o nazwie `thm`, `prop`, `lem`, `cor`, `fact axiom`, `conj` oraz `assum`. Wynik umieszczany jest na liście `$theorems`.

Następnie otwieramy plik z rozszerzeniem **.dot** do zapisu.

```
open(DOTFILE, "> $dotfile")
    || error("Cannot open file for writing: $dotfile: $!");

print DOTFILE "digraph G {\n";
```

Zgodnie ze składnią języka *dot* zaczynamy od deklaracji grafu skierowanego `digraph` i nadajemy mu nazwę `G`.

```
foreach my $theorem (@$theorems) {
```

Powyższa pętla stanowi pierwszy przebieg naszego programu. Zmienna `$theorem` przyjmować będzie kolejne adresy środowisk zawierających treść twierdzeń.

```
    my $auxlabel = get_auxlabel(get_label($theorem));
    if (! $auxlabel) {
        warning("Missing label: %s position: %s.",
                    $theorem->getEnvironmentClass,
                    $theorem->getNodeStartingPosition);
        next;
    }
    my $proof = get_proof($theorem);
```

W zmiennej `$auxlabel` zapamiętujemy wartość etykiety do przetwarzanego środowiska. Zmienna `$auxlabel` jest określona, jeśli do danego środowska jest w ogóle podana etykieta i ma ona skojarzoną wartość w pliku **.aux**. Jeśli nie, to po wypisaniu ostrzeżenia z podaniem nazwy i położenia środowiska w pliku, przeskakujemy do następnego twierdzenia.

W zmiennej `$proof` adres środowiska z ewentualnym dowodem.

```
    if ($proof) {
        my $refs = $proof->getChildTree->getCommandNodesByName("ref");
        my @handledrefs;
```

Procedujemy, jeśli do przetwarzanego twierdzenia jest dowód. W zmiennej `$refs` umieszczamy listę wszystkich makr `\ref` występujących w dowodzie.

Na liście `@handledrefs` będą kolekcjonowane powołania, które zostały już obsłużone tak, aby nie powtarzały się powołania wielokrotne na te same twierdzenia. W przeciwnym razie w pliku **.dot** pojawiłaby się kilkukrotnie deklaracja tej samej krawędzi, a co za tym idzie, na rysunku grafu mielibyśmy kilka strzałek pomiędzy dwoma twierdzeniami.

```
        foreach my $ref (@$refs) {
```

W tej pętli analizujemy kolejne powołania znajdujące się w dowodzie.

```
            my $refarg = $ref->getFirstChild->getNodeText;
```

Analizę zaczynamy od pobrania argumentu makra `\ref`, który jest nazwą etykiety, i zapamiętania go w zmiennej `$refarg`.

```
            my $skip = 0;
            foreach my $handledref (@handledrefs) {
                if ($handledref eq $refarg) {
                    $skip = 1;
                    last;
                }
            }
            next if ($skip == 1);
            $skip = 1;
            foreach my $theorem (@$theorems) {
                if ($refarg eq get_label($theorem)) {
                    $skip = 0;
                    last;
                }
            }
            if ($skip == 1) {
                warning("Not a theorem reference: %s.", $refarg);
                next;
            }
```

Pierwsza z powyższych pętli to przeszukanie listy `@handledrefs` obsłużonych już wcześniej powołań tak, aby nie powtarzały się powołania wielokrotne na te same twierdzenia. Przeskakujemy do następnego powołania, jeśli bieżące było już obsłużone.

W drugiej pętli przeszukujemy listę wszystkich twierdzeń i sprawdzamy, czy etykieta powołania jest etykietą do twierdzenia. Zabieg ten jest konieczny, gdyż przypadkowo możemy natrafić na etykiety do rysunków lub innych obiektów, do których odsyła się autor pracy w dowodzie. Gdy nie znajdziemy etykiety `$refarg` wśród etykiet do twierdzeń, to wypisywane jest stosowne ostrzeżenie i przeskakujemy do następnego powołania.

```
            my $auxref = get_auxlabel($refarg);
            if (! $auxref) {
                warning("Missing reference: %s in aux file.", $refarg);
                next;
            }
            if ($auxref ne $auxlabel) {
                printf(DOTFILE "\t%s -> %s;\n", $auxref, $auxlabel);
            }
            push(@handledrefs, $refarg);
        }
    }
}
```

Wyszukujemy wartość etykiety powołania i zapamiętujemy ją w zmiennej `$auxref`. Jeśli wartość ta nie jest określona, na przykład z powodu nieaktualnego pliku **.aux**, to wyświetlane jest ostrzeżenie na ten temat. Następnie sprawdzamy, czy etykieta powołania nie jest przypadkowo etykietą bieżącego twierdzenia. Jeśli tak nie jest, to na pliku **.dot** zapisywana jest odpowiednia deklaracja krawędzi w naszym grafie.

Dalej wykonywany jest drugi przebieg programu, wyszukujący niejawne powołania we wnioskach.

```
my $premise;

foreach my $theorem (@$theorems) {
    if ($theorem->getEnvironmentClass eq "cor") {
        my $label = get_label($theorem);
        if ($label && $premise && ! get_proof($theorem)) {
            my $auxlabel = get_auxlabel($label);
            my $auxpremise = get_auxlabel($premise);
            printf(DOTFILE "\t%s -> %s;\n", $auxpremise, $auxlabel);
        }
    } else {
        $premise = get_label($theorem);
    }
}
```

Analizujemy w pętli kolejne twierdzenia na liście `$theorems`. W zmiennej `premise` będzie zapamiętana nazwa etykiety ostatniego twierdzenia, nie będącego wnioskiem. Jeśli dane twierdzenie jest wnioskiem, to pobierana jest jego etykieta i zapisywana w zmiennej `$label`. Jeśli etykieta jest niepusta, wystąpiła wcześniej jakaś przesłanka w postaci twierdzenia i wniosek nie ma dowodu, to do pliku **.dot** dodajemy opis kolejnej krawędzi. Gdy dane twierdzenie nie jest wnioskiem, to jego etykieta jest zapamiętywana w zmiennej `premise`.

Na zakończenie domykana jest deklaracja grafu i zamykany jest plik **.dot**.

```
print DOTFILE "}\n";

close(DOTFILE);
```

Skrypt `ref2dot.pl` uruchamia się podając jako argument ścieżkę do pliku źródłowego pracy matematycznej w LaTeX-u:

```
ref2dot.pl plik.tex
```

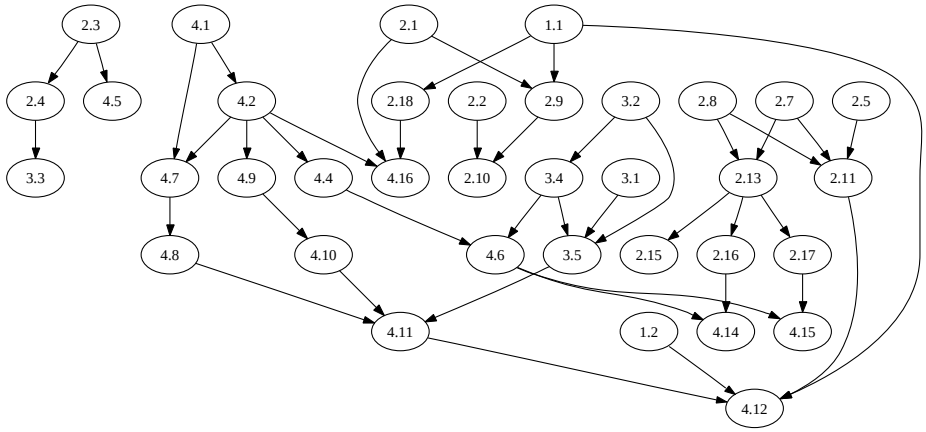
Jeśli plik jest w bieżącym katalogu, to wystarczy sama nazwa pliku. Gdy zapomnimy argumentu, podamy `-h` lub przypadkowo wprowadzimy więcej argumentów niż jeden, to program wypisze informację jakich parametrów oczekuje. W wyniku działania skryptu zostanie utworzony plik o takiej samej nazwie jak przetwarzany tylko z rozszerzeniem **.dot**. Jeśli przed wykonaniem skryptu taki plik istnieje, to zostanie nadpisany bez ostrzeżenia.

3.3 Wizualizacja

Tak jak pisaliśmy wcześniej, nasz skrypt `ref2dot.pl` czyta pracę matematyczną w LaTeX-u oraz jej plik pomocniczy **.aux** i na tej podstawie generuje opis grafu skierowanego w języku *dot*. Graf ten odzwierciedla schemat powołań między twierdzeniami. Przy pomocy programu `dot` możemy teraz utworzyć obraz tego grafu. Do wyboru mamy kilka formatów rastrowych (m.in. PNG, JPG i GIF), format Postscript (EPS) oraz formaty grafik wektorowych (SVG). W zależności od potrzeb możemy wybrać format najbardziej nam odpowiadający. Jeśli chcemy wstawić obraz grafu do tekstu w LaTeX-u, to najlepiej jest użyć format EPS. I tak na przykład:

```
dot -Tps -o plik.eps plik.dot
```

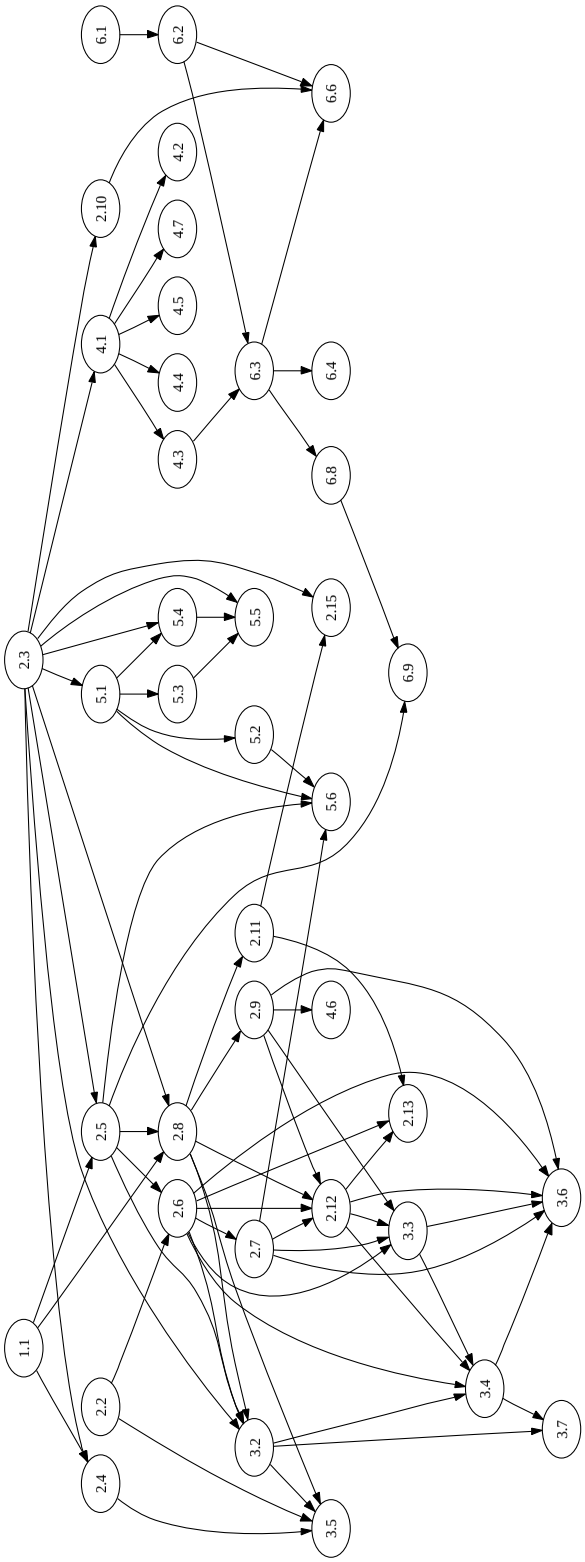
spowoduje utworzenie pliku Postscript-owego o nazwie `plik.eps`.



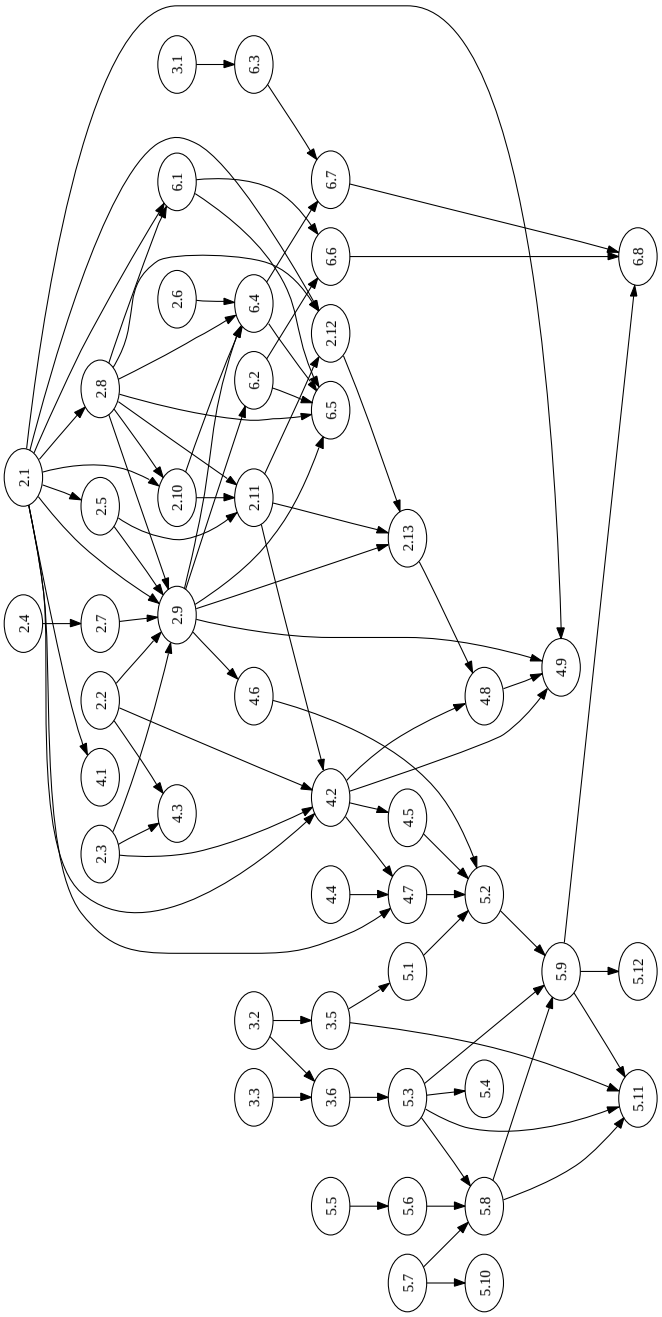
Rysunek 3.2: Drzewo powołań w przykładowej pracy magisterskiej.

Drzewo powołań z tematu niniejszej pracy kojarzy się z grafem skierowanym od dołu do góry tak, że podstawowe fakty są na dole, a wynikające z nich większe twierdzenia na górze. Ograniczenie programu `dot` polega na tym, że obrazy grafów, które tworzy, są skierowane albo od góry do dołu, albo od lewej do prawej. Domyślnie tworzone są obrazy pierwszym sposobem, ale można to zmienić dodatkowymi opcjami.

Nasz skrypt `ref2dot.pl` testowaliśmy na kilku pracach matematycznych. Na rysunku 3.2 przedstawione jest drzewo wygenerowane na podstawie pewnej pracy magisterskiej, natomiast rysunki 3.3 i 3.4 ilustrują drzewa powołań wybranych prac naukowych.



Rysunek 3.3: Drzewo powołań w przykładowej publikacji naukowej.



Rysunek 3.4: Drzewo powołań w przykładowej publikacji naukowej.

Dodatek A

Składania języka *dot*

graph	:	[**strict**] (**graph** | **digraph**) [*ID*] '**{**' *stmt_list* '**}**'
stmt_list	:	[stmt ['**;**'] [*stmt_list*]]
stmt	:	*node_stmt*
	|	*edge_stmt*
	|	*attr_stmt*
	|	ID '=' ID
	|	*subgraph*
attr_stmt	:	(**graph** | **node** | **edge**) *attr_list*
attr_list	:	'[' [*a_list*] ']' [*attr_list*]
a_list	:	ID ['=' ID] ['**,**'] [*a_list*]
edge_stmt	:	(*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS	:	*edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt	:	*node_id* [*attr_list*]
node_stmt	:	*node_id* [*attr_list*]
node_id	:	ID [*port*]
port	:	'**:**' ID ['**:**' *compass_pt*]
	|	'**:**' *compass_pt*
subgraph	:	[**subgraph** [ID]] '**{**' *stmt_list* '**}**'
	|	**subgraph** ID
compass_pt	:	(**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw**)

W zależności od tego czy graf jest skierowany czy nie, operator *edgeop* to odpowiednio `->` lub `--`.

Dodatek B

Kod skryptu ref2dot.pl

```
#!/opt/cfw/bin/perl
#
# Synopsis: Parse math article in LaTeX and generate a tree of references
#
# Last modified: Oct 11, 2006
#
# Copyright (c) 2006 Aneta Staskielunas & Mariusz Zynel
#
# This software is FREE. You can use and/or redistribute it for any purpose
# in either, modified, or unmodified form, provided that the above copyright
# notice and this permission are included in all copies or substantial
# portions of this software.
#
# THIS SOFTWARE IS PROVIDED AS IS AND COME WITH NO WARRANTY OF ANY KIND,
# EITHER EXPRESSED OR IMPLIED. IN NO EVENT WILL THE COPYRIGHT HOLDER BE
# LIABLE FOR ANY DAMAGES RESULTING FROM THE USE OF THIS SOFTWARE.

use strict;
use LaTeX::TOM;

(my $self = $0) =~ s/.*\///;

my @aux;

sub usage {
    print "usage: $self filename\n";
    exit 1;
}

sub error {
    my ($msg) = @_;
    print "ERROR: $self: $msg\n";
    exit 1;
}

sub get_label {
    my ($theorem) = @_;
    my $labels = $theorem->getChildTree->getCommandNodesByName("label");
    if (@$labels[0]) {
        return @$labels[0]->getFirstChild->getNodeText;
    } else {
        return "";
    }
}
```

```
sub get_proof {
    my ($theorem) = @_;
    my $next = $theorem->getPreviousSibling;
    while ($next->getNodeType eq "COMMENT" ||
           ($next->getNodeType eq "TEXT" && $next->getNodeText =~ /^\s*$/)) {
        $next = $next->getPreviousSibling;
    }
    if ($next->getNodeType eq "ENVIRONMENT" &&
           $next->getEnvironmentClass eq "proof") {
        return $next;
    } else {
        return undef;
    }
}

sub get_auxlabel {
    my ($label) = @_;
    foreach my $aux (@aux) {
        if (@$aux[0] eq $label) {
            return @$aux[1];
        }
    }
}

if ($#ARGV != 0) {
    usage();
}

my $latexfile = @ARGV[0];
(my $auxfile = $latexfile) =~ s/\.tex/.aux/;
(my $dotfile = $latexfile) =~ s/\.tex/.dot/;

open(AUXFILE, $auxfile) ||
                error("Cannot open file for reading: $auxfile: $!");

while (<AUXFILE>) {
    chomp;
    if (/^\\newlabel/) {
        m/\\newlabel{([^}]+)}{{([0-9.]+)}{.*}}/;
        push(@aux, [ $1 => $2 ]);
    }
}

close(AUXFILE);


my $parser = new Parser;
my $article = $parser->parseFile($latexfile);

my $theorems = $article->getNodesByCondition(
  '$node->getNodeType eq \'ENVIRONMENT\' &&
   $node->getEnvironmentClass =~
                /^(thm|prop|lem|cor|fact|conj|axiom|assum)$/');

open(DOTFILE, "> $dotfile") ||
                error("Cannot open file for writing: $dotfile: $!");

print DOTFILE "digraph G {\n";

foreach my $theorem (@$theorems) {
    my $auxlabel = get_auxlabel(get_label($theorem));
    if (! $auxlabel) {
        warning("Missing label: %s position: %s.",
                    $theorem->getEnvironmentClass,
                    $theorem->getNodeStartingPosition);
        next;
```

```
    }
    my $proof = get_proof($theorem);
    if ($proof) {
        my $refs = $proof->getChildTree->getCommandNodesByName("ref");
        my @handledrefs;
        foreach my $ref (@$refs) {
            my $refarg = $ref->getFirstChild->getNodeText;
            my $skip = 0;
            foreach my $handledref (@handledrefs) {
                if ($handledref eq $refarg) {
                    $skip = 1;
                    last;
                }
            }
            next if ($skip == 1);
            $skip = 1;
            foreach my $theorem (@$theorems) {
                if ($refarg eq get_label($theorem)) {
                    $skip = 0;
                    last;
                }
            }
            if ($skip == 1) {
                warning("Not a theorem reference: %s.", $refarg);
                next;
            }
            my $auxref = get_auxlabel($refarg);
            if (! $auxref) {
                warning("Missing reference: %s in aux file.", $refarg);
                next;
            }
            if ($auxref ne $auxlabel) {
                printf(DOTFILE "\t%s -> %s;\n", $auxref, $auxlabel);
            }
            push(@handledrefs, $refarg);
        }
    }
}

my $premise;

foreach my $theorem (@$theorems) {
    if ($theorem->getEnvironmentClass eq "cor") {
        my $auxlabel = get_auxlabel(get_label($theorem));
        if ($auxlabel && $premise && ! get_proof($theorem)) {
            my $auxpremise = get_auxlabel($premise);
            printf(DOTFILE "\t%s -> %s;\n", $auxpremise, $auxlabel);
        }
    } else {
        $premise = get_label($theorem);

    }
}

print DOTFILE "}\n";

close(DOTFILE);
```

Bibliografia

[1] Lamport L., *LaTeX system przygotowywania dokumentów*, Wydawnictwo Naukowo-Techniczne, Warszawa, 2004.

[2] Comprehensive Perl Archive Network:
`http://cpan.org/`

[3] Strona domowa projektu *Graphviz*:
`http://graphviz.org/`

[4] Strona domowa projektu *LaTeX::TOM*:
`http://br.endernet.org/~akrowne/elaine/latex_tom/`

[5] Partl H., Hyna I. i Schlegl E., *Nie za krótkie wprowadzenie do systemu LaTeX2E*, wersja 3.2, wrzesień 1998.

[6] Gajda W., *HTML kontra LaTeX.*:
`http://www.gajdaw.pl/latex/html-kontra-latex.html`

[7] `http://pl.wikipedia.org/wiki/latex`

[8] `http://www.tug.org/tex-virtual-academy-pl/cototex.html`