Języki formalne i automaty
JFA

Marcin Kubica
e-mail: kubica@mimuw.edu.pl

2006

Niniejsze materiały powinny być pierwszym źródłem informacji dotyczących przedmiotu *Języki formalne i automaty* (JFA). Czytelnikom, którzy oprócz lektury tych notatek, chcieliby sięgnąć do podręcznika, polecam w pierwszej kolejności książkę D. Kozena, *Automata and Computability*. Jest ona napisana przystępnie i zawiera mnóstwo zadań, choć nie jest (jak dotąd) dostępna w języku polskim. W drugiej kolejności polecam książkę J. Hopcrofta i J. Ullmana, *Wprowadzenie do teorii automatów, języków i obliczeń*. Oprócz tematów wchodzących w zakres tego kursu można w niej znaleźć też wiele informacji na zaawansowane pokrewne tematy. Ponadto została ona przetłumaczona na język polski. Obie te książki nie zawierają informacji na temat generatorów analizatorów leksykalnych i składniowych. W pierwszej kolejności polecam tutaj dokumentację Flex'a i Bison'a. Dodatkowo, można przeczytać odpowiednie rozdziały z książki A. Aho, R. Sethi i J. D. Ullmana, *Kompilatory*.

Do każdego wykładu dołączono szereg zadań. W przypadku studiów internetowych, rozwiązania zadań z p. *Praca domowa* należy wykonywać w określonych terminach i zgłaszać poprzez serwis `edu`. Pozostałe zadania mają charakter uzupełniający i są przeznaczone do samodzielnego rozwiązywania.

Materiały te są również dostępne w formacie PS i PDF. Uwagi dotyczące niniejszych materiałów proszę przesyłać na adres: `kubica@mimuw.edu.pl`.

Literatura

- [HU] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman *Wprowadzenie do teorii automatów, języków i obliczeń*, PWN, 2005 r.
- [K] Dexter C. Kozen, *Automata and Computability*,
- [ASU] A. V. Aho, R. Sethi, J. D. Ullman, *Kompilatory*, WNT, 2002,

Podziękowania

Niniejsze materiały powstały na podstawie notatek do prowadzonych przeze mnie, na przestrzeni kilku lat, wykładów z teorii języków i automatów oraz budowy kompilatorów. Chciałbym gorąco podziękować moim kolegom, którzy w tym czasie prowadzili ćwiczenia z tych przedmiotów, a w szczególności: Łukaszowi Krzeszczakowskiemu, Łukaszowi Maśko, Marcinowi Sawickiemu i Tomkowi Waleniowi. Ich uwagi miały wpływ na postać prowadzonych zajęć, a więc i również na te materiały. W szczególności część zamieszczonych tu zadań pochodzi od nich.

Szczególnie gorąco chciałbym podziękować Tomkowi Waleniowi za pomoc przy konwertowaniu niniejszych materiałów na format html i przygotowanie umieszczonych tu elementów multimedialnych.

Materiały te zostały opracowane w PJWSTK w projekcie współfinansowanym ze środków EFS.

1 Sylabus

Niniejsze materiały powstały pierwotnie z myślą o studiach internetowych z *Języków Formalnych i Automatów* (JFA). Mają one również sotoswane na studiach stacjonarnych, w ramach kursu z *AUtomatów i Gramatyk* (AUG). Poniżej zamieszczamy sylabusy dla obydwu tych przedmiotów.

1.1 Języki Formalne i Automaty (Studia internetowe)

W ramach kursu przedstawiane są klasy języków tworzące hierarchię Chomsky'ego: języki regularne, bezkontekstowe, kontekstowe i częściowo obliczalne (rekurencyjnie przeliczalne). Dodatkowo przedstawiona jest klasa języków obliczalnych.

Dla każdej z tych klas przedstawione są formalizmy służące do opisu języków: automaty skończone deterministyczne, niedeterministyczne i z ε-przejściami, wyrażenia regularne, automaty stosowe, gramatyki bezkontekstowe, gramatyki kontekstowe, maszyny Turinga oraz gramatyki ogólne (typu 0). Przedstawione są też fakty dotyczące przynależności języków do poszczególnych klas, w tym: lematy o pompowaniu dla języków regularnych i bezkontekstowych.

1.1.1 Program wykładów

1. Wprowadzenie, podstawowe pojęcia.

2. Wyrażenia regularne i wzorce.

3. Analiza leksykalna i Flex/Lex.
4. Deterministyczne automaty skończone.
5. Niedeterministyczne automaty skończone.
6. Równoważność automatów skończonych i wyrażeń regularnych.
7. Lemat o pompowaniu dla języków regularnych.
8. Minimalizacja deterministycznych automatów skończonych.
9. Języki i gramatyki bezkontekstowe.
10. Postać normalna Chomsky'ego i algorytm Cocke-Younger'a-Kasami.
11. Lemat o pompowaniu dla języków bezkontekstowych.
12. Automaty stosowe.
13. Analiza składniowa i Bison/Yacc.
14. Maszyny Turinga i obliczalność.
15. Języki obliczalne, częściowo obliczalne i nieobliczalne.

1.1.2 Kryteria zaliczeń

- prace domowe (przez Internet),
- egzamin (w uczelni).

Podstawą do wystawienia oceny z ćwiczeń będą punkty zdobyte za prace domowe, a z egzaminu punkty zdobyte na egzaminie.

1.1.3 Wymagane oprogramowanie

- Flex/Lex,
- Bison/Yacc,
- kompilator C/C++.

W przypadku Linuksa (zalecany) są to: `flex`, `bison`, `gcc`, `g++`. W przypadku Windows są to: `DJGPP` (z `gcc`), pakiety `flex` i `bison` z `GnuWin32`.

1.1.4 Powiązania merytoryczne

Nazwa przedmiotu poprzedzającego	Wymagany materiał
Matematyka dyskretna	Podstawy teorii mnogości, ciągi, relacje i funkcje.
Programowanie I i II	Podstawy programowania, podstawowe typy danych, programowanie w C lub C++.
Algorytmy i struktury danych	Złożoność asymptotyczna, podstawowe struktury danych (stosy), programowanie dynamiczne.

1.2 Automaty i Gramatyki (Studia stacjonarne)

Kurs obejmuje techniki analizy leksykalnej i składniowej, wykorzystanie narzędzi wspomagającymi generowanie analizatorów leksykalnych i składniowych (lex i yacc), oraz niezbędne podstawy z teorii języków formalnych i automatów, n.t. języków regularnych i bezkontekstowych. Przedstawione są również elementarne wiadomości z ' teorii obliczeń: maszyny Turinga, klasy języków obliczalnych i częściowo obliczalnych.

1.2.1 Program wykładów

1. Wprowadzenie, podstawowe pojęcia.
2. Wyrażenia regularne i wzorce.
3. Analiza leksykalna i Flex/Lex.
4. Deterministyczne automaty skończone.
5. Niedeterministyczne automaty skończone.
6. Równoważność automatów skończonych i wyrażeń regularnych.
7. Języki i gramatyki bezkontekstowe.
8. Automaty stosowe.
9. Analiza składniowa i Bison/Yacc.
10. Analiza składniowa zstępująca i parsery $LL(1)$.
11. Analiza składniowa wstępująca i parsery $SLR(1)$.
12. Parsery kanoniczne $LR(1)$.
13. Maszyny Turinga i obliczalność.

14. Języki obliczalne, częściowo obliczalne i nieobliczalne.

1.2.2 Program ćwiczeń

1. Operacje na słowach i językach.
2. Wyrażenia regularne.
3. Flex.
4. Deterministyczne automaty skończone.
5. Niedeterministyczne automaty skończone i determinizacja.
6. Równoważność automatów skończonych i wyrażeń regularnych.
7. Gramatyki bezkontekstowe.
8. Gramatyki bezkontekstowe, c.d.
9. Automaty stosowe.
10. Kolokwium.
11. Yacc
12. Praca nad progamami zaliczeniowymi.
13. Zaliczanie programów.

1.2.3 Kryteria zaliczeń

Do zaliczenia ćwiczeń konieczne jest:

- zaliczenie kolokwium,
- zaliczenie programu.

Dobrze zaliczone ćwiczenia mogą (ale nie muszą) skutkować zwolnieniem z egzaminu. W przypadku braku zwolnienia, egzamin jest obowiązkowy.

1.2.4 Wymagane oprogramowanie

- Flex/Lex,
- Bison/Yacc,
- kompilator C/C++.

W przypadku Linuksa (zalecany) są to: `flex`, `bison`, `gcc`, `g++`. W przypadku Windows są to: `DJGPP` (z `gcc`), pakiety `flex` i `bison` z `GnuWin32`.

1.2.5 Powiązania merytoryczne

Nazwa przedmiotu poprzedzającego	Wymagany materiał
Matematyka dyskretna	Podstawy teorii mnogości, ciągi, relacje i funkcje.
Programowanie I i II	Podstawy programowania, podstawowe typy danych, programowanie w C lub C++.
Algorytmy i struktury danych	Złożoność asymptotyczna, podstawowe struktury danych (stosy), programowanie dynamiczne.

1.2.6 Tematy programów zaliczeniowych

Programy zaliczeniowe muszą być tworzone przy użyciu Lex'a i Yacc'a/Bison'a. Nie muszą to być duże programy. Programy zaliczeniowe mogą być opracowywane wyłącznie indywidualnie. Prace zbiorowe są niedopuszczalne. Oto **przykładowa** lista tematów programów zaliczeniowych. Nie należy jej traktować jako listy tematów do wyboru, ale raczej jako źródło inspiracji.

- Kalkulatory i inne arytmetyczne:
  - Wykonujący operacje na napisach. Dostępne operacje to: sklejanie, odwracanie, wyłuskanie pierwszego/ostatniego słowa, obcięcie pierwszego/ostatniego słowa, maksymalny prefikso-sufiks. Możliwość przypisywania wartości zmiennym.
  - Wzbogacony o funkcje. Umożliwia definiowanie funkcji (bez rekurencji), postaci: `function f (x,y) = <wyrazenie>;` oraz obliczanie wyrażeń, w których mogą pojawiać się zdefiniowane funkcje.
  - Obliczanie pochodnych wyrażeń zawierajacych jedną zmienną (x), operacje $+$, $-$, $*$, $/$, $\hat{}$ oraz funkcje: $\ln, \sin, \cos, \ldots$.
  - Możliwość pracy w trybach: dziesiętnym, binarnym i szesnastkowym.
  - Ułamki (ze skracaniem) i potegowanie.
  - Operujący na liczbach w zapisie rzymskim.
  - Konwersja dat z reprezentacji słownej do liczbowej.
  - Konwersja kwot z reprezentacji słownej do liczbowej.
- HTML, XML, CSS i inne języki znacznikowe:
  - pretty-printer HTML $\rightarrow$ HTML,
  - pretty-printer XML $\rightarrow$ HTML,

  - walidator HTML sprawdzający czy wiersze w tabelce mają po tyle samo komórek,
  - konwerter HTML → HTML domykający wszystkie niedomknięte tagi `<li>`, `<td>` i `<tr>`,
  - walidator HTML sprawdzający czy w danym tagu podane atrybuty mają różne nazwy,
  - walidator CSS sprawdzający czy w każdej sekcji atrybuty mają różne nazwy,
  - konwerter CSS dopisujący wszystkie brakujące znaki `:`, `;` i `}`,
  - analizator HTML wyłuskujący strukturę tagów (bez atrybutów i wyświetlanej treści) i ładnie ją wypisujący (w XML-u).

- Języki programowania:
  - zamiana instrukcji `while`/`case`/`for` w C na skoki,
  - dopisywanie do programu w Pascalu/C/C++/Javie po każdym `end`-ie/nawiasie zamykającym `}` do czego się on odnosi,
  - walidator deklaracji klas w Delphi/Javie — sprawdza, czy kazda użyta klasa została zadeklarowana,
  - pretty-printer dla wybranego języka programowania (np.: Pascala, C, C++, Javy) tworzący wcięcia i odpowiednie odstępy,
  - dopisywanie przy każdej funkcji/procedurze w wybranym języku komentarza z listą używanych zmiennych i wywoływanych innych funkcji,
  - walidator zapytań w SQL'u uwzględniający konstrukcje: `select`, `from` (`join`), `where`: (operacje logiczne, porównywanie atrybutów, `order by`),
  - walidator wybranego języka programowania (np. C) sprawdzający, czy każda funkcja jest zdefiniowana tylko raz, oraz czy każda zadeklarowana funkcja jest zdefiniowana,
  - konwerter wybranego języka programowania (np. Java-script) analizujący wszystkie definiowane w programie nazwy funkcji i zmiennych, i zamieniający je na jak najkrótsze identyfikatory,
  - walidator sprawdzający wybrane aspekty poprawności programów w wybranym języku programowania.

- Inne:
  - Konwerter RTF → HTML.
  - Konwerter MediaWiki → HTML.
  - Dekodowanie plików skompresowanych za pomocą kodu Huffmana. Plik zawiera opis drzewa kodowania i zakodowaną treść (trzeba wymyślić składnię). Program wczytuje taki plik i wypisuje jego zdekodowaną postać.

Wykład 1. Wprowadzenie

Celem niniejszego wykładu jest:

- poznanie podstawowych środków opisu składni (wyrażenia regularne, gramatyki bezkontekstowe),
- poznanie narzędzi wspomagających przetwarzanie danych o określonej składni (Lex, Yacc).
- zrozumienie podstawowych pojęć dotyczących obliczalności i odpowiedzenie sobie na kilka podstawowych pytań jej dotyczących:
  - Co to znaczy, że dany problem może zostać rozwiązany za pomocą programu komputerowego?
  - Czy istnieją problemy obliczeniowe, których nie można rozwiązać za pomocą programu komputerowego?
  - W jakim stopniu to co możemy obliczyć zależy od modelu obliczeń czy dostępnych konstrukcji programistycznych?

Nie są to proste pytania.

W trakcie kolejnych wykładów będziemy poznawać kolejne klasy języków tworzące tzw. hierarchię Chomsky'ego. Hierarchia ta jest owocem prac Noama Chomsky'ego, lingwisty, który próbował sformalizować pojęcia *gramatyki* i *języka*. Hierarchia ta składa się z czterech rodzajów gramatyk:

0. gramatyki ogólne,
1. gramatyki kontekstowe,
2. gramatyki bezkontekstowe,
3. gramatyki liniowe.

W trakcie badań nad pojęciem obliczalności pojawiło się wiele modeli, o różnej sile wyrazu:

1. automaty skończone, wyrażenia regularne (stała skończona pamięć),
2. automaty stosowe (stała skończona pamięć + nieograniczony stos),
3. bez ograniczeń:
   - maszyny Turinga,
   - systemy Posta,
   - rachunek λ, i inne.

Niektóre z nich powstały na długo przed tym zanim powstały komputery i informatyka. Okazało się, że modele te w zadziwiający sposób odpowiadają kolejnym klasom w hierarchii Chomsky'ego:

- gramatyki liniowe = automaty skończone = wyrażenia regularne,
- gramatyki bezkontekstowe = automaty stosowe,
- gramatyki kontekstowe = liniowo ograniczone maszyny Turinga,
- gramatyki ogólne = maszyny Turinga.

Nie są to czysto teoretyczne formalizmy. Niektóre z nich mają wiele praktycznych zastosowań. Szczególny nacisk położymy na dwa najważniejsze z tych formalizmów: wyrażenia regularne i gramatyki bezkontekstowe. W szczególności służą do specyfikowania analizatorów leksykalnych i składniowych — typowych modułów pojawiających się wszędzie tam, gdzie program wczytuje dane o określonej składni. Poznamy też narzędzia wspomagające tworzenie takich analizatorów — generatory, które same tworzą kod źródłowy analizatorów na podstawie ich specyfikacji.

1.1 Języki jako problemy decyzyjne

We wprowadzeniu wspomnieliśmy o *obliczeniach* i *językach*. Gdy mówimy o obliczaniu, to zwykle mamy na myśli obliczenie wartości takiej czy innej funkcji. Gdy zaś mówimy o języku, to mamy na myśli (potencjalnie nieskończony) zbiór napisów złożonych ze znaków ustalonego (skończonego) alfabetu. (Napisy takie będziemy nazywać *słowami*.) Jak połączyć te dwa pojęcia?

Def. 1. *Problem decyzyjny* to funkcja, która możliwym danym wejściowym przyporządkowuje wartości logiczne tak/nie. □

Inaczej mówiąc problem decyzyjny, to taki problem, w którym szukany wynik to wartość logiczna. Na problem decyzyjny możemy też patrzeć jak na zbiór danych — zbiór tych danych, dla których odpowiedź brzmi „tak". I odwrotnie, na dowolny zbiór danych możemy patrzeć jak na problem decyzyjny — czy dane należą do określonego zbioru?

Jak to jednak ma się do języków? Jeśli interesuje nas tylko składnia języka (a nie jego semantyka, czyli znaczenie), to każdy język możemy przedstawić sobie jako zbiór (być może nieskończony) słów. Tak więc każdy język nie jest niczym więcej niż problemem decyzyjnym określonym dla słów. Powyższe intuicje są przedstawione bardziej formalnie poniżej.

W trakcie tego kursu będziemy zajmować się m.inn. tym, dla jakich języków (gdy spojrzymy na nie jak na problemy decyzyjne) można skonstruować automatyczne mechanizmy odpowiadające na pytanie, czy dane słowo należy do języka.

1.2 Podstawowe pojęcia dotyczące słów

Def. 2. *Alfabet* to dowolny niepusty, skończony zbiór. Elementy alfabetu nazywamy *znakami.* □

Alfabetem może to być np. zbiór cyfr dziesiętnych, zbiór znaków ASCII, czy zbiór bitów $\{0, 1\}$. Alfabet będziemy zwykle oznaczać przez Σ, a elementy alfabetu przez $a, b, c, \ldots$.

Def. 3. *Słowo* (lub *napis*) nad alfabetem Σ, to dowolny skończony ciąg znaków z Σ. W szczególności, pusty ciąg jest też słowem (i to nad dowolnym alfabetem). Oznaczamy go przez ε. □

Słowa będziemy zwykle reprezentować przez zmienne $x, y, z, \ldots$.

W przypadku języków programowania mówimy zwykle o napisach. Natomiast w teorii języków używa się raczej terminu słowo. Dodatkowo, w odniesieniu do języków naturalnych bardziej właściwe niż „słowo" wydaje się „zdanie". Tak naprawdę jednak chodzi o to samo pojęcie.

Def. 4. *Długość* słowa oznaczamy przez $|\cdot|$. □

Na przykład $|ala| = 3$, $|\varepsilon| = 0$.

Def. 5. *Sklejenie* (konkatenacja) słów to słowo powstałe z połączenia słów, tak jakby zostały one zapisane kolejno po sobie. Sklejanie zapisujemy pisząc po sobie sklejane słowa. □

Na przykład, jeżeli $x = ala, y = ma, z = kota$, to $xyz = alamakota$. Sklejanie słów jest łączne, tzn. $x(yz) = (xy)z$, a ε jest elementem neutralnym sklejania, tzn. $x\varepsilon = \varepsilon x = x$.

Sposób zapisu sklejania słów przypomina zwyczajowy zapis iloczynów (bez znaku mnożenia). Przez analogię, możemy wprowadzić operacje „potęgowania" słów:

Def. 6. Przez a^n oznaczamy n-krotne powtórzenie znaku a. Podobnie, przez x^n oznaczamy n-krotne powtórzenie słowa x. □

Na przykład, $a^5 = aaaaa$, $a^0 = \varepsilon$, $a^{n+1} = a^n a$, oraz $x^0 = \varepsilon$, $x^{n+1} = x^n x$. Jeżeli $x = ala$, to $x^3 = alaalaala$.

Def. 7. Przez $\#_a(x)$ oznaczamy *liczbę wystąpień* znaku a w słowie x. □

Na przykład, $\#_a(ala) = 2$.

Def. 8. *Prefiksem* słowa nazywamy dowolny jego początkowy fragment, tzn. x jest prefiksem y wtw., gdy istnieje takie z, że $xz = y$. Jeżeli ponadto $x \neq \varepsilon$ i $x \neq y$, to mówimy, że x jest *właściwym* prefiksem y.

Analogicznie *sufiksem* słowa nazywamy dowolny jego końcowy fragment, tzn. x jest sufiksem y wtw., gdy istnieje takie z, że $zx = y$. Jeżeli ponadto $x \neq \varepsilon$ i $x \neq y$, to mówimy, że x jest *właściwym* sufiksem y.

Podsłowem danego słowa nazywamy dowolny jego spójny fragment, tzn. powiemy, że x jest podsłowem y wtw., gdy istnieją takie v i w, że $vxw = y$. □

Na przykład, *kaj* i *kaja* są prefiksami (właściwymi) słowa *kajak*, natomiast *jak* jest jego sufiksem (właściwym).

Def. 9. Przez $\text{rev}(x)$ oznaczamy słowo x czytane wspak. Na przykład $\text{rev}(abbab) = babba$.
Jeżeli słowo x czytane wprost i wspak jest takie samo (tzn. $x = \text{rev}(x)$), to mówimy, że x jest *palindromem*. $\square$

1.3 Podstawowe pojęcia dotyczące języków

Przejdziemy teraz o jeden poziom wyżej i zajmijmy się zbiorami słów czyli językami.

Def. 10. *Język* (nad alfabetem Σ), to dowolny zbiór słów (nad alfabetem Σ). $\square$

Jeżeli alfabet Σ jest znany z kontekstu, to będziemy go czasami pomijać. Języki będziemy zwykle oznaczać przez $A, B, C, \dots$. Język złożony ze wszystkich możliwych słów nad alfabetem Σ będziemy oznaczać przez Σ^*.

Język, to podzbiór zbioru Σ^*. Będziemy więc (w kolejnych wykładach) utożsamiać język z problemem decyzyjnym, dla zbioru danych Σ^*.

Skoro języki to zbiory słów (napisów), a więc określone są na nich wszystkie podstawowe operacje na zbiorach:

- $\emptyset$ to pusty język, który nie zawiera żadnego słowa,
- $A \cup B$ to suma języków A i B,
- $A \cap B$ to część wspólna (przecięcie) języków A i B,
- $A \setminus B$ to różnica języków A i B.

Na przykład, $(\{ala, ola, ula\} \cap \{abba, ala, ula\}) \setminus \{ula, bula\} = \{ala\}$.

Przyda nam się też kilka operacji charakterystycznych dla języków:

Def. 11. $\overline{A}$ to dopełnienie języka A, czyli $\overline{A} = \Sigma^* \setminus A$. $\square$

Na przykład $\overline{\Sigma^*} = \emptyset$.

Def. 12. AB oznacza sklejenie (konkatenację) języków A i B, czyli język zawierający wszystkie możliwe sklejenia słów z A ze słowami z B, $AB = \{xy : x \in A, y \in B\}$. $\square$

Na przykład $\{a, ab\}\{b, bb\} = \{ab, abb, abbb\}$.

Zauważmy, że sklejanie języków jeszcze bardziej przypomina mnożenie, niż to miało miejsce w przypadku sklejania słów. Elementem neutralnym (czyli „jedynką") sklejania języków jest język zawierający tylko słowo puste $\{\varepsilon\}$, $\{\varepsilon\}A = A\{\varepsilon\} = A$. Natomiast „zerem" sklejania języków jest język pusty, $\emptyset A = A\emptyset = \emptyset$. W odróżnieniu od operacji mnożenia, sklejanie języków nie jest przemienne.

Sklejanie języków jest rozdzielne względem sumowania języków:

$$A(B \cup C) = AB \cup AC$$

Jest tak dlatego, że każde sklejenie słowa należącego do A ze słowem należącym do $(B \cup C)$ jest sklejeniem słowa należącego do A ze słowem należącym do B, lub sklejeniem słowa należącego do A ze słowem należącym do C, i vice versa. Analogicznie:

$$(A \cup B)C = AC \cup BC$$

Mając zdefiniowane sklejanie języków, możemy analogicznie do potęgowania słów zdefiniować potęgowanie języków:

Def. 13. Język A^n definiujemy rekurencyjnie:

- $A^0 = \{\varepsilon\}$,
- $A^{n+1} = A^n A$.

Czyli $A^n = \underbrace{A \dots A}_{n \text{ razy}}$. □

Inaczej mówiąc, A^n to język zawierający wszystkie możliwe sklejenia n (niekoniecznie różnych) słów wziętych z A.

Języki, jako zbiory, mogą być nieskończone. Możemy więc rozważać sklejenie dowolnej liczby słów pochodzących z A. Prowadzi to do tzw. domknięcia Kleene'ego:

Def. 14. Domknięcie Kleene'ego języka A, oznaczane jako A^*, to $A^* = \bigcup_{n \geq 0} A^n$. Inaczej mówiąc, A^* to język zawierający wszystkie możliwe sklejenia dowolnej liczby (łącznie z 0) słów należących do A. □

Dla każdego języka A mamy $\varepsilon \in A^*$, gdyż $\{\varepsilon\} = A^0 \subseteq A^*$. Jeżeli tylko A zawiera jakieś niepuste słowo, to A^* jest zbiorem nieskończonym. Na przykład, $\{ab, aa\}^* = \{\varepsilon, ab, aa, abab, abaa, aaab, aaaa, \dots\}$.

Jeżeli spojrzymy na Σ nie jak na alfabet, ale jak na zbiór słów długości 1, to wyjaśni się dlaczego Σ^* jest zbiorem wszystkich słów nad alfabetem Σ — po prostu każde słowo jest sklejeniem pewnej liczby znaków.

Czasami będzie nas interesowało sklejenie dowolnej, ale dodatniej, liczby słów z A:

Def. 15. Przez A^+ oznaczamy język zawierający wszystkie możliwe sklejenia dowolnej dodatniej liczby słów należących do A, $A^+ = AA^*$. □

1.4 Powtórzenie pojęć dotyczących relacji

Przypomnijmy sobie kilka pojęć dotyczących relacji.

Def. 16. Niech X będzie dowolnym zbiorem, a $\rho \subseteq X \times X$ relacją (binarną) określoną na tym zbiorze. Powiemy, że relacja ρ jest:

- zwrotna, jeżeli dla każdego $x \in X$ zachodzi $x \rho x$,

- symetryczna, gdy dla dowolnych $x, y \in X$ jeżeli mamy $x\rho y$, to mamy również $y\rho x$,
- przechodnia, gdy dla dowolnych $x, y, z \in X$ jeżeli mamy $x\rho y$ i $y\rho z$, to mamy również $x\rho z$,
- antysymetryczna, gdy dla dowolnych $x, y \in X$ jeśli $x\rho y$ i $y\rho x$, to $x = y$,
- relacją równoważności, jeśli jest zwrotna, symetryczna i przechodnia,
- częściowym porządkiem, jeśli jest zwrotna, przechodnia i antysymetryczna.

□

Relacja równoważności dodatkowo dzieli zbiór X na rozłączne klasy abstrakcji.

Def. 17. Jeśli ρ jest relacją równoważności, $x \in X$, to przez $[x]_\rho$ oznaczamy klasę abstrakcji x:

$$[x]_\rho = \{y \in X : x\rho y\}$$

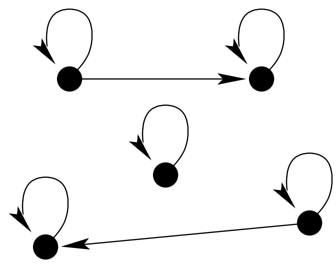
Przez $X_{/\rho}$ oznaczamy zbiór klas abstrakcji elementów zbioru X:

$$X_{/\rho} = \{[x]_\rho : x \in X\}$$

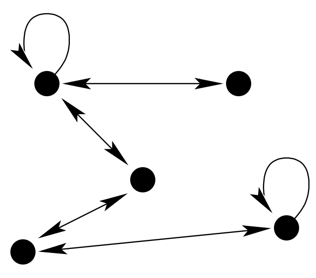
□

Pojęcia te są bardziej intuicyjne, gdy przedstawimy sobie relację jako graf skierowany (z pętelkami). Wyobraźmy sobie, że elementy zbioru X, to wierzchołki grafu. Z wierzchołka x prowadzi krawędź do wierzchołka y wtedy i tylko wtedy, gdy $x\rho y$. (W szczególności, gdy $x\rho x$, to mamy „pętelkę” prowadzącą z x do x.)

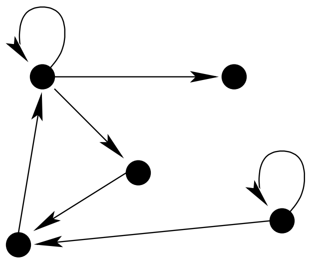
Relacja jest zwrotna, gdy w każdym wierzchołku jest „pętelka”.



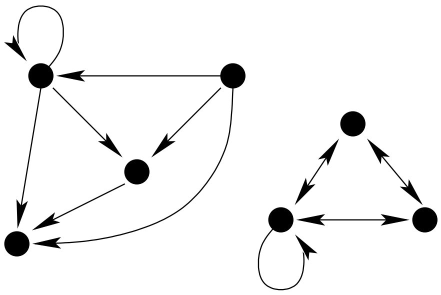
Relacja jest symetryczna, gdy krawędzie między (różnymi) wierzchołkami są dwukierunkowe (tzn. jeżeli jest krawędź w jednym kierunku, to jest i w drugim).



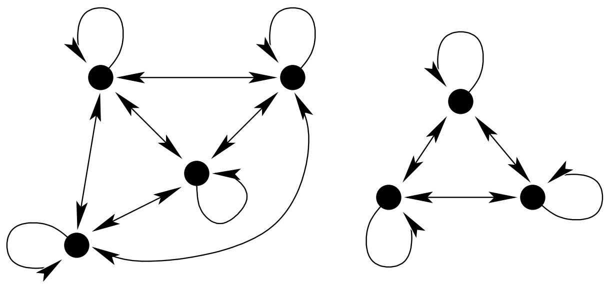
Relacja jest antysymetryczna, gdy krawędzie między (różnymi) wierzchołkami mogą być tylko jednokierunkowe.



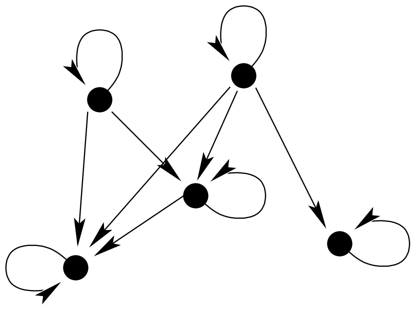
Relacja jest przechodnia, jeżeli dla każdej ścieżki w grafie (długości przynajmniej 1) istnieje w nim krawędź łącząca początek ścieżki z jej końcem. Inaczej mówiąc, graf musi zawierać wszystkie możliwe krawędzie idące „na skróty".



Relacja jest relacją równoważności, jeżeli graf jest podzielony na ileś spójnych składowych, każda składowa to klika (tzn. między każdymi dwoma wierzchołkami tej samej składowej mamy krawędź), natomiast między wierzchołkami należącymi do różnych składowych nie mamy krawędzi. Spójne składowe takiego grafu są nazywane *klasami abstrakcji*.



Relacja jest częściowym porządkiem, jeżeli jest zwrotna, przechodnia i nie zawiera cykli (z wyjątkiem pętelek, które są we wszystkich wierzchołkach).



Pojęcia zwrotności, przechodniości i symetryczności wszystkie polegają na tym, że pewne pary/krawędzie muszą być obecne. Będziemy mówić o odpowiednim *domknięciu* relacji, jako o relacji powstałej przez dodanie odpowiednich par/krawędzi, niezbędnych do spełnienia określonej własności. Szczególnie przydatne będzie nam domknięcie zwrotno-przechodnie.

Def. 18. Niech $\rho \subseteq X \times X$ będzie dowolną relacją binarną. *Domknięciem zwrotno-przechodnim* relacji ρ nazywamy najmniejszą taką relację $\rho' \subseteq X \times X$, która:

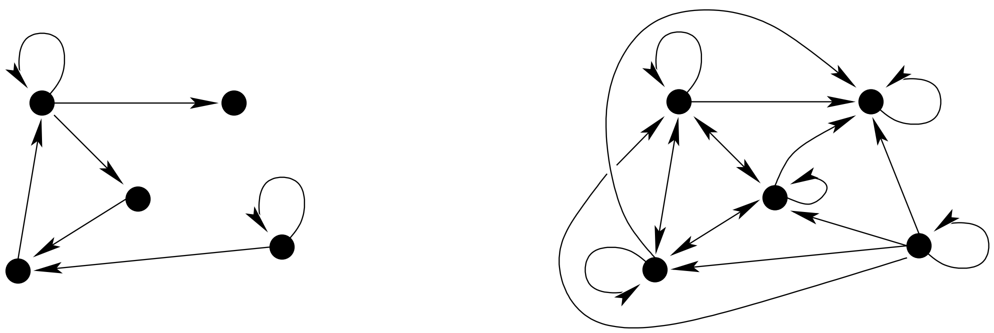
- zawiera relację ρ, $\rho \subseteq \rho'$,
- jest zwrotna i przechodnia.

$\square$

Podobnie definiujemy domknięcie przechodnie, czy symetryczne.

Intuicyjnie, domknięcie zwrotno-przechodnie polega na dodaniu wszystkich takich krawędzi, które w oryginalnym grafie mogły być realizowane przez ścieżki — łącznie z pętelkami, które odpowiadają ścieżkom długości 0.

Przykład: Relacja i jej domknięcie zwrotno-przechodnie:



1.5 Podsumowanie

W tym wykładzie poznaliśmy podstawowe pojęcia dotyczące słów i języków. Przypomnieliśmy też podstawowe informacje na temat relacji, które będą nam potrzebne w dalszych wykładach.

1.6 Skorowidz

- **Alfabet** to dowolny niepusty, skończony zbiór. Elementy alfabetu nazywamy znakami.

- **Domknięcie Kleene'ego** języka A, oznaczane jako A^*, to $A^* = \bigcup_{n \geq 0} A^n$. Inaczej mówiąc, A^* to język zawierający wszystkie możliwe sklejenia dowolnej liczby (łącznie z 0) słów należących do A.
- **Domknięcie zwrotno-przechodnie** relacji ρ to najmniejsza taka relacja $\rho' \subseteq X \times X$, która: zawiera relację ρ, $\rho \subseteq \rho'$, oraz jest zwrotna i przechodnia.
- **Dopełnienie języka** A, oznaczane jako $\overline{A}$, to $\overline{A} = \Sigma^* \setminus A$.
- **Język** (nad alfabetem Σ), to dowolny zbiór słów (nad alfabetem Σ).
- **Palindrom** to takie słowo x, które czytane wprost i wspak jest takie samo, $x = \mathrm{rev}(x)$.
- **Podsłowo** danego słowa to dowolny jego spójny fragment, tzn. powiemy, że x jest podsłowem y wtw., gdy istnieją takie v i w, że $vxw = y$.
- **Prefiks** słowa to dowolny jego początkowy fragment, tzn. x jest prefiksem y wtw., gdy istnieje takie z, że $xz = y$. Jeżeli ponadto $x \neq \varepsilon$ i $x \neq y$, to mówimy, że x jest *właściwym* prefiksem y.
- **Problem decyzyjny** to funkcja, która możliwym danym wejściowym przyporządkowuje wartości logiczne tak/nie.
- **Sklejenie (konkatenację) języków** to język zawierający wszystkie możliwe sklejenia słów z A ze słowami z B, $AB = \{xy : x \in A, y \in B\}$.
- **Sklejenie (konkatenacja) słów** to słowo powstałe z połączenia słów, tak jakby zostały one zapisane kolejno po sobie.
- **Słowo** (lub napis) nad alfabetem Σ, to dowolny skończony ciąg znaków z Σ.
- **Sufiks** słowa to dowolny jego końcowy fragment, tzn. x jest sufiksem y wtw., gdy istnieje takie z, że $zx = y$. Jeżeli ponadto $x \neq \varepsilon$ i $x \neq y$, to mówimy, że x jest *właściwym* sufiksem y.

1.7 Praca domowa

1. (2 p.) Jaka jest różnica między $\emptyset$ i $\{\varepsilon\}$? Ile słów zawierają te języki? Czy są elementami neutralnymi (czyli nie zmieniają wyniku) jakichś operacji na językach?
2. (2 p.) Podaj które prefiksy słowa *bbabbbaabbab* są równocześnie jego sufiksami.
3. (2 p.) Podaj wszystkie słowa należące do języka $\{ab, abb, bab\}\{ab, bab\}$.
4. (2 p.) Podaj wszystkie słowa należące do języka $\{a, ab, ba, bbb, baba\} \setminus \{a\}^*\{b\}^*$.
5. (2 p.) Podaj wszystkie słowa należące do języka $\{a, ab, abba, baba\} \cap \{a\}^*\{b\}^+\{a\}^*$.

1.8 Ćwiczenia

1. Niech A, B i C będą dowolnymi zbiorami. Dla każdego zestawu zbiorów poniżej określ, czy są one sobie zawsze równe, jeden zawiera się w drugim, lub nie da się ich porównać. Porównując zbiory możesz przedstawić je na diagramie.

   (a) $(A \cup B) \setminus (A \cap B)$, $(A \setminus B) \cup (B \setminus A)$, Rozwiązanie[1]

   (b) $(C \setminus A) \cap (C \setminus B)$, $C \setminus (A \cap B)$, Rozwiązanie[2]

   (c) $(C \setminus A) \cap (C \setminus B)$, $C \setminus (A \cup B)$, Rozwiązanie[3]

   (d) $(C \setminus A) \cup (C \setminus B)$, $C \setminus (A \cap B)$, Rozwiązanie[4]

   (e) $(A \setminus B) \cup (B \setminus A)$, $(A \cup B) \setminus (A \cap B)$ Rozwiązanie[5]

   (f) $C \setminus ((C \setminus A) \cup (C \setminus B))$, $A \cap B$ Rozwiązanie[6]

   (g) $(A \setminus B) \cup (B \setminus C) \cup (C \setminus A)$, $A \cup B \cup C$, Rozwiązanie[7]

2. Wybieramy dowolny numer telefonu i traktujemy go jak słowo złożone z cyfr. Podaj:

   - wszystkie prefiksy tego ciągu,
   - wszystkie sufiksy tego ciągu,
   - kilka podciągów (niekoniecznie spójnych); ile jest wszystkich?
   - kilka spójnych podciągów (podsłów); ile jest wszystkich?
   - podsłowa, które występują więcej niż raz w danym numerze.

3. Wybieramy zbiór 2–3 elementowy. Jakie są jego podzbiory? Ile podzbiorów ma zbiór n-elementowy? Rozwiązanie[8]

---

1 $(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$

2 $(C \setminus A) \cap (C \setminus B) \subseteq C \setminus (A \cap B)$

3 $(C \setminus A) \cap (C \setminus B) = C \setminus (A \cup B)$

4 $(C \setminus A) \cup (C \setminus B) = C \setminus (A \cap B)$

5 $(A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$

6 $C \setminus ((C \setminus A) \cup (C \setminus B)) \subseteq A \cap B$

7 $(A \setminus B) \cup (B \setminus C) \cup (C \setminus A) \subseteq A \cup B \cup C$

8 Zbiór n elementowy ma 2^n podzbiorów. Na przykład, wszystkie podzbiory zbioru $\{a, b\}$, to: $\emptyset$, $\{a\}$, $\{b\}$ i $\{a, b\}$.

4. Podaj które prefiksy słowa *ababbbabab* są równocześnie jego sufiksami. Rozwiązanie[9]

5. Które z poniższych tożsamości są prawdziwe? W przypadku prawdziwych uzasadnij, a w przypadku fałszywych podaj przykład słowa, które przeczy tożsamości.

   - $(A \cup B) \cup C = A \cup (B \cup C)$, Rozwiązanie[10]
   - $(A \cup B) \cap C = A \cup (B \cap C)$, Rozwiązanie[11]
   - $(A \cup B) \cap C \subseteq A \cup (B \cap C)$, Rozwiązanie[12]
   - $A(B \cup C) = AB \cup AC$, Rozwiązanie[13]
   - $A(B \cap C) = AB \cap AC$, Rozwiązanie[14]
   - $A(B \cap C) \subseteq AB \cap AC$, Rozwiązanie[15]
   - $(AB)^* = A^*B^*$, Rozwiązanie[16]
   - $(A \cup B)^* = (A^*B^*)^*$, Rozwiązanie[17]
   - $(A^+)^+ = A^+$, Rozwiązanie[18]

---

[9]

[10] Są to: ε, *ab*, *abab* i *ababbbabab*.

[11] Prawda, suma zbiorów jest operacją łączną.

[12] Fałsz. Zbiór $A \setminus C$ jest zawarty w drugim zbiorze, ale nie w pierwszym.

[13] To już jest prawda.

$$(A \cup B) \cap C = (A \cap C) \cup (B \cap C) \subseteq A \cup (B \cap C)$$

[14] Prawda, wynika to wprost z definicji sklejania języków.

[15] Fałsz. Jeżeli pewne słowo należy zarówno do AB i AC, ale zostało uzyskane na różne sposoby, to może nie należeć do $A(B \cap C)$. Na przykład, dla $A = \{a, ab\}$, $B = \{b\}$ i $C = \{bb\}$ mamy:

$$A(B \cap C) = \{a, ab\}\emptyset = \emptyset$$

$$AB \cap AC = \{a, ab\}\{b\} \cap \{a, ab\}\{bb\} = \{ab, abb\} \cap \{abb, abbb\} = \{abb\}$$

[16] A to już jest prawda. Jeżeli pewne słowo należy do $A(B \cap C)$, to należy zarówno do AB, jak i AC.

[17] Fałsz. Pierwszy język zawiera każdy z języków postaci $ABABAB \ldots AB$, a drugi każdy język postaci $AAA \ldots ABB \ldots B$.

[18] Prawda. Obydwa języki zawierają wszystkie słowa jakie możemy uzyskać sklejając słowa z A i/lub B.

Prawda. Powtarzanie powtarzania nic nowego nie wnosi.

- $A^{*+} = A^{+*}$, Rozwiązanie[19]
- $\overline{(A \cup B)} = \overline{A} \cap \overline{B}$. Rozwiązanie[20]

6. Co to za języki:

- $(A \cap A^*)^*$, Rozwiązanie[21]
- $A^* \setminus A^+$, Rozwiązanie[22]
- $(A \cup B)^* \setminus (A^* \cup B^*)$, Rozwiązanie[23]
- $\{aa, ba, bb, ab\}^*$, Rozwiązanie[24]
- $(\Sigma\Sigma)^*$, Rozwiązanie[25]
- $\overline{(\Sigma\Sigma)^*}$, Rozwiązanie[26]
- $\Sigma(\Sigma\Sigma)^*$ Rozwiązanie[27]
- ΠΙΣΣΑ Rozwiązanie[28]

7. Podaj wszystkie słowa należące do języka:

- $\{aba, ba, a\}\{ba, baba\}$, Rozwiązanie[29]

---

[19]

[20] Prawda. Powtarzanie powtarzania nic nowego nie wnosi, a w obu przypadkach sklejamy A z sobą dowolną liczbę razy (łącznie z zero).

[21] Prawda. Jest to wersja prawa De Morgana.

[22] $A \subseteq A^*$, a więc $A \cap A^* = A$. Jest to więc A^*.

[23] Jeśli $\varepsilon \in A$, to jest to $\emptyset$, gdyż $A^* = A^+$. W przeciwnym przypadku jest to $\{\varepsilon\}$.

[24] Język ten zawiera słowa powstałe ze sklejenia słów z A i B, przy czym konieczne jest wzięcie słów z obu tych języków.

[25] Słowa (nad alfabetem $\{a, b\}$) parzystej długości.

[26] Słowa (nad alfabetem Σ) parzystej długości.

[27] Słowa (nad alfabetem Σ) nieparzystej długości.

[28] Również słowa (nad alfabetem Σ) nieparzystej długości.

[29] A to jest po grecku pizza. :-)

Są to: $aba, baba, ababa, bababa, abababa$.

- $\{a, ab, ba, baba\} \cap \{ab, ba\}^2$, Rozwiązanie[30]
- $\{a, ab, ba\} \setminus \{a\}^*\{bb\}^*\{a\}^*$, Rozwiązanie[31]

8. Używając symboli i operacji wprowadzonych na pierwszym wykładzie (czyli operacji na słowach, zbiorach, językach i matematycznych definicji zbiorów) zdefiniuj podane języki (nad alfabetem Σ):
   - słowa zawierające tyle samo liter a i b, Rozwiązanie[32]
   - słowa nad alfabetem $\{a, b\}$ (nie)parzystej długości, Rozwiązanie[33]
   - palindromy, Rozwiązanie[34]
   - słowa długości 42. Rozwiązanie[35]

9. Podać domknięcie zwrotno-przechodnie i domknięcie symetryczne relacji:

$$\{(1,2),(2,3),(3,1),(2,4),(5,3)\}$$

   określonej na zbiorze $\{1, 2, \ldots, 6\}$. Rozwiązanie[36]

10. Pokazać, że następujące relacje są relacjami równoważności:
    - słowa x i y są w relacji wtw., gdy $|x| = |y|$,
    - słowa x i y są anagramami,
    - języki A i B są w relacji wtw., gdy $A^* = B^*$.

11. Pokazać, że następujące relacje są częściowymi porządkami:
    - relacja zawierania $\subseteq$ na językach nad ustalonym alfabetem,

---

[30] Jest to tylko jedno słowo: $baba$.

[31] Są to: ab i ba.

[32] $\{w \in \Sigma^* : \#_a(w) = \#_b(w)\}$

[33] Parzystej długości: $(\{a,b\}\{a,b\})^*$. Nieparzystej długości: $\{a,b\}(\{a,b\}\{a,b\})^*$.

[34] $\{w \in \Sigma^* : w = \mathrm{rev}(w)\}$

[35] $\{w \in \Sigma^* : |w| = 42\}$

[36] Domknięcie zwortno-przechodnie to: $\{(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4),(4,4),(5,1),(5,2),(5,3),(5,4),(5,5),(6,6)\}$.
Domknięcie synetryczne to: $\{(1,2),(1,3),(2,1),(2,3),(2,4),(3,1),(3,2),(3,5),(4,2),(5,3)\}$.

- relacja porządku leksykograficznego (alfabetycznego) na słowach,
- relacja bycia prefiksem,
- relacja $\prec$, określona na językach nad ustalonym alfabetem w następujący sposób: $A \prec B$ wtw., gdy istnieje taki język C, że $AC = B$.

Wykład 2. Wzorce i wyrażenia regularne

2.1 Wstęp

Wyrażenia regularne to rodzaj wzorców, do których dopasowuje się fragmenty różnych tekstów. Nazwy tej używa się w dwóch kontekstach. Pierwszy z nich dotyczy wyrażeń regularnych, jakie można znaleźć np. przy wyszukiwaniu tekstu, w bardziej rozbudowanych edytorach tekstów. Są one również szeroko stosowane w Uniksie, w różnego rodzaju programach do wyszukiwania i przetwarzania informacji tekstowych, np. `grep`, `awk` i `sed`, a także w generatorze analizatorów leksykalnych Lex.

Terminu tego używa się również w teorii języków formalnych w podobnym znaczeniu. Jednak w tym przypadku liczba konstrukcji jakie mogą być używane do budowy wyrażeń regularnych jest ograniczona do absolutnego minimum.

Jest to poniekąd naturalne, gdyż oba rodzaje wyrażeń regularnych służą innym celom. W pierwszym przypadku ma to być wygodny w użyciu język, pozwalający na zwięzłe zapisywanie wzorców. Stąd jest od dosyć rozbudowany i zawiera wiele konstrukcji. W drugim przypadku bardziej nas interesuje badanie samych wyrażeń regularnych i ich siły wyrazu. Stąd język tych wyrażeń regularnych jest maksymalnie uproszczony, a wszystkie konstrukcje, które można wyrazić w inny sposób zostały z niego usunięte.

W trakcie tego kursu będziemy używali obu rodzajów „wyrażeń regularnych". Musimy więc nadać im różne nazwy. Wyrażenia regularne takie, jak są używane w Lex'ie, będziemy nazywać *wzorcami*. Natomiast sam termin „wyrażenia regularne" zarezerwujemy dla takiego jego znaczenia, jakie ma w teorii języków formalnych.

Na potrzeby tego rozdziału ustalamy jako domyślny alfabet Σ zbiór znaków ASCII.

2.2 Wzorce

Poniżej podajemy definicję wzorców. Z jednej strony jest ona wzorowana na wyrażeniach regularnych w Lex'ie, ale nie wymieniliśmy tutaj wszystkich dostępnych tam konstrukcji. Z drugiej strony dodaliśmy dwie konstrukcje, które nie są dostępne w Lex'ie. Definicja ta, oprócz składni wzorców, opisuje też nieformalnie ich semantykę. Semantyka ta jest bardziej formalnie zdefiniowana dalej.

Def. 19. *Wzorce*, to wyrażenia, które możemy budować w podany poniżej sposób.

- dla $a \in \Sigma$, a jest wzorcem, do którego pasuje tylko słowo a (chyba, że a ma jakieś specjalne znaczenie opisane poniżej),
- ε jest wzorcem, do którego pasuje tylko słowo puste (ta konstrukcja nie jest dostępna w Lex'ie),
- $\emptyset$ jest wzorcem, do którego nie pasuje żadne słowo (ta konstrukcja nie jest dostępna w Lex'ie),
- . to wzorzec, do którego pasuje dowolny znak (oprócz końca wiersza),

- "x" to wzorzec, do którego pasuje tylko słowo x (nawet jak zawiera jakieś znaki o specjalnym znaczeniu),
- $\backslash a$ — ma analogiczne znaczenie jak w C, np. $\backslash n$ oznacza znak nowej linii,
- $[\ldots]$ to wzorzec, do którego pasuje dowolny ze znaków wymienionych w kwadratowych nawiasach, np. do $[abc]$ pasuje dowolny ze znaków a, b i c,
- $[a-b]$ to wzorzec, do którego pasuje dowolny znak od a do b, zakresy takie można łączyć, np. $[a-zA-Z]$ to wzorzec, do którego pasuje dowolna litera, mała lub wielka,
- $[\hat{}\ldots]$ to wzorzec, do którego pasuje dowolny znak oprócz znaków wymienionych wewnątrz kwadratowych nawiasów, np. do $[\hat{}xyz]$ pasuje dowolny znak oprócz x, y i z,
- jeśli α jest wzorcem, to $\alpha?$ jest wzorcem, do którego pasuje ε oraz wszystkie te słowa, które pasują do α,
- jeśli α jest wzorcem, to α^* jest wzorcem, do którego pasuje sklejenie zera lub więcej słów pasujących do α,
- jeśli α jest wzorcem, to α^+ jest wzorcem, do którego pasuje sklejenie jednego lub więcej słów pasujących do α,
- jeśli α i β są wzorcami, to $\alpha\beta$ jest wzorcem, do którego pasują sklejenia słów pasujących do α i słów pasujących do β,
- jeśli α i β są wzorcami, to $\alpha \mid \beta$ jest wzorcem, do którego pasują te słowa, które pasują do α lub do β,
- jeśli α i β są wzorcami, to $\alpha \cap \beta$ jest wzorcem, do którego pasują te słowa, które pasują równocześnie do α i β (ta konstrukcja nie jest dostępna w Lex'ie),
- jeśli α jest wzorcem, to $\overline{\alpha}$ jest wzorcem, do którego pasują wszystkie te słowa, które nie pasują do α (ta konstrukcja nie jest dostępna w Lex'ie),
- jeśli α jest wzorcem, to jest nim również (α) i pasują do niego te same słowa, co do α — inaczej mówiąc, do budowy wzorców możemy używać nawiasów,

□

Przykład: Oto garść przykładowych wzorców:

- $[0-9]^+$ opisuje (niepuste) ciągi cyfr, czyli zapisy dziesiętne liczb naturalnych.
- Do wzorca $(a^*bba^*) \cap (ab \mid ba)^*$ pasuje tylko słowo $abba$. Do a^*bba^* pasują słowa zawierające dokładnie dwie litery b i to położone obok siebie, a do $(ab \mid ba)^*$ pasują słowa zbudowane z „cegiełek” ab i ba. Jedynym słowem, które pasuje do obydwu tych wzorców jest właśnie $abba$.

- Wzorzec $[A-Z][A-Z][0-9][0-9][0-9][0-9][0-9]$ opisuje jedną z form numerów rejestracyjnych, złożonych z dwóch liter i pięciu cyfr.

Zdefiniujemy teraz bardziej formalnie semantykę wzorców.

Def. 20. Niech α i β będą wzorcami. Przez $L(\alpha)$ oznaczamy język opisywany przez wzorzec α, czyli język złożony z tych słów, które pasują do wzorca α. $L(\alpha)$ definiujemy indukcyjnie ze względu na budowę wzorca α:

- $L(a) = \{a\}$, dla $a \in \Sigma$, o ile a nie ma specjalnego znaczenia opisanego niżej,
- $L(\varepsilon) = \{\varepsilon\}$,
- $L(\emptyset) = \emptyset$,
- $L(.) = \Sigma \setminus \{\backslash n\}$,
- $L(\text{“}x\text{”}) = \{x\}$,
- $L([a_1 a_2 \dots a_k]) = \{a_1, a_2, \dots, a_k\}$,
- $L([a-b]) = \{a, \dots, b\}$ — zależy od kolejności znaków w kodowaniu ASCII,
- $L([\hat{}\,a-b]) = \Sigma \setminus L([a-b])$
- $L(\alpha?) = L(\alpha) \cup \{\varepsilon\}$,
- $L(\alpha^*) = L(\alpha)^*$,
- $L(\alpha^+) = L(\alpha)^+ = L(\alpha)L(\alpha)^*$,
- $L(\alpha\beta) = L(\alpha)L(\beta)$,
- $L(\alpha \mid \beta) = L(\alpha) \cup L(\beta)$,
- $L(\alpha \cap \beta) = L(\alpha) \cap L(\beta)$,
- $L(\overline{\alpha}) = \overline{L(\alpha)}$,
- $L(\,(\alpha)\,) = L(\alpha)$.

□

Badając wzorce często będziemy chcieli porównywać nie tyle je same, ale ich znaczenia. Przyda nam się do tego pojęcie równoważności wzorców.

Def. 21. Powiemy, że dwa wzorce α i β są równoważne, $\alpha \equiv \beta$, wtw., gdy $L(\alpha) = L(\beta)$. □

Oto garść tożsamości dotyczących wzorców. Niektóre z nich, tak naprawdę, poznaliśmy już w poprzednim wykładzie jako tożsamości dotyczące języków.

- $\alpha(\beta \mid \gamma) \equiv \alpha\beta \mid \alpha\gamma$ — to nic innego jak rozdzielność sklejania języków względem ich sumowania, zastosowana do wzorców,
- $(\beta \mid \gamma)\alpha \equiv \beta\alpha \mid \gamma\alpha$ — j.w.,
- $\emptyset\alpha \equiv \alpha\emptyset \equiv \emptyset$ — $\emptyset$ jest „zerem" sklejania, również dla wzorców,
- $\alpha^* \equiv \varepsilon \mid \alpha\alpha^* \equiv \varepsilon \mid \alpha^*\alpha$ — oto dwie rekurencyjne definicje operacji *,
- $\alpha^+ \equiv \alpha \mid \alpha\alpha^+ \equiv \alpha \mid \alpha^+\alpha$ — oto dwie rekurencyjne definicje operacji $^+$,
- $(\alpha\beta)^*\alpha \equiv \alpha(\beta\alpha)^*$, — oba wzorce opisują naprzemienne ciągi α i β zaczynające i kończące się α-ą,
- $(\alpha^*)^* \equiv \alpha^*$ — domykanie domknięcia Kleene'ego nie wnosi nic dodatkowego,
- $(\alpha \mid \beta)^* \equiv (\alpha^*\beta^*)^*$ — ciąg α i β można zawsze przedstawić jako poprzeplatane ze sobą ciągi α i ciągi β (wliczając w to ciągi długości zero), i vice versa.

2.3 Wyrażenia regularne

Okazuje się, że wiele z konstrukcji występujących we wzorcach może być zastąpionych prostszymi konstrukcjami. Na przykład zamiast $[a-c]$ można napisać $a \mid b \mid c$. Postępując w ten sposób, każdy wzorzec można przerobić na taki, który został zapisany przy użyciu tylko pewnego minimalnego zestawu konstrukcji. Wzorce, które możemy zapisać przy użyciu owego minimalnego zestawu operacji, nazywamy *wyrażeniami regularnymi*.

Def. 22. Wyrażenia regularne, to takie wzorce, które są zbudowane tylko przy użyciu:

- ε,
- $\emptyset$,
- a, dla $a \in \Sigma$,
- $\alpha\beta$, gdzie α i β to wyrażenia regularne,
- $\alpha \mid \beta$, gdzie α i β to wyrażenia regularne,
- α^*, gdzie α to wyrażenie regularne,
- (α), gdzie α to wyrażenie regularne.

$\square$

Jak zobaczymy dalej, wszystkie pozostałe konstrukcje są redundantne, choć w praktyce są przydatne. Wyrażenia regularne będą nam potrzebne wówczas, gdy będziemy się zajmowali tym, co można wyrazić za pomocą wzorców i ogólnymi właściwościami języków, które można opisywać za pomocą wzorców. Dzięki ograniczeniu zestawu możliwych konstrukcji do niezbędnego minimum, nasze rozważania będą prostsze.

2.4 Języki regularne

Klasę języków, dla których istnieją opisujące je wzorce nazywamy *językami regularnymi.*

Def. 23. Powiemy, że język A jest *regularny*, wtw., gdy istnieje wzorzec α opisujący A, czyli $A = L(\alpha)$. □

Jak później pokażemy, wzorce mają taką samą siłę wyrazu co wyrażenia regularne, tzn. jeśli dla danego języka istnieje opisujący go wzorzec, to istnieje również opisujące go wyrażenie regularne.

Jeden z problemów, jakim będziemy się zajmować w trakcie tego kursu, to: jakie języki są regularne? Okazuje się, że nie wszystkie. Poniżej podajemy kilka prostych własności klasy języków regularnych.

Fakt 1. *Każdy język skończony jest regularny.*

Dowód: Niech $\{x_1, x_2, \dots, x_n\}$ będzie niepustym skończonym językiem. Język ten można opisać wzorcem (a dokładniej wyrażeniem regularnym) postaci $x_1 \mid x_2 \mid \dots \mid x_n$. Język pusty można opisać wzorcem $\emptyset$. □

Fakt 2. *Niech A i B będą językami regularnymi. Języki $A \cup B$, $A \cap B$, AB, $\overline{A}$ i A^* są też regularne.*

Dowód: Skoro A i B są regularne, to istnieją opisujące je wzorce α i β, $L(\alpha) = A$, $L(\beta) = B$. Interesujące nas języki są regularne, bo opisują je wzorce: $\alpha \mid \beta$, $\alpha \cap \beta$, $\alpha\beta$, $\overline{\alpha}$ i α^*. □

2.5 Podsumowanie

W tym wykładzie poznaliśmy wzorce oraz wyrażenia regularne. Języki jakie można za ich pomocą opisać tworzą klasę języków regularnych.

2.6 Skorowidz

- **Język opisywany** przez wzorzec to język złożony z tych słów, które pasują do wzorca α.
- **Język regularny**, to taki język, dla którego istnieje opisujący go wzorzec.
- **Wyrażenia regularne** to szczególny przypadek wzorców, do których zapisu użyto wyłącznie: symboli alfabetu, ε, $\emptyset$, sklejania, $\mid$, * i nawiasów.
- **Wzorce** to rodzaj wyrażeń, do których dopasowujemy słowa nad ustalonym alfabetem. Wzorzec opisuje język złożony ze słów, które do niego pasują.

2.7 Praca domowa

1. (3 p.) Podaj wyrażenie regularne równoważne wzorcowi: $(a(b?)a)?(a[a-c]a)^+$
2. (3 p.) Podaj wzorzec opisujący numery telefonów (stacjonarne i komórkowe). (Możesz pominąć numery międzynarodowe, specjalne itp.)
3. (4 p.) Uprość następujący wzorzec: $(a^*(aba \mid abb) \mid b^*(bab \mid baa))^*(a(a^*ba \mid a^*bb) \mid b(b^*ab \mid b^*aa))$

2.8 Ćwiczenia

1. Podaj wzorce/wyrażenia regularne opisujące język złożony ze słów:
   (a) poprawne numery indeksów studentów, np. postaci `s0124`, Rozwiązanie[37]
   (b) poprawne numery dowodów osobistych, np. `ABC123456`. Rozwiązanie[38]
   (c) nad alfabetem $\{a, b\}$, które zawierają podsłowo $bbab$, Rozwiązanie[39]
   (d) nad alfabetem $\{a, b, c\}$, które zaczynają się i kończą tym samym znakiem, Rozwiązanie[40]
   (e) nad alfabetem $\{a, b\}$, które nie zawierają podsłowa aa, Rozwiązanie[41]
   (f) nad alfabetem $\{a, b\}$, które nie zawierają podsłowa ab, Rozwiązanie[42]
   (g) nad alfabetem $\{a, b, c\}$, które nie zawierają podsłowa aa, Rozwiązanie[43]
   (h) nad alfabetem $\{a, b, c\}$, które nie zawierają podsłowa ab, Rozwiązanie[44]

---

[37] $s[0-9][0-9][0-9][0-9]$

[38] $[A-Z][A-Z][A-Z][0-9][0-9][0-9][0-9][0-9][0-9]$

[39] $(a \mid b)^*bbab(a \mid b)^*$

[40] Ten znak to może być a, b lub c. Należy więc wziąć sumę tych trzech przypadków, nie zapominając o słowach jednoliterowych:

$$a(a \mid b \mid c)^*a \mid b(a \mid b \mid c)^*b \mid c(a \mid b \mid c)^*c \mid a \mid b \mid c$$

[41] Po każdej literze a (być może z wyjątkiem ostatniej) musi występować litera b: $(b \mid ab)^*(a \mid \varepsilon)$

[42] Po literze a nie może występować b, tylko ew. kolejna litera a. Tak więc najpierw muszą występować litery b, a potem a: b^*a^*.

[43] Po każdej literze a (być może z wyjątkiem ostatniej) musi występować litera b lub c: $(b \mid c \mid ab \mid ac)^*a?$.

[44] Po każdej sekwencji liter a (być może z wyjątkiem ostatniej) może wystąpić tylko litera c: $(b \mid a^*c)^*a^*$.

(i) nad alfabetem $\{a, b, c\}$ złożonych tylko z jednego rodzaju symboli, Rozwiązanie[45]

(j) nad alfabetem $\{a, b, c\}$ złożonych co najwyżej z dwóch rodzai symboli, Rozwiązanie[46]

(k) nad alfabetem $\{a, b, c\}$ złożonych dokładnie z dwóch rodzai symboli, Rozwiązanie[47]

(l) nad alfabetem $\{a, b\}$, które zawierają parzystą liczbę liter a. Rozwiązanie[48]

2. Porównaj podane wzorce pod kątem równoważności/zawierania się odpowiadających im języków. Jeśli są równoważne, to uzasadnij to. Jeżeli nie, to podaj przykład słowa, które pasuje do jednego wzorca, ale nie do drugiego.

   (a) $(a?b?)^*$ i $(a \mid b)^*$, Rozwiązanie[49]

   (b) $(a \mid b)^*$ i $a^* \mid b^*$, Rozwiązanie[50]

   (c) $(a^*b)^*$ i $(b^*a)^*$, Rozwiązanie[51]

   (d) $(a \mid b)^*$ i $(aa \mid ab \mid ba \mid bb)^*$, Rozwiązanie[52]

   (e) $aa(bbaa \mid baa)^*$ i $(aabb \mid aab)^*aa$. Rozwiązanie[53]

---

45 Pamiętajmy, że puste słowo składa się z zera rodzajów symboli, $a^+ \mid b^+ \mid c^+$.

46 Któregoś rodzaju literek musi brakować. Należy więc zsumować trzy przypadki: $(a \mid b)^* \mid (b \mid c)^* \mid (a \mid c)^*$.

47 Mamy trzy przypadki — któregoś rodzaju literek musi brakować. Równocześnie musimy zapewnić, aby dwie pozostałe literki występowały. Tak więc w słowie musi być przynajmniej po jednej sekwencji literek pierwszego i drugiego rodzaju.

$$((a^+b^+ \mid b^+a^+)(a \mid b)^*) \mid ((b^+c^+ \mid c^+b^+)(b \mid c)^*) \mid ((a^+c^+ \mid c^+a^+)(a \mid c)^*)$$

48 Ciąg liter a parzystej długości łatwo opisać: $(aa)^*$. Następnie należy „powciskać” litery b wszędzie tam, gdzie mogą się pojawić. W ten sposób uzyskujemy: $b^*(ab^*ab^*)^*$.

49 Są tożsame. Oba opisują wszystkie słowa nad alfabetem $\{a, b\}$.

50 Pierwszy wzorzec opisuje wszystkie słowa nad alfabetem $\{a, b\}$. Do drugiego wzorca nie pasuje np. słowo ba.

51 Nie są tożsame. Na przykład, do pierwszego z nich nie pasuje słowo a, a do drugiego nie pasuje słowo b.

52 Nie są tożsame. Pierwszy wzorzec opisuje wszystkie słowa nad alfabetem $\{a, b\}$, a drugi tylko parzystej długości.

53 Są tożsame. Jest to przykład bardziej ogólnej tożsamości: $\alpha(\beta\alpha)^* \equiv (\alpha\beta)^*\alpha$. Oba te wyrażenia regularne opisują słowa postaci: $\alpha\beta\alpha\beta\ldots\alpha\beta\alpha$.

3. Podaj wyrażenie regularne równoważne wzorcowi:
   - $((ab^+)? \mid [a-z])^*$, Rozwiązanie[54]
   - $([a-c] \mid a(b?))^+$. Rozwiązanie[55]
4. Uprość podane wzorce:
   - $(a \mid b)(aa \mid ab \mid bb \mid ba)^*(a \mid b)$, Rozwiązanie[56]
   - $(a(b \mid \varepsilon) \mid ba(b \mid \varepsilon))^+ \mid \varepsilon$. Rozwiązanie[57]

---

[54] Jedno z możliwych rozwiązań to: $(ab(b^*) \mid \varepsilon \mid x \mid y \mid z)^*$.

[55] Jedno z możliwych rozwiązań to: $(a \mid b \mid c \mid ab)(a \mid b \mid c \mid ab)^*$.

[56]

$$\begin{aligned}(a \mid b)(aa \mid ab \mid bb \mid ba)^*(a \mid b) &\equiv (a \mid b)((a \mid b)(a \mid b))^*(a \mid b) \equiv \\ &\equiv ((a \mid b)(a \mid b))^*(a \mid b)(a \mid b) \equiv ((a \mid b)(a \mid b))^+\end{aligned}$$

[57]

$$\begin{aligned}(a(b \mid \varepsilon) \mid ba(b \mid \varepsilon))^+ \mid \varepsilon &\equiv (a \mid ab \mid ba \mid bab)^+ \mid \varepsilon \equiv \\ &\equiv ((b?)a(b?))^+ \mid \varepsilon \equiv \\ &\equiv ((b?)a(b?))^*\end{aligned}$$

Wykład 3. Analiza leksykalna i generatory analizatorów leksykalnych

Wzorce możemy spotkać w wielu narzędziach — wszędzie tam, gdzie chcemy wyszukiwać pewne fragmenty tekstu, wyszukiwany fragment możemy opisać właśnie za pomocą wzorca:

- w edytorach tekstów często możemy podać nie tylko wyszukiwany tekst, ale właśnie wzorzec,
- programy systemowe służące do wyszukiwania informacji, np. `grep`, `sed`, czy `awk`,
- biblioteki programistyczne zawierające procedury rozpoznające wystąpienia wzorców w tekście,
- generatory analizatorów leksykalnych (skanerów).

Tym ostatnim zastosowaniem zajmiemy się bliżej. Najpierw jednak musimy przybliżyć pojęcie analizy leksykalnej.

3.1 Analiza leksykalna

Miejsce na analizę leksykalną jest wszędzie tam, gdzie wczytujemy dane o określonej składni. Wczytując takie dane, zanim będziemy mogli je przetwarzać, musimy rozpoznać ich składnię. Pomysł polega na tym, aby najpierw wczytywany ciąg znaków podzielić na elementarne cegiełki składniowe —- nazywane *leksemami* — a dopiero dalej analizować ciąg leksemów. Analizator leksykalny (nazywany też skanerem) to wyodrębniony moduł zajmujący się tym zadaniem.

Analiza leksykalna powstała i rozwinęła się w ramach prac nad budową kompilatorów. Wszak kompilator musi najpierw wczytać i zanalizować składnię wczytywanego programu. Jej zastosowania są jednak dużo szersze. Praktycznie analizę leksykalną można zastosować w każdym programie, który wczytuje dane posiadające jakąś składnię, na przykład w: przeglądarkach internetowych, edytorach, systemach składu tekstu, programach konwertujących, czy jakichkolwiek aplikacjach posiadających pliki konfiguracyjne o określonej składni.

Czemu wyodrębniać analizę leksykalną jako osobny moduł? Jest kilka powodów:

- uproszczenie konstrukcji programu — łatwiej opracować osobno analizę leksykalną i resztę analizy składniowej,
- zwiększenie przenośności — w module analizie leksykalnej można ukryć wszystkie szczegóły związane ze sposobem wczytywania plików, reprezentacją znaków, itp.,
- zwiększenie efektywności — analiza leksykalna, choć koncepcyjnie prosta, może zajmować dużą część czasu pracy programu; właściwe jej zaimplementowanie może poprawić efektywność programów,

Napisanie efektywnego skanera nie jest proste. Nie musimy jednak tego robić ręcznie. Praktycznie dla każdego języka programowania istnieją narzędzia, tzw. generatory analizatorów leksykalnych, które zrobią to za nas. Musimy im jedynie dostarczyć specyfikacji opisującej jak wyglądają leksemy, owe elementarne cegiełki, na które chcemy podzielić wczytywany ciąg znaków i jakie jest ich znaczenie.

3.2 Leksemy, żetony i atrybuty

W trakcie analizy leksykalnej wczytywany ciąg znaków jest dzielony na leksemy. Jednak to co jest przekazywane dalej, to nie są dokładnie leksemy. Formalnie, leksem to ciąg znaków. To co jest przekazywane, to informacja reprezentująca znaczenie leksemu. Informacja ta jest reprezentowana za pomocą tzw. *żetonu* i opcjonalnego *atrybutu*. Żeton niesie informację o rodzaju leksemu. Jeżeli leksemy danego rodzaju niosą ze sobą pewną „wartość”, to żetonowi towarzyszy atrybut i jest on równy tej wartości. Podsumujmy więc znaczenie tych trzech terminów:

leksem to ciąg kolejnych znaków stanowiących semantycznie niepodzielną całość,

żeton (ang. *token*) to stała (całkowita) reprezentująca rodzaj wczytanego leksemu,

atrybut to opcjonalna wartość reprezentująca znaczenie leksemu.

Typowe leksemy, to:

- identyfikatory,
- słowa kluczowe,
- napisy,
- liczby całkowite i rzeczywiste,
- operacje arytmetyczne i relacje,
- nawiasy i innego rodzaju znaki „interpunkcyjne”.

Najlepiej powyższe pojęcia zilustrować na przykładach.

Przykład: Rozważmy instrukcję (z kompilowanego programu) postaci:

$$\mathtt{E} := \mathtt{m} * \mathtt{c}\hat{\ }2;$$

Możemy ją rozbić na następujący ciąg leksemów:

$$\mathtt{E},\ :=,\ \mathtt{m},\ *,\ \mathtt{c},\ \hat{\ },\ \mathtt{2},\ ;$$

Natomiast ich reprezentacja za pomocą żetonów i atrybutów będzie następująca:

Leksem	Żeton	Atrybut
`E`	identyfikator	„E”
`:=`	przypisanie	
`m`	identyfikator	„m”
`*`	mnożenie	
`c`	identyfikator	„c”
`^`	potęgowanie	
2	liczba całkowita	2
`;`	średnik	

Przykład: Rozważmy fragment źródeł HTML:

```
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
  <title>PJWSTK - JFA</title>
```

Jak ten fragment należy podzielić na leksemy? Pamiętajmy, że leksemy muszą być niepodzielne semantycznie. Oto jeden z możliwych podziałów:

Leksem	Żeton	Atrybut
`<head`	open-start-tag	„`head`”
`>`	close-tag	
`<meta`	open-start-tag	„`meta`”
`http-equiv`	identyfikator	„`http-equiv`”
`=`	równość	
`"Content-Type"`	napis	„`Content-Type`
`content`	identyfikator	„`content`”
`=`	równość	
`"text/html; charset=iso-8859-2"`	napis	„`text/html; charset=iso-8859-2`”
`>`	close-tag	
`<title`	open-start-tag	„`title`”
`>`	close-tag	
`PJWSTK - JFA`	text	„`PJWSTK - JFA`”
`</title`	open-end-tag	„`title`”
`>`	close-tag	

Jak widać z powyższych przykładów, typowe atrybuty leksemów to

- identyfikator — nazwa identyfikatora (jeśli nie odróżnia się małych i wielkich liter, to w znormalizowanej postaci, np. pisana samymi wielkimi literami),
- napis — treść napisu,
- liczba — jej wartość,

Pisząc specyfikację dla Lex'a musimy dla każdego żetonu opisać jaką postać mogą mieć leksemy odpowiadające temu żetonowi. Robimy to podając dla każdego żetonu wzorzec. Następnie dla każdego żetonu podajemy fragment kodu, który na podstawie leksemu oblicza wartość atrybutu.

Przykład: Oto kilka przykładów żetonów i wzorców opisujących odpowiadające im leksemy:

Żeton	Przykład leksemu	Wzorzec
sł_klucz_if	`if`	`if`
identyfikator (złożony tylko z liter)	`Pom`	$[\mathtt{a}-\mathtt{zA}-\mathtt{Z}]^+$
liczba_całkowita	-42	$-?[\mathtt{0}-\mathtt{9}]^+$

Projektując podział wejścia na leksemy i dobierając żetony powinniśmy kierować się następującymi zasadami:

- Leksemy powinny być niepodzielne semantycznie.
- Leksemom, które mają tego samego rodzaju semantykę powinien odpowiadać ten sam żeton.
- Leksemy odpowiadające jednemu żetonowi powinno się dać opisać wzorcem.

Skaner jest zwykle zrealizowany w postaci modułu, który udostępnia procedurę „daj kolejny leksem". Procedura ta rozpoznaje jeden leksem i zwraca odpowiadający mu żeton i atrybut. Można więc powiedzieć, że skaner przetwarza strumień znaków w strumień par: żeton, atrybut.

Proste atrybuty, takie jak liczby, są przekazywane wprost. Bardziej złożone atrybuty, takie jak nazwy identyfikatorów, mogą być przez skaner umieszczane w specjalnym słowniku, nazywanym tablicą symboli. Wówczas jako atrybut przekazywany jest wskaźnik lub pozycja w tablicy symboli. Z jednej strony, dzięki temu oszczędzamy na pamięci. Z drugiej strony, sama pozycja w tablicy symboli jest wystarczająca, gdyż zwykle nie jest istotne, jak się dany identyfikator nazywa, tylko który z identyfikatorów to jest.

3.3 Generator analizatorów leksykalnych Lex

Generator analizatorów leksykalnych na podstawie specyfikacji skanera sam generuje jego kod. Jest wiele różnych generatorów analizatorów leksykalnych, różnego pochodzenia i przeznaczonych dla różnych języków programowania. Na wykładzie skupimy się na generatorze Flex/Lex przeznaczonym dla C/C++. Nazwa [F]lex to skrót od *[Fast] LEXical analyzer generator*.

Specyfikacja taka zawiera z jednej strony opis składni leksemów różnych rodzajów, a z drugiej zawiera fragmenty kodu, które mają być wykonane w przypadku napotkania leksemu określonego rodzaju.

Dla każdego rodzaju leksemów (tj. dla każdego żetonu) specyfikacja skanera zawiera:

- wzorzec opisujący postać leksemów (danego rodzaju),
- fragment kodu wykonywany w momencie napotkania leksemu danego rodzaju.

Fragmenty kodu wykonywane w momencie rozpoznania leksemu mają oczywiście dostęp do słowa (napisu) stanowiącego rozpoznany leksem. Ich głównym zadaniem jest przekazanie żetonu i obliczenie ew. atrybutu. Jednak z punktu widzenia Lex'a mogą one robić cokolwiek. Jeżeli np. naszym zadaniem jest przetłumaczenie jednej postaci danych na inną, takie fragmenty mogą wypisywać przetłumaczone odpowiedniki wczytanych leksemów.

3.3.1 Definicje nazwanych wzorców

W specyfikacji dla Lex'a zwykle pewne wzorce pojawiają się wielokrotnie. Żeby nie powtarzać ich, wprowadzono możliwość definiowania pomocniczych nazwanych wzorców. Definicje takie mają postać:

```
nazwa     wzorzec
```

Od momentu zdefiniowania nazwanego wzorca, można się do niego odwoływać w innych wzorcach poprzez: `{nazwa}`. W definicjach nazwanych wzorców można się odwoływać do nazwanych wzorców zdefiniowanych wcześniej, ale tylko wcześniej — tym samym definicje wzorców nie mogą być rekurencyjne.

Przykład:

```
cyfra          [0-9]
cyfry          {cyfra}+
cz_ułamkowa    "."{cyfry}
znak_opc       ("+"|"-")?
wykładnik_opc  (E{znak_opc}{cyfry})?
liczba         {znak_opc}({cyfry}{cz_ułamkowa}|{cyfry}|{cz_ułamkowa}){wykładnik_opc}
```

Definicje nazwanych wzorców pozwalają w zwięzły sposób powiązać nazwy z określonymi wzorcami. Nie wnoszą one jednak nic do siły wyrazu wzorców. Definicji nazwanych wzorców można się zawsze pozbyć rozwijając wszystkie definicje.

3.3.2 Specyfikacje skanerów dla Lex'a

Język specyfikacji dla Lex'a zawiera mnóstwo szczegółów, których nie będziemy tutaj omawiać. W tej chwili naszym zadaniem jest wprowadzenie Czytelnika w typowy sposób wykorzystania Lex'a. Gdy zawarte tu informacje okażą się niewystarczające, odsyłamy Czytelnika do dokumentacji generatora skanerów (`man lex`).

Specyfikacja dla Lex'a składa się z trzech części:

- deklaracji programistycznych (stałych i zmiennych), oraz definicji regularnych,

- reguł rozpoznawania leksemów,
- dodatkowych procedur zdefiniowanych przez użytkownika.

Części te są oddzielone od siebie wierszami postaci: „%%".

Deklaracje programistyczne Dowolne globalne **deklaracje** programistyczne można umieścić ujmując je w nawiasy `%{ ...%}`. Dotyczy to również dyrektyw `#include`.

Definicje nazwanych wzorców Zapisane zgodnie z opisaną powyżej składnią.

Reguły rozpoznawania leksemów Reguły rozpoznawania leksemów mają postać:

```
wzorzec     fragment kodu
```

Jeżeli fragment kodu jest dłuższy niż jedna instrukcja, to musi być ujęty w nawiasy `{ ...}`). Wzorzec musi zaczynać się na początku wiersza i nie może zawierać żadnych (nie ujętych w cudzysłowy) odstępów.

Dodatkowe procedury W trzeciej części można umieścić deklaracje własnych procedur. W szczególności można tam zdefiniować procedurę `main`.

3.3.3 Rozpoznawanie leksemów

Zanim zobaczymy przykłady specyfikacji, musimy zrozumieć, jak działają wygenerowane przez Lex'a skanery. Skaner udostępnia funkcję (`int yylex()`) służącą do wczytania jednego leksemu. Jeżeli plik został wczytany, to `yylex` zwraca 0. Wpp. stara się dopasować pewien fragment początkowy wejścia do jednego z podanych wzorców, wg. następujących zasad:

- Jeżeli istnieje kilka możliwych dopasowań leksemu, przede wszystkim wybierany jest jak najdłuższy leksem.
- Jeśli jest kilka wzorców pasujących do najdłuższego możliwego leksemu, to wybierany jest pierwszy z nich (zgodnie z kolejnością w specyfikacji). Następnie wykonywany jest fragment kodu odpowiadający dopasowanemu wzorcowi. Dopasowany leksem jest dostępny poprzez zmienną `yytext` i ma długość `yyleng`. (Zmienna `yytext` może być typu `char yytext[]` lub `char *yytext`.)
- Jeśli wykonywany fragment kodu zakończy się poprzez `return`, to jest to powrót z wywołania procedury `yylex`. Wpp. rozpoczyna się dopasowywanie kolejnego leksemu itd.

  Żeton powinien być zwracany właśnie jako wynik procedury `yylex`. Jeśli chcemy przekazywać atrybuty, to powinniśmy je przypisać (zadeklarowanym w pierwszej części specyfikacji) zmiennym.

- Jeżeli nic się nie udało dopasować, to jeden znak z wejścia jest przepisywany na wyjście i rozpoznawanie leksemów zaczyna się od początku.

Przykład: Oto specyfikacja skanera, który każdy napis „`FOO`” zamienia na „`Foo`”, „`BAR`” na „`Bar`” i „`FOOBAR`” na „`Foobar`”. Wszystko inne jest przepisywane z wejścia na wyjście bez zmian.

```
%{
  #include <stdio.h>
%}

%%

FOO     printf("Foo");
BAR     printf("Bar");
FOOBAR  printf("Foobar");

%%

int main() {
    yylex();
    return 0;
}
```

Przykład ten znajduje się w pliku `foobar.l`.

Przykład: Oto specyfikacja skanera, który wyszukuje w wejściu liczby zmiennopozycyjne i wypisuje je. Wszystko inne jest ignorowane.

```
%{
  #include <stdio.h>
%}

cyfra          [0-9]
cyfry          {cyfra}+
cz_ułamkowa    "."{cyfry}
znak_opc       ("+"|"-")?
wykładnik_opc  ((e|E){znak_opc}{cyfry})?
liczba         {znak_opc}({cyfry}{cz_ułamkowa}|{cyfry}"."?|{cz_ułamkowa}){wykładnik_opc}

%%
```

```
{liczba}      { printf(" %g ", atof(yytext)); }
.             { /* nic */ }

%%

int main() {
  yylex();
  return 0;
}
```

Przykład ten znajduje się w pliku `liczby.l`.

W powyższych przykładach nie używaliśmy żetonów ani atrybutów. Jedno wywołanie funkcji `yylex` przetwarzało całe wejście. Poniższy przykład ilustruje wykorzystanie żetonów i atrybutów.

Przykład: Oto specyfikacja prostego skanera, który sumuje wszystkie liczby całkowite znajdujące się na wejściu, pomijając wszystko inne.

```
%{
  #include <stdio.h>
  #define LICZBA 1
  int attr;
%}

INTEGER -?[0-9]+

%%

{INTEGER} { attr = atoi(yytext);
            return LICZBA; }

.|\n      { /* Nic */ }

%%

int main() {
    int sum=0;

    while (yylex() == LICZBA) sum += attr;
    printf("suma = %d\n", sum);
    return 0;
}
```

Przykład ten znajduje się w pliku `sum.l`.

Przykład: Kolejny przykład pokazuje jak przekazywać inne atrybuty niż liczby całkowite. Poniższy skaner wychwytuje wszystkie identyfikatory i przekazuje ich nazwy jako atrybuty.

```
%{
  #include <stdio.h>
  #include <string.h>
  #define IDENTYFIKATOR 6942
  #define ZNAK 1234
  char* attr;
%}

IDENT [a-zA-Z][a-zA-Z0-9_]*

%%

{IDENT} {
  attr = malloc (yyleng+1);
  strcpy(attr, yytext);
  return IDENTYFIKATOR;
}

.|\n      { return ZNAK; }

%%

int main() {
  int token;

  while ((token=yylex()) > 0) {
    if (token == IDENTYFIKATOR) printf("`%s'\n", attr);
  };
  return 0;
}
```

Przykład ten znajduje się w pliku `identyfikatory.l`.

Przykład: Następująca specyfikacja opisuje skaner, który usuwa komentarze postaci `/* ...*/`. Uwaga: nie uwzględnia ona innych rodzajów komentarzy, ani napisów.

```
%%

"/*"([ˆ∧*]|("*"+[ˆ∧*/]))*"*"*"*/"  { /* Nic */ }

"/*"                              { printf("Nieprawidłowy komentarz."); }

%%
int main() {
    yylex();
    return 0;
}
```

Przykład ten znajduje się w pliku `uncomment.l`.

3.3.4 Tryby

Domyślnie, analizator leksykalny dopasowuje standardowe wejście do wszystkich wzorców reguł podanych w specyfikacji. Można jednak określić kilka *trybów* działania skanera i w określonym trybie dopasowywać wzorce tylko określonych reguł. Tryby deklarujemy w pierwszej części specyfikacji, zaczynając wiersz od `%s` lub `%x` i wymieniając dalej nazwy trybów. Jeżeli tryb został zadeklarowany za pomocą `%s` to obejmuje dodatkowo reguły bez określonego trybu, a jeśli został zadeklarowany za pomocą `%x` to ich nie obejmuje. Jeżeli chcemy, żeby dany wzorzec był dopasowywany tylko w określonym trybie, to dodajemy przed nim nazwy tych trybów ujęte w trójkątne nawiasy `<...>` i pooddzielane przecinkami. Tryb możemy zmieniać za pomocą polecenia `BEGIN(`*tryb*`)`. Tryb, w którym skaner pracuje w momencie uruchomienia i który obejmuje wszystkie reguły bez podanego trybu, nazywa się `INITIAL`. Możliwe jest też sprawdzenie w jakim trybie jesteśmy. Aktualny tryb jest pamiętany na zmiennej `YY_START`.

Przykład: Używając trybów możemy napisać dużo prościej skaner usuwający komentarze postaci `/* ...*/`. Wystarczy, że wprowadzimy dodatkowy tryb `KOMENTARZ`, w którym będziemy analizować treść komentarzy.

```
%x KOMENTARZ

%%

"/*"                    { BEGIN(KOMENTARZ); }
<KOMENTARZ>"*/"         { BEGIN(INITIAL); }
<KOMENTARZ>(.|\ n)      { /* Nic */ }
```

```
%%

int main() {
  yylex();
  if (YY_START == KOMENTARZ) printf("Nieprawidłowy komentarz.");
  return 0;
}
```

Przykład ten znajduje się w pliku `uncomment-start.l`.

3.4 Kompilacja programów generowanych przez Flex'a

Zodnie z przyjętą konwencją, specyfikacje dla Flex'a to pliki z rozszerzeniem `.l`. Załóżmy, że w pliku `spec.l` znajduje się taka specyfikacja. Najprostsze wywołanie Flex'a wygląda tak:

```
flex spec.l
```

W jego wyniku Flex tworzy plik `lex.yy.c` z kodem źródłowym skanera. Możemy oczywiście określić (za pomocą opcji `-o`) inną nazwę pliku z kodem źródłowym, na przykład polecenie:

```
flex  -ospec.yy.c spec.l
```

spowoduje umieszczenie kodu źródłowego skanera w pliku `spec.yy.c`. Tak wygenerowany kod korzysta ze standardowej biblioteki o nazwie `fl`. Można go skompilować za pomocą polecenia postaci: `gcc ...spec.yy.c -lfl`.

3.5 Podsumowanie

Wykład ten wyjaśnia czym jest analiza leksykalna i generatory analizatorów leksykalnych, takie jak Lex. Pokazuje też jak pisać specyfikacje analizatorów leksykalnych dla Lex'a. Umiejętność korzystania z Lex'a będzie jeszcze przedmiotem ćwiczeń laboratoryjnych.

3.6 Skorowidz

- **Analiza leksykalna** polega na wczytaniu strumienia znaków, podzieleniu go na leksemy i przekształceniu na żetony i ew. atrybuty.
- **Atrybut** (leksemu) to opcjonalna wartość reprezentująca znaczenie/wartość leksemu.
- **Definicje nazwanych wzorców** to część specyfikacji dla Lex'a wiążąca z określonymi nazwami określone wzorce.

- **Generator analizatorów leksykalnych** to program generujący, na podstawie specyfikacji, kod analizatora leksykalnego (skanera).
- **Leksem** to ciąg kolejnych znaków stanowiących semantycznie niepodzielną całość.
- **Lex** to generator analizatorów leksykalnych.
- **Skaner** (analizator leksykalny) to moduł zajmujący się analizą leksykalną.
- **Żeton** to stała reprezentująca rodzaj leksemu.

3.7 Praca domowa

1. (3 p.) Dla następującego fragmentu pliku `ini`:

   ```
   [main]
   version=3.0.0.9
   LinuxDesktop=0
   OpenFileCMD=kfmclient exec
   IndiPanelCmd=cd /usr/local/dcd ; python ./dcd.py
   [catalog]
   starcatpath=/usr/share/apps/skychart/cat/bsc5
   StarmagMax=10
   NebmagMax=14
   ```

   podaj:

   - jak byś go podzielił na leksemy,
   - jakie żetony byś wyodrębnił
   - jakie atrybuty będą towarzyszyć leksemom.

2. (7 p.) Znaki interpunkcyjne, to: kropka, przecinek, znak zapytania, wykrzyknik i średnik. Napisz specyfikację skanera, który usuwa wszelkie białe znaki poprzedzające znaki interpunkcyjne. Inne znaki powinny być wypisywane bez zmian. Na przykład, tekst:

   ```
   Ala ma kota   ,psa i kanarka
     , chyba . . .
   ```

   powinien zostać zamieniony na:

   ```
   Ala ma kota,psa i kanarka, chyba...
   ```

3.8 Ćwiczenia

1. Podaj leksemy pojawiające się w poniższych programach.

   - ```
     function abs (i : integer): integer;
     { Oblicza wartość bezwzględną i }
     begin
       if i > 0 then abs := i else abs := -i
     end;
     ```
   - ```
     int abs (i)
     int i;
     /* Oblicza wartość bezwzględną i */
     {
        return i>0?i:-i;
     }
     ```

2. Podaj wyrażenia wzorce opisujące następujące leksemy:

   - numery rejestracyjne samochodów (takie rodzaje, jakie znasz),
   - identyfikatory (w Twoim ulubionym języku programowania),
   - napisy (w Twoim ulubionym języku programowania) wraz z możliwością umieszczania ogranicznika napisu w napisie,
   - komentarze (w Twoim ulubionym języku programowania).

3. Napisz specyfikację skanera, który:

   (a) Usuwa białe znaki z początku/końca każdej linii. (Przeczytaj w dokumentacji Lex'a o znaczeniu ^ i `$` we wzorcach.)
   Rozwiązanie[58]

   (b) Zamienia ciąg białych znaków na jeden odstęp.
   Rozwiązanie[59]

   (c) Zamienia wszystkie słowa tak, że są pisane z wielkiej litery, ale pozostałe litery są małe. Wszystkie inne znaki, powinny być wypisywane bez zmian. Rozwiązanie[60]

---

58 Przykładowe rozwiązanie znajduje się w pliku `unindent.l`.

59 Przykładowe rozwiązanie znajduje się w pliku `compress-whitespace.l`.

60 Przykładowe rozwiązanie znajduje się w pliku `capitalize.l`.

(d) Wstawia odstęp po każdej kropce i przecinku, chyba że taki odstęp już tam jest. (Przeczytaj w dokumentacji Lex'a o znaczeniu wzorców postaci r/s.) Rozwiązanie[61]

(e) Usuwa komentarze typu `//` z programu w C/C++. (Ignorujemy problemy z komentarzami postaci `/*...//...*/`):

- nie uwzględniając stałych napisowych,
- uwzględniając je.

Rozwiązanie[62]

(f) Zamienia pierwszą literę w każdym zdaniu na wielką, a pozostałe na małe. (Zdanie, to dowolny ciąg znaków zakończony kropką. Inne znaki niż litery powinny być wypisywane bez zmian.) Rozwiązanie[63]

---

61 Przykładowe rozwiązanie znajduje się w pliku `interpunkcja.l`.

62 Przykładowe rozwiązanie znajduje się w pliku `uncomment2.l`.

63 Przykładowe rozwiązanie znajduje się w pliku `capitalize-sentence.l`. Wstawka w kodzie zamieniająca litery odpowiednio na wielkie i małe może wyglądać rozmaicie.

Wykład 4. Deterministyczne automaty skończone

4.1 Wstęp

Przypomnijmy, że na język można patrzeć jak na problem decyzyjny. W ramach tego wykładu zajmiemy się modelowaniem takich mechanizmów liczących, które są przystosowane do rozstrzygania, czy dane słowo należy do języka i są wyposażone w pomocniczą pamięć stałej wielkości. Takie modele to właśnie automaty skończone. Języki, dla których takie mechanizmy istnieją, jak się później okaże, to języki regularne. Zacznijmy jednak od ustalenia pewnych, być może oczywistych i intuicyjnych pojęć.

4.2 Intuicje

Ustalmy najpierw pewne intuicyjne pojęcia: *stanu* i *przejścia*. *Stan* jakiegokolwiek systemu (czy mechanizmu), to pełna informacja na jego temat w danym momencie czasu. W szczególności stan zawiera wszelką „pamięć” w jaką wyposażony jest dany system. Również wszystko to co zdarzyło się do danego momentu ma wpływ na przyszłość tylko o tyle, o ile wpłynęło na stan systemu w danej chwili.

Będziemy rozpatrywać takie systemy i mechanizmy, których stan zmienia się w sposób dyskretny (tj. skokowy), a nie ciągły. Podejście takie ma zastosowanie np. do komputerów i urządzeń cyfrowych. Praca podzespołów komputerowych jest taktowana odpowiednimi zegarami. Ich stan może się zmieniać tylko wraz z kolejnymi taktami odpowiednich zegarów. Na cały komputer możemy spojrzeć jak na system, w którym czas biegnie nie w sposób ciągły, ale drobnymi skokami. W przypadku wielu innych systemów takie podejście też może być poprawne, jeżeli ich stan, który będzie nas interesował, będzie miał charakter dyskretny. Wówczas jego zmiany będą musiały mieć charakter skokowy. Dlatego też w naszych modelach czas będzie *dyskretny*, tzn. będzie on płynął drobnymi kroczkami.

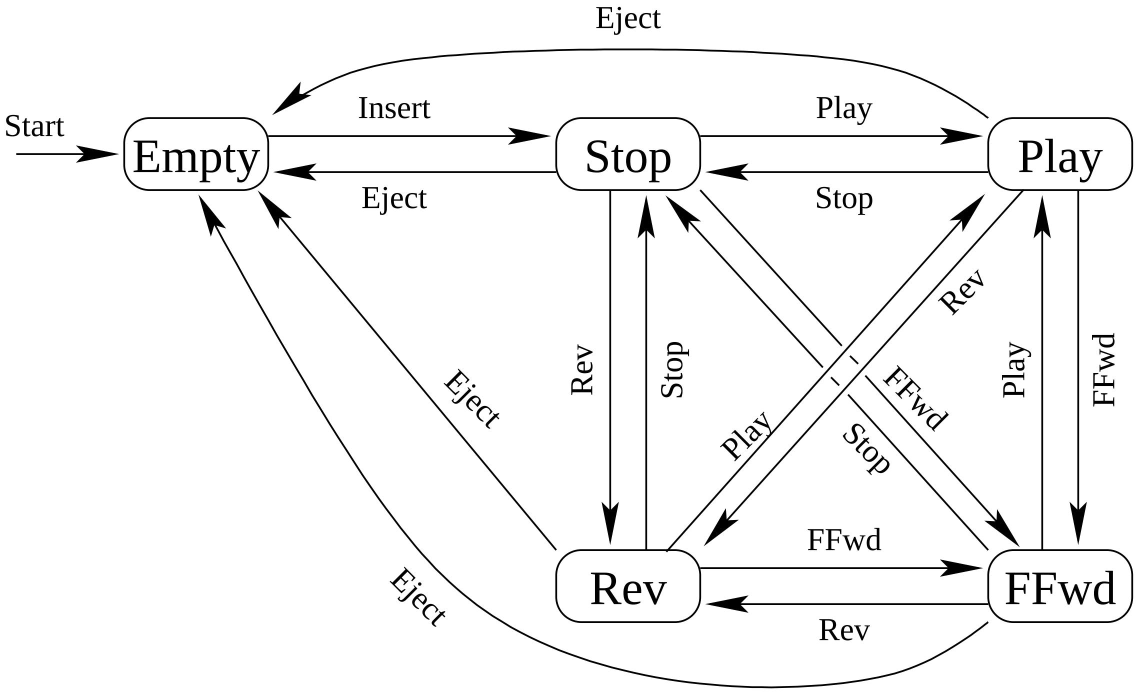
Zmianę stanu będziemy nazywać *przejściem* — od stanu w chwili poprzedniej do stanu w chwili następnej. Zwykle przejście ma miejsce jako reakcja na pewne zdarzenie. Załóżymy przy tym, że przejścia są natychmiastowe — jest to oczywiście uproszczenie, tak samo jak założenie, że czas jest dyskretny.

Najlepiej powyższe intuicje zilustrować na przykładach:

- Kostka Rubik'a — jej stany to możliwe ułożenia, a przejścia zachodzą na skutek wykonywania pojedynczych ruchów.
- Zegarek cyfrowy — Stan zegarka to aktualny czas (z dokładnością do jego pomiaru przez zegarek) oraz cała pamięć zegarka (np. godzina, na którą ustawiony jest budzik). Przejścia zachodzą na skutek upływu czasu lub naciskania przycisków zegarka.
- Sterownik windy — stany tego sterownika zawierają informację m.inn. nt. położenia windy (z pewną dokładnością), kierunku jej jazdy, otwarcia lub zamknięcia drzwi, oraz tego, jakie guziki (w windzie i na zewnątrz) były naciśnięte.

- Wszechświat(?) — pytanie czy wszechświat jest systemem dyskretnym pozostawimy filozofom i fizykom.

Przykład: Na poniższym diagramie przedstawiono uproszczony schemat działania bardzo prostego odtwarzacza video. Stany zaznaczono owalami. W momencie włączenia system znajduje się w stanie „Empty". Przejścia przedstawiono jako strzałki i umieszczono przy nich etykiety reprezentujące zdarzenia powodujące określone przejścia.



4.3 Deterministyczne automaty skończone

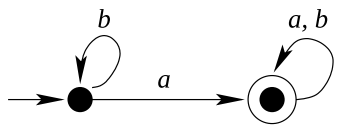
Powróćmy do języków. Interesuje nas modelowanie mechanizmów obliczeniowych odpowiadających na pytanie, czy dane słowo należy do ustalonego języka. Automat skończony wczytuje dane słowo znak po znaku i po wczytaniu całego słowa udziela odpowiedzi, czy słowo to należy do ustalonego języka. Przejścia między stanami zachodzą w automacie skończonym na skutek wczytywania kolejnych znaków analizowanego słowa. W automacie deterministycznym, dla każdego znaku jaki może się pojawić na wejściu i dla każdego stanu automatu, ze stanu tego wychodzi dokładnie jedno przejście odpowiadające danemu znakowi — w każdej sytuacji działanie automatu musi być określone jednoznacznie. (W tym wykładzie będziemy się zajmować wyłącznie automatami deterministycznymi. Automaty niedeterministyczne pojawią się w kolejnym wykładzie i dopiero wówczas będziemy je odróżniać.)

To, jaka jest odpowiedź udzielana przez automat zależy od stanu osiąganego po wczytaniu całego słowa. Jeżeli odpowiedź jest pozytywna, to mówimy, że automat *akceptuje* słowo, a w przeciwnym przypadku mówimy, że je *odrzuca*. Stany, w których automat akceptuje nazywamy *stanami akceptującymi*. Pamiętajmy jednak, że o tym, czy automat akceptuje słowo decyduje wyłącznie ostatni stan — ten, w którym automat jest po wczytaniu całego słowa.

Dodatkowo, automat skończony, jak sama nazwa wskazuje, ma skończoną liczbę stanów. Oznacza to, że dysponuje on stałą pamięcią dodatkową, tzn. wielkość pamięci dodatkowej nie zależy od długości wczytywanego słowa i jest ograniczona przez stałą. Jeżeli do stwierdzenia, czy słowo należy do ustalonego języka potrzebna jest pamięć pomocnicza, której nie da się ograniczyć przez stałą, to taki język nie jest akceptowany przez żaden automat skończony. Przykłady takich języków, oraz techniki dowodzenia tego poznamy w dalszych wykładach.

Przykład: Zanim podamy formalną definicję automatu skończonego zobaczmy prosty przykład takiego automatu.

Do prezentowania automatów skończonych będziemy używać diagramów podobnych do diagramu odtwarzacza wideo z poprzedniego punktu. Stany automatu oznaczamy na diagramie punktami, a przejścia strzałkami. Przejścia etykietujemy znakami, których wczytanie powoduje dane przejście. Początkowy stan automatu oznaczamy strzałką „znikąd". Stany akceptujące zaznaczamy otaczając je dodatkowym kółkiem.



Powyższy automat wczytuje słowa nad alfabetem $\Sigma = \{a, b\}$. Akceptuje on wszystkie słowa zawierające choć jedną literę a — tylko takie słowa mogą spowodować przejście do stanu po prawej stronie.

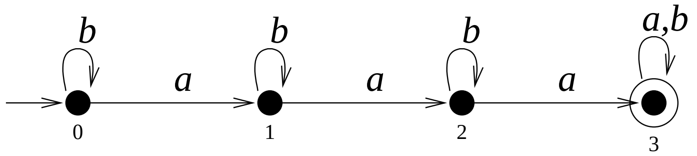
Def. 24. Automat skończony (deterministyczny) M, to dowolna taka piątka $M = \langle Q, \Sigma, \delta, s, F\rangle$, w której:

- Q to skończony zbiór *stanów*,
- Σ to (skończony) *alfabet wejściowy*,
- $\delta : Q \times \Sigma \to Q$ to *funkcja przejścia* — $\delta(q, a)$ to stan do którego przechodzimy ze stanu q na skutek wczytania znaku a,
- s to stan początkowy,
- $F \subseteq Q$ to zbiór stanów akceptujących.

$\Box$

Przedstawiając automat musimy podać informacje o wszystkich pięciu elementach. Formalne podawanie piątki, takiej jak w definicji, jest mało czytelne. Dlatego też zwykle przedstawia się automat w postaci tabelki lub diagramu.

Przykład: Następująca diagram i tabelka przedstawiają ten sam automat.



		a	b
$\rightarrow$	0	1	0
	1	2	1
	2	3	2
F	3	3	3

Tabelka przedstawia przede wszystkim funkcję przejścia δ. Wiersze są indeksowane stanami. Kolumny są indeksowane znakami z alfabetu wejściowego. Komórki tabeli zawierają stany, do których przechodzimy na skutek wczytania pojedynczych znaków. Dodatkowo, stan początkowy jest zaznaczony za pomocą strzałki $\rightarrow$, a stany akceptujące są zaznaczone literami F.

Zaletą diagramów jest ich przejrzystość oraz to, że nie ma konieczności nazywania stanów. Natomiast zaletą tabel jest ich zwarta postać.

Dane dla automatu to dowolne słowo $x \in \Sigma^*$. Działanie automatu możemy zasymulować w następujący sposób:

1. Na początku automat jest w stanie początkowym.
2. Wczytujemy kolejne znaki słowa na wejściu i zmieniamy stan zgodnie z przejściami odpowiadającymi tym znakom.
3. Po wczytaniu całego słowa i wykonaniu wszystkich przejść sprawdzamy, czy automat jest w stanie akceptującym. Jeżeli tak, to słowo jest akceptowane przez automat, wpp. nie.

Mając automat przedstawiony w formie diagramu możemy potraktować ten diagram jak planszę. Na początku stawiamy pionek na polu / w stanie początkowym. Następnie przesuwamy pionek zgodnie z wczytywanymi znakami. Jeśli na koniec jesteśmy na polu / w stanie akceptującym, to słowo jest akceptowane.

Przykład:

Czas formalnie zdefiniować działanie deterministycznych automatów skończonych. Niech $M = \langle Q, \Sigma, \delta, s, F\rangle$ będzie ustalonym deterministycznym automatem skończonym.

Def. 25. Funkcję przejścia $\delta : Q \times \Sigma \rightarrow Q$ rozszerzamy do funkcji $\delta^* : Q \times \Sigma^* \rightarrow Q$ w następujący sposób:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, xa) = \delta(\delta^*(q, x), a)$$

□

Podczas gdy funkcja δ określa działanie automatu dla pojedynczego znaku, funkcja δ^* określa jego działanie dla całych słów.

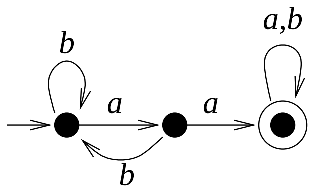
Def. 26. Automat *akceptuje słowo* x wtw., gdy $\delta^*(s, x) \in F$.
Język akceptowany przez automat M, to:

$$L(M) \equiv \{x \in \Sigma^* : \delta^*(s, x) \in F\}$$

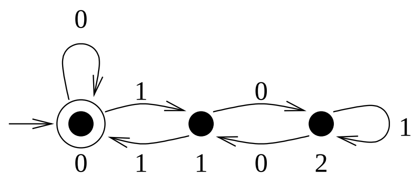
□

Inaczej mówiąc, język akceptowany przez automat składa się dokładnie z tych słów, które ten automat akceptuje.

Przykład: Automat akceptujący słowa nad alfabetem $\{a, b\}$ zawierające podsłowo aa.



Przykład: Automat akceptujący $\{x \in \{0, 1\}^* : x$ w układzie binarnym jest podzielne przez 3 $\}$.



Stany reprezentują reszty modulo 3. Po wczytaniu dowolnego ciągu cyfr automat jest w stanie odpowiadającym reszcie z dzielenia wczytanej liczby przez 3. Stan początkowy (odpowiadający wczytaniu ε) to 0. Budując przejścia automatu musimy się zastanowić, jak zmienia się reszta z dzielenia danej liczby przez 3, gdy do jej zapisu binarnego dopiszemy (z prawej strony) 0 lub 1. Korzystamy tu z następujących tożsamości:

$$x0 = 2 \cdot x \equiv (2 \cdot (x \bmod 3)) \bmod 3$$

$$x1 = 2 \cdot x + 1 \equiv (2 \cdot (x \bmod 3) + 1) \bmod 3$$

Czyli $\delta(q, x) = (2 \cdot q + x)\mathrm{mod}3$.

4.4 Własności klasy języków akceptowanych przez deterministyczne automaty skończone

Zajmiemy się teraz własnościami klasy języków akceptowanych przez deterministyczne automaty skończone. Jak się później okaże, jest to dokładnie klasa języków regularnych. Zanim się to jednak stanie, pokażemy, że automaty skończone mają niektóre z cech wzorców, które pokazaliśmy w poprzednim wykładzie.

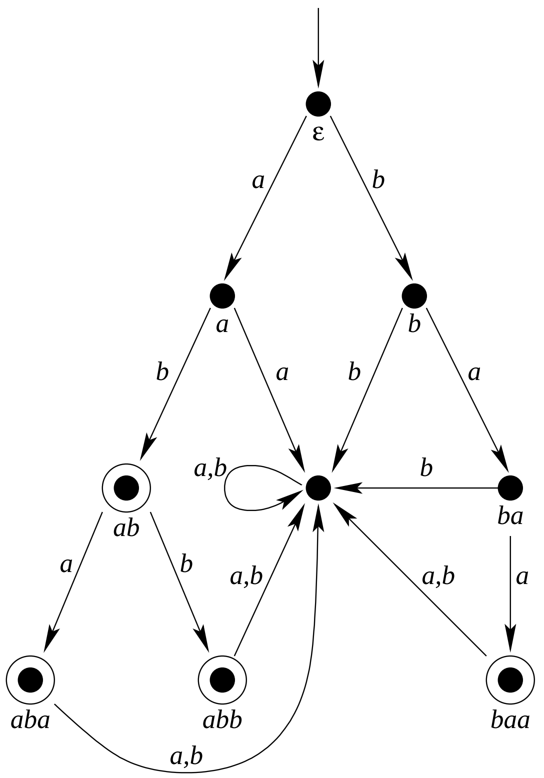
Fakt 3. *Każdy język skończony jest akceptowany przez pewien deterministyczny automat skończony.*

Dowód: Niech A będzie skończonym językiem. Wszystkie słowa z A razem wzięte mają skończoną liczbę prefiksów. Stanami automatu akceptującego A będą właśnie te prefiksy, plus jeden dodatkowy stan, który będziemy nazywać „śmietnikiem". Po wczytaniu dowolnego słowa, jeżeli jest ono prefiksem pewnego słowa z A, nasz automat będzie dokładnie w takim stanie jak to słowo. Natomiast po wczytaniu słowa, które nie jest prefiksem żadnego słowa z A automat przejdzie do śmietnika.

Stan początkowy to ε. Jeśli q i qa są prefiksami pewnego słowa z A, to $\delta(q,a) = qa$, wpp. $\delta(q,a) = \textit{śmietnik}$. Stany akceptujące to te prefiksy, które należą do A.

W oczywisty sposób, język akceptowany przez automat to A. Natomiast dzięki temu, że słowa z A mają skończoną liczbę prefiksów, jest to poprawny automat skończony. □

Przykład: Weźmy język $A = \{ab, aba, abb, baa\}$. Prefiksy słów z tego języka tworzą zbiór: $\{\varepsilon, a, b, ab, ba, aba, abb, baa\}$. W rezultacie otrzymujemy automat postaci:



Twierdzenie: Niech A i B będą językami (nad tym samym alfabetem Σ) akceptowanymi odpowiednio przez automaty deterministyczne $M_1 = \langle Q_1, \Sigma, \delta_1, s_1, F_1 \rangle$ i $M_2 = \langle Q_2, \Sigma, \delta_2, s_2, F_2 \rangle$, $L(M_1) = A$, $L(M_2) = B$. Wówczas następujące języki są również akceptowane przez pewne deterministyczne automaty skończone:

- $\overline{A}$,
- $A \cap B$,
- $A \cup B$.

Dowód:

- Dopełnienie: Weźmy $M_1' = \langle Q_1, \Sigma, \delta_1, s_1, Q \setminus F_1 \rangle$. Mamy:

$$\begin{aligned} L(M_1') &= \{x \in \Sigma^* : \delta_1^*(s_1, x) \in Q \setminus F_1\} = \\ &= \{x \in \Sigma^* : \delta_1^*(s_1, x) \notin F_1\} = \\ &= \Sigma^* \setminus \{x \in \Sigma^* : \delta_1^*(s_1, x) \in F_1\} = \\ &= \Sigma^* \setminus L(M_1) = \Sigma^* \setminus A = \overline{A} \end{aligned}$$

Tak więc $\overline{A}$ jest akceptowany przez M_1'.

- Weźmy $M_3 = \langle Q_1 \times Q_2, \Sigma, \delta_3, (s_1, s_2), F_1 \times F_2 \rangle$, przy czym $\delta_3((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$.

  Automat taki nazywamy *automatem produktowym.* Jego stany to stany automatów M_1 i M_3. Symuluje on równocześnie działanie obydwóch tych automatów. Jednocześnie akceptuje tylko takie słowa, które byłyby zaakceptowane równocześnie przez M_1 i M_2.

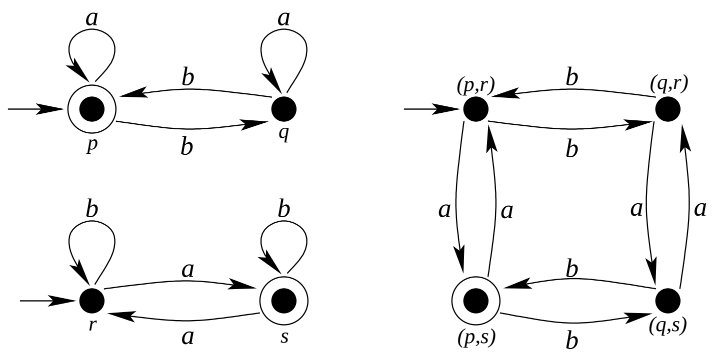
  Można pokazać (przez indukcję po długości słowa x), że $\delta_3^*((p, q), x) = (\delta_1^*(p, x), \delta_2^*(q, x))$. Stąd uzyskujemy:

$$
\begin{aligned}
L(M_3) &= \{x \in \Sigma^* : \delta_3^*((s_1, s_2), x) \in F_1 \times F_2\} = \\
&= \{x \in \Sigma^* : (\delta_1^*(s_1, x), \delta_2^*(s_2, x)) \in F_1 \times F_2\} = \\
&= \{x \in \Sigma^* : \delta_1^*(s_1, x) \in F_1 \wedge \delta_2^*(s_2, x) \in F_2\} = \\
&= \{x \in \Sigma^* : \delta_1^*(s_1, x) \in F_1\} \cap \{x \in \Sigma^* : \delta_2^*(s_2, x) \in F_2\} = \\
&= L(M_1) \cap L(M_2) = A \cap B
\end{aligned}
$$

  Tak więc $A \cap B$ jest akceptowany przez M_3.

- Z praw De Morgana mamy $A \cup B = \overline{\overline{A} \cap \overline{B}}$. Stąd $A \cup B$ jest akceptowany przez pewien deterministyczny automat skończony. $\square$

Przykład: Oto dwa automaty, pierwszy akceptujący słowa zawierające parzystą liczbę liter b i drugi akceptujący słowa zawierające nieparzystą liczbę liter a, oraz ich automat produktowy (akceptujący te słowa, które zawierają parzystą liczbę liter b i nieparzystą liczbę liter a).



4.5 Podsumowanie

Ten wykład był poświęcony deterministycznym automatom skończonym. Poznaliśmy ich budowę i sposób działania. Poznaliśmy też podstawowe operacje, na które domknięta jest klasa języków akceptowanych przez automaty deterministyczne. Jak się okaże w dalszej części kursu, jest to dokładnie klasa języków regularnych.

4.6 Skorowidz

- **Automat skończony, deterministyczny** to dowolna taka piątka $M = \langle Q, \Sigma, \delta, s, F \rangle$, w której: Q to skończony zbiór *stanów*, Σ to (skończony) *alfabet wejściowy*, $\delta : Q \times \Sigma \to Q$ to *funkcja przejścia*, s to stan początkowy, $F \subseteq Q$ to zbiór stanów akceptujących.
- **Automat produktowy** to automat symulujący równocześnie działanie dwóch danych automatów deterministycznych — jego zbiór stanów to produkt kartezjański zbiorów stanów danych automatów.
- **Język akceptowany** przez deterministyczny automat skończony $M = \langle Q, \Sigma, \delta, s, F \rangle$ to język postaci:
$$L(M) \equiv \{x \in \Sigma^* : \delta^*(s, x) \in F\}$$
- **Przejście** to zmiana stanu — od stanu w chwili poprzedniej do stanu w chwili następnej. Zwykle przejście ma miejsce jako reakcja na pewne zdarzenie.
- **Stan** jakiegokolwiek systemu (czy mechanizmu), to pełna informacja na jego temat w danym momencie czasu. W szczególności stan zawiera wszelką „pamięć" w jaką wyposażony jest dany system.

4.7 Praca domowa

1. (3p.) Podaj deterministyczny automat skończony akceptujący te słowa nad alfabetem $\{0, 1\}$, w których kazda seria zer i każda seria jedynek jest parzystej długości.
2. (3p.) Podaj deterministyczny automat skończony akceptujący te słowa nad alfabetem $\{a, b\}$, które zawierają podsłowo *babbb*.
3. (4p.) Podaj deterministyczny automat skończony akceptujący przecięcie języków akceptowanych przez automaty:

		a	b
$\to F$	1	2	3
	2	3	1
	3	1	2

		a	b
$\to$	1	2	1
F	2	1	2

4.8 Ćwiczenia

Podaj deterministyczne automaty skończone akceptujące następujące języki:

1. słowa nad alfabetem $\{0, 1\}$, które zawierają parzystą liczbę zer, Rozwiązanie[64]

2. słowa nad alfabetem $\{a, b\}$ zakończone na aa, Rozwiązanie[65]

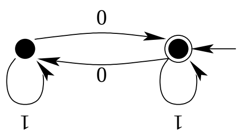
3. słowa nad alfabetem $\{0, 1\}$, w których wszystkie 1-ki znajdują się na parzystych pozycjach, Rozwiązanie[66]

4. Podaj deterministyczny automat skończony akceptujący te słowa nad alfabetem $\{0, 1\}$, w których między każdymi dwoma kolejnymi jedynkami jest parzysta liczba zer. Rozwiązanie[67]

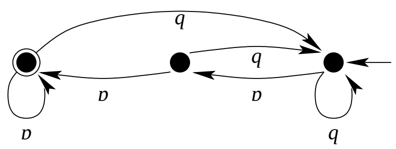
5. słowa nad alfabetem $\{0, 1\}$, w których na wszystkich parzystych pozycjach występują

---

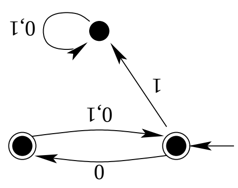
[64] Jedno z możliwych rozwiązań to:



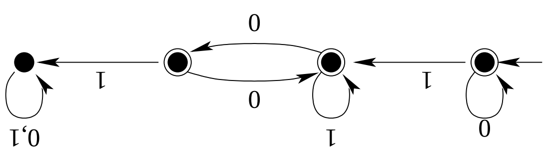
[65] Jedno z możliwych rozwiązań to:



[66] Jedno z możliwych rozwiązań to:



[67] Jedno z możliwych rozwiązań to:

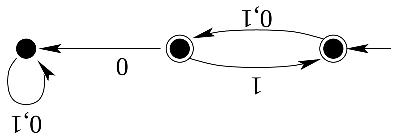


1-ki, Rozwiązanie[68]

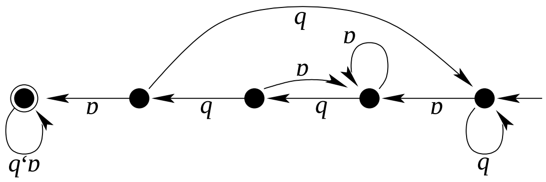
6. słowa nad alfabetem $\{1, 2, 3, 4\}$ zawierające podsłowo 42,
7. słowa nad alfabetem $\{a\}$ o długości podzielnej przez 2 lub 3,
8. słowa nad alfabetem $\{a, b\}$ zawierające podsłowo abb,
9. słowa nad alfabetem $\{a, b\}$ nie zawierające podsłowa aba,
10. słowa nad alfabetem $\{a, b\}$ zawierające podsłowo $abba$, Rozwiązanie[69]
11. słowa nad alfabetem $\{a, b\}$ zawierające podsłowo $ababb$, Rozwiązanie[70]
12. słowa nad alfabetem $\{a, b\}$ zawierające przynajmniej 2 wystąpienia słowa aba (być może zachodzące na siebie),
13. te słowa nad alfabetem $\{a, b\}$, w których pierwszy i ostatni znak są takie same (uwaga, słowa a i b powinny zostać zaakceptowane),
14. te słowa nad alfabetem $\{0, 1, \ldots, 9\}$, które zawierają jako podsłowo wybrany przez Ciebie numer telefonu,
15. zapisy 10-tne liczb podzielnych przez 3,
16. te słowa nad alfabetem $\{0, 1\}$, w których każde dwa kolejne znaki, licząc od początku słowa, zawierają choć jedną 1-kę,

---

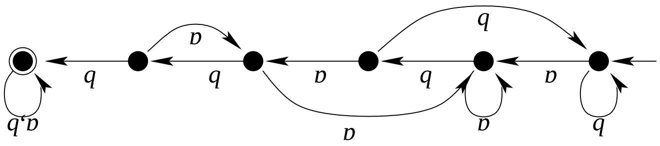
68



69 Jedno z możliwych rozwiązań to:



70



17. przecięcie/sumę języków akceptowanych przez automaty:

		a	b
$\to$	1	2	2
F	2	1	1

		a	b
$\to$	1	1	2
F	2	1	2

Rozwiązanie[71]

18. przecięcie/sumę języków akceptowanych przez automaty:

		a	b
$\to F$	1	2	2
	2	1	1

		a	b
$\to$	1	2	1
F	2	1	1

Rozwiązanie[72]

19. przecięcie/sumę języków akceptowanych przez automaty:

		a	b
$\to F$	1	2	2
	2	1	1

		a	b
$\to$	1	1	2
F	2	2	1

20. przecięcie/sumę języków akceptowanych przez automaty:

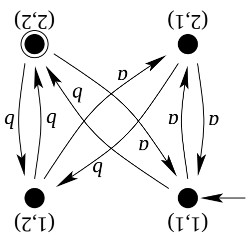
		a	b
$\to$	1	1	2
F	2	2	1

		a	b
$\to F$	1	2	1
	2	1	2

---

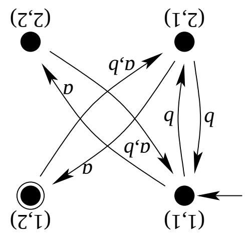
71 Oto automat produktowy akceptujący przecięcie tych języków:

		a	b
$\to$	(1,1)	(2,1)	(2,2)
	(1,2)	(2,1)	(2,2)
	(2,1)	(1,1)	(1,2)
F	(2,2)	(1,1)	(1,2)



72 Oto automat produktowy akceptujący przecięcie tych języków:

		a	b
$\to$	(1,1)	(2,2)	(2,1)
F	(1,2)	(2,1)	(2,1)
	(2,1)	(1,2)	(1,1)
	(2,2)	(1,1)	(1,1)



21. przecięcie/sumę języków akceptowanych przez automaty:

		a	b
$\rightarrow$	1	2	2
F	2	1	1

		a	b
$\rightarrow F$	1	2	3
	2	3	1
	3	1	2

Wykład 5. Niedeterministyczne automaty skończone

5.1 Wstęp

W poprzednim wykładzie poznaliśmy deterministyczne automaty skończone. Czym jednak jest ów *determinizm*? Determinizm to koncepcja filozoficzna, według której w Świecie nie ma miejsca na przypadek — gdyby mieć pełną informację o stanie świata, można by przewidzieć jego przyszłość. W determinizmie brak też miejsca na wolną wolę — nasze decyzje są określone przez stan Świata, którego jesteśmy częścią.

Niedeterminizm stanowi przeciwną koncepcję. Nawet gdybyśmy mieli pełną wiedzę na temat stanu Świata w danej chwili, to i tak nie bylibyśmy w stanie przewidzieć przyszłości. Co najwyżej moglibyśmy określić jakie są możliwe przyszłe dzieje.

W informatyce determinizm oznacza, że jeżeli będziemy wielokrotnie uruchamiać ten sam program czy mechanizm obliczeniowy, podając te same dane, to zawsze uzyskamy te same wyniki. Niedeterminizm zaś przeciwnie — dla tych samych danych możemy uzyskać różne wyniki. Prostych programy komputerowe (bez randomizacji) zwykle działają w sposób deterministyczny. Niedeterminizm pojawia się np. w algorytmach randomizacyjnych czy programowaniu współbieżnym, czyli tam, gdzie jest jakieś źródło przypadkowości. Wówczas zwykle wymagamy, aby bez względu na przebieg obliczeń była zachowana jakaś wymagana przez nas własność. Niedeterminizm jest też używany przy specyfikowaniu systemów — specyfikacja nie określa jednoznacznie jak powinien zachowywać się system, każde zachowanie zgodne ze specyfikacją jest dopuszczalne.

Automat niedeterministyczny jest więc automatem, którego działania nie da się w pełni przewidzieć. Będąc w tym samym stanie i wczytując ten sam znak może wykonać różne przejścia. Przestaje więc obowiązywać wymóg, że dla danego znaku, z każdego stanu wychodzi dokładnie jedno przejście odpowiadające wczytaniu tego znaku. Takich przejść może być dowolnie wiele, a nawet może ich nie być wcale (co to znaczy wyjaśni się za chwilę). Automat taki może również mieć wiele stanów początkowych — każde uruchomienie automatu może rozpocząć się w dowolnym z tych stanów.

Może więc się zdarzyć, że ten sam automat raz zaakceptuje dane słowo, a raz nie. Czy więc takie słowo należy do języka akceptowanego przez automat, czy nie? Otóż, jeśli tylko automat **może** zaakceptować słowo, to powiemy, że należy ono do języka akceptowanego przez automat. Jest to dosyć nietypowa interpretacja niedeterminizmu. Nie odpowiada ona intuicyjnemu rozumieniu przypadku.

Na taki niedeterministyczny automat możemy spojrzeć jak na mechanizm o dwoistej budowie: jedna jego część to automat skończony określający jakie przejścia są możliwe, druga część, to tzw. *wyrocznia*, która w przypadku wielu możliwych do wykonania przejść wybiera przejście, które prowadzi do zaakceptowania słowa. (Zapachniało metafizyką? Chwila cierpliwości, zaraz wszystko będzie ściśle zdefiniowane.) Jedno z zagadnień, którymi się zajmiemy, to czy taka wyrocznia zmienia siłę obliczeniową automatów, czyli czy dla tych samych języków potrafimy zbudować automaty deterministyczne i nie, czy też są jakieś różnice? Jak się okaże, w przypadku automatów skończonych nie ma żadnej różnicy, a wyrocznię i niedeterminizm można zastąpić większą liczbą stanów.

5.2 Działanie niedeterministycznych automatów skończonych

Jak już powiedzieliśmy, w automacie niedeterministycznym, dla danego znaku, nie musi z każdego stanu wychodzi dokładnie jedno przejście odpowiadające wczytaniu tego znaku. Takich przejść może nie być, może być jedno, lub może być wiele. Podobnie, automat niedeterministyczny może mieć wiele stanów początkowych.

Działanie automatu niedeterministycznego możemy symulować w następujący sposób. Spójrzmy na diagram automatu jak na planszę, po której będziemy przesuwać pionki — stany to pola, na których będziemy stawiać pionki, a przesuwać będziemy je zgodnie z przejściami.

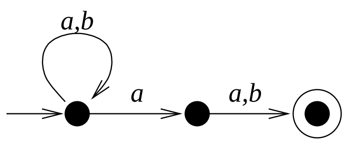
1. Na początku symulacji ustawiamy pionki we wszystkich stanach początkowych.

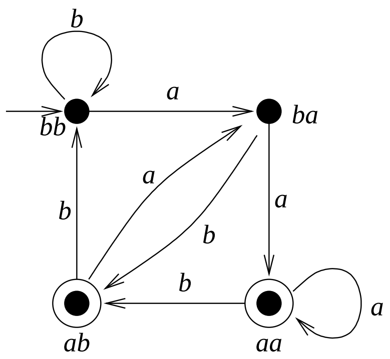
2. Wczytując pojedynczy znak przesuwamy zgodnie z przejściami wszystkie pionki na planszy. Przy czym, jeżeli z pola, na którym stoi pionek wychodzi wiele przejść odpowiadających wczytanemu znakowi, to stawiamy pionki na wszystkich polach docelowych. W szczególności, jeżeli z pola, na którym stoi pionek nie wychodzi żadne takie przejście, to taki pionek zdejmujemy z planszy. Jeżeli na jednym polu znajdzie się kilka pionków, to zostawiamy jeden, a resztę zdejmujemy.

3. Automat akceptuje słowo jeśli po wczytaniu całego słowa w którymkolwiek z pól/stanów akceptujących znajduje się pionek.

Pionki reprezentują wszystkie te stany, w których automat może się znaleźć po wczytaniu odpowiedniego (prefiksu) słowa. Automat akceptuje słowo jeśli choć jeden pionek dotrze do stanu akceptującego, czyli może się zdarzyć takie obliczenie, które prowadzi do jego zaakceptowania.

Przykład: Oto automat niedeterministyczny, który akceptuje te słowa nad alfabetem $\{a, b\}$, w których przedostatni znak to a. Obok przedstawiono dla porównania deterministyczny automat skończony akceptujący taki sam język. Automat deterministyczny musi pamiętać ostatnie dwa znaki wczytywanego słowa — dlatego też ma cztery stany.





		a	b
$\rightarrow$	1	1, 2	1
	2	3	3
F	3		

		a	b
$\rightarrow$	bb	ba	bb
	ba	aa	ab
F	ab	ba	bb
F	aa	aa	ab

Def. 27. Automat niedeterministyczny, to dowolna taka piątka $M = \langle Q, \Sigma, \delta, S, F\rangle$, gdzie:

- Q to skończony zbiór stanów,
- Σ to (skończony) alfabet wejściowy,
- δ to funkcja[73] przejścia $\delta : Q \times \Sigma \to 2^Q$,
- $S \subseteq Q$ to zbiór stanów początkowych,
- $F \subseteq Q$ to zbiór stanów akceptujących.

□

Def. 28. Rozszerzenie funkcji przejścia $\delta^* : 2^Q \times \Sigma^* \to 2^Q$ definiujemy następująco:

$$\delta^*(A, \varepsilon) = A$$

$$\delta^*(A, ax) = \delta^*(\bigcup_{q \in A} \delta(q, a), x)$$

□

Funkcja δ^* zamiast na pojedynczych stanach operuje na zbiorach stanów. Dokładnie, opisuje ona jak zmienia się ustawienie pionków pod wpływem wczytywanych słów.

Def. 29. Niedeterministyczny automat skończony $M = \langle Q, \Sigma, \delta, S, F\rangle$ akceptuje słowo x wtw., gdy $\delta^*(S, x) \cap F \neq \emptyset$. Język akceptowany przez automat M, to:

$$L(M) = \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\}$$

□

Formalnie automaty niedeterministyczne są czymś innym niż deterministyczne. Jednak my będziemy traktować automaty deterministyczne jak szczególny przypadek automatów niedeterministycznych. Patrząc na diagramy reprezentujące automaty, automaty deterministyczne muszą spełnić więcej wymagań niż automaty niedeterministyczne. Następujący fakt utwierdza nas w takim spojrzeniu:

[73] W literaturze można spotkać definicje, w których niedeterministyczny automat skończony ma *relację* przejścia postaci $\delta \subseteq Q \times \Sigma \times Q$. Relacje takie są jednak izomorficzne z funkcjami podanej przez nas postaci.

Fakt 4. *Niech $M = \langle Q, \Sigma, \delta, s, F\rangle$ będzie automatem deterministycznym. Wówczas automat niedeterministyczny $M' = \langle Q, \Sigma, \delta', \{s\}, F\rangle$, gdzie $\delta'(q, a) = \{\delta(q, a)\}$ akceptuje dokładnie ten sam język, $L(M) = L(M')$.*

Kilka własności funkcji δ^*, które przydadzą nam się w dalszej części wykładu:

Fakt 5. *Dla dowolnych $x, y \in \Sigma^*$ mamy $\delta^*(A, xy) = \delta^*(\delta^*(A, x), y)$.*

Dowód przebiega przez indukcję ze względu na $|y|$.

Fakt 6. *Funkcja δ^* jest rozłączna ze względu na sumowanie zbiorów, tzn.:*

$$\delta^*(\bigcup_i A_i, x) = \bigcup_i \delta^*(A_i, x)$$

Dowód przebiega przez indukcję ze względu na $|x|$.

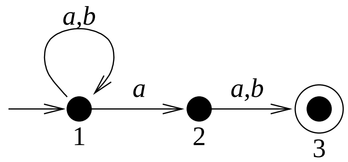
5.3 Determinizacja automatów skończonych

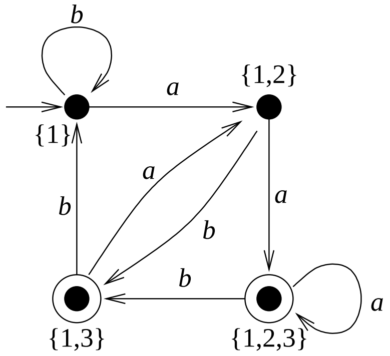
Pokażemy teraz, że siła wyrazu automatów niedeterministycznych jest dokładnie taka sama, jak deterministycznych. W tym celu musimy pokazać, że dla każdego niedeterministycznego automatu skończonego istnieje równoważny mu automat deterministyczny. To, że automaty deterministyczne możemy traktować jak szczególny przypadek automatów niedeterministycznych powiedzieliśmy już wcześniej.

W trakcie symulacji działania automatu niedeterministycznego istotny jest zbiór pól, na których w danym momencie znajdują się pionki. Nasza konstrukcja będzie polegała na zbudowaniu automatu deterministycznego, którego stanami będą *zbiory* stanów automatu niedeterministycznego (mogące się pojawić w czasie symulacji), jego stanem początkowym będzie zbiór stanów początkowych automatu niedeterministycznego, a akceptujące będą te stany, które zawierają choć jeden stan akceptujący automatu niedeterministycznego.

Konstruując automat deterministyczny liczba stanów może wzrosnąć nawet wykładniczo. Jednak nie zawsze tak musi być, gdyż nie zawsze wszystkie możliwe zbiory stanów automatu niedeterministycznego mogą się pojawić w czasie symulacji. Zobaczmy kilka przykładów.

Przykład: Przypomnijmy sobie automat niedeterministyczny akceptujący słowa nad alfabetem $\{a, b\}$, w których przedostatni znak to a. W trakcie dowolnego obliczenia, na polu początkowym nr 1 będzie cały czas stał pionek. Tak więc mogą się pojawić cztery różne konfiguracje pionków (czy też cztery różne zbiory pól zajmowanych przez pionki). Ponieważ konfiguracje nieosiągalne nie mają znaczenia dla działania automatu, możemy się ograniczyć się do tych czterech konfiguracji. Odpowiada to automatowi zapamiętującemu dwa ostatnie znaki w słowie.

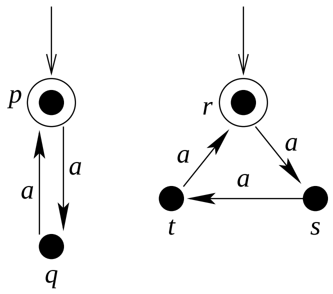


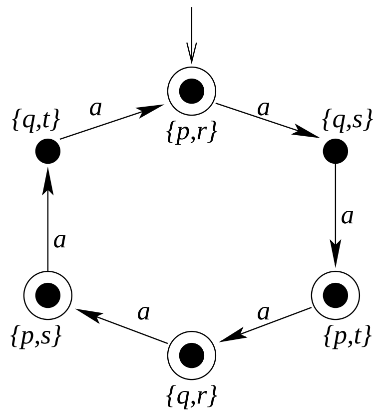


		a	b
$\to$	1	$1,2$	1
	2	3	3
F	3		

		a	b
$\to$	$\{1\}$	$\{1,2\}$	$\{1\}$
	$\{1,2\}$	$\{1,2,3\}$	$\{1,3\}$
F	$\{1,3\}$	$\{1,2\}$	$\{1\}$
F	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,3\}$

Przykład: Automat przedstawiony na poniższym rysunku akceptuje słowa (nad alfabetem $\{a\}$) o długości podzielnej przez 2 lub 3. Jedyny niedeterminizm polega tu na wybraniu stanu początkowego. Automat deterministyczny ma 6 stanów.

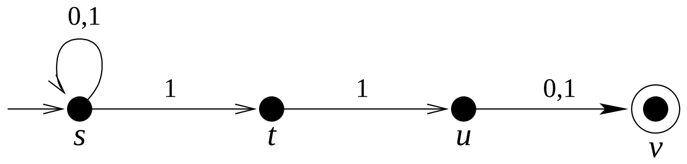




		a
$\to F$	p	q
	q	p
$\to F$	r	s
	s	t
	t	r

		a
$\to F$	$\{p,r\}$	$\{q,s\}$
	$\{q,s\}$	$\{p,t\}$
F	$\{p,t\}$	$\{q,r\}$
F	$\{q,r\}$	$\{p,s\}$
F	$\{p,s\}$	$\{q,t\}$
	$\{q,t\}$	$\{p,r\}$

Przykład: Automat przedstawiony na rysunku poniżej akceptuje takie ciągi zer i jedynek, w których przedostatnim i przed-przedostatnim znakiem są jedynki.



[Zdeterminizować i porównać z deterministycznym skonstruowanym od początku.....]

Konstrukcja automatu potęgowego Konstruowany automat deterministyczny, równoważny automatowi niedeterministycznemu, jest nazywany *automatem potęgowym* — gdyż jego zbiór stanów to zbiór potęgowy zbioru stanów automatu niedeterministycznego. Formalnie, jego konstrukcja przebiega następująco.

Niech $M = \langle Q, \Sigma, \delta, S, F\rangle$ będzie niedeterministycznym automatem skończonym. Jego determinizacją będziemy nazywać automat deterministyczny $M' = \langle 2^Q, \Sigma, \delta', S, F'\rangle$, gdzie:

$$\delta'(A, a) = \delta^*(A, a)$$

$$F' = \{A \subseteq Q : A \cap F \neq \emptyset\}$$

Pokażemy, że oba automaty akceptują te same języki.

Lemat 1. *Dla dowolnych $A \subseteq Q$ i $x \in \Sigma^*$ zachodzi*

$$\delta^*(A, x) = \delta'^*(A, x)$$

Dowód: Dowód przebiega przez indukcja ze względu na $|x|$.

1. $\delta^*(A, \varepsilon) = A = \delta'^*(A, \varepsilon)$,

2. Załóżmy, że lemat zachodzi dla słowa x. Pokażemy, że zachodzi również dla słowa postaci ax:

$$\delta^*(A, ax) = \delta^*(\delta^*(A, a), x) = \delta^*(\delta'(A, a), x) = \delta^*(\delta'^*(A, a), x)$$

   Z założenia indukcyjnego:

$$\delta^*(\delta'^*(A, a), x) = \delta'^*(\delta'^*(A, a), x) = \delta'^*(A, ax)$$

□

Korzystając z lematu, dowód, że jest natychmiastowy:

$$\begin{aligned} L(M) &= \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\} = \\ &= \{x \in \Sigma^* : \delta'^*(S, x) \cap F \neq \emptyset\} = \\ &= \{x \in \Sigma^* : \delta'^*(S, x) \in F'\} = L(M') \end{aligned}$$

W tak skonstruowanym automacie potęgowym możemy, bez zmiany akceptowanego przez niego języka, usunąć wszystkie stany nieosiągalne.

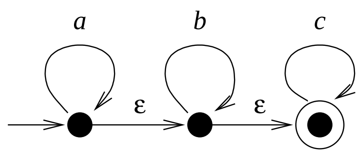
5.4 Automaty z ε-przejściami

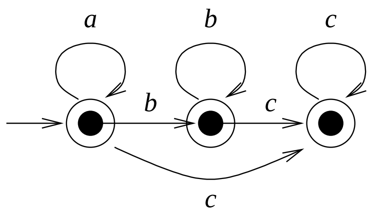
Automaty z ε-przejściami stanowią rozszerzenie automatów niedeterministycznych. Mogą one dodatkowo zawierać tzw. ε-przejścia. Są to przejścia, które automat może w każdej chwili wykonywać, nie wczytując nic z wejścia. Tak więc automat z ε-przejściami nie musi w każdym kroku wczytywać jednego znaku.

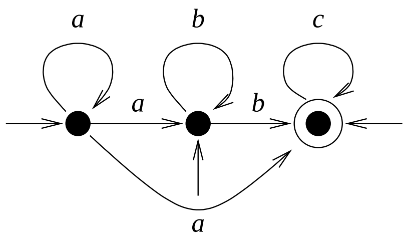
Podobnie, jak automaty deterministyczne stanowią szczególny przypadek automatów niedeterministycznych, tak automaty niedeterministyczne stanowią szczególny przypadek automatów z ε-przejściami — są to automaty z ε-przejściami, w których nie ma ε-przejść.

Okazuje się, że automaty z ε-przejściami mają taką samą siłę wyrazu co automaty niedeterministyczne. Czasami jednak ε-przejścia pozwalają uprościć budowę automatu.

Przykład: Poniższy rysunek przedstawia automat z ε-przejściami akceptujący język $a^*b^*c^*$. Poniżej przedstawiono dwa równoważne mu automaty niedeterministyczne. Jak widać, ε-przejścia można usunąć zachowując ten sam zbiór stanów, ale kosztem zwiększenia liczby przejść oraz większej liczby stanów początkowych lub akceptujących.







		a	b	c	ε
$\rightarrow$	1	1			2
	2		2		3
F	3			3	

		a	b	c
$\rightarrow F$	1	1	2	3
F	2		2	3
F	3			3

		a	b	c
$\rightarrow$	1	$1,2,3$		
$\rightarrow$	2		$2,3$	
$\rightarrow F$	3			3

Def. 30. Automat z ε-przejściami, to dowolna taka piątka $M = \langle Q, \Sigma, \delta, S, F\rangle$, w której:

- Q jest skończonym zbiorem stanów,
- Σ to (skończony) alfabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ to funkcja przejścia,
- $S \subseteq Q$ to zbiór stanów początkowych, oraz

- $F \subseteq Q$ to zbiór stanów akceptujących.

□

Jak widać, jedyna różnica między definicją automatu niedeterministycznego i ε-przejściami polega na dodatkowej etykiecie jaka może towarzyszyć przejściom: ε. Zobaczmy jak definiujemy wpływ ε-przejść na działanie automatów:

Def. 31. Niech A będzie zbiorem stanów, $A \subseteq Q$. Przez $D(A)$ oznaczamy zbiór stanów, które można osiągnąć z A poprzez ε-przejścia.

Rozszerzenie funkcji przejścia $\delta^* : 2^Q \times \Sigma^* \to 2^Q$ definiujemy następująco:

$$\delta^*(A, \varepsilon) = D(A)$$

$$\delta^*(A, ax) = \delta^*(\bigcup_{q \in D(A)} \delta(q, a), x)$$

□

Każde obliczenie automatu z ε-przejściami to przeplot ε-przejść i zwykłych przejść. Odpowiada temu w definicji δ^* przeplatanie funkcji D i δ.

Def. 32. Język akceptowany przez automat z ε-przejściami to:

$$L(M) = \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\}$$

□

Powyższa definicja jest identyczna jak w przypadku automatów niedeterministycznych. Pamiętajmy jednak, że funkcja δ^*, opisująca działanie automatu, jest zdefiniowana inaczej.

Pokażemy teraz, że automaty z ε-przejściami i automaty niedeterministyczne mają taką samą siłę wyrazu. (Tym samym, automaty z ε-przejściami mają taką samą siłę wyrazu co automaty deterministyczne.) Powiedzieliśmy już wcześniej, że automaty niedeterministyczne są szczególnym przypadkiem automatów z ε-przejściami. Musimy pokazać, że ε-przejścia nie zwiększają siły wyrazu i że zawsze można je wyeliminować, zamieniając automat z ε-przejściami w automat niedeterministyczny.

Eliminacja ε-przejść Eliminacja ε-przejść opiera się na dodaniu dodatkowych przejść i stanów początkowych, tak aby definicje funkcji δ^* dla automatów niedeterministycznych i z ε-przejściami były sobie równoważne. W tym celu dodamy dodatkowe stany początkowe oraz dodatkowe przejścia.

Niech $M = \langle Q, \Sigma, \delta, S, F \rangle$ będzie automatem skończonym z ε-przejściami. Równoważny mu niedeterministyczny automat skończony M' ma postać $M' = \langle Q, \Sigma, \delta', S', F \rangle$, gdzie $S' = D(S)$, $\delta'(q, a) = D(\delta(q, a))$ (dla $a \neq \varepsilon$).

Dowód poprawności Zbiory stanów i stanów akceptujących są w obu automatach takie same. Pokażemy przez indukcję (ze względu na $|x|$), że dla dowolnego $A \subseteq Q$ zachodzi $\delta^*(A, x) = \delta'^*(D(A), x)$.

1. Dla $x = \varepsilon$ mamy:
$$\delta^*(A, \varepsilon) = D(A) = \delta'^*(D(A), \varepsilon)$$

2. Załóżmy, że teza jest prawdziwa dla x. Pokażemy jej prawdziwość dla ax (dla dowolnego $a \in \Sigma$).

$$\begin{aligned}
\delta^*(A, ax) &= \delta^*(\bigcup_{q \in D(A)} \delta(q, a), x) = \\
&= \delta'^*(D(\bigcup_{q \in D(A)} \delta(q, a)), x) = \\
&= \delta'^*(\bigcup_{q \in D(A)} D(\delta(q, a)), x) = \\
&= \delta'^*(\bigcup_{q \in D(A)} \delta'(q, a), x) = \delta'^*(D(A), ax)
\end{aligned}$$

Stąd mamy:

$$L(M') = \{x \in \Sigma^* : \delta'^*(S', x) \cap F \neq \emptyset\} = \{x \in \Sigma^* : \delta^*(D(S), x) \cap F \neq \emptyset\} = L(M)$$

Czyli oba automaty, M i M', akceptują ten sam język. □

5.5 Podsumowanie

W tym wykładzie przedstawiliśmy automaty niedeterministyczne i automaty z ε-przejściami. Pokazaliśmy, że mają one taką samą siłę wyrazu Siła wyrazu automatów deterministycznych, niedeterministycznych i z ε-przejściami jest dokładnie taka sama.

5.6 Skorowidz

- **Automat potęgowy** to deterministyczny automat skończony równoważny danemu niedeterministycznemu automatowi skończonemu $M = \langle Q, \Sigma, \delta, S, F\rangle$ następującej postaci: $M' = \langle 2^Q, \Sigma, \delta', S, F'\rangle$, gdzie $\delta'(A, a) = \delta^*(A, a)$, $F' = \{A \subseteq Q : A \cap F \neq \emptyset\}$.

- **Automat skończony, niedeterministyczny** to dowolna taka piątka $M = \langle Q, \Sigma, \delta, S, F\rangle$, w której: Q to skończony zbiór stanów, Σ to (skończony) alfabet wejściowy, δ to funkcja przejścia $\delta : Q \times \Sigma \to 2^Q$, $S \subseteq Q$ to zbiór stanów początkowych, $F \subseteq Q$ to zbiór stanów akceptujących.

- **Automat skończony z ε-przejściami** to dowolna taka piątka $M = \langle Q, \Sigma, \delta, S, F\rangle$, w której: Q jest skończonym zbiorem stanów, Σ to (skończony) alfabet, $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ to funkcja przejścia, $S \subseteq Q$ to zbiór stanów początkowych, oraz $F \subseteq Q$ to zbiór stanów akceptujących.

- **Determinizacja** niedeterministycznego automatu skończonego polega na skonstruowaniu równoważnego mu automatu potęgowego.

- **Eliminacja ε-przejść** to konstrukcja niedeterministycznego automatu skończonego równoważnego danemu automatowi skończonemu z ε-przejściami.

- **Język akceptowany** przez niedeterministyczny automat skończony $M = \langle Q, \Sigma, \delta, S, F\rangle$ to:
$$L(M) = \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\}$$

- **Język akceptowany** przez automat skończony z ε-przejściami to:
$$L(M) = \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\}$$

- **Niedeterminizm** stanowi koncepcję przeciwną do determinizmu: Nawet gdybyśmy mieli pełną wiedzę na temat stanu Świata w danej chwili, to i tak nie bylibyśmy w stanie przewidzieć przyszłości. Co najwyżej moglibyśmy określić jakie są możliwe przyszłe dzieje.

  W informatyce niedeterminizm oznacza, że uruchamiając kilka razy ten sam program czy inny mechanizm obliczeniowy, dla tych samych danych możemy uzyskać różne wyniki.

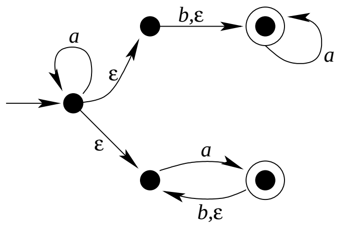
5.7 Praca domowa

1. (2 p.) Podaj automat niedeterministyczny akceptujący słowa nad alfabetem $\{a, b\}$ zawierające podsłowo *babbaab*.

2. (5 p.) Zdeterminizuj następujący automat niedeterministyczny, pomiń stany nieosiągalne:

	a	b
$\to$ 1	2, 4	–
2	3	–
F 3	–	1
4	4	3

3. (3 p.) Wyeliminuj ε-przejścia w poniższym automacie. Podaj wzorzec opisujący język akceptowany przez ten automat.

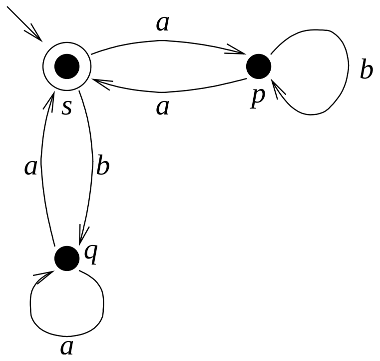


5.8 Ćwiczenia

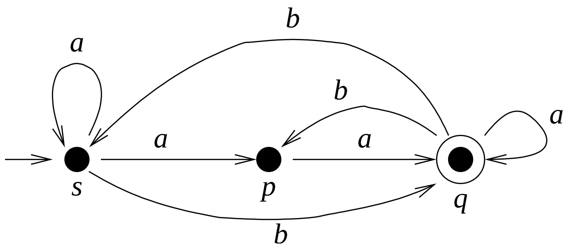
1. Podaj automat niedeterministyczny akceptujący słowa nad alfabetem $\{a, b\}$ zawierające podsłowo *ababb*. Rozwiązanie[74]

2. Zdeterminizuj następujące automaty niedeterministyczne. Pomiń stany nieosiągalne.

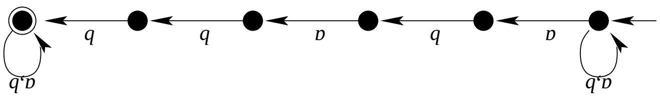
   (a)



   (b)



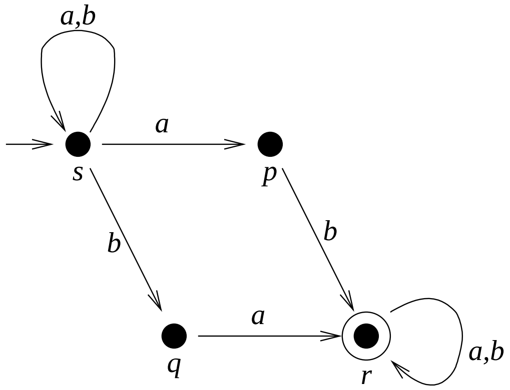
74



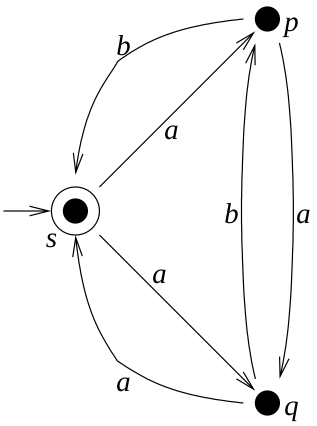
Taki automat wystarczy, że „zgadnie" gdzie występuje szukane słowo i sprawdzi czy ono faktycznie tam jest.

Rozwiązanie[75]

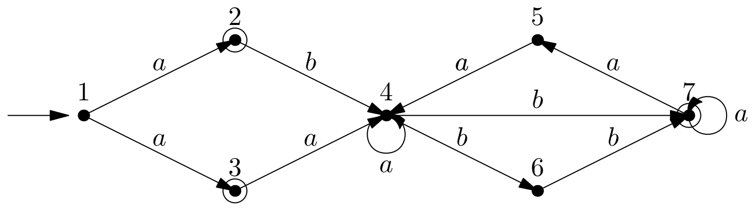
(c)



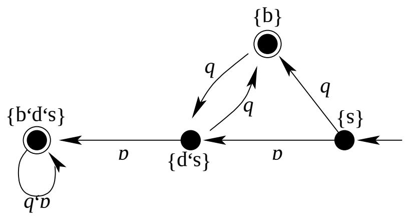
(d)



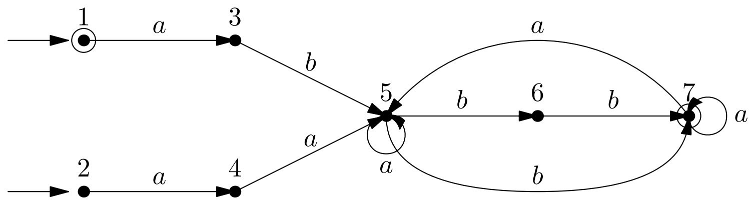
(e)



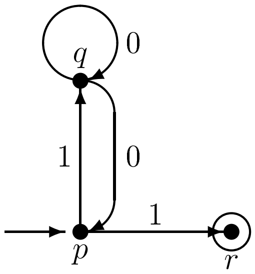
[75] Oto odpowiedni automat potęgowy (ograniczony do stanów osiągalnych):



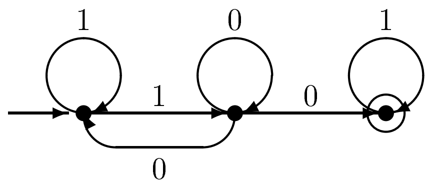
(f)



(g)

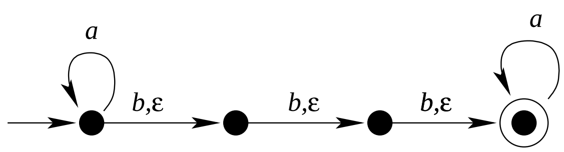


(h)



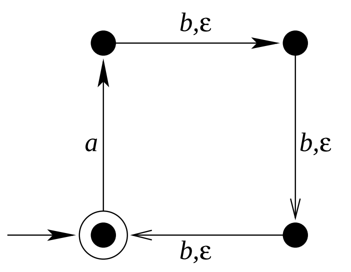
3. Wyeliminuj ε-przejścia w następujących automatach:

(a)

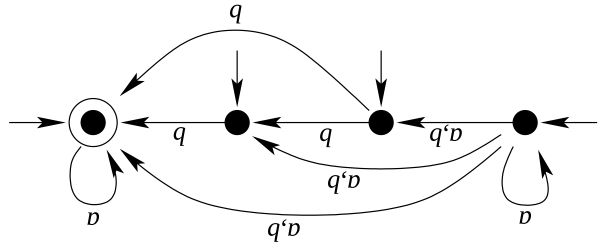


Rozwiązanie[76]

(b)



[76] Oto jedno z możliwych rozwiązań. Nie jest ono najprostsze, ale jest wynikiem zastosowania konstrukcji przedstawionej w wykładzie. Język akceptowany przez automat, to $a^*b?b?b?a^*$.



Wykład 6. Równoważność wzorców, wyrażeń regularnych i automatów skończonych

Tematem tego wykładu jest pokazanie, że wzorce, wyrażenia regularne i automaty skończone mają taką samą siłę wyrazu. Wszystkie one opisują dokładnie języki regularne.

Tw. 1. *Niech Σ będzie ustalonym alfabetem, a A niech będzie językiem nad tym alfabetem, $A \subseteq \Sigma^*$. Następujące stwierdzenia są równoważne:*

a) *A istnieje taki automat skończony M, że $A = L(M)$,*

b) *A jest regularny, tzn. $A = L(\alpha)$ dla pewnego wzorca α,*

c) *$A = L(\alpha)$ dla pewnego wyrażenia regularnego α.*

Dowód: Pokażemy, że $c) \Rightarrow b)$, $b) \Rightarrow a)$ i $a) \Rightarrow c)$.

$c) \Rightarrow b)$ Oczywiste, bo każde wyrażenie regularne jest wzorcem.

$b) \Rightarrow a)$ Indukcyjnie ze względu na strukturę wzorca budujemy automat z ε-przejściami, o jednym stanie początkowym i jednym stanie akceptującym, który akceptuje dokładnie słowa pasujące do wzorca.

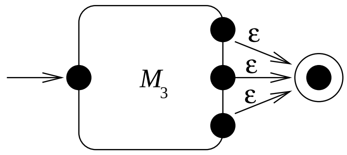
- Dla wzorców postaci ε, $\emptyset$ i a (dla $a \in \Sigma$) odpowiednie automaty przedstawiono na poniższym rysunku.

ε $\emptyset$ a

- Dla wzorca postaci $\overline{\alpha}$ konstrukcja przebiega w kilku krokach:
  1. Niech M_1 będzie automatem skonstruowanym dla wzorca α, $L(M_1) = L(\alpha)$.
  2. Niech $M_2 = \langle Q, \Sigma, \delta, s, F\rangle$ będzie automatem powstałym w wyniku eliminacji ε-przejść i determinizacji automatu M_1.
  3. Niech $M_3 = \langle Q, \Sigma, \delta, s, Q \setminus F\rangle$. Analogicznie jak w dowodzie twierdzenia 4.4, M_3 akceptuje dopełnienie języka akceptowanego przez M_2:

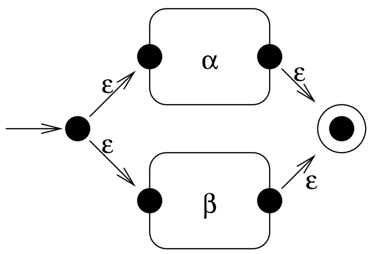
  $$L(M_3) = \overline{L(M_2)} = \overline{(L(\alpha))} = L(\overline{\alpha})$$

  4. Niech M_4 będzie automatem powstałym z M_3 poprzez dodanie: nowego stanu akceptującego, ε-przejść z dotychczasowych stanów akceptujących do nowego stanu akceptującego i zmianą dotychczasowych stanów akceptujących na nieakceptujące. Konstrukcję tę przedstawia poniższy rysunek:

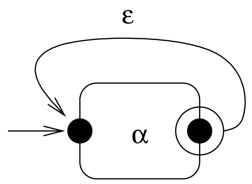


Konstrukcja ta nie zmienia języka akceptowanego przez automat, $L(M_4) = L(M_3) = L(\overline{\alpha})$, natomiast automat ten spełnia przyjęte założenia o jednym stanie początkowym i jednym stanie akceptującym.

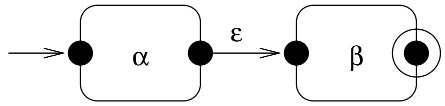
- Jeśli wzorzec jest postaci: $\alpha \mid \beta$, α^+ lub $\alpha\beta$, to konstruujemy automat z automatów akceptujących $L(\alpha)$ i $L(\beta)$, w sposób przedstawiony na rysunku. Stany akceptujące zmieniamy tak, że tylko stan zaznaczony na rysunku pozostaje akceptujący:



$\alpha \mid \beta$



α^+



$\alpha\beta$

- Dla wzorców postaci $\alpha = \beta \cap \gamma$ i $\alpha = \beta^*$ wykorzystujemy tożsamości:

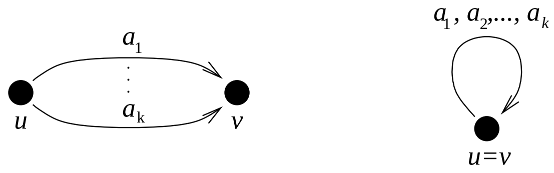
$$\beta \cap \gamma \equiv \overline{\overline{\beta} \mid \overline{\gamma}}$$

$$\beta^* \equiv \varepsilon \mid \beta^+$$

i wcześniej opisane konstrukcje.

$a) \Rightarrow c)$ Niech $M = \langle Q, \Sigma, \delta, S, F \rangle$ będzie danym automatem skończonym (deterministycznym lub niedeterministycznym). Konstrukcja wyrażenia regularnego opisującego język akceptowany przez M przebiega rekurencyjnie. Budujemy wyrażenia regularne opisujące coraz większe fragmenty automatu. Niech u i v będą stanami, a X będzie zbiorem stanów, $u, v \in Q$, $X \subseteq Q$. Wyrażenie regularne $\alpha_{u,v}^X$ opisuje te słowa, za pomocą których można przejść z u do v zatrzymując się po drodze jedynie w stanach z X. Wyrażenie $\alpha_{u,v}^X$ definiujemy indukcyjnie ze względu na $|X|$.

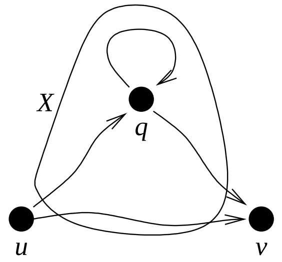
1. Załóżmy, że $X = \emptyset$. Oznaczmy przez $a_1, \ldots, a_k$ wszystkie znaki powodujące przejście z u do v. Możliwe są dwa przypadki: $u = v$ lub $u \neq v$:



Jeśli $u \neq v$ i nie ma żadnych przejść z u do v, to wczytanie żadnego słowa nie spowoduje przejścia z u do v. Tak więc:

$$\alpha^{\emptyset}_{u,v} = \begin{cases} a_1 \mid \ldots \mid a_k \mid \varepsilon & ; \quad u = v \\ a_1 \mid \ldots \mid a_k & ; \quad u \neq v \land k > 0 \\ \emptyset & ; \quad u \neq v \land k = 0 \end{cases}$$

2. Załóżmy, że $X \neq \emptyset$. Niech q będzie dowolnym stanem należącym do X, $q \in X$. Wówczas możliwe są dwa rodzaje przejścia z u do v:
   - Przechodzimy z u do v zatrzymując się po drodze w stanach należących do X, ale nie w q. Czyli po drodze zatrzymujemy się w stanach należących do $X \setminus \{q\}$. Słowa, które możemy wczytać w ten sposób są opisane przez $\alpha^{X\setminus\{q\}}_{u,v}$.
   - Przechodząc z u do v przynajmniej raz, a być może więcej razy, przechodzimy przez q. Całą trasę możemy podzielić na odcinki, na każdym przejściu przez stan q.
     Pierwszy odcinek to przejście z u do q (pierwsza wizyta w q), przechodząc po drodze przez stany z $X \setminus \{q\}$. To co możemy wczytać przechodząc przez ten pierwszy odcinek jest opisane przez $\alpha^{X\setminus\{q\}}_{u,q}$.
     Następnie pewną liczbę razy (być może zero) wychodzimy z q i wracamy do q. To co możemy wczytać w trakcie pojedynczego przejścia z q do q jest opisane przez $\alpha^{X\setminus\{q\}}_{q,q}$.
     Na koniec, wychodzimy z q (ostatnia wizyta w q) i przechodzimy do v. To co możemy w trakcie tego przejścia wczytać opisuje $\alpha^{X\setminus\{q\}}_{q,v}$.



Łącząc obie powyższe możliwości, wszystko co może zostać wczytane w trakcie przejścia z u do v, po drodze zatrzymując się w stanach należących do X, jest opisane przez:

$$\alpha^{X}_{u,v} = \alpha^{X\setminus\{q\}}_{u,v} \mid \alpha^{X\setminus\{q\}}_{u,q}(\alpha^{X\setminus\{q\}}_{q,q})^*\alpha^{X\setminus\{q\}}_{q,v}$$

Słowa, które są akceptowane przez M to te, które mogą zostać wczytane przy przejściu z jednego ze stanów początkowych do jednego ze stanów akceptujących. Niech $S = \{s_1, \dots, s_k\}$, $F = \{f_1, \dots, f_l\}$. Wówczas:

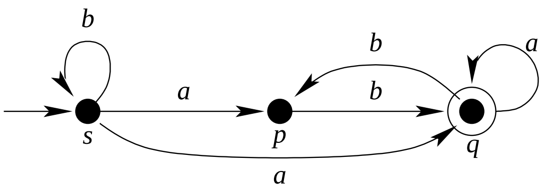
$$L(M) = L(\alpha^Q_{s_1,f_1} \mid \dots \mid \alpha^Q_{s_1,f_l} \mid \dots \mid \alpha^Q_{s_k,f_1} \mid \dots \mid \alpha^Q_{s_k,f_l})$$

$\square$

Tym samym pokazaliśmy, że: wzorce, wyrażenia regularne, oraz automaty skończone (wszystkie trzy poznane rodzaje), wszystkie opisują dokładnie klasę języków regularnych.

Konstrukcja wyrażenia regularnego odpowiadającego automatowi skończonemu, podana w powyższym dowodzie daje dosyć skomplikowane wyrażenie regularne. Zauważmy, że rozmiar wynikowego wyrażenia regularnego rośnie czterokrotnie z każdym kolejnym stanem automatu! Jednocześnie wiele powstających podwyrażeń można pominąć, gdyż nie wnoszą nic do wyniku. Jeśli więc będziemy chcieli przekształcić jakiś automat na wyrażenie regularne, to nie musimy sztywno trzymać się pokazanej powyżej konstrukcji. Gdy jednak zdrowy rozsądek nie podpowiada nam jak sobie poradzić, wówczas powinniśmy zastosować konstrukcję z dowodu, zmniejszając rozmiar rozważanego automatu. Ilustruje to poniższy przykład:

Przykład: Przekształćmy następujący automat na wyrażenie regularne:



		a	b
$\rightarrow$	s	p, q	s
	p		q
F	q	q	p

Język akceptowany przez automat jest opisany wyrażeniem.

$$\alpha^{\{s,p,q\}}_{s,q} = \alpha^{\{s,q\}}_{s,q} \mid \alpha^{\{s,q\}}_{s,p}(\alpha^{\{s,q\}}_{p,p})^*\alpha^{\{s,q\}}_{p,q}$$

Patrząc na automat możemy wywnioskować:

$$\alpha^{\{s,q\}}_{s,q} \equiv b^*aa^*$$

$$\alpha^{\{s,q\}}_{p,p} \equiv \varepsilon \mid ba^*b$$

$$\alpha^{\{s,q\}}_{p,q} \equiv ba^*$$

Stąd:

$$\alpha^{\{s,p,q\}}_{s,q} \equiv b^*aa^* \mid \alpha^{\{s,q\}}_{s,p}(\varepsilon \mid ba^*b)^*ba^*$$

Dalej:

$$\alpha^{\{s,q\}}_{s,p} = \alpha^{\{s\}}_{s,p} \mid \alpha^{\{s\}}_{s,q}(\alpha^{\{s\}}_{q,q})^*\alpha^{\{s\}}_{q,p}$$

Zauważmy, że:

$$\alpha^{\{s\}}_{s,p} \equiv \alpha^{\{s\}}_{s,q} \equiv b^*a$$

$$\alpha^{\{s\}}_{q,q} \equiv \varepsilon \mid a$$

$$\alpha^{\{s\}}_{q,p} \equiv b$$

Czyli:

$$\alpha^{\{s,q\}}_{s,p} \equiv b^*a \mid b^*a(\varepsilon \mid a)^*b$$

Stąd:

$$\begin{aligned}\alpha^{\{s,p,q\}}_{s,q} &\equiv b^*aa^* \mid (b^*a \mid b^*a(\varepsilon \mid a)^*b)\,(\varepsilon \mid ba^*b)^*ba^* \\ &\equiv b^*aa^* \mid (b^*a \mid b^*aa^*b)(ba^*b)^*ba^*\end{aligned}$$

6.1 Zastosowania automatów skończonych

Automaty skończone nie są tylko teoretycznym narzędziem do badania języków regularnych. Mają one również wiele zastosowań praktycznych. Na przykład, stosuje się je do wyszukania wzorca lub podsłowa w tekście — dla danego wzorca czy podsłowa należy skonstruować odpowiedni automat (deterministyczny), a następnie wczytać analizowany tekst; każde przejście takiego automatu przez stan akceptujący reprezentuje jedno wystąpienie wzorca czy podsłowa w tekście.

Dalej, analizatory leksykalne są realizowane jako automaty skończone. Lex na podstawie specyfikacji tworzy odpowiedni automat skończony, który wyszukuje wystąpienia leksemów i równocześnie określa które fragmenty kodu ze specyfikacji należy zastosować.

Automaty skończone mają też zastosowanie w bardzo młodej dziedzinie informatyki, weryfikacji modelowej (ang. *model-checking*). Weryfikacja modelowa jest stosowana do badania, czy określony mechanizm (np. układ elektroniczny, protokół komunikacyjny, prosty program o ograniczonej liczbie stanów) posiada wymagane własności. W tym celu tworzy się automat skończony będący modelem systemu. Język akceptowany przez ten automat reprezentuje możliwe przebiegi zdarzeń. Następnie bada się, czy język akceptowany przez automat zawiera się w języku złożonym z dopuszczalnych przez specyfikację przebiegów zdarzeń. Zaletą weryfikacji modelowej w porównaniu z testowaniem jest to, że weryfikacja modelowa obejmuje **wszystkie** możliwe przebiegi zdarzeń, a nie tylko niektóre.

6.2 Podsumowanie

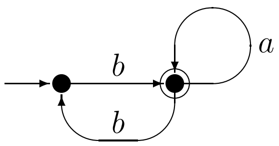
W niniejszym wykładzie pokazaliśmy, że: wzorce, wyrażenia regularne, oraz automaty skończone (wszystkie trzy poznane rodzaje), wszystkie opisują dokładnie klasę języków regularnych. Ponieważ dowód był konstruktywny, widać z niego jak można tłumaczyć wzorce na automaty i odwrotnie.

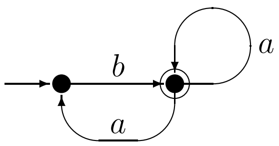
6.3 Skorowidz

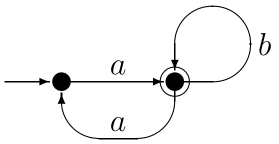
- **Weryfikacja modelowa** (ang. *model-checking*) to metoda weryfikacji polegająca na stworzeniu modelu weryfikowanego systemu w postaci np. automatu skończonego i wykazaniu jego poprawności poprzez badanie stworzonego modelu.

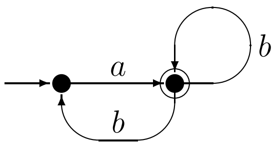
6.4 Praca domowa

1. (3 p.) Dopasuj do siebie automaty i wyrażenia regularne:

   a) 

   b) 

   c) 

   d) 

   1) $b(ab \mid a)^*$.
   2) $a(ba \mid b)^*$,
   3) $a(aa \mid b)^*$,
   4) $b(bb \mid a)^*$,

2. (4 p.) Podaj automat skończony (dowolnego rodzaju) akceptujący język opisany wzorcem: $b(aba)^+b^* \mid (aa)^*(ab)^+$.

3. (3 p.) Podaj wyrażenie regularne opisujące język akceptowany przez następujący automat niedeterministyczny:

	a	b
$\to F1$	$2, 4$	
2	2	4
3	1	
4	3	$1, 3$

6.5 Ćwiczenia

1. Dopasuj do siebie automaty i wyrażenia regularne:

a)

	a	b
$\to 1$	2	2
$F2$		2

b)

	a	b
$\to F1$	2	2
2	1	1

c)

	a	b
$\to F1$		2
$F2$	1	2

d)

	a	b
$\to 1$	2	2
$F2$		1

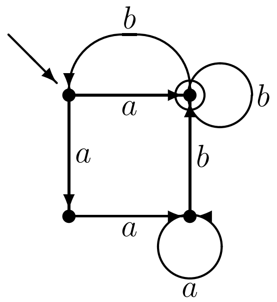
e)

	a	b
$\to F1$	1	2
$F2$	1	2

1) $(b \mid ba)^*$,
2) $(a \mid b)(ba \mid bb)^*$,
3) $(a \mid b)^*$,
4) $(a \mid b)b^*$,
5) $((a \mid b)(a \mid b))^*$.

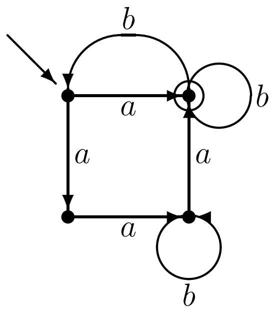
Rozwiązanie[77]

2. Dopasuj automaty do podanych wyrażeń regularnych:

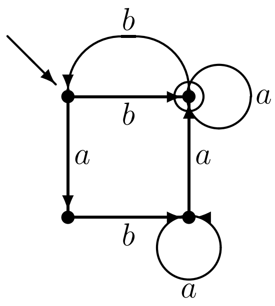


a)

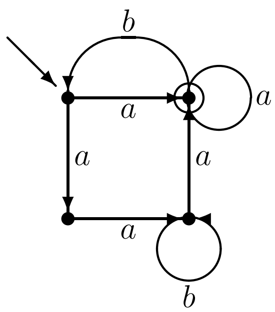
[77] 1-c, 2-d, 3-e, 4-a, 5-b



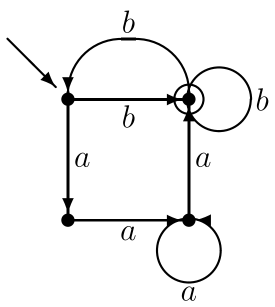
b)



c)



d)



e)

1) $(aaa^*a \mid b)(b^* \mid bb \mid baaa^*a)^*$,

2) $(b \mid aba^*a)(a^* \mid bb \mid baba^*a)^*$,

3) $(aab^*a \mid a)(b^* \mid ba \mid baab^*a)^*$,

4) $(a \mid aaa^*b)(b^* \mid ba \mid baaa^*b)^*$,

5) $(a \mid aab^*a)(a^* \mid ba \mid baab^*a)^*$.

Rozwiązanie[78]

3. Podaj automaty skończone (dowolnego rodzaju) odpowiadające następującym wzorcom:

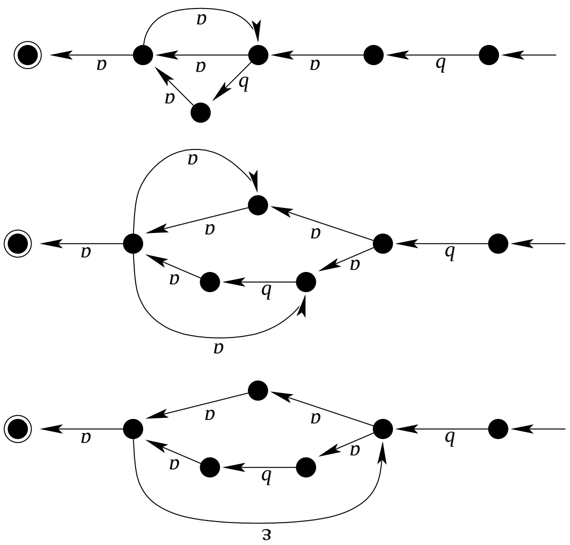
   - $(a(ab)^* \mid a^*bb)^*$,
   - $(ab^*a)^* \mid (ba^*b)^*$,
   - $b(aba \mid aa)^+a$, Rozwiązanie[79]
   - $(ab \mid abb \mid aabbb)^+$,
   - $aa \mid (a \mid bab)b^*$,
   - $a(bab)^* \mid a((aa)^* \mid (bb)^*)$,
   - $a^*[((ba)^* \mid bb)^* \mid aa]^*ab$.

4. Podaj deterministyczne automaty skończone akceptujące języki opisane wyrażeniami regularnymi:

   - $a^*b^*a^*$,
   - $a^*b^* \mid b^*a^*$,

[78] 1-e, 2-c, 3-b, 4-a, 5-d.

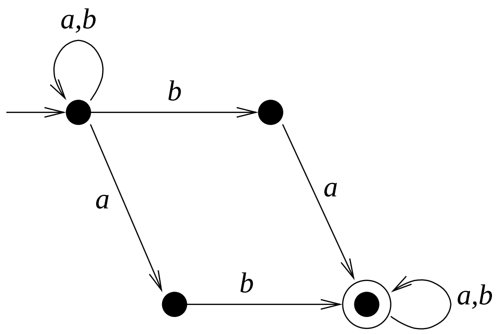
[79] Oto kilka możliwych rozwiązań:



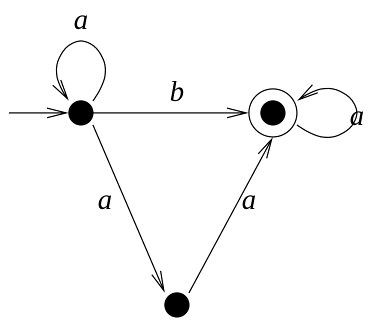
- $(ab)^* \mid (ba)^*$,
- $a(baba)^+a$,
- $(bab \mid abba)^*aba$,
- $(aba)^* \mid (bab)^*$,
- $(aba \mid abba)^*bab$,
- $((bab \mid bba)(bab \mid baba))^+$.

5. Podaj wyrażenia regularne odpowiadające automatom:

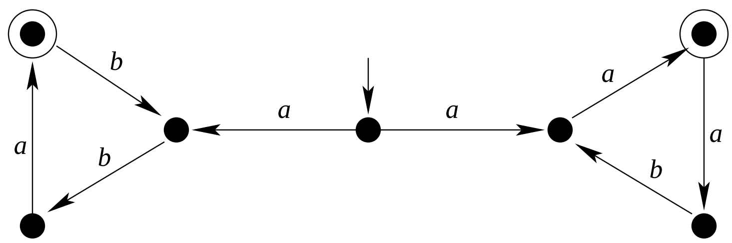
(a)



(b)



(c)



(d)

	a	b
$\rightarrow F1$	$2, 3$	4
2		1
3	1	
4	1	1

Rozwiązanie[80]

---

[80] Oto kilka możliwych odpowiedzi:

$$(ab \mid aa \mid b(a \mid b))^*$$

$$(ab \mid aa \mid ba \mid bb)^*$$

$$((a \mid b)(a \mid b))^*$$

Wykład 7. Lemat o pompowaniu dla języków regularnych

7.1 Wstęp

Okazuje się, że nie każdy język jest regularny. Standardowy przykład języka, który nie jest regularny, to $A = \{a^n b^n : n \geq 0\}$. Intuicyjnie jest to dosyć oczywiste. Żeby język był regularny, musi być możliwe stwierdzenie czy słowo należy do języka przy użyciu stałej pamięci. Jednak żeby sprawdzić, czy słowo jest postaci $a^n b^n$, musimy najpierw wczytać litery a i zapamiętać ich dokładną liczbę. Zapamiętanie liczby n wymaga przynajmniej $\log_2 n$ bitów pamięci, a to nie jest stała. Tak więc, jaką byśmy pamięcią nie dysponowali, zawsze znajdzie się dostatecznie duże n, takie że po wczytaniu a^n nie będziemy dokładnie wiedzieli ile tych liter a było. Tym samym nie będziemy w stanie sprawdzić, czy liter b jest dokładnie tyle samo.

Spróbujmy sformalizować powyższe rozumowanie. Dowodzimy nie wprost, że A nie jest regularne. Załóżmy przeciwnie, że A jest regularne. Niech $M = (Q, \Sigma, \delta, s, F)$ będzie automatem akceptującym A i niech $n = |Q|$. Wówczas z zasady szufladkowej Dirichleta istnieje taki stan q, $u \geq 0$ oraz $v > 0$, że $n = u + v$, $\delta^*(s, a^u) = \delta^*(q, a^v) = q$. Wówczas tak samo jest akceptowane, lub nie, słowo $a^u b^n$ i $a^n b^n$. Sprzeczność. A więc założenie, że A jest regularny było fałszywe.

7.2 Lemat o pompowaniu

Powyższe rozumowanie można uogólnić, uzyskując bardziej ogólne narzędzie do wykazywania, że pewne języki nie są regularne.

Tw. 2 (Lemat o pompowaniu dla języków regularnych)**.** *Niech A będzie językiem regularnym. Wówczas istnieje takie k, że dla dowolnych słów x, y i z takich, że $xyz \in A$ oraz $|y| \geq k$, można y podzielić na słowa u, v i w, $y = uvw$ w taki sposób, że $v \neq \varepsilon$ i dla wszystkich $i \geq 0$ $xuv^iwz \in A$.*

Dowód: Skoro A jest językiem regularnym, to istnieje deterministyczny automat skończony rozpoznający A — oznaczmy go przez $M = \langle Q, \Sigma, \delta, s, F \rangle$. Wybieramy $k = |Q|$.

Niech x, y i z będą takimi słowami, że $xyz \in A$ oraz $|y| \geq k$. Zauważmy, że w obliczeniu automatu M akceptującego słowo xyz, w trakcie wczytywania słowa y, pojawia się przynajmniej $k + 1$ stanów.

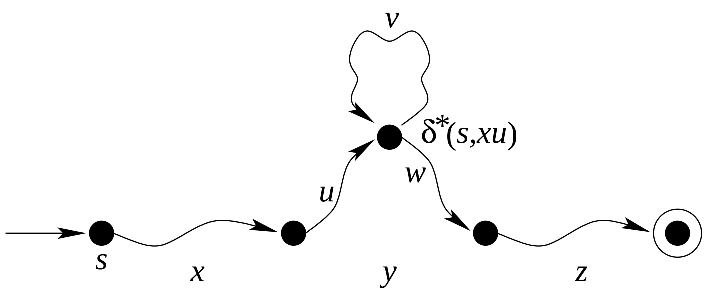
Z zasady szufladkowej Dirichleta wynika, że przynajmniej jeden z tych stanów musi się powtarzać. Weźmy pierwsze powtórzenie się stanu w trakcie wczytywania słowa y.

Możemy więc podzielić słowo y na trzy części $y = uvw$:

- u — od początku słowa y do pierwszego wystąpienia powtarzającego się stanu,
- v — od pierwszego do drugiego wystąpienia powtarzającego się stanu, oraz
- w — od drugiego wystąpienia powtarzającego się stanu do końca słowa y.

Zauważmy, że słowu v odpowiada „pętelka”, tzn. $v \neq \varepsilon$, oraz:

$$\delta^*(s, xu) = \delta^*(s, xuv)$$



Stąd, dla dowolnego $i \geq 0$ mamy:

$$\begin{aligned}
\delta^*(s, xuv^i) &= \delta^*(\delta^*(s, xuv), v^{i-1}) = \\
&= \delta^*(\delta^*(s, xu), v^{i-1}) = \\
&= \delta^*(s, xuv^{i-1}) = \\
&\vdots \\
&= \delta^*(s, xu)
\end{aligned}$$

Stąd zaś otrzymujemy:

$$\delta^*(s, xuv^iwz) = \delta^*(s, xuvwz) \in F$$

Czyli $xuv^iwz \in A$. $\square$

7.3 Zastosowanie lematu o pompowaniu

Lemat o pompowaniu służy pokazywaniu, że określone języki **nie są** regularne. Schemat takiego dowodu przebiega nie wprost. Lemat o pompowaniu ma postać implikacji: jeśli język jest regularny, to ma pewne własności. Dowód, że język nie jest regularny przebiega według schematu: skoro dany język nie posiada tych własności, to nie może być regularny.

Specjalnie z myślą o takim zastosowaniu powstało alternatywne sformułowanie lematu o pompowaniu:

Lemat o pompowaniu dla języków regularnych — sformułowanie alternatywne: Niech A będzie językiem. Jeżeli dla każdego $k \geq 0$ istnieją takie słowa x, y, z, że $xyz \in A$, $|y| \geq k$ oraz dla dowolnego podziału słowa y na słowa u, v, w, $y = uvw$ takiego, że $v \neq \varepsilon$, istnieje takie $i \geq 0$, że $xuv^iwz \notin A$, wówczas A nie jest regularny.

Nawet w postaci odwróconej lemat o pompowaniu jest trudny do zrozumienia, gdyż mamy tu do czynienia z czterema poziomami zagnieżdżonych naprzemiennych kwantyfikatorów uniwersalnych i egzystencjalnych. Takie zagnieżdżenie naprzemiennych kwantyfikatorów możemy sobie wyobrazić jako ruchy dwóch graczy — gracze na przemian wskazują jakie wartości powinny przyjąć zmienne spod kwantyfikatorów, jeden gracz spod uniwersalnych, a drugi spod egzystencjalnych. Pierwszy gracz stara się pokazać, że dany język jest regularny, a drugi, że nie. O wygranej decyduje, czy dla wybranych wartości zmiennych zachodzi własność objęta kwantyfikatorami. Gra jest skończona, więc jeden z graczy ma strategię wygrywającą. W zależności od tego, który z graczy to jest, język może być regularny lub nie.

Lemat o pompowaniu dla języków regularnych — gra z Demonem: Prowadzimy z Demonem grę. Staramy się pokazać, że język nie jest regularny, a Demon stara się pokazać, że język mimo wszystko jest regularny. Zasady gry są następujące:

1. Demon wybiera $k \geq 0$. (Jeśli A jest faktycznie regularny, to Demon może wybrać za k liczbę stanów automatu akceptującego A).
2. Wybieramy takie słowa x, y, z, że $xyz \in A$ i $|y| \geq k$. Jeżeli takie słowa nie istnieją, czyli nie możemy wykonać ruchu, przegrywamy.
3. Demon dzieli y na takie słowa u, v, w, $y = uvw$, że $v \neq \varepsilon$. (Jeśli A jest faktycznie regularny, to Demon może podzielić y na pierwszych dwóch wystąpieniach stanu, który jako pierwszy powtarza się w trakcie wczytywania słowa y.)
4. Wybieramy $i \geq 0$.

Jeśli $xuv^iwz \notin A$, to wygraliśmy, wpp. wygrał Demon.

Oryginalne sformułowanie lematu o pompowaniu mówi, że jeżeli język jest regularny, to Demon ma strategię wygrywającą. Natomiast odwrotne sformułowanie mówi, że jeżeli my mamy strategię wygrywającą, to język nie jest regularny. Pamiętajmy, że lemat o pompowaniu nie służy do pokazywania, że język jest regularny. Nie zawsze pozwala on na pokazanie, że język, który nie jest regularny faktycznie taki nie jest — istnieją języki, które nie są regularne i spełniają lemat o pompowaniu.

Zobaczmy na przykładach jak można zastosować sformułowanie lematu o pompowaniu jak gry z Demonem:

Przykład: Pokażemy, że język $A = \{a^n b^n : n \geq 0\}$ nie jest regularny.

- Demon wybiera k.
- Wybieramy $x = \varepsilon$, $y = a^k$ i $z = b^k$.
- Jakkolwiek Demon by nie podzielił słowa y, to słowo v składa się wyłącznie z literek a i jest niepuste. Możemy wybrać $i = 2$.

Słowo $xuv^2wz \notin A$, bo ma więcej literek a niż b, czyli wygraliśmy.

Powyższa strategia jest dla nas strategią wygrywającą, a zatem język A nie jest regularny.

Przykład: Pokażemy, że język $B = \{a^{2^n} : 0 \leq n\}$ nie jest regularny.

- Demon wybiera k.
- Wybieramy $x = \varepsilon$, $y = a^{2^k-1}$, $z = a$.
- Jakkolwiek Demon by nie podzielił słowa y, to słowo v składa się wyłącznie z literek a oraz $0 < |v| < 2^k$. Możemy wybrać $i = 2$.

Słowo $xuv^2wz \not\in B$, bo $2^k < |xuv^2wz| < 2 \cdot 2^k = 2^{k+1}$ a zatem wygraliśmy.

Powyższa strategia jest dla nas strategią wygrywającą, a zatem język B nie jest regularny.

Przykład: Pokażemy, że język $C = \{a^{n!} : n \geq 0\}$ nie jest regularny.

- Demon wybiera k.
- Wybieramy $x = \varepsilon$, $y = a^{(k+1)!-1}$, $z = a$.
- Demon wybiera słowa u, v, w. Mamy $0 < |v| < (k + 1)!$.
- Wybieramy $i = k + 2$.

Mamy $|xuv^iwz| = (k+1)!+(i-1)\cdot|v|$. Tak więc, z jednej strony mamy $|xuv^iwz| > (k+1)!$, ale z drugiej:

$$|xuv^iwz| = (k + 1)! + (k + 1) \cdot |v| < (k + 1)! + (k + 1) \cdot (k + 1)! = (k + 2)(k + 1)! = (k + 2)!$$

Stąd, $|xuv^iwz| \not\in C$, czyli wygraliśmy.

Powyższa strategia jest dla nas strategią wygrywającą, a zatem język C nie jest regularny.

7.4 Inne metody dowodzenia nieregularności języków

Wykorzystanie lematu o pompowaniu nie jest jedynym sposobem pokazywania, że jakiś język nie jest regularny. Jeśli już wiemy o jakimś języku (B), że nie jest regularny, to możemy pokazać, że jakiś inny język (A) nie jest regularny korzystając z własności domknięcia klasy języków regularnych. Rozumujemy przy tym nie wprost — gdyby język A był regularny, to B też, a że B nie jest, to i A nie może być.

Przykład: Pokażemy, że język $D = \{a^nb^m : n \neq m\}$ nie jest regularny. Gdyby był on regularny, to (ze względu na domknięcie klasy języków regularnych na dopełnienie) język $A = \{a^nb^n : n \geq 0\}$ byłby też regularny, a nie jest. Tak więc język D nie może być regularny.

Przykład: Rozważmy język $E \subseteq \{[,]\}$ złożonych z poprawnych wyrażeń nawiasowych. Pokażemy, że nie jest on regularny. Gdyby był on regularny, to $E \cap L([^*]^*) = \{[^n]^n : n \geq 0\}$ też byłby regularny. Natomiast język $\{[^n]^n : n \geq 0\}$, tak jak $A = \{a^n b^n : n \geq 0\}$, nie jest regularny. Tak więc język E nie może być regularny.

7.5 Podsumowanie

Wykład ten był poświęcony przede wszystkim lematowi o pompowaniu dla języków regularnych. Poznaliśmy trzy równoważne sformułowania tego lematu, oraz przykłady jego zastosowania do wykazywania, że określone języki nie są regularne. Poznaliśmy też metodę wykazywania, że dane języki nie są regularne, w oparciu o własności domknięcia klasy języków regularnych.

7.6 Skorowidz

- **Lemat o pompowaniu dla języków regularnych** mówi, że dla każdego języka regularnego, każde dostatecznie długie słowo w tym języku można „pompować” dowolnie je wydłużając. Wykazywanie braku tej własności jest metodą pokazywania, że dany język nie jest regularny.
- **Gra z Demonem** — alternatywne sformułowanie lematu o pompowaniu dla języków regularnych jako gry.

7.7 Praca domowa

1. (2 p.) Które z poniższych języków są regularne, a które nie? (Odpowiedzi nie musisz formalnie uzasadniać.)
   a) $\{a^{n^3} : n \geq 0\}$,
   b) $\{a^i b^j : i \cdot j \leq 42\}$,
   c) $\{a^i b^j : i \equiv j \bmod 42\}$,
   d) $\{ww : w \in \{a,b\}^*\}$,
   e) $\{a^n b^m : |n - m| \leq 42\}$
2. (4 p.) Pokaż, że język $\{a^m b^n a^{m \cdot n} : m, n \geq 0\}$ nie jest regularny.
3. (4 p.) Pokaż, że język zawierający te słowa nad alfabetem $\{a, b\}$, które zawierają tyle samo liter a i b, $\{x \in \{a,b\}^* : \#_a(x) = \#_b(x)\}$, nie jest regularny.

7.8 Ćwiczenia

1. Pokaż, że następujące języki nie są regularne:
   - $\{a^n b^{2 \cdot n} :\geq 0\}$,
   - język złożony z palindromów nad alfabetem $\{a, b\}$, $\{x \in \{a, b\}^* : x = \mathrm{rev}(x)\}$,
   - $\{a^p : p$ jest liczbą pierwszą $\}$,
   - $\{a^{n^2} : n \geq 0\}$,
   - $\{a^i b^j c^k : i = j \vee j = k\}$

2. Które z podanych języków są regularne, a które nie? Odpowiedzi uzasadnij.

   a) $\{a^i b^j : 0 \leq i \leq j \leq 42\}$,

   b) komentarze w Pascalu,

   c) $\{a^i b^j c^k : i \neq j \vee j \neq k\}$,

   d) $\{a^i b^j a^{2(i+j)} : i, j \geq 0\}$,

   e) $\{x \in \{a, b\} : \#_a(x) \neq \#_b(x)\}$,

   f) $\{a^{2n} : n \geq 0\}$,

   g) $\{a^i b^j c^k : i + j + k \leq 42\}$,

   h) poprawne programy w Pascalu,

   i) $\{a^{42^n} : n \geq 0\}$,

   j) $\{x \in \{a, b\}^* : \#_a(x)$ jest podzielne przez $42\}$.

   Rozwiązanie[81]

---

81 Regularne są: a), b), f), g), j).

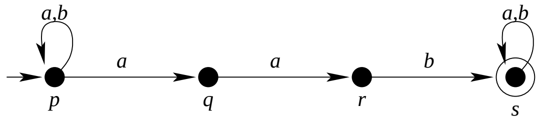
Wykład 8. Minimalizacja deterministycznych automatów skończonych

8.1 Wstęp

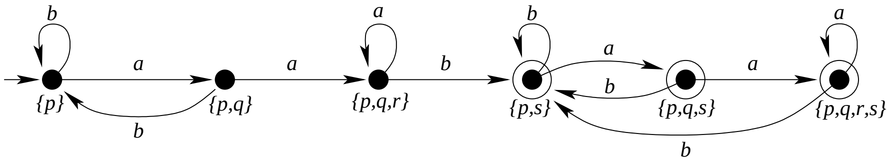
W tym wykładzie zajmiemy się problemem minimalizacji liczby stanów w deterministycznych automatach skończonych. Konstruując automaty skończone często możemy dość do kilu rozwiązań o różnej liczbie stanów. Jak znaleźć automat o najmniejszej możliwej liczbie stanów? W przypadku deterministycznych automatów skończonych znany jest efektywny algorytm przekształcający dowolny automat akceptujący dany język w automat o minimalnej liczbie stanów.

Zanim poznamy ten algorytm i udowodnimy jego poprawność, zobaczmy na przykładzie jak można zminimalizować liczbę stanów.

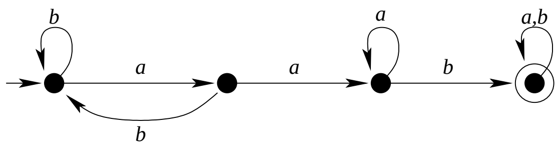
Przykład: Weźmy na początek niedeterministyczny automat skończony akceptujący te słowa (nad alfabetem $\{a, b\}$), które zawierają podsłowo aab.



Automat ten ma 4 stany, a więc odpowiadający mu automat potęgowy ma 16 stanów. Oczywiście możemy się ograniczyć do stanów osiągalnych, a tych jest 6.



Zauważmy, że możemy skleić ze sobą wszystkie stany akceptujące — po ich osiągnięciu nie można już ich opuścić, a więc słowo musi zostać zaakceptowane.



Uzyskaliśmy automat deterministyczny o 4 stanach. Czy to jest minimum? Zastanówmy się ile stanów jest niezbędnych. Potrzebny jest jeden stan akceptujący, do którego wchodzimy, gdy wczytamy podsłowo aab. Dodatkowo potrzebujemy przynajmniej 3 inne stany. Możemy je rozróżnić m.inn. po najkrótszych słowach koniecznych do osiągnięcia stanu akceptującego. Dla stanu początkowego jest to oczywiście słowo aab. Dla stanu w jakim jesteśmy po wczytaniu a jest to ab, a dla stanu, w którym jesteśmy po wczytaniu aa jest to b. Razem uzyskujemy 4 stany, a więc nasz automat jest minimalny.

8.2 Dowód istnienia automatu minimalnego

Jak widać z powyższego przykładu niektóre stany w automacie deterministycznym mogą być sobie „równoważne” i możemy je skleić. Natomiast badając, czy stany są równoważne powinniśmy zwrócić uwagę na to jakie słowa z danych stanów prowadzą do stanów akceptujących. Jeśli są tu jakieś różnice, to stanów nie możemy sklejać. Narazie są to mało precyzyjne intuicje, ale za chwilę sformalizujemy je.

Nadal jednak pozostaje otwarte pytanie, czy usunięcie stanów nieosiągalnych i sklejenie stanów „równoważnych” prowadzi do jednego automatu minimalnego? Czy też zaczynając od różnych automatów możemy uzyskać różne automaty minimalne? Pokażemy, że dla każdego języka regularnego istnieje jeden (z dokładnością do izomorfizmu) minimalny deterministyczny automat skończony.

Def. 33. Niech A będzie ustalonym językiem. Przez A/x oznaczamy język postaci:

$$A/x = \{y \in \Sigma^* : xy \in A\}$$

Języki postaci A/x dla $x \in \Sigma^*$ nazywamy *językami ilorazowymi.* □

Inaczej mówiąc, A/x to język złożony ze wszystkich tych ciągów dalszych, które uzupełniają wystąpienia prefiksów x w słowach z A. W szczególności $A/\varepsilon = A$. Wprost z definicji języka A/x wynika następujący fakt:

Fakt 7. *Dla dowolnych $x, y \in \Sigma^*$ mamy $A/xy = (A/x)/y$.*

Stąd zaś wynika kolejny fakt:

Fakt 8. *Dla dowolnych $x, y, z \in \Sigma^*$, jeśli $A/x = A/y$, to $A/xz = A/yz$.*

Jest tak gdyż:

$$A/xz = (A/x)/z = (A/y)/z = A/yz$$

Jeżeli dodatkowo A jest regularny to pojawiają się dodatkowe intuicje. Niech $M = \langle Q, \Sigma, \delta, s, F\rangle$ będzie deterministycznym automatem skończonym akceptujący A, $L(M) = A$, w którym wszystkie stany są osiągalne. A/x to język złożony z tych słów, które prowadzą ze stanu $\delta^*(s, x)$ do stanów akceptujących, (czyli jest to język akceptowany przez automat postaci $\langle Q, \Sigma, \delta, \delta^*(s, x), F\rangle$). Ponieważ to, jakie słowa prowadzą z danego stanu do stanów akceptujących zależy od tego stanu, ale nie od tego jak dostaliśmy się do tego stanu, więc zachodzi następujący fakt:

Fakt 9. *Niech $x, y \in \Sigma^*$. Jeśli $\delta^*(s, x) = \delta^*(s, y)$, to $A/x = A/y$.*

Możemy więc każdemu stanowi q automatu M przyporządkować jeden z języków ilorazowych A/x języka A — dokładnie taki, dla którego $\delta^*(s, x) = q$. Wynika stąd kolejny fakt:

Fakt 10. *Jeśli A jest regularny, to istnieje skończenie wiele języków ilorazowych języka A. Dokładniej, jest ich co najwyżej tyle, co stanów osiągalnych w deterministycznym automacie skończonym akceptującym A.*

Okazuje się, że odwrotna implikacja jest również prawdziwa.

Tw. 3 (uproszczone twierdzenie Myhilla-Nerode'a)**.** *Niech A będzie ustalonym językiem. Następujące stwierdzenia są równoważne:*

- *A jest regularny,*
- *istnieje skończenie wiele języków ilorazowych języka A.*

Dowód: Implikacja w jedną stronę wynika z poprzedniego faktu. Musimy pokazać, że jeżeli istnieje skończenie wiele języków ilorazowych języka A, to A jest regularny. Skonstruujemy deterministyczny automat skończony akceptujący A. Stanami tego automatu będą języki ilorazowe języka A, $Q = \{A/x : x \in \Sigma^*\}$. Nasz automat skonstruujemy tak, że każdy stan jako język będzie zawierał dokładnie te słowa, które z tego stanu prowadzą do stanów akceptujących:

$$A/x = \{y \in \Sigma^* : \delta^*(A/x, y) \in F$$

Stanem początkowym jest oczywiście $A = A/\varepsilon$, gdyż nasz automat ma akceptować właśnie język A. Stanami akceptującymi są te stany, które jako języki zawierają słowo puste,

$$F = \{A/x : \varepsilon \in A/x\}$$

Czyli:

$$F = \{A/x : x \in A\}$$

Natomiast funkcja przejścia jest postaci:

$$\delta(A/x, a) = A/xa$$

Powyższa definicja jest poprawna, gdyż jeśli $A/x = A/y$, to $A/xa = A/ya$.

Trzeba jeszcze pokazać, że automat $M = \langle Q, \Sigma, \delta, A, F\rangle$ akceptuje język A. Niech $y \in \Sigma^*$, $y = a_1a_2 \dots a_k$. Zauważmy, że:

$$\begin{array}{rcl}
\delta^*(A/x, y) & = & \delta^*(A/x, a_1a_2 \dots a_k) = \\
& = & \delta^*(\delta(A/x, a_1), a_2 \dots a_k) = \\
& = & \delta^*(A/xa_1, a_2 \dots a_k) = \\
& & \vdots \\
& = & \delta^*(A/xa_1, a_2 \dots a_k, \varepsilon) = \\
& = & A/xa_1, a_2 \dots a_k = A/xy
\end{array}$$

Stąd, język akceptowany przez M to:

$$\begin{array}{rcl}
L(M) & = & \{x \in \Sigma^* : \delta^*(A, x) \in F\} = \\
& = & \{x \in \Sigma^* : \delta^*(A/\varepsilon, x) \in F\} = \\
& = & \{x \in \Sigma^* : A/x \in F\} = \\
& = & \{x \in \Sigma^* : \varepsilon \in A/x\} = \\
& = & \{x \in \Sigma^* : x \in A\} = A
\end{array}$$

□

8.3 Konstrukcja automatu minimalnego

Zauważmy, że automat skonstruowany w powyższym dowodzie ma minimalną liczbę stanów — gdyż ma ich dokładnie tyle ile jest języków ilorazowych języka A. Pokażemy jak z dowolnego deterministycznego automatu skończonego akceptującego język A zrobić automat izomorficzny ze skonstruowanym powyżej.

Niech $M = \langle Q, \Sigma, \delta, s, F\rangle$ będzie dowolnym ustalonym deterministycznym automatem skończonym akceptującym język A, $L(M) = A$, w którym wszystkie stany są osiągalne. (Jeśli M zawiera stany nieosiągalne, to najpierw je usuwamy.) Każdemu stanowi $q \in Q$ możemy przypisać język złożony z tych słów, które prowadzą z q do stanów akceptujących. Język taki to A/x, gdzie x prowadzi z s do q, $\delta^*(s, x) = q$.

Niech $M_{/\approx}$ będzie automatem powstałym z M przez sklejenie ze sobą stanów, którym przypisaliśmy takie same języki ilorazowe. Ściśle rzecz biorąc, na stanach automatu M definiujemy relację równoważności $\approx \subseteq Q \times Q$ określoną w następujący sposób:

$$p \approx q \text{ w.t.w. } \forall_{x\in\Sigma^*}\delta^*(p, x) \in F \Leftrightarrow \delta^*(q, x) \in F$$

Inaczej mówiąc, niech A/x i A/y będą językami ilorazowymi przypisanymi stanom p i q, wówczas $p \approx q$ wtw., gdy $A/x = A/y$.

Sklejamy te stany automatu M, które są sobie równoważne:

$$M_{/\approx} = \langle Q_{/\approx}, \Sigma, \delta_{/\approx}, [s]_\approx, F_{/\approx}\rangle$$

gdzie:

$$\delta_{/\approx}([q]_\approx, a) = [\delta(q, a)]_\approx$$

Automat $M_{/\approx}$ jest nazywany automatem *ilorazowym*.

Po pierwsze musimy się upewnić, czy $\delta_{/\approx}$ jest poprawnie zdefiniowana. Musimy w tym celu pokazać, że jeżeli sklejamy ze sobą dwa stany $p, q \in Q$, $p \approx q$, to dla dowolnego $a \in \Sigma$ mamy również $\delta(p, a) \approx \delta(q, a)$. Skoro $p \approx q$, to $A/x = A/y$, a zatem $A/xa = A/ya$, czyli $\delta(p, a) \approx \delta(q, a)$.

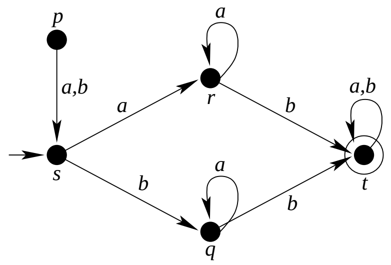
Zauważmy, że nigdy nie skleimy stanu akceptującego z nieakceptującym, gdyż stany akceptujące charakteryzują się tym, że przypisane im języki zawierają słowa puste.

Czy automat $M_{/\approx}$ akceptuje ten sam język co M, $L(M_{/\approx}) = L(M)$? Można pokazać (przez indukcję), że dla dowolnego stanu $q \in Q$ i słowa $x \in \Sigma^*$ mamy $\delta^*_{/\approx}([q]_\approx, x) = [\delta^*(q, x)]_\approx$. Ponieważ jednak nie skleiliśmy żadnego stanu akceptującego z żadnym ze stanów nieakceptujących, mamy:

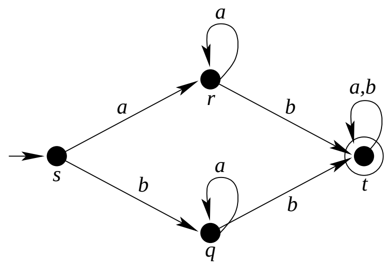
$$\begin{aligned} L(M_{/\approx}) &= \{x \in \Sigma^* : \delta^*_{/\approx}([s]_\approx, x) \in F_{/\approx}\} = \\ &= \{x \in \Sigma^* : [\delta^*(s, x)]_\approx \in F_{/\approx}\} = \\ &= \{x \in \Sigma^* : \delta^*(s, x) \in F\} = L(M) = A \end{aligned}$$

Tak więc usuwając stany nieosiągalne i sklejając ze sobą stany, którym odpowiadają te same języki ilorazowe otrzymujemy ten sam minimalny deterministyczny automat skończony akceptujący dany język regularny.

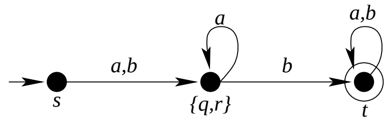
Przykład: Weźmy następujący automat deterministyczny:



Stan p jest nieosiągalny. Po jego usunięciu mamy:



Tylko dwa stany możemy skleić ze sobą: r i q, $r \approx q$. Po ich sklejeniu uzyskujemy;



8.4 Algorytm minimalizacji

Pokażemy jak można zaimplementować opisaną w poprzednim punkcie konstrukcję. Algorytm minimalizacji składa się z dwóch faz:

1. usunięcia stanów nieosiągalnych,
2. wyznaczenia relacji równoważności $\approx$ i sklejenia ze sobą stanów równoważnych.

Znalezienie stanów nieosiągalnych jest prostsze. Trudniej natomiast znaleźć stany równoważne. Przypomnijmy, że dwa stany p i q są sobie równoważne, gdy:

$$\forall_{x\in\Sigma^*}\delta^*(p,x) \in F \Leftrightarrow \delta^*(q,x) \in F$$

Pokażemy algorytm, który znajduje wszystkie pary nierównoważnych sobie stanów. Siłą rzeczy, pozostałe pary stanów są sobie równoważne i można je skleić ze sobą.

Jeśli stany p i q nie są sobie równoważne, to istnieje rozróżniające je słowo x, takie że:

$$\delta^*(p,x) \in F \not\Leftrightarrow \delta^*(q,x) \in F$$

Początkowo zakładamy, że wszystkie stany można skleić ze sobą. Następnie sukcesywnie wyznaczamy pary stanów które nie są sobie równoważne — w kolejności wg. rosnącej długości najkrótszych rozróżniających je słów.

Algorytm

1. Tworzymy tablicę wartości logicznych $T_{\{p,q\}}$ indeksowaną nieuporządkowanymi parami $\{p, q\}$ stanów $p, q \in Q$. Początkowo $T_{\{p,q\}} = \texttt{true}$ dla wszystkich $p, q \in Q$.

2. Dla wszystkich takich par stanów $\{p, q\}$, że $p \in F$ i $q \notin F$ zaznaczamy $T_{\{p,q\}} = \texttt{false}$. Jeśli p jest akceptujący, a q nie, to stany te rozróżnia słowo puste ε.

3. Dla wszystkich par stanów $\{p, q\}$ i znaków $a \in \Sigma$ takich, że $T_{\{\delta(p,a),\delta(q,a)\}} = \texttt{false}$, zaznaczamy również $T_{\{p,q\}} = \texttt{false}$.

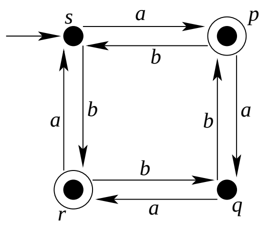
4. Jeżeli w kroku 3 zmieniliśmy choć jedną komórkę tablicy T z `true` na `false`, to powtarzamy krok 3 — tak długo, aż nie będzie on powodował żadnych zmian w tablicy T.

5. Na koniec mamy: $p \approx q \Leftrightarrow T_{\{p,q\}}$.

Dowód poprawności algorytmu: Poprawność algorytmu wynika stąd, że dla każdej pary stanów $\{p, q\}$, które nie są sobie równoważne istnieje pewne słowo, które je rozróżnia. Weźmy najkrótsze takie słowo. W kroku 2 wyznaczamy wszystkie takie pary, które są rozróżniane przez ε. Z każdym powtórzeniem kroku 3 wyznaczamy wszystkie takie pary, które są rozróżniane przez słowa o jeden dłuższe.

Ponieważ tablica T ma skończoną liczbę komórek, więc krok 3 jest powtarzany ograniczoną liczbę razy, a powyższy algorytm zawsze się zatrzyma.

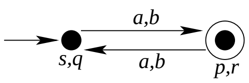
Przykład: Weźmy następujący automat deterministyczny:



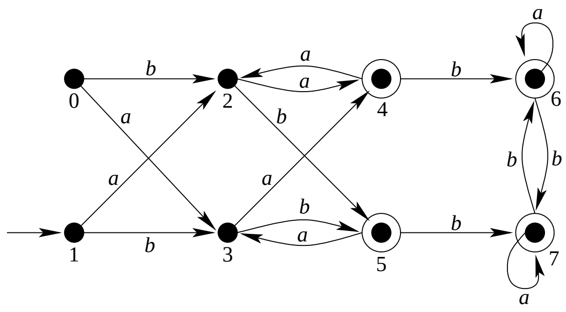
Wszystkie stany są osiągalne. Tablica T obliczona dla tego automatu uzyskuje swą postać już w kroku 2:

p			
F	q		
T	F	r	
F	T	F	s

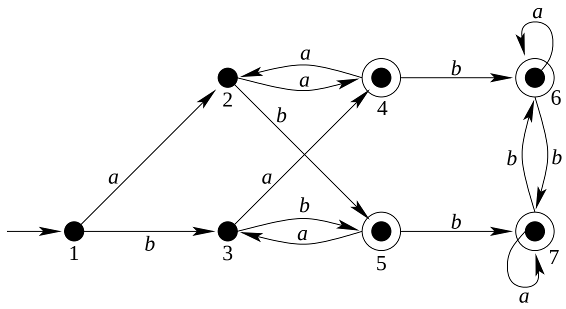
Co oznacza, że możemy skleić ze sobą stany s i q oraz p i r:



Przykład: Weźmy następujący automat deterministyczny:



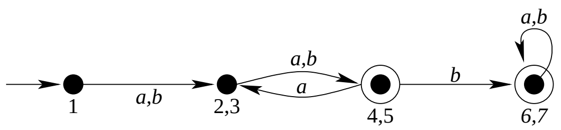
Stan 0 jest nieosiągalny. Po jego usunięciu mamy:



Tablica T obliczona dla tego automatu uzyskuje swą ostateczną postać po jednokrotnej iteracji kroku 3:

1						
F	2					
F	T	3				
F	F	F	4			
F	F	F	T	5		
F	F	F	F	F	6	
F	F	F	F	F	T	7

Co oznacza, że możemy skleić ze sobą stany: 2 z 3, 4 z 5, oraz 6 z 7.



8.5 Podsumowanie

W tym wykładzie poznaliśmy (uproszczone) twierdzenie Myhilla-Nerode'a. Twierdzenie to dostarcza nam jeszcze jednego kryterium do badania czy języki są regularne — każdy język regularny ma skończenie wiele języków ilorazowych. Jako wniosek z dowodu tego twierdzenia uzyskaliśmy, że istnieje jeden (z dokładnością do izomorfizmu) minimalny automat deterministyczny akceptujący dany język. Poznaliśmy też algorytm wyznaczania tego minimalnego automatu.

8.6 Skorowidz

- **Automat ilorazowy** to minimalny deterministyczny automat skończony powstały przez sklejenie stanów równoważnych (zgodnie z daną relacją równoważności na stanach, spełniającą pewne dodatkowe warunki).
- **Algorytm minimalizacji** deterministycznych automatów skończonych — algorytm wyznaczania minimalnego deterministycznego automatu skończonego na podstawie dowolnego deterministycznego automatu skończonego akceptującego dany język.
- **Język ilorazowy** języka A, to każdy z języków postaci $A/x = \{y \in \Sigma^* : xy \in A\}$ (dla $x \in \Sigma^*$).

8.7 Praca domowa

Zminimalizuj następujące deterministyczne automaty skończone:

1. (4 p.)

	a	b
$\to 1$	4	3
2	1	4
$F3$	1	1
$F4$	1	1

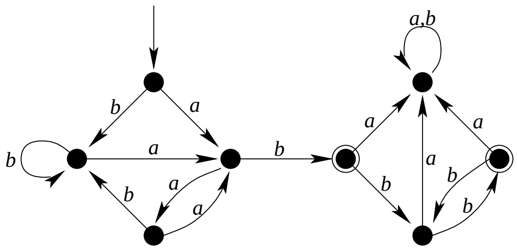
2. (6 p.)

	a	b
$\to F1$	2	4
2	5	1
3	6	5
4	1	6
5	1	2
6	4	1

8.8 Ćwiczenia

Zminimalizuj następujące deterministyczne automaty skończone:

1.



Rozwiązanie[82]

2.

	a	b
$\to 1$	3	4
2	4	3
$F3$	2	1
$F4$	1	2

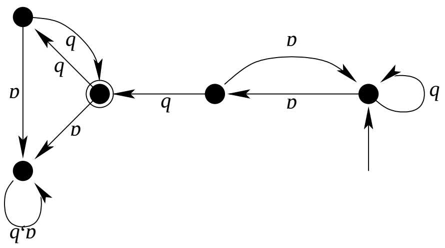
3.

	a	b
$\to 1$	3	4
2	5	6
3	2	3
4	3	1
$F5$	1	5
6	5	2

4.

	a	b
$\to 1$	4	2
$F2$	3	2
3	4	2
$F4$	4	3

82



Rozwiązanie[83]

5.

	a	b
$\to F1$	2	1
2	3	2
$F3$	4	3
4	1	4

Rozwiązanie[84]

6.

	a	b
$\to 1$	3	4
2	4	3
3	4	1
$F4$	1	3

Rozwiązanie[85]

---

[83] Możemy skleić stany 1 i 3.

	a	b
$\to 1,3$	4	2
$F2$	$1,3$	2
$F4$	4	$1,3$

[84] Możemy skleić stany 1 i 3, oraz 2 i 4.

	a	b
$\to F1,3$	$2,4$	$1,3$
$2,4$	$1,3$	$2,4$

[85] Stan 2 nie jest osiągalny i można go pominąć.

	a	b
$\to 1$	3	4
3	4	1
$F4$	1	3

7.

	a	b
$\to 1$	6	3
2	2	6
$F3$	5	4
4	5	1
5	1	4
6	5	1

Rozwiązanie[86]

8.

	a	b
1	2	4
$F2$	5	1
$\to 3$	2	3
4	5	1
$F5$	2	4
6	5	6

Rozwiązanie[87]

9.

	a	b
$\to 1$	6	6
$F2$	3	2
3	6	1
4	1	6
5	6	1
$F6$	5	3

---

86 Stan 6 jest nieosiągalny. Można skleić stany 1,3 i 4, oraz 2 i 5.

	a	b
$\to 1,3,4$	$2,5$	$1,3,4$
$F2,5$	$2,5$	$1,3,4$

87 Stan 2 jest nieosiągalny. Stany 4 i 6 można skleić.

	a	b
$\to 1$	$4,6$	3
$F3$	5	$4,6$
$4,6$	5	1
5	1	$4,6$

Rozwiązanie[88]

10.

	a	b
0	4	4
1	0	6
2	3	6
3	0	6
4	0	0
$\rightarrow 5$	3	1
$F6$	5	4

Rozwiązanie[89]

11.

	a	b
0	2	4
1	4	5
2	4	3
$\rightarrow 3$	1	5
4	6	3
$F5$	2	5
6	6	3

[88] Stan 2 jest nieosiągalny. Stany 0 i 4, oraz 1 i 3 można skleić.

	a	b
$0,4$	$0,4$	$0,4$
$1,3$	$0,4$	6
$\rightarrow 5$	$1,3$	$1,3$
$F6$	5	$0,4$

[89] Stany 2 i 4 są nieosiągalne. Można skleić stany 3 i 5.

	a	b
$\rightarrow 1$	6	6
$3,5$	6	1
$F6$	$3,5$	$3,5$

12.

	a	b
$\rightarrow 0$	2	5
1	2	4
$F2$	5	1
3	2	5
4	5	0
5	1	1
6	5	2
7	5	3

13.

	a	b
$\rightarrow 0$	2	4
$F1$	2	3
2	3	0
3	0	0
4	3	1
$F5$	2	3
6	3	2
7	3	5

14.

	a	b
$\rightarrow F1$	2	5
$2F$	1	4
3	7	2
4	5	7
5	4	3
6	3	6
7	3	1

Wykład 9. Języki bezkontekstowe

9.1 Wstęp

W tym wykładzie poznamy *gramatyki bezkontekstowe* i opisywaną przez nie klasę *języków bezkontekstowych*. Gramatyki bezkontekstowe są formalizmem służącym do opisu składni — podobnie jak wyrażenia regularne. Podobnie również jak wyrażenia regularne są maksymalnie uproszczonym formalizmem, a wzorce są ich rozbudowaną wersją przeznaczoną do praktycznych zastosować, tak i gramatyki bezkontekstowe mają swoją rozbudowaną wersję przeznaczoną do praktycznych zastosowań. Są to: notacja Backusa-Naura (BNF, ang. *Backus-Naur form*) oraz rozszerzona notacja Backusa-Naura (EBNF, ang. *extended Backus-Naur form*). Jest bardzo prawdopodobne, że Czytelnik zetknął się z nimi. Notacje te są zwykle używane w podręcznikach do ścisłego opisu składni języków programowania.

Będziemy również badać właściwości klasy języków, które można opisać gramatykami bezkontekstowymi — tzw. klasy języków bezkontekstowych. Jak zobaczymy, siła wyrazu gramatyk bezkontekstowych będzie większa niż wyrażeń regularnych i automatów skończonych. Inaczej mówiąc, klasa języków bezkontekstowych zawiera klasę języków regularnych.

9.2 BNF

W pierwszym przybliżeniu BNF jest formalizmem przypominającym definicje nazwanych wzorców ze specyfikacji dla Lex'a. Główna różnica polega na tym, że zależności między poszczególnymi nazwami mogą być rekurencyjne.

Specyfikacja w BNF-ie określa składnię szeregu *konstrukcji składniowych*, nazywanych też *nieterminalami*. Każda taka konstrukcja składniowa ma swoją nazwę. Specyfikacja składa się z szeregu reguł (nazywanych *produkcjami*), które określają jaką postać mogą przybierać konstrukcje składniowe. Nazwy konstrukcji składniowych ujmujemy w trójkątne nawiasy $\langle \dots \rangle$. Produkcje mają postać:

$$\langle \texttt{definiowana konstrukcja} \rangle \quad ::= \quad \text{wyrażenie opisujące definiowaną konstrukcję}$$

Wyrażenie opisujące definiowaną konstrukcję może zawierać:

- $\langle \texttt{konstrukcja} \rangle$ — nazwy konstrukcji, w tym również definiowanej, rekurencyjne zależności są dozwolone,
- tekst, ujęty w cudzysłów, który pojawia się dosłownie w danej konstrukcji,
- $\dots | \dots$ — oddziela różne alternatywne postaci, jakie może mieć dana konstrukcja,
- $[\dots]$ — fragment ujęty w kwadratowe nawiasy jest opcjonalny.

Produkcje traktujemy jak możliwe reguły przepisywania — nazwę konstrukcji stojącej po lewej stronie produkcji możemy zastąpić dowolnym napisem pasującym do wyrażenia podanego po prawej stronie. Każdej konstrukcji składniowej odpowiada pewien język. Składa sie on z wszystkich tych napisów, które można uzyskać zaczynając od nazwy konstrukcji i stosując produkcje, aż do momentu uzyskania napisu, który nie zawiera już żadnych nazw konstrukcji.

Przykład: BNF może służyć do opisywania składni najrozmaitszych rzeczy. Oto opis składni adresów pocztowych:

```
⟨adres⟩          ::=  ⟨adresat⟩ ⟨adres lokalu⟩ ⟨adres miasta⟩ ⟨adres kraju⟩
⟨adresat⟩        ::=  ["W.P." | "Sz.Pan."] ⟨napis⟩ ","
⟨adres lokalu⟩   ::=  ⟨ulica⟩ ⟨numer⟩ ["/" ⟨numer⟩ ] ["m" ⟨numer⟩ | "/" ⟨numer⟩ ]","
⟨adres miasta⟩   ::=  [ ⟨kod⟩ ] ⟨napis⟩
⟨adres kraju⟩    ::=  ["," ⟨napis⟩ ]
⟨kod⟩            ::=  ⟨cyfra⟩ ⟨cyfra⟩ "–" ⟨cyfra⟩ ⟨cyfra⟩ ⟨cyfra⟩
⟨cyfra⟩          ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
⟨numer⟩          ::=  ⟨cyfra⟩ [ ⟨numer⟩ ]
```

Specyfikacja ta nie jest oczywiście kompletna, gdyż nie definiuje czym jest `napis`. Zwróćmy uwagę na to, że definicja `numer` jest rekurencyjna.

9.3 Gramatyki bezkontekstowe

Gramatyki bezkontekstowe są uproszczoną wersją notacji BNF. Konstrukcje składniowe nazywamy tu *nieterminalami* lub *symbolami nieterminalnymi* i będziemy je oznaczać wielkimi literami alfabetu łacińskiego: A, B, $\ldots$, X, Y, Z. Wśród nieterminali jest jeden wyróżniony symbol, nazywany *aksjomatem*. To właśnie język związany z tym nieterminalem opisuje gramatyka.

Oprócz nieterminali używamy również *terminali* (*symboli terminalnych*) — są to znaki stanowiące alfabet opisywanego przez gramatykę języka. Będziemy je oznaczać małymi literami alfabetu łacińskiego: a, b, $\ldots$.

Produkcje w gramatykach bezkontekstowych mają bardzo prostą postać: $A \to \alpha$, gdzie A to nieterminal, a α to słowo, które może zawierać terminale i nieterminale. Produkcja taka oznacza, że nieterminal A **możemy** zastąpić napisem α. Dla jednego nieterminala możemy mieć wiele produkcji. Oznacza to, że możemy je dowolnie stosować. Zwykle zamiast pisać:

$$\begin{array}{ccc} A & \to & \alpha \\ A & \to & \beta \\ & \vdots & \\ A & \to & \gamma \end{array}$$

Będziemy pisać krócej:

$$A \to \alpha \mid \beta \mid \ldots \mid \gamma$$

Przykład: Zanim formalnie zdefiniujemy gramatyki bezkontekstowe i opisywane przez nie języki, spójrzmy na gramatykę opisującą język: $\{a^n b^n : n \geq 0\}$.

$$A \to aAb \mid \varepsilon$$

Oto sekwencja zastosować produkcji, która prowadzi do uzyskania słowa *aaabbb*:

$$A \to aAb \to aaAbb \to aaaAbbb \to aaabbb$$

Trzykrotnie stosowaliśmy tutaj produkcję $A \to aAb$, aby na koniec zastosować $A \to \varepsilon$.
Przykład ten pokazuje, że gramatyki bezkontekstowe mogą opisywać języki, które nie są regularne.

Def. 34. *Gramatyka bezkontekstowa*, to dowolna taka czwórka $G = \langle N, \Sigma, P, S \rangle$, gdzie:

- N to (skończony) alfabet symboli *nieterminalnych*,
- Σ to (skończony) alfabet symboli *terminalnych*, $\Sigma \cap N = \emptyset$,
- P to skończony zbiór *produkcji*, reguł postaci $A \to \alpha$ dla $A \in N$, $\alpha \in (N \cup \Sigma)^*$,
- $S \in N$ to *aksjomat* wyróżniony symbol nieterminalny.

$\square$

Zwykle nie będziemy podawać gramatyki jako formalnej czwórki. Zamiast tego będziemy podawać zbiór produkcji. Zwykle, w sposób niejawny z produkcji wynika zbiór nieterminali i terminali. Jeżeli nie będzie oczywiste, który nieterminal jest aksjomatem, będziemy to dodatkowo zaznaczać.

Def. 35. Niech $G = \langle N, \Sigma, P, S \rangle$ będzie ustaloną gramatyką bezkontekstową. Przez $\to$ będziemy oznaczać zbiór produkcji P traktowany jako relacja, $\to \subseteq N \times (N \cup \Sigma)^*$. Przez $\to^*$ będziemy oznaczać zwrotno-przechodnie domknięcie $\to$, $\to^* \subseteq (N \cup \Sigma)^* \times (N \cup \Sigma)^*$. $\square$

Relacja $\to$ opisuje pojedyncze zastosowanie produkcji, jako relacja między nieterminalem, a zastępującym go słowem. Relacje $\to^*$ opisuje co można zrobić stosując dowolną liczbę (łącznie z zero) dowolnych produkcji. Możemy też przedstawić sobie relację $\to^*$ jako skrót do wielokrotnego iterowania relacji $\to$:

Fakt 11. *Jeżeli $x \to^* y$, to istnieje ciąg słów (z_i) taki, że:*

$$x = z_0 \to z_1 \to \cdots \to z_k = y$$

Gramatyka opisuje język złożony z tych wszystkich słów (nad alfabetem terminalnym), które możemy uzyskać z aksjomatu stosując produkcje.

Def. 36. Niech $G = \langle N, \Sigma, P, S\rangle$ będzie ustaloną gramatyką bezkontekstową. Język *generowany* (lub *opisywany*) przez gramatykę G, to:

$$L(G) = \{x \in \Sigma^* : S \to^* x\}$$

Alternatywnie, $L(G)$ to zbiór takich słów $x \in \Sigma^*$, dla których istnieją ciągi słów (z_i), $z_i \in (N \cup \Sigma)^*$ takie, że:

$$S = z_0 \to z_1 \to \cdots \to z_k = x$$

Ciąg (z_i) nazywamy *wyprowadzeniem* słowa x (w gramatyce G). □

Def. 37. Powiemy, że język jest bezkontekstowy, jeżeli istnieje generująca go gramatyka. □

Wyprowadzenie stanowi coś w rodzaju dowodu, że dane słowo faktycznie należy do języka generowanego przez gramatykę. Istnieje jeszcze inny, bardziej strukturalny, sposób ilustrowania jak dane słowo można uzyskać w danej gramatyce. Jest to *drzewo wyprowadzenia.*

Def. 38. *Drzewo wyprowadzenia*, to sposób ilustrowania jak dane słowo może być wyprowadzone w danej gramatyce, w postaci drzewa. Drzewo wyprowadzenia musi spełniać następujące warunki:

- W korzeniu drzewa znajduje się aksjomat.
- W węzłach wewnętrznych drzewa znajdują się nieterminale, a w liściach terminale lub słowa puste ε.
- Jeśli w danym węźle mamy nieterminal X, a w jego kolejnych synach mamy $x_1, x_2, \ldots, x_k$, to musi zachodzić $X \to x_1 x_2 \ldots x_k$.
- Terminale umieszczone w liściach, czytane od lewej do prawej, tworzą wyprowadzone słowo.

□

Przyjrzyjmy się zdefiniowanym powyżej pojęciom na przykładach:

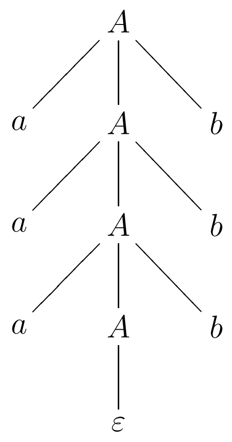
Przykład: Weźmy gramatykę generującą język $\{a^n b^n : n \geq 0\}$:

$$A \to aAb \mid \varepsilon$$

Wyprowadzenie słów *aaabbb* ma postać:

$$A \to aAb \to aaAbb \to aaaAbbb \to aaabbb$$

a jego drzewo wyprowadzenia wygląda następująco:

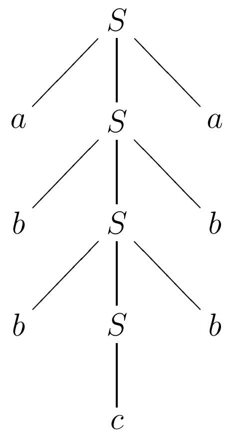


Przykład: Oto gramatyka generująca język palindromów nad alfabetem $\{a, b, c\}$:

$$S \to aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \varepsilon$$

Weźmy słowo *abbcbba*. Oto jego wyprowadzenie i drzewo wyprowadzenia:

$$S \to aSa \to abSba \to abbSbba \to abbcbba$$



Przykład: Przyjrzyjmy się językowi złożonemu z poprawnych wyrażeń nawiasowych (dla czytelności zbudowanych z nawiasów kwadratowych). Język ten możemy zdefiniować na dwa sposoby. Po pierwsze, wyrażenia nawiasowe, to takie ciągi nawiasów, w których:

- łączna liczba nawiasów otwierających i zamykających jest taka sama,
- każdy prefiks wyrażenia nawiasowego zawiera przynajmniej tyle nawiasów otwierających, co zamykających.

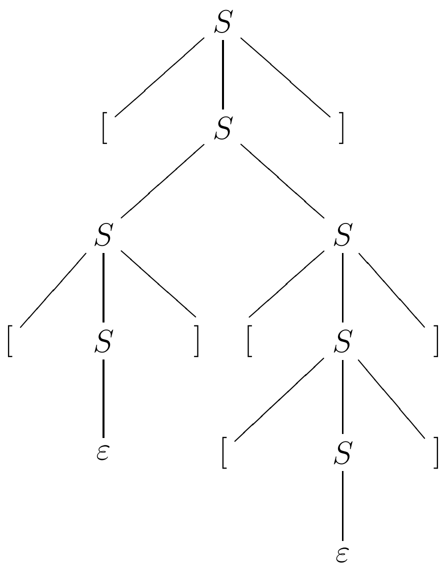
Jeżeli zdefiniujemy funkcje $L(x) = \#_{[}(x)$ i $R(x) = \#_{]}(x)$, to język wyrażeń nawiasowych możemy zdefiniować następująco:

$$W = \{x \in \{[,]\}^* : L(x) = R(x) \text{ i dla każdego prefiksu } y \text{ słowa } x \text{ mamy} L(y) \geq R(y)\}$$

Drugi sposób zdefiniowania języka wyrażeń nawiasowych, to podanie gramatyki bezkontekstowej.

$$S \to [S] \mid SS \mid \varepsilon$$

Oznaczmy tę gramatykę przez G. Oto przykładowe drzewo wyprowadzenia dla wyrażenia nawiasowego [[][[]]]:



Pokażemy, że obie definicje są sobie równoważne. W tym celu pokażemy najpierw, że każde słowo, które można wyprowadzić w G należy do W, a następnie, że każde słowo z W można wyprowadzić w G.

To, że $L(G) \subseteq W$ będziemy dowodzić przez indukcję ze względu na długość wyprowadzenia.

1. Jedynym słowem jakie można wyprowadzić w jednym kroku jest ε. Oczywiście $\varepsilon \in W$.

2. Załóżmy, że wyprowadzenie ma postać $S \to [S] \to^* [x]$. Z założenia indukcyjnego x jest wyrażeniem nawiasowym. W takim razie $[x]$ też, gdyż:

   $$L([x]) = L(x) + 1 = R(x) + 1 = R([x])$$

   oraz dla każdego prefiksu y słowa x mamy:

   $$L([y) = L(y) + 1 > L(y) \geq R(y) = R([y)$$

3. Załóżmy, że wyprowadzenie ma postać:

$$\begin{tikzcd} S \arrow[r] & SS \arrow[dl, "*"'] \arrow[dr, "*"] & \\ x & & y \end{tikzcd}$$

   Z założenia indukcyjnego x i y są wyrażeniami nawiasowymi. W takim razie xy też, gdyż:

$$L(xy) = L(x) + L(y) = R(x) + R(y) = R(xy)$$

   oraz dla każdego prefiksu z słowa y mamy:

$$L(xz) = L(x) + L(z) \geq L(x) + R(z) = R(x) + R(z) = R(xz)$$

Z zasady indukcji otrzymujemy $L(G) \subseteq W$.

Pokażemy teraz, że $W \subseteq L(G)$. Dowód będzie przebiegał przez indukcję po długości słowa.

1. Mamy $\varepsilon \in W$ oraz $S \to \varepsilon$.

2. Niech $x \in W$, $|x| > 0$, $x = a_1 a_2 \ldots a_k$. Określmy funkcję $f : \{0, 1, \ldots, k\} \to \mathbb{N}$:

$$f(i) = L(a_1 \ldots a_i) - R(a_1 \ldots a_i)$$

   Zauważmy, że z tego, że x jest wyrażeniem nawiasowym wynika, że:

$$f(0) = f(k) = 0$$

$$f(i) \geq 0$$

   Mamy dwa możliwe przypadki:

   - Dla pewnego $0 < i < k$ mamy $f(i) = 0$. Wówczas $a_1 \ldots a_i$ oraz $a_{i+1} \ldots a_k$ są wyrażeniami nawiasowymi. Stąd:

$$S \to SS \to^* a_1 \ldots a_i \; a_{i+1} \ldots a_k = x$$

   - Dla wszystkich $0 < i < k$ mamy $f(i) > 0$. Wówczas $a_2 \ldots a_{k-1}$ jest wyrażeniem nawiasowym. Stąd:

$$S \to [S] \to^* [a_2 \ldots a_{k-1}] = x$$

   Tak więc $x \in L(G)$.

Na mocy zasady indukcji uzyskujemy $W \subseteq L(G)$. Tak więc $W = L(G)$.

Często chcemy, aby gramatyka nie tylko opisywała język, ale również jego strukturę. Wymagamy wówczas, aby gramatyka była dodatkowo *jednoznaczna*.

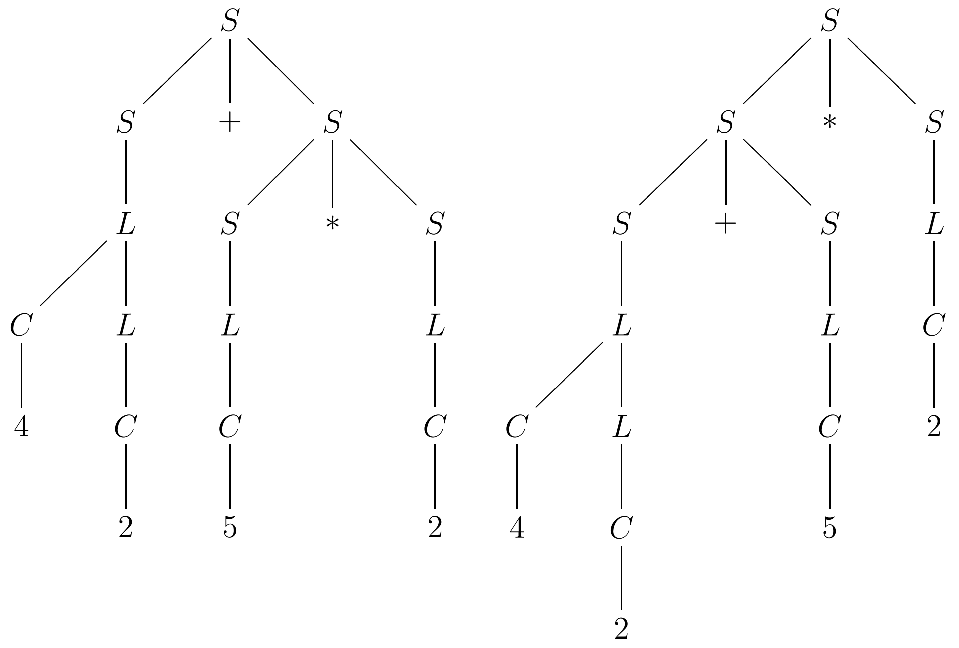
Def. 39. Powiemy, że gramatyka G jest *jednoznaczna*, gdy dla każdego słowa $x \in L(G)$ istnieje tylko jedno drzewo wyprowadzenia słowa x w gramatyce G. □

Jeśli gramatyka jest jednoznaczna, to drzewa wyprowadzeń jednoznacznie określają strukturę (składniową) słów z języka. Nadal oczywiście słowa mogą mieć wiele wyprowadzeń, ale różnią się one tylko kolejnością stosowanych produkcji, a nie strukturą wyprowadzania słowa.

Przykład: Oto gramatyka opisująca proste wyrażenia arytmetyczne:

$$\begin{array}{rcl} S & \to & S+S \mid S-S \mid S*S \mid S/S \mid L \mid (S) \\ L & \to & C \mid CL \\ C & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Gramatyka ta jest niestety niejednoznaczna. Na przykład, wyrażenie $42+5*2$ ma dwa różne drzewa wyprowadzenia:

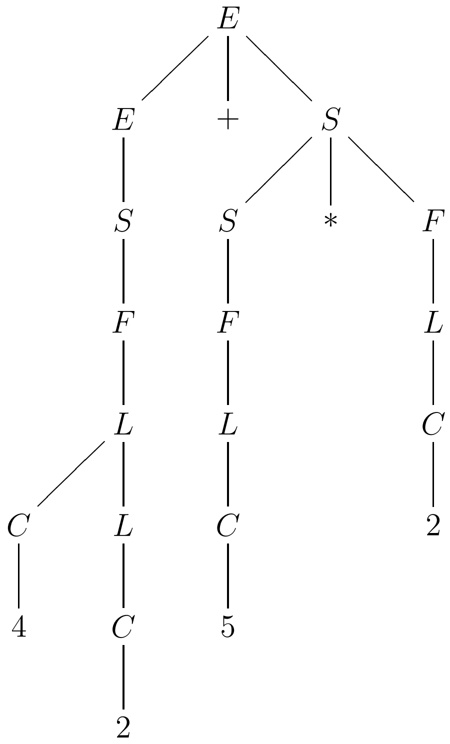


Struktura wyrażenia ma wpływ na sposób jego interpretacji. Drzewo wyprowadzenia po lewej prowadzi do wartości wyrażenia 52, a to po prawej do 94. Wszędzie tam, gdzie gramatyka opisuje składnię języka, któremu towarzyszy ściśle zdefiniowana semantyka, będziemy chcieli, aby była to jednoznaczna gramatyka.

W przypadku wyrażeń arytmetycznych musimy ustalić kolejność wiązania operatorów, oraz wymusić, aby operatory były albo lewostronnie, albo prawostronnie łączne. Oto jednoznaczna gramatyka wyrażeń arytmetycznych:

$$\begin{array}{lcl} E & \to & E+S \mid E-S \mid S \\ S & \to & S*F \mid S/F \mid F \\ F & \to & L \mid (E) \\ L & \to & C \mid CL \\ C & \to & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

W tej gramatyce analizowane wyrażenie ma tylko jedno drzewo wyprowadzenia:



9.4 Języki regularne a bezkontekstowe

Pokażemy teraz, że wszystkie regularne są bezkontekstowe. Jak już wcześniej widzieliśmy, odwrotne stwierdzenie nie jest prawdziwe, gdyż $a^n b^n$ jest językiem bezkontekstowym, ale nie regularny.

Def. 40. Gramatyka (prawostronnie) liniowa, to taka, w której wszystkie produkcje mają postać: $A \to \alpha B$ lub $A \to \alpha$, dla $A, B \in N$ i $\alpha \in \Sigma^*$.

Gramatyka silnie (prawostronnie) liniowa, to taka, w której wszystkie produkcje mają postać: $A \to aB$ lub $A \to \varepsilon$, dla $A, B \in N$ i $a \in \Sigma$. □

Okazuje się, że gramatyki liniowe i silnie liniowe opisują dokładnie klasę języków regularnych.

Fakt 12. *Gramatyki (silnie) liniowe opisują dokładnie klasę języków regularnych.*

Dowód: Pokażemy, jak gramatykę liniową przerobić na równoważną jej gramatykę silnie liniową. Następnie pokażemy jak można gramatykę silnie liniową przekształcić na równoważny jej niedeterministyczny automat skończony i vice versa.

Żeby gramatykę liniową przerobić na silnie liniową, musimy zastąpić produkcje postaci $A \to a_1 \dots a_k B$ (dla $k > 1$) oraz produkcje postaci $A \to B$, równoważnymi jej produkcjami odpowiednimi dla gramatyki silnie liniowej.

Jeżeli mamy w gramatyce produkcję postaci $A \to B$, to dla każdej produkcji postaci $B \to \alpha$ dodajemy do gramatyki produkcję $A \to \alpha$. Krok ten powtarzamy tak długo, jak długo wprowadza on nowe produkcje. Następnie usuwamy wszystkie produkcje postaci $A \to B$. W ten sposób uzyskujemy gramatykę generującą ten sam język, ale nie zawięrającą produkcji postaci $A \to B$.

Dla każdej produkcji postaci:

$$A \to a_1 \dots a_k B$$

(dla $k > 1$) dodajemy do gramatyki nowe terminale $A_1, \dots, A_{k-1}$, a samą produkcję zastępujemy produkcjami:

$$\begin{array}{rcl} A & \to & a_1 A_1 \\ A_1 & \to & a_2 A_2 \\ & \vdots & \\ A_{k-1} & \to & a_k B \end{array}$$

Podobnie, dla każdej produkcji postaci (dla $k > 1$):

$$A \to a_1 \dots a_k$$

dodajemy do gramatyki nowe terminale $A_1, \dots, A_k$, a samą produkcję zastępujemy produkcjami:

$$\begin{array}{rcl} A & \to & a_1 A_1 \\ A_1 & \to & a_2 A_2 \\ & \vdots & \\ A_{k-1} & \to & a_k A_k \\ A_k & \to & \varepsilon \end{array}$$

1. Gramatykę liniową można sprowadzić do silnie liniowej.
2. Gramatykę silnie liniową można przerobić na automat niedeterministyczny.
3. Automat niedeterministyczny można przerobić na gramatykę silnie liniową.

W rezultacie uzyskujemy gramatykę akceptującą ten sam język, ale silnie liniową.

Zauważmy, że jeśli gramatyka jest silnie liniowa (lub liniowa), to w wyprowadzeniu pojawia się na raz tylko jeden nieterminal, i to zawsze na samym końcu słowa. Konstruując niedeterministyczny automat skończony równoważny danej gramatyce silnie liniowej opieramy się na następującej analogii: Miech $G = \langle N, \Sigma, P, S\rangle$ będzie daną gramatyką silnie liniową. Stanami automatu będą nieterminale gramatyki N. Dla każdego słowa $x \in \Sigma^*$ i $A \in N$, takich, że $S \to^* xA$ nasz automat po wczytaniu słowa x będzie mógł przejść do stanu A. Formalnie konstrukcja naszego automatu M wygląda następująco $M = (N, \Sigma, \delta, \{S\}, F)$, gdzie:

$$F = \{A : A \to \varepsilon \in P\}$$

oraz:

$$\delta(A, a) = \{B \in N : A \to aB \in P\}$$

Można pokazać (przez indukcję po długości słowa), że:

$$\delta^*(A, x) = \{B \in N : A \to^* xB\}$$

Stąd:

$$\begin{aligned}
L(M) &= \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\} = \\
&= \{x \in \Sigma^* : \exists_{A \in N} A \in \delta^*(S, x) \wedge A \in F\} = \\
&= \{x \in \Sigma^* : \exists_{A \in N} S \to^* xA \wedge A \to \varepsilon \in P\} = \\
&= \{x \in \Sigma^* : S \to^* x\} = L(G)
\end{aligned}$$

Tak więc faktycznie języki generowane przez gramatyki silnie liniowe są regularne.

Co więcej, powyższą konstrukcję można odwrócić. Jeśli $M = (Q, \Sigma, \delta, \{s\}, F)$ jest danym niedeterministycznym automatem skończonym (z jednym stanem początkowym), to równoważna mu gramatyka silnie liniowa ma postać $G = \langle Q, \Sigma, P, s\rangle$, gdzie P zawiera produkcje postaci:

$$\begin{array}{lcll}
q & \to & \varepsilon & \text{dla} \quad q \in F \\
q & \to & ap & \text{dla} \quad p, q \in Q,\ a \in \Sigma,\ p \in \delta(q, a)
\end{array}$$

Ponownie można pokazać (przez indukcję po długości słowa), że:

$$\delta^*(q, x) = \{p \in Q : q \to^* xp\}$$

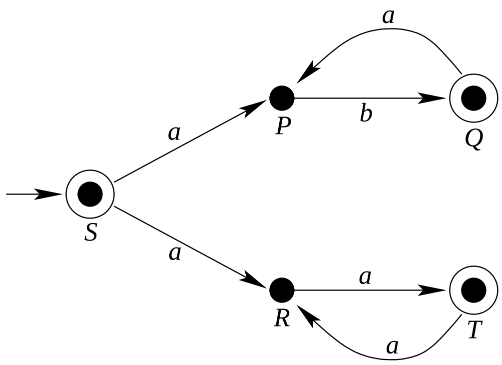
Stąd:

$$\begin{aligned}
L(G) &= \{x \in \Sigma^* : s \to^* x\} = \\
&= \{x \in \Sigma^* : \exists_{q \in Q} s \to^* xq \wedge q \to \varepsilon \in P\} = \\
&= \{x \in \Sigma^* : \exists_{q \in Q} q \in \delta^*(s, x) \wedge q \in F\} = \\
&= \{x \in \Sigma^* : \delta^*(S, x) \cap F \neq \emptyset\} = L(M)
\end{aligned}$$

□

Tak więc dla wszystkich języków regularnych istnieją generujące je gramatyki silnie liniowe. Tym samym pokazaliśmy, że klasa języków regularnych zawiera się ściśle w klasie języków bezkontekstowych.

Przykład: Rozważmy automat skończony akceptujący język $(ab)^* \mid (aa)^*$:



Jest on równoważny gramatyce silnie liniowej postaci:

$$\begin{array}{rcl} S & \rightarrow & aP \mid aR \mid \varepsilon \\ P & \rightarrow & bQ \\ Q & \rightarrow & aP \mid \varepsilon \\ R & \rightarrow & aT \\ T & \rightarrow & aR \mid \varepsilon \end{array}$$

Oto wyprowadzenie przykładowego słowa *abab*:

$$S \rightarrow aP \rightarrow abQ \rightarrow abaP \rightarrow ababQ \rightarrow abab$$

Jak widać, odpowiada ono dokładnie obliczeniu automatu akceptującemu słowo *abab*.

Jeśli wystarczy nam, aby gramatyka była tylko liniowa, to wystarczy mniejsza liczba nieterminali:

$$\begin{array}{rcl} S & \rightarrow & abQ \mid aaT \mid \varepsilon \\ Q & \rightarrow & abQ \mid \varepsilon \\ T & \rightarrow & aaT \mid \varepsilon \end{array}$$

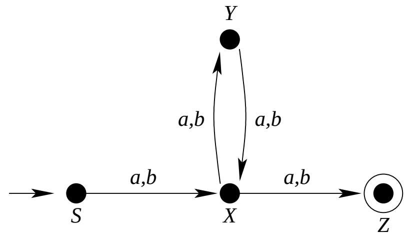
A oto wyprowadzenie przykładowego słowa *abab*:

$$S \rightarrow abQ \rightarrow ababQ \rightarrow abab$$

Przykład: Niech A będzie językiem złożonym ze słów nad alfabetem $\{a, b\}$ parzystej długości i różnych od ε:

$$A = \{x \in \{a, b\} : |x| > 0 \wedge |x| \bmod 2 = 0\}$$

Język ten jest akceptowany przez następujący automat:



Na podstawie tego automatu można stworzyć następującą gramatykę liniową (ale nie silnie liniową) generującą język A:

$$
\begin{array}{rcl}
S & \rightarrow & aX \mid bX \\
X & \rightarrow & aaX \mid abX \mid baX \mid bbX \mid a \mid b
\end{array}
$$

Na przykład, słowo *abbaaa* ma w niej następujące wyprowadzenie:

$$S \rightarrow aX \rightarrow abbX \rightarrow abbaaX \rightarrow abbaaa$$

Automat ten jest również równoważny następującej gramatyce silnie liniowej:

$$
\begin{array}{rcl}
S & \rightarrow & aX \mid bX \\
X & \rightarrow & aY \mid bY \mid aZ \mid bZ \\
Y & \rightarrow & aX \mid bX \\
Z & \rightarrow & \varepsilon
\end{array}
$$

Słowo *abbaaa* ma w tej gramatyce następujące wyprowadzenie:

$$S \rightarrow aX \rightarrow abY \rightarrow abbX \rightarrow abbaY \rightarrow abbaaX \rightarrow abbaaaZ \rightarrow abbaaa$$

Jak widać, wyprowadzenie w gramatyce silnie liniowej dokładnie odpowiada akceptującemu obliczeniu automatu skończonego. Wyprowadzenie w gramatyce liniowej (ale nie silnie liniowej) również odpowiada akceptującemu obliczeniu automatu, ale jeden krok w wyprowadzeniu może odpowiadać kilku krokom automatu.

9.5 Podsumowanie

W tym wykładzie poznaliśmy języki i gramatyki bezkontekstowe wraz z podstawowymi związanymi z nimi pojęciami. Pokazaliśmy też, że klasa języków bezkontekstowych zawiera ściśle w sobie klasę języków regularnych.

9.6 Skorowidz

- **BNF** — notacja Backusa-Naura, formalizm służący do opisu składni, oparty na gramatykach bezkontekstowych. Zobacz też artykuł w Wikipedii.

- **Drzewo wyprowadzenia** — drzewo potwierdzające, że dane słowo należy do języka generowanego przez daną gramatykę; odzwierciedla również strukturę składniową słowa.
- **Gramatyka bezkontekstowa** — formalizm do opisu języków.
- **Gramatyka liniowa** — gramatyka bezkontekstowa, w której występują produkcje jedynie postaci $A \to \alpha B$ lub $A \to \alpha$ (dla $A, B \in N$ i $\alpha \in \Sigma^*$). Gramatyki liniowe opisują języki regularne.
- **Gramatyka silnie liniowa** — gramatyka bezkontekstowa, w której występują produkcje jedynie postaci $A \to aB$ lub $A \to \varepsilon$ (dla $A, B \in N$ i $a \in \Sigma$). Gramatyki silnie liniowe opisują języki regularne.
- **Język generowany przez gramatykę bezkontekstową** — to język złożony z tych słów $x \in \Sigma^*$, które można wyprowadzić z aksjomatu S, $S \to^* x$.
- **Wyprowadzenie** — ciąg słów potwierdzający, że dane słowo należy do języka generowanego przez daną gramatykę.

9.7 Praca domowa

1. (6 p.) Podaj gramatykę bezkontekstową generującą język $\{a^n b^{n+k} c^k : n, k \geq 1\}$. Podaj drzewo wyprowadzenia dla słowa *aabbbbbccc*.
2. (4 p.) Podaj liniową gramatykę generująca język $ab(b^* \mid a^*)$.

9.8 Ćwiczenia

1. Dla gramatyki:

$$\begin{array}{rcl} S & \to & aB|Ab|SS \\ A & \to & a|aS \\ B & \to & b|Sb \end{array}$$

   i słowa *aabbab*:

   - podaj jego wyprowadzenie,
   - podaj jego drzewo wyprowadzenia,
   - czy jest to gramatyka jednoznaczna (podaj kontrprzykład, lub krótko uzasadnij)?

Rozwiązanie[90]

2. Podaj gramatykę bezkontekstową generującą język: $\{a^i b^{2i+1} : i \geq 0\}$. Narysuj drzewo wyprowadzenia dla słowa *aabbbbb*. Rozwiązanie[91]

3. Podaj gramatykę bezkontekstową generującą język: $\{a^i b^{i+2 \cdot j} c^j : i, j \geq 0\}$. Narysuj drzewo wyprowadzenia dla słowa *aabbbbbbbbccc*. Rozwiązanie[92]

4. Dla podanej poniżej gramatyki, podaj wyprowadzenie i drzewo wyprowadzenia słowa *baabab*.

$$\begin{array}{lcl} S & \rightarrow & AB \mid BA \\ A & \rightarrow & a \mid BBA \\ B & \rightarrow & b \mid AAB \end{array}$$

Czy podana gramatyka jest jednoznaczna (odpowiedź uzasadnij lub podaj kontrprzykład)?

5. Dla podanej poniżej gramatyki, podaj wyprowadzenie i drzewo wyprowadzenia słowa

---

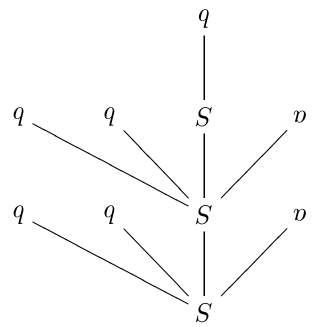
90

....

$$S \rightarrow SS \rightarrow aBS \rightarrow aBAb \rightarrow aSbAb \rightarrow aAbbAb \rightarrow aabbAb \rightarrow aabbab$$

91

$$S \rightarrow aSbb \mid b$$



92

....

$$\begin{array}{lcl} S & \rightarrow & XY \\ X & \rightarrow & aXb \mid \varepsilon \\ Y & \rightarrow & bbYa \mid \varepsilon \end{array}$$

abbabbaa.

$$\begin{aligned} S &\rightarrow A \mid B \mid AB \\ A &\rightarrow BA \mid aaA \mid abaA \mid aa \\ B &\rightarrow abbB \mid \varepsilon \end{aligned}$$

Czy podana gramatyka jest jednoznaczna (odpowiedź uzasadnij lub podaj kontrprzykład)?

6. Dla podanej poniżej gramatyki, podaj wyprowadzenie i drzewo wyprowadzenia słowa *abaabb*.

$$\begin{aligned} S &\rightarrow aSB \mid ASb \mid ab \\ A &\rightarrow Sa \mid bAA \mid a \\ B &\rightarrow bS \mid BBa \mid b \end{aligned}$$

Czy podana gramatyka jest jednoznaczna (odpowiedź uzasadnij lub podaj kontrprzykład)?

7. Podaj gramatykę bezkontekstową generującą język:

- $\{a^n b^{2n} : n \geq 1\}$, Rozwiązanie[93]
- $\{a^k b^n : k \geq 2n\}$, Rozwiązanie[94]
- $\{a^k b^n : 3k \geq n\}$, Rozwiązanie[95]
- $\{a^n b^k : 2n \geq 3k\}$.
- $\{a^i b^j c^i\}$,
- $\{a^i b^j a^j b^i\}$,
- $\{a^i b^i a^j b^j\}$,
- $\{a^i b^j c^k : i + 2 \cdot j = k\}$,
- $\{a^i b^j c^k : 3i = 2j + k\}$,
- $\{a^n b^m : n \geq 2m \geq 0\}$,

93

$$S \rightarrow aSbb \mid abb$$

94

$$S \rightarrow aaSb \mid aS \mid \varepsilon$$

95

$$S \rightarrow aSbbb \mid aSbb \mid aSb \mid aS \mid \varepsilon$$

- $\{a^n b^m c^k d^l :\ n+m = k+l\}$,
- $\{a^n b^m c^k d^l :\ n+l = k+m\}$,
- $\{a^n b^m c^k d^l :\ n+k = m+l\}$,
- $\{xc\,\mathrm{rev}(x) : x \in \{a,b\}^*\}$,
- $\{xcy : x, y \in \{a,b\}^* \wedge \#_a(x) = \#_b(y)\}$,
- $\{a^n b^m c^k :\ n+k = 2 \cdot m\}$,
- język złożony ze wszystkich słów nad alfabetem $\{a,b\}^*$, które nie są palindromami.

8. Rozszerz podaną w treści wykładu gramatykę wyrażeń arytmetycznych o unarny minus. Zapewnij aby była jednoznaczna. Jak silnie powinien wiązać unarny minus? Jak byś wstawił nawiasy do wyrażenia $3+3*-4*8$? Zdecyduj, czy $2+--4$ powinno być poprawnym wyrażeniem.

9. Rozszerz gramatykę będącą rozwiązaniem poprzedniego zadania o operację potęgowania ($**$). Zapewnij aby była jednoznaczna. Potęgowanie powinno wiązać mocniej niż mnożenie, ale słabiej niż unarny minus. Zdecyduj, jak byś wstawił nawiasy do wyrażenia $2**3**4$ i tak też zrealizuj gramatykę.

10. Podać gramatykę generującą język złożony z takich wyrażeń arytmetycznych składających się z symboli $+, *, 0, 1, (,)$, których wartość jest większa niż 2.

11. Dla zadanego automatu/wyrażenia regularnego podaj gramatyki (prawostronnie) liniowe / silnie liniowe opisujące odpowiedni język:

- $a(a \mid b)^* aa$,
- $(a(ab \mid ba)^*) \mid (b(bb \mid aa)^*)$,
-

	a	b
$\rightarrow F\ A$	A	B
B	B	A

-

	a	b
$\rightarrow A$	A	B
$F\ B$	A	C
$F\ C$	C	A

-

	a	b
$\rightarrow A$	B	$-$
$F\ B$	B	A

-

	a	b
$\to F\ A$	A	B
B	B	C
C	C	A

12. Podaj gramatykę wyrażeń regularnych (nad alfabetem $\{a, b\}$):

    - dowolną,
    - jednoznaczną;
    - narysuj drzewo wyprowadzenia (w jednoznacznej gramatyce) dla wyrażenia $(a \mid b^*)a^* \mid ab$.

Wykład 10. Postać normalna Chomsky'ego, algorytm Cocke-Younger'a-Kasami'ego

10.1 Wstęp

W tym wykładzie zajmiemy się problemem jak efektywnie rozpoznawać, czy dane słowo należy do języka generowanego przez daną gramatykę. W tym celu poznamy najpierw szczególną postać gramatyk bezkontekstowych, tzw. postać normalną Chomsky'ego i dowiemy się jak przekształcać gramatyki do tej postaci. Następnie poznamy dynamiczny algorytm rozpoznający czy dane słowo można wyprowadzić w danej gramatyce w postaci normalnej Chomsky'ego.

10.2 Postać normalna Chomsky'ego

Def. 41. Powiemy, że gramatyka jest w *postaci normalnej Chomsky'ego*, jeśli wszystkie produkcje w gramatyce są postaci: $A \to BC$ lub $A \to a$ (dla pewnych nieterminali A, B i C oraz terminala a). □

Zauważmy, że jeśli gramatyka jest w postaci normalnej Chomsky'ego, to każda produkcja przekształca jeden nieterminal na dwa nieterminale, lub jeden nieterminal na jeden terminal. Ułatwia to rozstrzygnięcie, czy dane słowo x da się wyprowadzić w danej gramatyce — wiadomo, że w takim wyprowadzeniu $|x| - 1$ razy należy zastosować produkcje przekształcające nieterminal na dwa nieterminale i $|x|$ razy zastosować produkcje przekształcające nieterminal na terminal.

Podstawowe pytanie, jakie można sobie zadać brzmi: czy dla każdego języka bezkontekstowego istnieje generująca go gramatyka w postaci normalnej Chomsky'ego? Odpowiedź brzmi: nie. Zauważmy, że zastosowanie dowolnej produkcji nie zmniejsza liczby symboli. Ponieważ zaczynamy od słowa złożonego z jednego symbolu (aksjomatu), więc nigdy nie uzyskamy słowa pustego. Jeśli więc język A jest bezkontekstowy i $\varepsilon \in A$, to A nie jest generowany przez żadną gramatykę w postaci normalnej Chomsky'ego.

Na szczęście słowo puste jest jedynym elementem, którego nie możemy uzyskać rozważając gramatyki w postaci normalnej Chomsky'ego. Jeżeli dopuścimy, aby słowo puste zniknęło z języka generowanego przez gramatykę, to każdą gramatykę można przekształcić do postaci normalnej Chomsky'ego.

Fakt 13. *Dla dowolnej gramatyki bezkontekstowej G istnieje taka gramatyka bezkontekstowa w postaci normalnej Chomsky'ego G', że $L(G') = L(G) \setminus \{\varepsilon\}$.*

Dowód: Niech $G = \langle N, \Sigma, P, S\rangle$. Konstrukcja gramatyki G' przebiega w kilku krokach:

1. Niech gramatyka $G_1 = \langle N, \Sigma, P_1, S\rangle$ będzie gramatyką powstałą z G przez domknięcie zbioru produkcji zgodnie z następującymi dwiema regułami:

- Jeśli mamy produkcje postaci $X \to \alpha Y \beta$ i $Y \to \varepsilon$ (dla $X, YinN$ i $\alpha, \beta \in (N \cup \Sigma)^*$), to dodajemy produkcję $X \to \alpha\beta$. Tak dodana produkcja realizuje w jednym kroku to, co można było zrobić w dwóch krokach: $X \to \alpha Y \beta \to \alpha\beta$.
- Jeśli mamy produkcje postaci $X \to Y$ i $Y \to \gamma$ (dla $X, YinN$ i $\gamma \in (N \cup \Sigma)^*$), to dodajemy produkcję $X \to \gamma$. Tak dodana produkcja realizuje w jednym kroku to, co można było zrobić w dwóch krokach: $X \to Y \to \gamma$.

Stosowanie powyższych reguł powtarzamy tak długo, jak długo wprowadzają one jakiekolwiek nowe produkcje. Ponieważ dodawane produkcje nie są dłuższe od już istniejących, więc cały proces doda tylko ograniczoną liczbę produkcji i w pewnym momencie się zakończy.

Zauważmy, że $L(G_1) = L(G)$, gdyż dodawane produkcje nie pozwalają wyprowadzić niczego, czego i tak nie dałoby się wcześniej wyprowadzić.

2. Dla dowolnego słowa z $x \in L(G_1)$, $x \neq \varepsilon$ rozważymy jego najkrótsze wyprowadzenie. W takim wyprowadzeniu nie będą się pojawiały produkcje postaci $X \to \varepsilon$ ani $X \to Y$. Niech G_2 będzie gramatyką powstałą z G_1 przez usunięcie tych produkcji, $G_2 = \langle N, \Sigma, P_2, S \rangle$, gdzie

$$P_2 = P_1 \setminus \{X \to \varepsilon, X \to Y : X, Y \in N\}$$

Wówczas:

$$L(G_2) = L(G_1) \setminus \{\varepsilon\} = L(G) \setminus \{\varepsilon\}$$

3. Gramatyka G_3 powstaje z G_2 przez dodanie do niej, dla każdego terminala $a, b, c, \dots \in N$, odpowiadającego mu nowego nieterminala $(A, B, C, \dots)$, oraz produkcji: $A \to a$, $B \to b$, $C \to c$, ... We wszystkich pozostałych produkcjach $X \to \alpha$, gdzie $|\alpha| > 1$, zamieniamy wszystkie terminale na odpowiadające im nieterminale. Nie zmienia to języka generowanego przez gramatykę:

$$L(G_3) = L(G_2) = L(G) \setminus \{\varepsilon\}$$

Natomiast w ten sposób w G_3 mamy tylko dwa rodzaje produkcji: $X \to a$ gdzie a jest terminalem, oraz $X \to Y_1 Y_2 \dots Y_k$ gdzie Y_i to nieterminale i $k > 1$.

4. Gramatyka G_4 powstaje z G_3 przez zastąpienie każdej produkcji postaci $X \to Y_1 Y_2 \dots Y_k$ (dla $k > 2$) przez zestaw produkcji:

$$\begin{array}{rcl} X & \to & Y_1 X_1 \\ X_1 & \to & Y_2 X_2 \\ & \vdots & \\ X_{k-2} & \to & Y_{k-1} Y_k \end{array}$$

gdzie $X_1, X_2, \ldots, X_{k-2}$ to nowe nieterminale dodane do G_4. Tak uzyskana gramatyka generuje dokładnie ten sam język — jedyne co się zmieniło to to, że zamiast zastosowania jednej produkcji, trzeba ich zastosować kilka,

$$L(G_4) = L(G_3) = L(G) \setminus \{\varepsilon\}$$

Natomiast G_4 jest już w postaci normalnej Chomsky'ego. $\square$

Przykład: Zobaczmy, jak powyższy algorytm działa na przykładzie gramatyki generującej język $\{a^n b^n : n \geq 0\}$.

$$S \to aSb \mid \varepsilon$$

Pierwszy krok powoduje dodanie produkcji $S \to ab$:

$$S \to aSb \mid ab \mid \varepsilon$$

W drugim kroku usuwamy produkcję $S \to \varepsilon$.

$$S \to aSb \mid ab$$

W trzecim kroku dodajemy nieterminale A i B, zastępujemy nimi w istniejących produkcjach terminale a i b, oraz dodajemy produkcje $A \to a$ i $B \to b$.

$$\begin{array}{lcl} S & \to & ASB \mid AB \\ A & \to & a \\ B & \to & b \end{array}$$

W ostatnim kroku algorytmu rozbijamy produkcję $S \to ASB$ na dwie produkcje.

$$\begin{array}{lcl} S & \to & AX \mid AB \\ X & \to & SB \\ A & \to & a \\ B & \to & b \end{array}$$

Tak uzyskana gramatyka jest w postaci normalnej Chomsky'ego. Na przykład, wyprowadzenie słowa $aaabbb$ ma postać:

$$\begin{array}{lcl} S & \to & AX \to ASB \to aSB \to aSb \to \\ & \to & aAXb \to aASBb \to aaSBb \to aaSbb \to \\ & \to & baABbb \to aaaBbb \to aaabbb \end{array}$$

Przykład: Zobaczmy, jak powyższy algorytm działa na przykładzie gramatyki generującej wyrażenia nawiasowe.

$$S \to [S] \mid SS \mid \varepsilon$$

Pierwszy krok powoduje dodanie produkcji: $S \to []$ i $S \to S$:

$$S \to [S] \mid [] \mid SS \mid S \mid \varepsilon$$

W drugim kroku usuwamy produkcje $S \to \varepsilon$ i $S \to S$.

$$S \to [S] \mid [] \mid SS$$

W trzecim kroku dodajemy nieterminale L i P, zastępujemy nimi w istniejących produkcjach odpowiednio terminale [i], oraz dodajemy produkcje $L \to [$ i $P \to]$.

$$\begin{array}{rcl} S & \to & LSP \mid LP \mid SS \\ L & \to & [\\ P & \to &] \end{array}$$

W ostatnim kroku algorytmu rozbijamy produkcję $S \to LSP$ na dwie produkcje.

$$\begin{array}{rcl} S & \to & LX \mid LP \mid SS \\ X & \to & SP \\ L & \to & [\\ P & \to &] \end{array}$$

Tak uzyskana gramatyka jest w postaci normalnej Chomsky'ego. Na przykład, wyprowadzenie słowa [[][[]]] w tej gramatyce ma postać:

$$\begin{array}{rcl} S & \to & LX \to LSP \to [SP \to [S] \to \\ & \to & [SS] \to [LPS] \to [[PS] \to [[]S] \to \\ & \to & [[]LX] \to [[]LSP] \to [[][SP] \to [[][S]] \to \\ & \to & [[][LP]] \to [[][[P]] \to [[][[]]] \end{array}$$

10.3 Algorytm rozpoznawania Cocke-Younger'a-Kasami (CYK)

Algorytm CYK pozwala stwierdzić, czy dane słowo jest wyprowadzalne w danej gramatyce (w postaci normalnej Chomsky'ego). Jeśli gramatyka nie jest w postaci normalnej Chomsky'ego, to można ją sprowadzić do tej postaci zgodnie z opisaną wyżej konstrukcją.

Tak jak to zwykle bywa w programowaniu dynamicznym, żeby rozwiązać dany problem musimy go najpierw uogólnić i zamiast szukać odpowiedzi na jedno pytanie, znajdziemy odpowiedzi na całe mnóstwo powiązanych ze sobą pytań.

Niech $G = \langle N, \Sigma, P, S\rangle$ będzie daną gramatyką w postaci normalnej Chomsky'ego, a dane słowo $x \in \Sigma^*$ będzie postaci $x = a_1a_2 \ldots a_n$. Dla każdej pary indeksów $1 \leq i \leq j \leq n$ wyznaczymy wszystkie takie nieterminale X, z których można wyprowadzić podsłowo $a_i \ldots a_j$. Dokładniej, wypełnimy tablicę:

$$T[i, j] = \{X \in N : X \to^* a_i \ldots a_j\}$$

Gdy już będziemy mieli obliczoną tablicę T, to słowo x można wyprowadzić w gramatyce G wtw., gdy $S \in T[1, n]$.

W jaki sposób możemy wypełniać tę tablicę?

- Zaczynamy od przekątnej. $T[i, i]$ to zbiór nieterminali, z których można wyprowadzić a_i. Jednak takie wyprowadzenie może składać się tylko z zastosowania jednej produkcji zastępującej nieterminal terminalem:

  $$T[i, i] = \{X : X \to a_i \in P\}$$

- Następnie wypełniamy kolejne przy-przekątne. Zauważmy, że jeżeli $X \to^* a_i \ldots a_j$, $i < j$, to wyprowadzenie to musi mieć postać: $X \to YZ$, $Y \to^* a_i \ldots a_k$, $Z \to^* a_{k+1} \ldots a_j$, dla pewnego k, $i \leq k < j$.

  Pomocna nam będzie następująca operacja na zbiorach nieterminali, która na podstawie zbioru wszystkich możliwych Y-ów i Z-ów wyznacza zbiór wszystkich możliwych X-ów:

  $$\mathcal{A} \otimes \mathcal{B} = \{X : X \to YZ \in P, Y \in \mathcal{A}, Z \in \mathcal{B}\}$$

  Operacja $\otimes$ może być skomplikowana, ale zauważmy, że jej koszt jest stały (tzn. zależy od gramatyki, ale nie od długości słowa x).

  Obliczając $T[i, j]$ uwzględniamy wszystkie możliwe punkty podziału k podsłowa $a_i \ldots a_j$:

  $$T[i, j] = \bigcup_{i \leq k < j} T[i, k] \otimes T[k + 1, j]$$

Tablica ma rozmiar $\Theta(n^2)$ przy czym aby wypełnić jeden element tablicy trzeba wykonać liniową liczbę operacji $\otimes$, tak więc złożoność czasowa tego algorytmu to $\Theta(n^3)$.

Przykład: Rozważmy gramatykę w postaci normalnej Chomsky'ego:

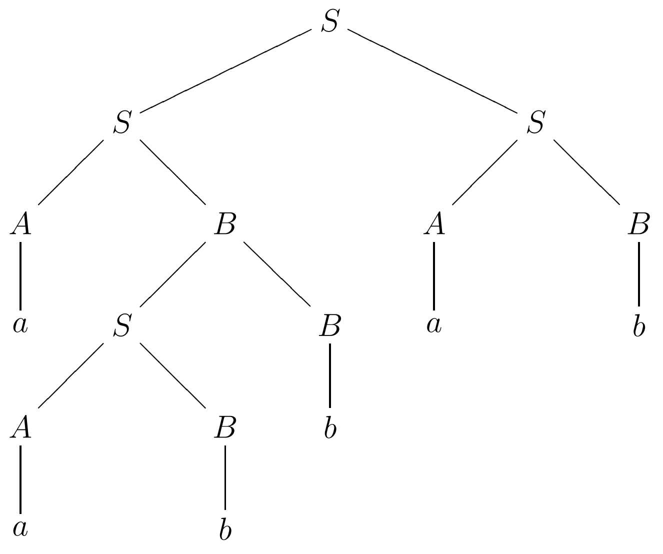
$$\begin{array}{rcl} S & \to & SS \mid AB \\ A & \to & AS \mid AA \mid a \\ B & \to & SB \mid BB \mid b \end{array}$$

oraz słowo *aabbab*. Algorytm CYK da nam w wyniku następującą tablicę T:

	1	2	3	4	5	6
1	A	A	A, S	A, B, S	A	A, S
2		A	S	B, S	$-$	S
3			B	B	$-$	$-$
4				B	$-$	$-$
5					A	S
6						B

Jeżeli spróbujemy odtworzyć skąd w $T[1, 6]$ wzięło się S, to uzyskamy informacje o drzewach wyprowadzenia badanego słowa. Poniżej podano przykładowe drzewo wyprowadzenia i zaznaczono odpowiadające mu nieterminale w tablicy T.

	1	2	3	4	5	6
1	$\mathbf{A}$	A	A, S	$A, B, \mathbf{S}$	A	$A, \mathbf{S}$
2		$\mathbf{A}$	$\mathbf{S}$	$\mathbf{B}, S$	$-$	S
3			$\mathbf{B}$	B	$-$	$-$
4				$\mathbf{B}$	$-$	$-$
5					$\mathbf{A}$	$\mathbf{S}$
6						$\mathbf{B}$



10.4 Podsumowanie

W tym wykładzie poznaliśmy postać normalną Chomsky'ego gramatyk bezkontekstowych i pokazaliśmy, że każdą gramatykę bezkontekstową można sprowadzić do tej postaci (z dokładnością do akceptowania słowa pustego ε). Poznaliśmy też algorytm CKY rozpoznający, czy dane słowo należy do języka generowanego przez daną gramatykę bezkontekstową w postaci normalnej Chomsky'ego.

10.5 Skorowidz

- **Postać normalna Chomsky'ego** — taka postać gramatyk bezkontekstowych, w której gramatyka może zawierać tylko produkcje postaci: $A \to BC$ lub $A \to a$ (dla pewnych nieterminali A, B i C oraz terminala a). Zobacz też artykuł w Wikipedii.
- **Algorytm CYK** — algorytm rozpoznawania czy dane słowo jest wyprowadzalne w danej gramatyce w postaci normalnej Chomsky'ego. Zobacz też artykuł w Wikipedii.

10.6 Praca domowa

1. (4 p.) Sprowadź do postaci normalnej Chomsky'ego następującą gramatykę bezkontekstową:

$$\begin{array}{lcl} S & \to & SS \mid PaS \mid P \\ P & \to & bbPa \mid varepsilon \end{array}$$

2. (6 p.) Dla podanej gramatyki bezkontekstowych w postaci normalnej Chomsky'ego i słowa *abaabb* zasymuluj działanie algorytmu CYK. Podaj zawartość tablicy T oraz przykładowe drzewo wyprowadzenia.

$$\begin{array}{lcl} S & \to & AB \mid BA \\ A & \to & AA \mid BB \mid a \\ B & \to & SB \mid b \end{array}$$

10.7 Ćwiczenia

1. Sprowadź do postaci normalnej Chomsky'ego następujące gramatyki bezkontekstowe:

   -
$$\begin{array}{lcl} S & \to & aSb \mid cS \mid Sc \mid T \\ T & \to & ab \mid c \end{array}$$

Stosujemy kolejne kroki sprowadzenia do postaci normalnej Chomsky'go. W pierwszym kroku dodajemy produkcje wynikające z $S \to T$:

$$\begin{array}{lcl} S & \to & aSb \mid cS \mid Sc \mid T \mid ab \mid c \\ T & \to & ab \mid c \end{array}$$

W drugim kroku usuwamy produkcję $S \to T$:

$$\begin{array}{lcl} S & \to & aSb \mid cS \mid Sc \mid ab \mid c \\ T & \to & ab \mid c \end{array}$$

W trzecim kroku zastępujemy terminale a, b i c nieterminalami A, B i C, oraz dodajemy odpowiednie produkcje tłumaczące dodane nieterminale na terminale:

$$\begin{array}{lcl} S & \to & ASB \mid CS \mid SC \mid AB \mid c \\ T & \to & AB \mid c \\ A & \to & a \\ B & \to & b \\ C & \to & c \end{array}$$

Zwróćmy uwagę, że w produkcjach $S \to c$ i $T \to c$ nie zamieniamy c na C, gdyż te produkcje są już w postaci normalnej Chomsky'ego. W czwartym kroku rozbijamy produkcję $S \to ASB$:

$$\begin{array}{lcl} S & \to & AX \mid CS \mid SC \mid AB \mid c \\ T & \to & AB \mid c \\ X & \to & SB \\ A & \to & a \\ B & \to & b \\ C & \to & c \end{array}$$

Uzyskaną gramatykę można uprościć. Zauważmy, że nieterminal T nie jest wykorzystywany:

$$\begin{array}{lcl} S & \to & AX \mid CS \mid SC \mid AB \mid c \\ X & \to & SB \\ A & \to & a \\ B & \to & b \\ C & \to & c \end{array}$$

96

$$\begin{array}{lcl} S & \to & aSc \mid B \\ B & \to & bBc \mid \varepsilon \end{array}$$

•

Rozwiązanie[96]

Stosujemy kolejne kroki sprowadzenia do postaci normalnej Chomsky'go. W pierwszym kroku dodajemy produkcje wynikające z $S \to B$ i $B \to \varepsilon$:

$$\begin{array}{rcl} S & \to & aSc \mid B \mid bBc \mid bc \mid \varepsilon \\ B & \to & bBc \mid \varepsilon \mid bc \end{array}$$

W trakcie pierwszego kroku pojawia się produkcja $S \to \varepsilon$, co powoduje, że trzeba dodać kolejną produkcję:

$$\begin{array}{rcl} S & \to & aSc \mid B \mid bBc \mid \varepsilon \mid bc \mid ac \\ B & \to & bBc \mid \varepsilon \mid bc \end{array}$$

W drugim kroku usuwamy produkcje $S \to B$, $S \to \varepsilon$ i $B \to \varepsilon$:

$$\begin{array}{rcl} S & \to & aSc \mid bBc \mid bc \mid ac \\ B & \to & bBc \mid bc \end{array}$$

W trzecim kroku zastępujemy terminale a, b i c nieterminalami A, D i C (nieterminal B jest już używany), oraz dodajemy odpowiednie produkcje tłumaczące dodane nieterminale na terminale:

$$\begin{array}{rcl} S & \to & ASC \mid DBC \mid DC \mid AC \\ B & \to & DBC \mid DC \\ A & \to & a \\ D & \to & b \\ C & \to & c \end{array}$$

W czwartym kroku rozbijamy produkcje $S \to ASC$, $S \to DBC$ i $B \to DBC$:

$$\begin{array}{rcl} S & \to & AX \mid DY \mid DC \mid AC \\ B & \to & DY \mid DC \\ X & \to & SC \\ Y & \to & BC \\ A & \to & a \\ D & \to & b \\ C & \to & c \end{array}$$

Rozwiązanie[97]

•

$$\begin{array}{rcl} S & \to & ab \mid ba \mid T \\ T & \to & SS \mid aSb \mid bSa \end{array}$$

97

Stosujemy kolejne kroki sprowadzenia do postaci normalnej Chomsky'go. W pierwszym kroku dodajemy produkcje wynikające z $S \to T$:

$$
\begin{aligned}
S &\to ab \mid ba \mid T \mid SS \mid aSb \mid bSa \\
T &\to SS \mid aSb \mid bSa
\end{aligned}
$$

W drugim kroku usuwamy produkcję $S \to T$:

$$
\begin{aligned}
S &\to ab \mid ba \mid SS \mid aSb \mid bSa \\
T &\to SS \mid aSb \mid bSa
\end{aligned}
$$

W trzecim kroku zastępujemy terminale a i b nieterminalami A i B, oraz dodajemy odpowiednie produkcje tłumaczące dodane nieterminale na terminale:

$$
\begin{aligned}
S &\to AB \mid BA \mid SS \mid ASB \mid BSA \\
T &\to SS \mid ASB \mid BSA \\
A &\to a \\
B &\to b
\end{aligned}
$$

W czwartym kroku rozbijamy produkcje $S \to ASB \mid BSA$ i $T \to ASB \mid BSA$:

$$
\begin{aligned}
S &\to AB \mid BA \mid SS \mid AX \mid BY \\
T &\to SS \mid AX \mid BY \\
X &\to SB \\
Y &\to SA \\
A &\to a \\
B &\to b
\end{aligned}
$$

Uzyskaną gramatykę można uprościć. Zauważmy, że nieterminal T nie jest wykorzystywany.

$$
\begin{aligned}
S &\to AB \mid BA \mid SS \mid AX \mid BY \\
X &\to SB \\
Y &\to SA \\
A &\to a \\
B &\to b
\end{aligned}
$$

86

$$
\begin{aligned}
X &\to aXa \mid Y \\
Y &\to Yc \mid bX \mid a
\end{aligned}
$$

- Rozwiązanie[98]

Rozwiązanie[99]

-
$$\begin{aligned} X &\rightarrow aXYb \mid Y \\ Y &\rightarrow \varepsilon \mid cY \end{aligned}$$

Rozwiązanie[100]

-
$$\begin{aligned} S &\rightarrow Y \mid aY \mid bbY \\ Y &\rightarrow \varepsilon \mid cS \end{aligned}$$

-
$$\begin{aligned} S &\rightarrow yY \mid xxSy \mid YSY \mid xy \\ X &\rightarrow Y \mid xYX \mid \varepsilon \\ Y &\rightarrow X \mid yX \mid \varepsilon \end{aligned}$$

-
$$\begin{aligned} S &\rightarrow aSbb \mid ASB \mid B \mid \varepsilon \\ A &\rightarrow ABA \mid a \mid \varepsilon \\ B &\rightarrow aB \mid bA \mid S \mid a \end{aligned}$$

99
$$\begin{aligned} X &\rightarrow AZ \mid YC \mid BX \mid a \\ Y &\rightarrow YC \mid BX \mid a \\ Z &\rightarrow XA \\ A &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

100
$$\begin{aligned} X &\rightarrow AP \mid CY \mid AQ \mid AR \mid AB \mid c \\ Y &\rightarrow CY \mid c \\ P &\rightarrow XQ \\ Q &\rightarrow YB \\ R &\rightarrow XB \\ A &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

- $$\begin{array}{rcl} S & \rightarrow & PSP \mid qqSq \mid Q \mid qq \mid \varepsilon \\ P & \rightarrow & pQp \mid xY \mid p \mid q \\ Q & \rightarrow & QP \mid PQ \mid S \mid \varepsilon \end{array}$$

- $$\begin{array}{rcl} S & \rightarrow & mSnn \mid mNm \mid m \mid \varepsilon \\ M & \rightarrow & MN \mid NM \mid N \\ N & \rightarrow & mM \mid nM \mid M \mid \varepsilon \end{array}$$

2. Dla każdej z poniższych gramatyk bezkontekstowych w postaci normalnej Chomsky'ego zasymuluj działanie algorytmu CYK dla podanego słowa. Czy podane słowo faktycznie można wyprowadzić w podanej gramatyce? Jeśli tak, to podaj przykładowe drzewo wyprowadzenia.

   - $$\begin{array}{rcl} S & \rightarrow & SS \mid AB \\ A & \rightarrow & AA \mid a \\ B & \rightarrow & BB \mid b \end{array}$$

Rozwiązanie[101]

- *aabbbab*

$$\begin{aligned} S &\rightarrow AB \mid AA \\ A &\rightarrow SS \mid BB \mid a \\ B &\rightarrow SB \mid AS \mid b \end{aligned}$$

- *aabaaab*

$$\begin{aligned} S &\rightarrow BB \mid AA \\ A &\rightarrow AS \mid BS \mid a \\ B &\rightarrow BA \mid SB \mid b \end{aligned}$$

- *aaaaaba.*

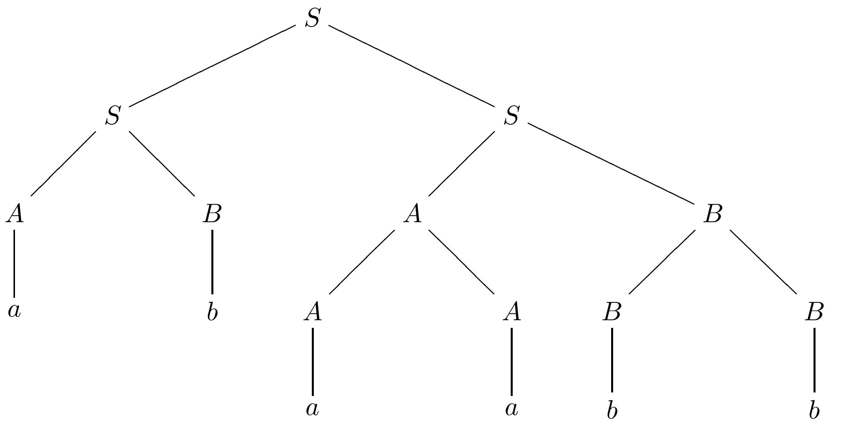
$$\begin{aligned} S &\rightarrow SS \mid AA \\ A &\rightarrow AS \mid SB \mid a \\ B &\rightarrow AB \mid BB \mid b \end{aligned}$$

101

W wyniku działania algorytmu CYK uzyskujemy następującą tablicę:

A	**S**	–	–	S	**S**
	B	–	–	–	–
		A	**A**	S	**S**
			A	S	S
				B	**B**
					B

Pogrubioną czcionką zaznaczono wartości odpowiadające węzłom drzewa wyprowadzenia. Samo drzewo wyprowadzenia wygląda tak:



- $aabbaba$

$$\begin{aligned} S &\rightarrow SS \mid BB \\ A &\rightarrow SA \mid BS \mid a \\ B &\rightarrow BA \mid AA \mid b \end{aligned}$$

- $aacacab$

$$\begin{aligned} S &\rightarrow AC \mid CB \mid AB \\ A &\rightarrow a \mid AA \mid AC \\ B &\rightarrow b \mid BB \mid CB \\ C &\rightarrow c \mid CB \mid CC \end{aligned}$$

Wykład 11. Lemat o pompowaniu dla języków bezkontekstowych

11.1 Wstęp

Okazuje się, że nie każdy język jest bezkontekstowy. Poznamy w tym wykładzie lemat o pompowaniu dla języków bezkontekstowych. Podobnie jak w przypadku lematu o pompowaniu dla języków regularnych, jest to narzędzie umożliwiające wykazanie, że dany język **nie jest** bezkontekstowy.

11.2 Lemat o pompowaniu dla języków bezkontekstowych

Niech A będzie językiem bezkontekstowym. Jeżeli A zawiera dowolnie długie słowa, to okazuje się, że drzewach wyprowadzeń tych słów muszą istnieć fragmenty, które można powielać. Ich powielanie „pompuje" słowa należące do języka.

Tw. 4 (lemat o pompowaniu dla języków bezkontekstowych)**.** *Niech A będzie językiem bezkontekstowym. Istnieje wówczas taka stała $k > 0$, że dla każdego słowa $z \in A$ o długości $|z| \geq k$ można to słowo podzielić na pięć części: $z = uvwxy$ w taki sposób, że: $vx \neq \varepsilon$, $|vwx| \leq k$ oraz dla każdego $i \geq 0$ mamy $uv^iwx^iy \in A$.*

Dowód: Jeśli język A jest bezkontekstowy, to istnieje opisująca go gramatyka bezkontekstowa. Zauważmy, że nie ma znaczenia, czy $\varepsilon \in A$ (gdyż $k > 0$). Załóżmy więc, że $\varepsilon \notin A$. Dzięki temu możemy założyć, że istnieje gramatyka bezkontekstowa w postaci normalnej Chomsky'ego $G = \langle N, \Sigma, P, S\rangle$ generująca język A, $L(G) = A$.

Zauważmy, że drzewa wyprowadzeń dla gramatyk w postaci normalnej Chomsky'ego mają postać regularnych drzew binarnych z „frędzelkami" długości 1. Produkcje przekształcające nieterminal na dwa nieterminale dają węzły wewnętrzne o dwóch synach. Natomiast produkcje przekształcające nieterminal na terminal odpowiadają ojcom liści — każdy ojciec liścia ma tylko jednego syna.

Zachodzi następujący fakt:

Fakt 14. *Jeśli drzewo wyprowadzenia ma wysokość h, to długość wyprowadzanego słowa nie przekracza 2^{h-1}.*

Dowód faktu: Niech x będzie wyprowadzanym słowem. Dowód przebiega przez indukcję po wysokości drzewa wyprowadzenia h:

1. Jeśli $h = 1$, to $|x| = 1 = 2^0 = 2^{h-1}$.

2. Jeśli $h > 1$, to można podzielić x na dwa takie słowa y i z, $x = yz$, że wyprowadzenie x wygląda następująco:

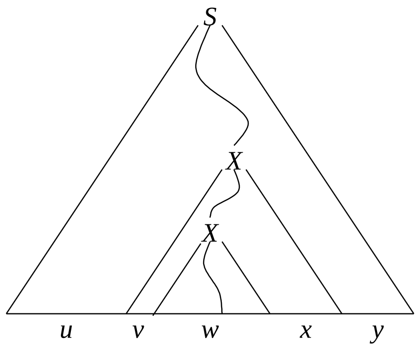
$$\begin{tikzcd} & S \arrow[r] & YZ \arrow[dl, "*"'] \arrow[dr, "*"] & \\ & y & & z \end{tikzcd}$$

dla pewnych $Y, Z \in N$. Poddrzewa wyprowadzenia y z Y i z z Z mają wysokość co najwyżej $h-1$. Tak więc, z założenia indukcyjnego mamy:

$$|x| = |xy| = |x| + |y| \leq 2^{h-2} + 2^{h-2} = 2^{h-1}$$

Na mocy zasady indukcji uzyskujemy lemat. □

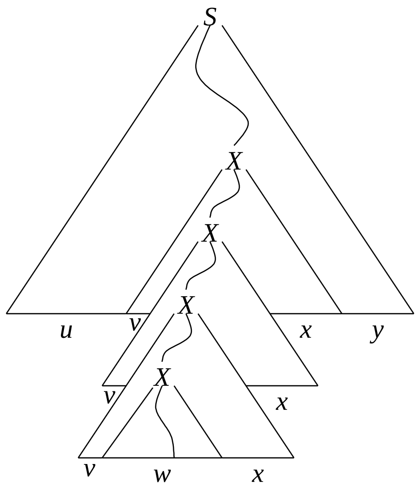
Niech $|N| = n$. Pokażemy, że można wziąć $k = 2^n$. Z powyższego faktu wynika, że jeżeli pewne słowo $z \in A$ ma długość przynajmniej $k = 2^n$, to jego drzewo wyprowadzenia ma wysokość przynajmniej $n+1$. W takim drzewie wyprowadzenia istnieje ścieżka, na której jest przynajmniej $n+1$ nieterminali. Wybierzmy i ustalmy jedną taką ścieżkę. Na takiej ścieżce pewien nieterminal musi się powtarzać. Niech X będzie pierwszym nieterminalem, który powtarza się na tej ścieżce idąc do góry (od liścia do korzenia). Jego powtórzenia wyznaczają podział słowa z na słowa u, v, w, x i y, $uvwxy = z$, w sposób przedstawiony na rysunku.



Drzewo wyprowadzenia jest regularne, a oba zaznaczone na rysunku wystąpienia nieterminala X to różne węzły drzewa. Wynika stąd, że v i x nie mogą być oba pustymi słowami. Czyli $vx \neq \varepsilon$.

Ponieważ X jest pierwszym terminalem, który powtarza się na wybranej ścieżce idąc od liścia do korzenia, więc poddrzewo wyprowadzenia vwx z X jest wysokości co najwyżej $n+1$. Stąd $|vwx| \leq 2^n = k$.

Nieterminal X występuje (przynajmniej) w dwóch miejscach na ścieżce. Zauważmy, że możemy powielać fragment drzewa wyprowadzenia między pierwszym i drugim wystąpieniem nieterminala X na wybranej ścieżce, z którego pochodzą słowa v i x.



Jak widać, powielając ów fragment drzewa i razy uzyskujemy drzewo wyprowadzenia słowa uv^iwx^iy. Czyli $uv^iwx^iy \in A$. $\square$

11.3 Zastosowanie lematu o pompowaniu

Lemat o pompowaniu służy pokazywaniu, że określone języki **nie są** bezkontekstowe. Schemat takiego dowodu przebiega nie wprost. Lemat o pompowaniu ma postać implikacji: jeśli język jest bezkontekstowy, to ma pewne własności. Dowód, że język nie jest bezkontekstowy przebiega według schematu: skoro dany język nie posiada tych własności, to nie może być bezkontekstowy. Myśląc o takim zastosowaniu, możemy odwrócić lemat o pompowaniu:

Lemat o pompowaniu dla języków bezkontekstowych— sformułowanie alternatywne: Niech A będzie językiem. Jeśli dla każdego $k > 0$ istnieje takie słowo z, że $|z| \geq k$ oraz dla dowolnego podziału słowa z na słowa u, v, w, x i y, $uvwxy = z$ takiego, że $vx \neq \varepsilon$ i $|vwx| \leq k$ istnieje takie $i \geq 0$, że $uv^iwx^iy \notin A$, wówczas A nie jest bezkontekstowy.

Nawet w postaci odwróconej lemat o pompowaniu jest trudny do zrozumienia, gdyż mamy tu do czynienia z czterema poziomami zagnieżdżonych naprzemiennych kwantyfikatorów uniwersalnych i egzystencjalnych. Takie zagnieżdżenie naprzemiennych kwantyfikatorów możemy sobie wyobrazić jako ruchy dwóch graczy — gracze na przemian wskazują jakie wartości powinny przyjąć zmienne spod kwantyfikatorów, jeden gracz spod uniwersalnych, a drugi spod egzystencjalnych. Pierwszy gracz stara się pokazać, że dany język jest bezkontekstowy, a drugi, że nie. O wygranej decyduje, czy dla wybranych wartości zmiennych zachodzi własność objęta kwantyfikatorami. Gra jest skończona, więc jeden z graczy ma strategię wygrywającą. W zależności od tego, który z graczy to jest, język może być bezkontekstowy lub nie.

Lemat o pompowaniu dla języków bezkontekstowych — gra z Demonem: Prowadzimy z Demonem grę. Staramy się pokazać, że język nie jest bezkontekstowy, a Demon stara się pokazać, że język mimo wszystko jest bezkontekstowy. Zasady gry są następujące:

1. Demon wybiera $k > 0$. (Jeśli A jest faktycznie bezkontekstowy, to Demon może wybrać za $k = 2^{|N|}$.)

2. Wybieramy takie słowo $z \in A$, że $|z| \geq k$.

3. Demon dzieli z na takie słowa u, v, w, x, y, $uvwxy = z$, że $vx \neq \varepsilon$ i $|vwx| \leq k$. (Jeśli A jest faktycznie bezkontekstowy, to może to być podział wyznaczony przez powtarzający się nieterminal na pewnej ścieżce od liścia do korzenia.)

4. Wybieramy $i \geq 0$.

Jeśli $uv^iwx^iy \notin A$, to wygraliśmy, wpp. wygrał Demon.

Oryginalne sformułowanie lematu o pompowaniu mówi, że jeżeli język jest bezkontekstowy, to Demon ma strategię wygrywającą. Natomiast odwrotne sformułowanie mówi, że jeżeli my mamy strategię wygrywającą, to język nie jest bezkontekstowy. Pamiętajmy, że lemat o pompowaniu nie służy do pokazywania, że język jest bezkontekstowy. Nie zawsze pozwala on na pokazanie, że język, który nie jest bezkontekstowy faktycznie taki nie jest — istnieją języki, które nie są bezkontekstowe i spełniają lemat o pompowaniu.

Przykład: Standardowym przykładem języka, który nie jest bezkontekstowy to $A = \{a^nb^nc^n : n \geq 0\}$. Udowodnimy to korzystając z lematu o pompowaniu sformułowanego jako gra z Demonem. Nasza strategia wygląda następująco:

- Demon wybiera k.
- Wybieramy słowo $z = a^kb^kc^k$.
- Demon dzieli słowo $z = uvwxy$.
- Wybieramy $i = 2$.

Wiemy, że $|vwx| \leq k$ i $vx \neq \varepsilon$. Tak więc słowa v i x mogą zawierać jeden rodzaj znaków lub dwa, ale nie wszystkie trzy. Czyli nie może zachodzić $\#_a(uv^iwx^iy) = \#_b(uv^iwx^iy) = \#_c(uv^iwx^iy)$. Stąd wynika, że $uv^iwx^iy \notin A$.

Tak więc powyższa strategia jest wygrywająca, czyli A nie jest bezkontekstowy. $\square$

Przykład: Pokażemy, że język $B = \{a^ib^ja^ib^j : i, j \geq 0\}$ nie jest bezkontekstowy. Nasza strategia wygląda następująco:

- Demon wybiera k.
- Wybieramy słowo $z = a^kb^ka^kb^k$.
- Demon dzieli słowo $z = uvwxy$.
- Wybieramy $i = 2$.

Wiemy, że $|vwx| \leq k$ i $vx \neq \varepsilon$. Możliwe są trzy przypadki:

Jeżeli słowo v lub x nie jest w całości zbudowane z jednego rodzaju znaków, to $uv^iwx^iy \notin L(a^*b^*a^*b^*)$ i tym samym $uv^iwx^iy \notin B$.

Jeżeli słowa v i x są zbudowane w całości z tych samych znaków, to znaczy że oba pochodzą z tego samego podsłowa postaci a^k lub b^k. Ale wówczas $uv^iwx^iy \notin B$.

Pozostaje jeszcze przypadek, gdy słowo v jest w całości złożone z liter a, a x z liter b (lub odwrotnie). Wówczas oba te słowa pochodzą z sąsiednich „bloków" znaków a i b. Stąd również $uv^iwx^iy \notin B$.

Tak więc powyższa strategia jest wygrywająca, czyli B nie jest bezkontekstowy. □

11.4 Inne metody dowodzenia niebezkontekstowości języków

Zachodzi następujący fakt, którego nie będziemy teraz dowodzić:

Fakt 15. *Przecięcie języka bezkontekstowego i regularnego jest bezkontekstowe.*

Zachodzi też inny fakt:

Fakt 16. *Suma języków bezkontekstowych jest bezkontekstowa.*

Dowód: Jeżeli A i B są językami bezkontekstowymi, to istnieją generujące je gramatyki $G_A = \langle N_A, \Sigma, P_A, S_A \rangle$ i $G_B = \langle N_B, \Sigma, P_B, S_B \rangle$. Co więcej, możemy założyć, że gramatyki te mają rozłączne zbiory nieterminali $N_A \cap N_B = \emptyset$. Niech S będzie nowym symbolem nieterminalnym nie występującym ani w G_A, ani w G_B. Wówczas gramatyka:

$$G = \langle N_A \cup N_B \cup \{S\}, \Sigma, P_A \cup P_B, S \rangle$$

generuje język $A \cup B$. □

Korzystając z tych faktów możemy pokazywać nie wprost, że języki nie są bezkontekstowe. Schemat rozumowania jest następujący: Zakładamy, że dany język jest bezkontekstowy. Z powyższych faktów wynika, że jakiś inny język, który nie jest bezkontekstowy, musiałby taki być. A więc uprzednie założenie musiało być fałszywe. Stąd dany język nie jest bezkontekstowy.

Przykład: Pokażemy, że język $C = \{ww : w \in \{a,b\}^*\}$ nie jest bezkontekstowy. Załóżmy przeciwnie, że jest. Wówczas:

$$C \cap L(a^*b^*a^*b^*) = \{a^ib^ja^ib^j : i,j \geq 0\} = B$$

Wiemy wszak, że B nie jest bezkontekstowy, a więc C też nie może być. □

11.5 Podsumowanie

Wykład ten był poświęcony przede wszystkim lematowi o pompowaniu dla języków bezkontekstowych. Poznaliśmy trzy równoważne sformułowania tego lematu, oraz przykłady jego zastosowania do wykazywania, że określone języki nie są bezkontekstowe. Poznaliśmy też metodę wykazywania, że dane języki nie są bezkontekstowe, w oparciu o własności klasy języków bezkontekstowych.

11.6 Skorowidz

- **Lemat o pompowaniu dla języków bezkontekstowych** mówi, że dla każdego języka bezkontekstowego, każde dostatecznie długie słowo w tym języku można „pompować" (w dwóch miejscach równocześnie) dowolnie je wydłużając. Wykazywanie braku tej własności jest metodą pokazywania, że dany język nie jest bezkontekstowy.
- **Gra z Demonem** — alternatywne sformułowanie lematu o pompowaniu dla języków bezkontekstowych jako gry.

11.7 Praca domowa

1. (5 p.) Udowodnij, że język $D = \{a^{2^i} : i \geq 0\}$ nie jest bezkontekstowy.
2. (5 p.) Dla każdego z poniższych języków określ, czy jest on: (i) regularny, (ii) bezkontekstowy, ale nie regularny, (iii) nie jest bezkontekstowy:
   (a) $\{a^i b^j c^k : 2i + j = k\}$,
   (b) $\{a^i b^j c^k : i + j + k = 42\}$,
   (c) $\{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}$,
   (d) $\{a^i b^j c^k : i + 2j = k \vee 3i + j = 2k\}$,
   (e) $\{a^{n^2} : n \geq 0\}$.

11.8 Ćwiczenia

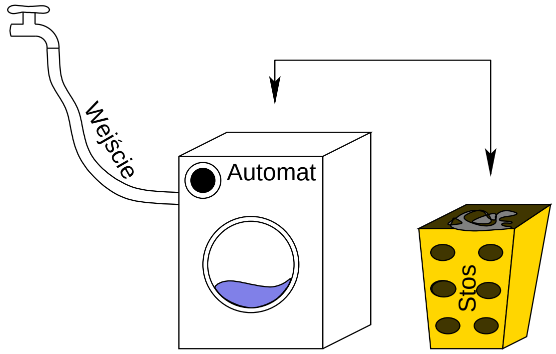
1. Udowodnij, że następujące języki nie są bezkontekstowe:
   - $\{a^{n^5} : n \geq 0\}$,
   - $\{a^i b^j c^k : k = i \cdot j\}$,
   - $\{a^p :\ p$ jest liczbą Fibonacciego$\}$,
   - $\{w \in \{a, b, c, d\} : \#_a(w) = \#_b(w) \wedge \#_c(w) = \#_d(w)\}$,
   - $\{a^n b^{n!} : n \geq 0\}$.
2. Dla każdego z poniższych języków określ, czy jest on: (i) regularny, (ii) bezkontekstowy, ale nie regularny, (iii) nie jest bezkontekstowy:

- $\{a^i b^j c^k : 3i = 2j + k\}$,
- $\{a^i b^j : i + j = k^3$ dla pewnego $k \geq 0\}$,
- $\{a^i b^i c^i : i \leq 42\}$,
- $\{aw_1bw_2c : w_1, w_2 \in \{a, b, c\}^*\}$,
- $\{a^i b^j : i > j\}$,
- $\{a^i b^j : i \neq j\}$,
- $\{awbwc : w \in \{a, b, c\}^*\}$,
- $\{a^i b^j : i + 2j = 42\}$,
- $\{a^i b^j c^k : i \cdot j = 2^k\}$,
- $\{a^i b^j c^k : i \cdot j \cdot k = 6\}$.

Wykład 12. Automaty stosowe

12.1 Wstęp

W tym wykładzie poznamy model obliczeń odpowiadający gramatykom bezkontekstowym (tak jak automaty skończone odpowiadały wzorcom). Ten model, to automaty stosowe. Automaty stosowe możemy sobie wyobrazić jako automaty skończone wyposażone w dodatkową nieograniczoną pamięć, ale działającą na zasadzie stosu.



To znaczy, na wierzch stosu można zawsze włożyć nowy element. Jeśli stos nie jest pusty, to można zawsze zobaczyć jaki element jest na wierzchu i ew. zdjąć ten element. Nie można natomiast oglądać ani wyjmować elementów z wnętrza stosu.

Zobaczymy na przykładach jak można konstruować automaty stosowe i pokażemy, że faktycznie ich siła wyrazu jest taka sama jak gramatyk bezkontekstowych.

Pod względem siły wyrazu automaty stosowe są też równoważne imperatywnym językom programowania bez plików (pomocniczych) dynamicznych struktur danych, wskaźników i parametrów proceduralnych. Przy tych ograniczeniach dane globalne w programie są stałego rozmiaru, a danych dynamicznych nie ma. Można korzystać z rekurencji i nie ma ograniczenia na rozmiar stosu, jednak wielkość danych związanych z każdym poziomem rekurencji jest ograniczona przez stałą i nie ma możliwości wyciągania danych z głębi stosu. Jak zobaczymy dokładnie odpowiada to automatom stosowym.

12.2 Automaty stosowe

Automat stosowy, podobnie jak automat skończony, ma skończony zbiór stanów i stan początkowy. Dodatkowo jest on wyposażony w stos, na który może wkładać elementy, podglądać element znajdujący się na wierzchu i zdejmować elementy. Przejścia automatu stosowego są trochę bardziej skomplikowane niż przejścia w automacie skończonym, gdyż oprócz wczytywania znaków z wejścia i zmiany stanów obejmują również operacje na stosie. Automat taki może być niedeterministyczny i może zawierać ε-przejścia, tzn. wykonaniu przejścia nie musi towarzyszyć wczytanie znaku z wejścia. Natomiast przedstawione tutaj automaty stosowe będą pozbawione stanów akceptujących — automat będzie sygnalizował akceptację wejścia opróżniając stos[102].

Chcąc opisać działanie automatów stosowych musimy wprowadzić pojęcie *konfiguracji*. Konfiguracja będzie obejmować stan automatu (ze skończonego zbioru stanów), zawartość stosu i pozostałe do wczytania wejście. Automat stosowy w jednym kroku obliczeń wykonuje następujące czynności:

1. Podgląda znak czekający na wczytanie na wejściu.
2. Podgląda element na wierzchołku stosu.
3. Na podstawie tych informacji wybiera jedno z przejść do wykonania.
4. Jeśli to nie jest ε-przejście, to wczytywany jest jeden znak z wejścia.
5. Zdejmowany jest element z wierzchołku stosu.
6. Pewna liczba określonych elementów może być włożona na stos.
7. Zgodnie z przejściem zmieniany jest stan.

Może wydawać się nienaturalne, że w każdym kroku zdejmujemy element z wierzchołka stosu. Jednak dzięki temu, opisany powyżej krok automatu jest na tyle elastyczny, że może zdejmować elementy ze stosu, wkładać nowe, podmieniać element na wierzchołku lub nie zmieniać stosu. Jeżeli chcemy zdjąć element z wierzchołku stosu, to wystarczy, że nie będziemy nic wkładać. Jeżeli chcemy podmienić element na wierzchołku, to wystarczy, że włożymy jeden element, który zastąpi element zdjęty z wierzchołka. Jeżeli chcemy włożyć elementy na stos, to wkładamy na stos właśnie zdjęty element, wraz z elementami, które mają być umieszczone na stosie. Jeśli nie chcemy zmieniać zawartości stosu, to po prostu wkładamy dokładnie ten sam element, który właśnie został zdjęty. Możemy włożyć na stos kilka elementów w jednym kroku, ale niestety nie możemy w jednym kroku zdjąć więcej niż jeden element.

Zauważmy, że powyższy schemat postępowania wymaga, żeby stos nie był pusty. Inaczej nie ma czego podglądać ani zdejmować ze stosu. Dlatego też, w momencie uruchomienia automatu stos nie jest pusty. Zawiera jeden wyróżniony element, który będziemy nazywać „pinezką” i oznaczać przez $\perp$. Jak tylko stos zostaje opróżniony, automat zatrzymuje się i dalsze jego działanie nie jest możliwe. Jeżeli automat zatrzymuje się z pustym stosem po wczytaniu **całego** słowa z wejścia, to przyjmujemy, że słowo to zostało zaakceptowane.

Zanim formalnie zdefiniujemy automaty stosowe i akceptowane przez nie języki, zobaczmy prosty nieformalny przykład:

[102]W literaturze można spotkać trzy różne definicje, wzajemnie równoważnych automatów stosowych: akceptujących pustym stosem, akceptujących stanem i akceptujących pustym stosem i stanem równocześnie. Tutaj przedstawiamy najprostszy z tych rodzajów automatów stosowych — akceptujące pustym stosem.

Przykład: Zbudujemy automat akceptujący słowa postaci $a^n b^n$, dla $n \geq 0$. Automat taki będzie miał dwa stany, s i q, przy czym s będzie stanem początkowym. Na stosie będziemy trzymać dwa rodzaje elementów: $\perp$ i A. Będąc w stanie s automat ma dwa przejścia do wyboru: może wczytać z wejścia znak a i włożyć na stos A, lub nic nie wczytywać, ani nie zmieniać zawartości stosu i przejść do stanu q. W stanie q automat ma również dwa przejścia do wyboru: jeżeli na wejściu czeka na wczytanie znak b i na wierzchołku stosu jest A, to może wczytać z wejścia b i zdjąć ze stosu A. Jeżeli natomiast na wierzchołku stosu jest $\perp$, to automat może zdjąć ze stosu $\perp$ i tym samym zakończyć działanie.

Jedyna możliwość, żeby zaakceptować słowo wygląda tak: Automat wczytuje sekwencję a^n i umieszcza na stosie A^n. Następnie przechodzi do stanu q i wczytuje znaki b zdejmując ze stosu A. Jednych i drugich musi być tyle samo, czyli wczytana zostaje sekwencja b^n. Po wczytaniu tej sekwencji odsłania się $\perp$ i jest zdejmowana ze stosu. Jeżeli automat ma zaakceptować słowo, to musi ono być w tym momencie całe wczytane, czyli było to słowo $a^n b^n$.

Poniższy diagram przedstawia obliczenie akceptujące słowo $aaabbb$. Widać na nim kolejne zawartości stosu i stany, w których jest automat.

$$
\begin{array}{ccccccccc}
 & & & A & A & & & & \\
 & & A & A & A & A & & & \\
 & A & A & A & A & A & A & & \\
\perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \\
\hline
s & s & s & s & q & q & q & q & q
\end{array}
$$

Def. 42. Automat stosowy to dowolna taka szóstka $M = \langle Q, \Sigma, \Gamma, \delta, s, \perp \rangle$, w której:

- Q to skończony zbiór stanów,
- Σ to (skończony) alfabet wejściowy,
- Γ to (skończony) alfabet stosowy,
- δ to skończona relacja przejścia, $\delta \subseteq (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times (Q \times \Gamma^*)$; określa ona dla danego aktualnego stanu, znaku na wierzchołku stosu i znaku czekającego na wczytanie (lub ε jeżeli nic nie jest wczytywane), do jakiego stanu należy przejść i co należy włożyć na stos (uprzednio zdjąwszy element z wierzchołka stosu),
- $s \in Q$ to stan początkowy,
- $\perp \in \Gamma$ to symbol początkowy na stosie, tzw. „pinezka”.

$\square$

Notacja: W przypadku automatów stosowych, przedstawienie przejść w postaci diagramu nie jest czytelne. Nie będziemy jednak pisać:

$$\delta = \{((s,a,A),(s,AA)),((s,a,\perp),(s,A\perp)),\dots\}$$

jakby wynikało z definicji, gdyż byłoby to zupełnie nieczytelne. Zamiast tego będziemy przedstawiać przejścia jako reguły postaci:

$$\begin{array}{rcl}(s,A) & \stackrel{a}{\to} & (s,AA)\\ (s,\perp) & \stackrel{a}{\to} & (s,A\perp)\end{array}$$

lub w postaci skrótów:

$$(s,x) \stackrel{a}{\to} (s,Ax) \text{ dla } x = A,\perp$$

Konfiguracje automatów stosowych definiujemy następująco:

Def. 43. Niech $M = \langle Q,\Sigma,\Gamma,\delta,s\rangle$ będzie ustalonym automatem stosowym. Konfiguracja takiego automatu stosowego, to dowolna trójka:

$$\langle q,\alpha,\beta\rangle \in Q\times\Sigma^*\times\Gamma^*$$

gdzie q reprezentuje aktualny stan, α pozostałe do wczytania wejście, a β zawartość stosu (czytaną od wierzchołka w głąb).

Konfiguracja początkowa dla słowa x, to $\langle s,x,\perp\rangle$. □

Jeden krok obliczeń automatu stosowego opisujemy relacją $\to$ określoną na konfiguracjach.

Def. 44. Niech $M = \langle Q,\Sigma,\Gamma,\delta,s\rangle$ będzie ustalonym automatem stosowym. Relacja przejścia między konfiguracjami tego automatu $\to$ to najmniejsza relacja spełniająca następujące warunki:

- jeśli δ zawiera przejście postaci $(p,A) \stackrel{a}{\to} (q,\gamma)$ (dla $p,q\in Q$, $a\in\Sigma$, $A\in\Gamma$ i $\gamma\in\Gamma^*$), to dla dowolnych $y\in\Sigma^*$ i $\beta\in\Gamma^*$ mamy: $\langle p,ay,A\beta\rangle \to \langle q,y,\gamma\beta\rangle$,
- jeśli δ zawiera przejście postaci $(p,A) \stackrel{\varepsilon}{\to} (q,\gamma)$, (dla $p,q\in Q$, $A\in\Gamma$ i $\gamma\in\Gamma^*$) to dla dowolnych $y\in\Sigma^*,\beta\in\Gamma^*$ mamy: $\langle p,y,A\beta\rangle \to \langle q,y,\gamma\beta\rangle$.

Przez $\to^*$ będziemy oznaczać zwrotno-przechodnie domknięcie relacji $\to$. □

Inaczej mówiąc, $(p,A) \stackrel{a}{\to} (q,\gamma)$ oznacza, że możemy będąc w stanie p i mając na wierzchołku stosu A przejść do stanu q wczytując z wejścia a i zastępując na stosie A przez γ. Podobnie $(p,A) \stackrel{\varepsilon}{\to} (q,\gamma)$ oznacza, że możemy będąc w stanie p i mając na wierzchołku stosu A przejść do stanu q nie wczytując nic z wejścia i zastępując na stosie A przez γ.

Relacja $\to$ opisuje pojedyncze kroki obliczenia, a relacja $\to^*$ dowolnie długie obliczenia.

Def. 45. Niech $M = \langle Q, \Sigma, \Gamma, \delta, s\rangle$ będzie ustalonym automatem stosowym. Język akceptowany przez automat stosowy M jest określony następująco:

$$L(M) = \{x \in \Sigma^* : \exists_{q \in Q}\ \langle s, x, \bot\rangle \to^* \langle q, \varepsilon, \varepsilon\rangle\}$$

$\square$

Inaczej mówiąc, automat stosowy akceptuje takie słowa x, dla których istnieją obliczenia prowadzące od konfiguracji początkowej (dla słowa x) do konfiguracji, w której całe słowo zostało wczytane, a stos opróżniony. Pamiętajmy, że automat stosowy może być niedeterministyczny, a więc wystarczy żeby istniało jedno obliczenie akceptujące, żeby słowo było akceptowane przez automat.

Przykład: Skonstruujemy automat akceptujący słowa zawierające tyle samo liter a, co b. Automat ten będzie używać stosu jako licznika. Z każdym wczytanym a licznik zwiększa się o 1, a z każdym wczytanym b zmniejsza się o 1. Dodatnie wartości licznika reprezentujemy jako $\bot$ i stos A, a ujemne, jako $\bot$ i stos B. Dodatkowo, w momentach gdy odsłonięta jest pinezka, automat może ją zdjąć ze stosu. Automat akceptuje tylko takie słowa x, po których wczytaniu licznik jest wyzerowany, czyli $\#_a(x) = \#_b(x)$.

Automat ten ma jeden stan s i działa w następujący sposób:

$$\begin{array}{lcl}
(s, \bot) & \xrightarrow{a} & (s, A\,\bot) \\
(s, A) & \xrightarrow{a} & (s, AA) \\
(s, B) & \xrightarrow{a} & (s, \varepsilon) \\
(s, \bot) & \xrightarrow{\varepsilon} & (s, \varepsilon)
\end{array}
\qquad
\begin{array}{lcl}
(s, \bot) & \xrightarrow{b} & (s, B\,\bot) \\
(s, B) & \xrightarrow{b} & (s, BB) \\
(s, A) & \xrightarrow{b} & (s, \varepsilon)
\end{array}$$

Przykładowe obliczenie dla $aabbba$:

$$\begin{array}{cccccccc}
 & & A & & & & & \\
 & A & A & A & & B & & \\
\bot & \bot & \bot & \bot & \bot & \bot & \bot & \\
\hline
s & s & s & s & s & s & s & s
\end{array}$$

Oczywiście taki automat może usunąć pinezkę gdy jeszcze nie całe słowo jest wczytane. Takie obliczenie nie doprowadzi do zaakceptowania słowa. Pamiętajmy jednak, że automaty stosowe są niedeterministyczne i wystarczy, aby istniało choć jedno obliczenie akceptujące dane słowo, aby należało ono do języka akceptowanego przez automat stosowy.

Przykład: Skonstruujemy automat akceptujący wyrażenia nawiasowe zbudowane z kwadratowych nawiasów [i]. Automat ten działa podobnie jak poprzedni, ale licznik może przyjmować tylko dodatnie wartości. Na stosie trzymamy pinezkę i tyle nawiasów zamykających ile wynosi wartość licznika. Z wczytaniem każdego nawiasu otwierającego licznik jest zwiększany o 1, a z wczytaniem każdego nawiasu zamykającego zmniejszany o 1.

$$\begin{array}{rcl} (s,x) & \xrightarrow{[} & (s,]x) \text{ dla } x =], \bot \\ (s,]) & \xrightarrow{]} & (s,\varepsilon) \\ (s,\bot) & \xrightarrow{\varepsilon} & (s,\varepsilon) \end{array}$$

Przykładowe obliczenie dla [[][[]]]:

$$\begin{array}{cccccccccc}
 & & & & &] & & & & \\
 & &] & &] &] &] & & & \\
 &] &] &] &] &] &] &] & & \\
\bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \\
\hline
s & s & s & s & s & s & s & s & s & s
\end{array}$$

Przykład: Automat akceptujący palindromy nad alfabetem $\{a,b\}$. Automat ten musi niedeterministycznie zgadnąć gdzie jest środek palindromu i czy jest on parzystej, czy nieparzystej długości.

Nasz automat będzie miał dwa stany: s i p. Na stosie będziemy przechowywać pinezkę i znaki alfabetu wejściowego. W pierwszej fazie (stan s) automat zapamiętuje pierwszą połowę słowa na stosie. W drugiej fazie (stan p) zdejmuje znaki ze stosu, sprawdzając, czy są one takie same jak wczytywane znaki. Jeżeli słowo jest nieparzystej długości, to przechodząc z pierwszej fazy do drugiej wczytujemy środkowy znak nie zmieniając zawartości stosu. Jeżeli zaś jest parzystej długości, to przechodząc do drugiej fazy nic nie wczytujemy.

$$\begin{array}{rcl l l} (s,x) & \xrightarrow{y} & (s,yx) & x = \bot, a, b & y = a, b \\ (s,x) & \xrightarrow{y} & (p,x) & x = \bot, a, b & y = a, b, \varepsilon \\ (p,x) & \xrightarrow{x} & (p,\varepsilon) & x = a, b & \\ (p,\bot) & \xrightarrow{\varepsilon} & (s,\varepsilon) & & \end{array}$$

Przykładowe obliczenia akceptujące słowa *abba* i *baabaab*:

$$\begin{array}{ccccccc}
 & & b & b & & & \\
 & a & a & a & a & & \\
\bot & \bot & \bot & \bot & \bot & \bot & \\
\hline
s & s & s & p & p & p & p
\end{array}$$

$$\begin{array}{ccccccccc}
 & & & a & a & & & & \\
 & & a & a & a & a & & & \\
 & b & b & b & b & b & b & & \\
\bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \\
\hline
s & s & s & s & p & p & p & p & p
\end{array}$$

Przykład: W poprzednim wykładzie pokazaliśmy, że język $\{ww : w \in \{a,b\}^*\}$ nie jest bezkontekstowy. Za chwilę pokażemy, że automaty stosowe akceptują dokładnie języki bezkontekstowe. Teraz jednak skonstruujemy automat akceptujący język $\overline{\{ww : w \in \{a,b\}^*\}}$, pokazując tym samym, że jest on bezkontekstowy.

Słowo $x = a_1 a_2 \dots a_k$ należy do języka $\overline{\{ww : w \in \{a,b\}^*\}}$ w dwóch przypadkach. Albo jest nieparzystej długości. Albo jest parzystej długości i istnieje taka pozycja i ($1 \le i \le \frac{k}{2}$) w pierwszej połowie tego słowa, że na i-tej pozycji w słowie x jest inny znak niż na odpowiadającej jej pozycji w drugiej połowie słowa, $a_i \neq a_{i+\frac{k}{2}}$.

Nasz automat będzie miał szereg stanów, przy czym stanem początkowym będzie s. Na stosie będziemy trzymać dwa rodzaje symboli: $\perp$ i $\circ$. Automat zgaduje najpierw, czy słowo jest parzystej (stan p), czy nieparzystej (stan n) długości. Jeśli nieparzystej, to tylko to sprawdza, że faktycznie tak jest.

$$\begin{array}{rcl}
(s,\perp) & \stackrel{\varepsilon}{\rightarrow} & (n,\perp) \\
(s,\perp) & \stackrel{\varepsilon}{\rightarrow} & (p,\perp) \\
(n,\perp) & \stackrel{x}{\rightarrow} & (m,\perp) \text{ dla } x = a,b \\
(m,\perp) & \stackrel{x}{\rightarrow} & (n,\perp) \text{ dla } x = a,b \\
(m,\perp) & \stackrel{\varepsilon}{\rightarrow} & (m,\varepsilon)
\end{array}$$

Jeśli słowo jest parzystej, to automat stosowy musi zgadnąć niedeterministycznie na jakiej pozycji i istnieje niezgodność między pierwszą i drugą połową słowa, a następnie sprawdzić, że faktycznie tak jest. Inaczej mówiąc, słowo x musi być postaci $uavbw$ lub $ubvaw$, przy czym $|v| = |u| + |w|$. To czy w pierwszej połowie na i-tej pozycji jest a czy b można zapamiętać w stanie automatu. Problemem jest natomiast odliczenie odpowiadających sobie pozycji w obu połówkach słowa. Zauważmy jednak, że automat nie musi wiedzieć gdzie jest połowa długości słowa. Wystarczy, że będzie zachodzić $|v| = |u| + |w|$.

Nasz automat najpierw wczytuje słowo u, wkładając na stos $|u|$ znaków $\circ$.

$$(p,i) \stackrel{j}{\rightarrow} (p,\circ i) \text{ dla } i = \perp,\circ,\ j = a,b$$

Następnie niedeterministycznie zgaduje kiedy jest koniec u i wczytuje jeden znak, zapamiętując go na stanie.

$$(p,i) \stackrel{j}{\rightarrow} (q_j,i) \text{ dla } i = \perp,\circ,\ j = a,b$$

Następnie nasz automat wczytuje tyle znaków z wejścia, ile $\circ$ jest na stosie. W ten sposób wczytujemy prefiks v długości $|u|$. Po zdjęciu ze stosu wszystkich $\circ$ przechodzimy do kolejnej fazy.

$$\begin{array}{rcl}
(q_i,\circ) & \stackrel{j}{\rightarrow} & (q_i,\varepsilon) \text{ dla } i = a,b,\ j = a,b \\
(q_i,\perp) & \stackrel{\varepsilon}{\rightarrow} & (r_i,\perp) \text{ dla } i = a,b
\end{array}$$

Następnie nasz automat powinien wczytać z wejścia $|w| = |v| - |u|$ znaków słowa v. Liczbę tę zgaduje niedeterministycznie, wkładając równocześnie $|w|$ znaków $\circ$ na stos. W momencie gdy automat zgaduje niedeterministycznie, że całe słowo v zostało wczytane, wczytuje jeden znak z wejścia sprawdzając równocześnie, że jest on różny od znaku zapamiętanego na stanie.

$$
\begin{array}{rcl}
(r_i, j) & \stackrel{l}{\to} & (r_i, \circ j) \text{ dla } i = a, b \ j = \perp, \circ, \ l = a, b \\
(r_i, j) & \stackrel{l}{\to} & (t, j) \text{ dla } i = a, b, \ j = \perp, \circ, \ l = a, b, \ l \neq i
\end{array}
$$

Pozostało jeszcze do wczytania w, przy czym na stosie powinno być $|w|$ znaków $\circ$. Po wczytaniu słowa w powinna odsłonić się pinezka, którą zdejmujemy.

$$
\begin{array}{rcl}
(t, \circ) & \stackrel{i}{\to} & (t, \varepsilon) \text{ dla } i = a, b \\
(t, \perp) & \stackrel{\varepsilon}{\to} & (t, \varepsilon)
\end{array}
$$

Oto przykładowe obliczenie akceptujące słowo *abbaabaa*.

$$
\begin{array}{cccccccccccc}
 & & & \circ & \circ & & & & & & & \\
 & & \circ & \circ & \circ & \circ & & & \circ & \circ & & \\
\perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \perp & \\
\hline
s & p & p & p & q_b & q_b & q_b & r_b & r_b & t & t & t
\end{array}
$$

Skonstruowany automat jest wysoce niedeterministyczny. Jeśli gdziekolwiek cokolwiek zostanie źle zgadnięte, to obliczenie nie doprowadzi do zaakceptowania słowa. Pamiętajmy jednak, iż wystarczy, że *można* tak zgadnąć, aby obliczenie było akceptujące, aby słowo było akceptowane przez automat.

12.3 Równoważność gramatyk bezkontekstowych i automatów stosowych

Pokażemy teraz, że siła wyrazu automatów stosowych jest dokładanie taka sama, jak gramatyk bezkontekstowych — tzn. akceptują one dokładnie języki bezkontekstowe.

Tw. 5. *Niech A będzie językiem nad alfabetem Σ. Następujące stwierdzenia są równoważne:*

- *A jest bezkontekstowy, tzn. A jest generowany przez pewną gramatykę bezkontekstową G, $A = L(G)$,*
- *istnieje automat stosowy M akceptujący A, $A = L(M)$.*

Przedstawimy tu jedynie szkic dowodu — oprzemy sie na trzech faktach. Pokażemy jak dla danej gramatyki bezkontekstowej skonstruować równoważny jej automat stosowy (z jednym stanem). Następnie pokażemy, że konstrukcję tę można odwrócić, tzn. dla danego automatu stosowego z jednym stanem można skonstruować równoważną mu gramatykę bezkontekstową. Na koniec pokażemy, jak zredukować liczbę stanów w automacie stosowym do jednego.

12.3.1 Automat stosowy równoważny danej gramatyce bezkontekstowej

Niech G będzie daną gramatyką bezkontekstową $G = \langle N, \Sigma, P, S\rangle$. Zbudujemy równoważny jej automat stosowy $M = \langle Q, \Sigma, \Gamma, \delta, s\rangle$. Automat ten będzie miał tylko jeden stan $Q = \{s\}$. Na stosie będzie przechowywał symbole terminalne i nieterminalne. $\Gamma = N \cup \Sigma$. Przyjmujemy, że $S = \perp$. Dla każdej produkcji $X \to \alpha$ mamy przejście:

$$(s, X) \ \stackrel{\varepsilon}{\to}\ (s, \alpha)$$

Dodatkowo, dla każdego $a \in \Sigma$ mamy przejście:

$$(s, a) \ \stackrel{a}{\to}\ (s, \varepsilon)$$

Nasz automat będzie działa w następujący sposób: Jeśli na wierzchołku stosu jest terminal, to oczekujemy, że czeka on na wejściu, wczytujemy go i zdejmujemy ze stosu. Jeśli na wierzchołku stosu jest nieterminal, to należy wczytać pewne słowo, które można z niego wyprowadzić. Najpierw jednak należy zdecydować, jaka produkcja rozpoczyna wyprowadzenie takiego słowa. Zdejmujemy więc dany nieterminal ze stosu i wkładamy na stos prawą stronę wybranej produkcji dla tego nieterminala. Akceptujemy pustym stosem, gdy całe wyprowadzone słowo zostało wczytane.

Można pokazać (przez indukcję po długości wyprowadzenia), że M realizuje tzw. lewostronne wyprowadzenia, to znaczy takie, w których zawsze jest rozwijany skrajnie lewy nieterminal. Jeśli wczytywane słowo to x, $x = yz$, to konfiguracji $\langle s, z, \alpha\rangle$ odpowiada w wyprowadzeniu słowo $y\alpha$. Konfiguracja początkowa $\langle s, x, S\rangle$ odpowiada aksjomatowi S, a konfiguracja akceptująca $(s, \varepsilon, \varepsilon)$ odpowiada w wyprowadzeniu słowu x. Stąd $L(M) = L(G)$.

Przykład: Weźmy, przedstawioną wcześniej, gramatykę bezkontekstową generującą wyrażenia nawiasowe:

$$S \to [S] \mid SS \mid \varepsilon$$

Stosując przedstawioną powyżej metodę otrzymujemy następujący automat stosowy:

$$\begin{array}{rcl}
(s, S) & \stackrel{\varepsilon}{\to} & (s, [S]) \\
(s, S) & \stackrel{\varepsilon}{\to} & (s, SS) \\
(s, S) & \stackrel{\varepsilon}{\to} & (s, \varepsilon) \\
(s, [) & \stackrel{[}{\to} & (s, \varepsilon) \\
(s,]) & \stackrel{]}{\to} & (s, \varepsilon)
\end{array}$$

Jest to inny automat niż skonstruowany przez nas wcześniej automat stosowy akceptujący wyrażenia nawiasowe. Oto jego przykładowe obliczenie akceptujące [[][[]]].

$$\begin{array}{cccccccccccccccc}
 & & & & [& & & & & & [& & & & & \\
 & & & & S & S & & & [& & S & S & & & & \\
 & [& & S &] &] &] & & S & S &] &] &] & & & \\
 & S & S & S & S & S & S & S &] &] &] &] &] &] & & \\
S &] &] &] &] &] &] &] &] &] &] &] &] &] &] & \\
\hline
s & s & s & s & s & s & s & s & s & s & s & s & s & s & s & s
\end{array}$$

12.3.2 Gramatyka bezkontekstowa równoważna danemu automatowi stosowemu z jednym stanem

Przedstawioną w poprzednim punkcie konstrukcję można odwrócić. Załóżmy, że mamy dany automat stosowy z jednym stanem $M = \langle \{s\}, \Sigma, \Gamma, \delta, s, \perp \rangle$. Bez zmniejszenia ogólności możemy założyć, że $\Gamma \cap \Sigma = \emptyset$. Nasza gramatyka ma postać $G = \langle \Gamma, \Sigma, P, \perp \rangle$, przy czym jeśli δ zawiera przejście $(s, A) \stackrel{a}{\rightarrow} (s, \gamma)$ (dla $a \in \Sigma \cup \{\varepsilon\}$), to w P mamy produkcję $A \rightarrow a\gamma$.

Odpowiedniość jest taka jak poprzednio. Jeśli wczytywane słowo to x, $x = yz$, to konfiguracji $\langle s, z, \alpha \rangle$ odpowiada w wyprowadzenie lewostronne $\perp \rightarrow^* y\alpha$. Konfiguracja początkowa $\langle s, x, \perp \rangle$ odpowiada aksjomatowi $\perp$, a konfiguracja akceptująca $(s, \varepsilon, \varepsilon)$ odpowiada w wyprowadzeniu $\perp \rightarrow^* x$. Stąd $L(M) = L(G)$.

Przykład: Weźmy, przedstawiony powyżej, automat stosowy akceptujący słowa zawierające tyle samo liter a, co b.

$$
\begin{array}{lcl}
(s, \perp) & \stackrel{a}{\rightarrow} & (s, A\perp) \\
(s, A) & \stackrel{a}{\rightarrow} & (s, AA) \\
(s, B) & \stackrel{a}{\rightarrow} & (s, \varepsilon) \\
(s, \perp) & \stackrel{\varepsilon}{\rightarrow} & (s, \varepsilon)
\end{array}
\qquad
\begin{array}{lcl}
(s, \perp) & \stackrel{b}{\rightarrow} & (s, B\perp) \\
(s, B) & \stackrel{b}{\rightarrow} & (s, BB) \\
(s, A) & \stackrel{b}{\rightarrow} & (s, \varepsilon)
\end{array}
$$

Stosując przedstawioną powyżej metodę, uzyskujemy następującą równoważną mu gramatykę (dla czytelności $\perp$ zastąpiono przez S):

$$
\begin{array}{lcl}
S & \rightarrow & aAS \mid bBS \mid \varepsilon \\
A & \rightarrow & b \mid aAA \\
B & \rightarrow & a \mid bBB
\end{array}
$$

Oto wyprowadzenie w tej gramatyce słowa $aabbba$:

$$S \rightarrow aAS \rightarrow aaAAS \rightarrow aabAS \rightarrow aabbS \rightarrow aabbbBS \rightarrow aabbbaS \rightarrow aabbba$$

12.3.3 Redukcja zbioru stanów do jednego

Pozostało nam pokazać, że dla każdego automatu stosowego $M = \langle Q, \Sigma, \Gamma, \delta, s, \perp \rangle$ istnieje równoważny mu automat z jednym stanem $M' = \langle \{s\}, \Sigma, \Gamma', \delta' s \rangle$. Bez zmniejszenia ogólności możemy założyć, że automat M w momencie opróżnienia stosu będzie zawsze w jednym stanie q.

Na stosie automaty M' będziemy trzymać trójki postaci $\langle p, a, r\rangle \in \Gamma' = Q \times \Gamma \times Q$. Jeżeli na wierzchołku stosu M' jest trójka $\langle p, A, r\rangle$, to odpowiada to sytuacji, gdy na wierzchołku stosu w M jest A, M jest w stanie p i po zdjęciu symbolu A ze stosu przejdzie do stanu r. Stany pamiętane na stosie M' muszą się „zazębiać", tzn. pod trójką $\langle p, A, r\rangle$ musi być trójka postaci $\langle r, B, t\rangle$ (dla pewnych $B \in \Gamma$ i $t \in Q$) itd.

Automat M' na początku ma na stosie trójkę $\langle s, \perp, q\rangle$ i akceptuje słowo x wtw., gdy M zaczynając w stanie s i z symbolem $\perp$ na stosie, wczytuje x i kończy w stanie q z pustym stosem.

Relacja δ' jest określona następująco:

- jeśli δ zawiera przejście:
$$(p, A) \xrightarrow{a} (r, B_1B_2 \dots B_k)$$
dla pewnych $a \in \Sigma \cup \{\varepsilon\}$, $p, r \in Q$, $k > 0$, $A, B_1, B_2, \dots, B_k \in \Gamma$, to δ' (dla dowolnych $q_1, \dots, q_k \in Q$) zawiera przejścia:
$$(s, \langle p, A, q_k\rangle) \xrightarrow{a} (s, \langle r, B_1, q_1\rangle\langle q_1, B_2, q_2\rangle\rangle \dots \langle q_{k-1}, B_k, q_k\rangle)$$

- jeśli δ zawiera przejście
$$(p, A) \xrightarrow{a} (r, \varepsilon))$$
dla pewnych $a \in \Sigma \cup \{\varepsilon\}$, $p, r \in Q$, $A \in \Gamma$, to δ' zawiera przejście:
$$(s, \langle p, A, r\rangle) \xrightarrow{a} (s, \varepsilon)$$

Automat ten jest wysoce niedeterministyczny. Wkładając trójki na stos musimy z góry przewidzieć w jakim stanie będzie automat M w momencie zdjęcia odpowiedniej trójki w M'. Oczywiście zgadnięcie odpowiednich stanów jest możliwe, a więc M' będzie mógł zasymulować każde obliczenie M (i tylko obliczenia M).

12.4 Podsumowanie

W tym wykładzie poznaliśmy automaty stosowe. Zobaczyliśmy na przykładach jak je można konstruować. Dowiedzieliśmy się też, że są one równoważne gramatykom bezkontekstowym.

12.5 Skorowidz

- **Automat stosowy** to dowolna taka piątka $M = \langle Q, \Sigma, \Gamma, \delta, s\rangle$, w której: Q to skończony zbiór stanów, Σ to alfabet wejściowy, Γ to alfabet stosowy, $\delta \subseteq (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times (Q \times \Gamma^*)$ to relacja przejścia, $s \in Q$ to stan początkowy, $\perp \in \Gamma$ to symbol początkowy na stosie.

- **Konfiguracja automatu stosowego** to dowolna trójka $\langle q, \alpha, \beta\rangle \in Q \times \Sigma^* \times \Gamma^*$ gdzie $M = \langle Q, \Sigma, \Gamma, \delta, s\rangle$ jest danym automatem stosowym.

- **Język akceptowany przez automat stosowy** to

$$L(M) = \{x \in \Sigma^* : \exists_{q \in Q} \ \langle s, x, \bot \rangle \to^* \langle q, \varepsilon, \varepsilon \rangle\}$$

12.6 Praca domowa

1. (4 p.) Skonstruuj automat stosowy akceptujący wyrażenia nawiasowe zawierające trzy rodzaje nawiasów: ()[]{}, przy czym:

   (a) nawiasy okrągłe mogą znajdować się wewnątrz nawiasów okrągłych, kwadratowych lub wąsatych,

   (b) nawiasy kwadratowe mogą znajdować się wewnątrz nawiasów kwadratowych lub wąsatych,

   (c) nawiasy wąsate mogą znajdować się tylko wewnątrz nawiasów wąsatych.

   Na przykład, ()[[()(())]][] jest poprawnym wyrażeniem nawiasowym, a [[]]([][]) nie jest.

2. (3 p.) Przekształć gramatykę:

   $$S \ \to \ aSa \mid bSb \mid a \mid b \mid \varepsilon$$

   generującą palindromy (nad alfabetem $\{a, b\}$), na automat stosowy (stosując konstrukcję podaną w tym wykładzie).

3. (3 p.) Przekształć automat stosowy akceptujący wyrażenia nawiasowe:

   $$\begin{aligned} (s, x) &\xrightarrow{[} (s,]x) \text{ dla } x =], \bot \\ (s,]) &\xrightarrow{]} (s, \varepsilon) \\ (s, \bot) &\xrightarrow{\varepsilon} (s, \varepsilon) \end{aligned}$$

   na równoważną mu gramatykę bezkontekstową (stosując konstrukcję podaną w tym wykładzie).

12.7 Ćwiczenia

Skonstruuj automaty stosowe akceptujące poniższe języki i zasymuluj ich działanie dla podanych słów. Czy potrafisz skonstruować takie automaty, które mają tylko jeden stan? Jeśli tak, to przekształć je na równoważne im gramatyki bezkontekstowe.

1. wyrażenia nawiasowe zawierające trzy rodzaje nawiasów: ()[]{}; przy tym: nawiasy okrągłe mogą otaczać tylko nawiasy okrągłe, nawiasy kwadratowe mogą otaczać nawiasy kwadratowe lub okrągłe, a nawiasy wąsate mogą otaczać nawiasy wąsate lub kwadratowe,

2. palindromy nad alfabetem $\{a, b, c\}$; zasymuluj działanie dla słowa *abcaacba*,
3. $\{a^i b^{2i}\}$; zasymuluj działanie dla słowa *aabbbb*,
4. $\{a^i b^j a^j b^i\}$; zasymuluj działanie dla słów *aabbbaaabb*, *aaabbb* i *bbaa*,
5. $\{w : \#_a(w) >= 2\#_b(w)\}$; zasymuluj działanie dla słowa *ababaaa*,
6. $\{a^i b^j c^k : i \neq j \vee j \neq k\}$,
7. wyrażenia arytmetyczne w notacji polskiej (dla uproszczenia liczby mogą być tylko jednocyfrowe),
8. wyrażenia arytmetyczne w odwrotnej notacji polskiej (dla uproszczenia liczby mogą być tylko jednocyfrowe).

Wykład 13. Analiza składniowa

13.1 Wstęp

Analiza składniowa ma miejsce wówczas, gdy wczytujemy dane o pewnej określonej składni. Zadaniem analizatora składniowego (tzw. *parsera*) jest sprawdzenie, czy dane są poprawne składniowo i rozpoznanie struktury składniowej wczytywanych danych.

Analiza składniowa powstała i rozwinęła się w ramach prac nad budową kompilatorów. Wszak kompilator musi najpierw wczytać i zanalizować składnię wczytywanego programu. Jej zastosowania są jednak dużo szersze. Praktycznie analizę składniową można zastosować w każdym programie, który wczytuje dane posiadające bardziej złożoną składnię, na przykład w: przeglądarkach internetowych, edytorach, systemach składu tekstu, programach konwertujących, czy jakichkolwiek aplikacjach posiadających pliki konfiguracyjne o określonej składni.

Konstrukcja parsera, zwłaszcza w przypadku kompilatorów lub bardziej skomplikowanych języków, to skomplikowane zadanie. Dlatego też powstały narzędzia wspomagające tworzenie parserów. Są to tzw. generatory analizatorów składniowych. Przykładem takiego generatora jest *Bison*, którego poznamy bliżej w tym wykładzie. Bison jest następcą generatora *Yacc* (ang. *yet another compiler compiler* — jeszcze jeden kompilator kompilatorów). Bison jest przeznaczony do generowania parserów w C/C++. Jednak istnieje wiele generatorów parserów i w zasadzie dla każdego języka programowania można znaleźć generator przeznaczony dla danego języka.

Generator parserów na podstawie odpowiedniej specyfikacji generuje kod źródłowy analizatora składniowego. Specyfikacja taka zawiera opis składni wczytywanego języka oraz fragmenty kodu, które są wykonywane dla poszczególnych konstrukcji składniowych. Do opisu składni używa się jednoznacznych gramatyk bezkontekstowych.

Analiza składniowa stanowi kolejną fazę przetwarzania wczytywanych danych po analizie leksykalnej. Tak więc wejściem dla analizy składniowej jest strumień żetonów i atrybutów. Żetony stanowią symbole terminalne w gramatyce bezkontekstowej opisującej składnię. Efektem pracy parsera jest odtworzenie drzewa wyprowadzenia (a ściślej mówiąc jego obejścia) dla wczytywanego tekstu.

Generowane automatycznie parsery mają postać odpowiednich, deterministycznych automatów stosowych. Sposób generowania parserów wykracza poza ramy tego kursu.

13.2 Gramatyki S-atrybutywne

Zanim poznamy język specyfikacji dla Bison'a i Yacc'a, musimy zapoznać się z pojęciem gramatyk S-atrybutywnych. (Nie jest to nic skomplikowanego, a ta odstraszająca nazwa pochodzi stąd, że jest wiele różnych gramatyk atrybutywnych. My jednak nie będziemy ich tutaj omawiać.)

Gramatyki S-atrybutywne to rozszerzenie gramatyk bezkontekstowych — w takim sensie, że podstawowym składnikiem gramatyki S-atrybutywnej jest jednoznaczna gramatyka bezkontekstowa. (Gramatyka musi być jednoznaczna, żeby dla każdego słowa z języka istniało tylko jedno możliwe drzewo wyprowadzenia.)

Natomiast dodatkowo w gramatyce S-atrybutywnej z każdym węzłem w drzewie wyprowadzenia wiążemy pewną obliczaną wartość nazywaną *atrybutem*. (Jeśli potrzebujemy wielu wartości, to możemy pomyśleć o atrybucie jak o n-ce, rekordzie czy strukturze złożonej z tych wartości.) Liśćmi drzewa wyprowadzenia są symbole terminalne, czyli żetony. Żetonom mogą towarzyszyć atrybuty, jeśli zostały wyznaczone przez skaner. Natomiast atrybuty towarzyszące nieterminalom, czyli węzłom wewnętrznym, są obliczane w trakcie analizy leksykalnej.

Atrybuty nieterminali obliczamy na podstawie atrybutów ich synów w drzewie wyprowadzenia. (Atrybuty takie są nazywane *syntezowanymi* — stąd „S" w nazwie tego rodzaju gramatyk.) Dla każdej produkcji podajemy w specyfikacji fragment kodu, który oblicza atrybut ojca na podstawie atrybutów synów. W kontekście produkcji postaci $X \to B_1 B_2 \ldots B_k$, atrybut X-a oznaczamy przez \$\$, a atrybut B_i przez $\$i$.

Parser wywołuje te fragmenty kodu obliczając atrybuty w odpowiedniej kolejności, od liści do korzenia (ang. *bottom-up*). Wynikiem całości obliczeń jest atrybut korzenia drzewa wyprowadzenia.

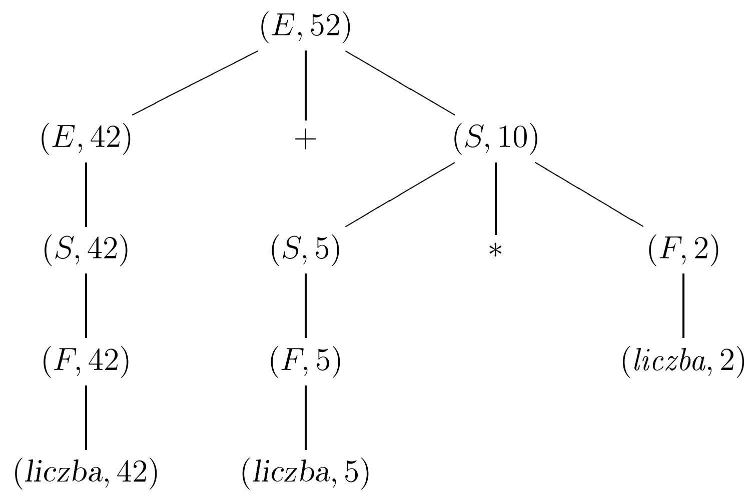
Przykład: Przypomnijmy sobie jednoznaczną gramatykę wyrażeń arytmetycznych:

$$\begin{array}{lcl} E & \to & E+S \mid E-S \mid S \\ S & \to & S*F \mid S/F \mid F \\ F & \to & liczba \mid (E) \end{array}$$

W tym przypadku liczba jest dla nas terminalem, gdyż jest to leksem rozpoznawany przez skaner. Oto gramatyka S-atrybutywna obliczająca wartość wyrażenia:

$$\begin{array}{lcll} E & \to & E+S & \$\$ = \$1 + \$3 \\ E & \to & E-S & \$\$ = \$1 - \$3 \\ E & \to & S & \$\$ = \$1 \\ S & \to & S*F & \$\$ = \$1 * \$3 \\ S & \to & S/F & \$\$ = \$1/\$3 \\ S & \to & F & \$\$ = \$1 \\ F & \to & liczba & \$\$ = \$1 \\ F & \to & (E) & \$\$ = \$2 \end{array}$$

A oto drzewo wyprowadzenia dla wyrażenia $42 + 5 * 2$ wzbogacone o obliczone atrybuty



Tak więc obliczona wartość całego wyrażenia to 52.

13.3 Sposób działania parsera

Bison został zaprojektowany tak, aby współpracował razem z Flex'em. Kod wygenerowany przez Bison'a na podstawie specyfikacji parsera może być kompilowany razem z kodem wygenerowanym przez Flex'a na podstawie specyfikacji skanera. W szczególności żetony są definiowane w specyfikacji parsera, a ich definicje w języku programowania są umieszczane w kodzie parsera. Natomiast w kodzie skanera można z nich korzystać.

Parser wywołuje procedurę `yylex` żeby wczytywać kolejne leksemy. Wynikiem tej procedury powinien być żeton odpowiadający wczytanemu leksemowi. Zakłada się, że atrybut żetonu, jeśli jest obecny, to jest przypisany zmiennej `yylval`. Procedura `yylex` może również zwrócić znak ASCII — wówczas leksemem jest pojedynczy wczytany znak.

Parser udostępnia funkcję `int yyparse()`. Wywołanie tej funkcji powoduje próbę wczytania słowa należącego do języka opisywanego przez gramatykę podaną w specyfikacji. Parser nie konstruuje drzewa wyprowadzenia jako struktury danych, ale wykonuje takie czynności, jakby obchodził je w porządku postfiksowym. Dla każdego odwiedzanego nieterminala w drzewie wyprowadzenia wykonywany jest fragment kodu odpowiadający zastosowanej dla niego produkcji. Jako ostatni wykonywany jest fragment kodu dla korzenia. Jeżeli w tym fragmencie kodu nie nastąpi powrót z procedury (`return`), to wejście ponownie jest parsowane. W przypadku wystąpienia błędu składniowego wywoływana jest procedura `void yyerror (char const *s)`. Procedura ta musi być zdefiniowana.

Fragmenty kodu umieszczone w specyfikacji nie muszą tylko obliczać atrybuty. Mogą wykonywać dowolne czynności, np. wypisywać informacje na wyjście czy modyfikować globalne struktury danych. Trzeba jednak zdawać sobie sprawę, w jakiej kolejności różne fragmenty kodu są wykonywane.

13.4 Specyfikacje parserów dla Bison'a

Specyfikacja dla Bison'a składa się z trzech części, oddzielonych od siebie wierszami postaci „%%":

- Pierwsza część zawiera deklaracje programistyczne (stałych i zmiennych) oraz deklaracje żetonów (postaci `%token ŻETON`).
- Druga część zawiera gramatykę S-atrybutywną. Gramatyka ta jest zapisana w następujący sposób:

```
nieterminal  :  prawa strona produkcji  {fragment kodu}
             |  prawa strona produkcji  {fragment kodu}
                          ⋮
             |  prawa strona produkcji  {fragment kodu}
             ;
```

  Czyli produkcje dla jednego nieterminala są zgrupowane razem, a dla każdej z tych produkcji mamy oddzielny fragment kodu ujęty w wąsate nawiasy. Fragmenty kodu są opcjonalne. Jeżeli dany fragment kodu jest pominięty, to tak jakby zostało podane: `{ $$ = $1 }`. Aksjomatem jest ten nieterminal, dla którego produkcje są podane jako pierwsze.

- Trzecia część to definicje procedur pomocniczych. W szczególności można tu umieścić:
  - definicję procedury `yylex`, jeżeli nie korzystamy ze skanera wygenerowanego przez Flex'a,
  - definicję procedury `main`,
  - powinna tu zostać zdefiniowana procedura `yerror`, która jest wywoływana przez parser w przypadku wystąpienia błędów składniowych.

Przykład: Zbudujemy prosty kalkulator wzbogacony o zmienne. Będzie on przyjmował dwa rodzaje poleceń:

```
SET identyfikator = wyrażenie
PRINT wyrażenie
```

Polecenie `SET` powoduje przypisanie wartości wyrażenia na zmienną. Polecenie `PRINT` powoduje wypisanie wartości wyrażenia. Każde polecenie musi znajdować się w osobnym wierszu. Wyrażenia mogą zawierać nazwy zmiennych. Nazwy zmiennych mają postać $[A-Za-z][A-Za-z0-9]^*$. Poza tym, składnia wyrażeń jest taka jak w poprzednich przykładach. Początkowo wszystkie zmienne mają wartość 0. W danych może pojawić się co najwyżej 1000 różnych nazw zmiennych.

Skaner odróżnia słowa kluczowe `SET` i `PRINT`, liczby i identyfikatory. Wszystkie inne znaki są przekazywane dosłownie. Kod źródłowy specyfikacji skanera znajduje się w pliku `calc.l`.

```
%{
// Definicje żetonów pochodzące z Bison'a
#include "skalk.tab.h"

// Tablica zmiennych
char *dict[1000];
int dict_size=0;

// Odnajduje nazwę zmiennej w tablicy, lub ją dodaje
int dict_find(const char *key) {
    int i;
    for(i=0; i<dict_size; i++) if (strcmp(key,dict[i])==0) return i;

    i=dict_size; dict_size++;
    dict[i]=(char *)malloc(strlen(key)+1);
    strncpy(dict[i],key,strlen(key)+1);
    return i;
}
%}

INT     [0-9]+
ID      [A-Za-z][A-Za-z0-9]*

%%

PRINT { return CMD_PRINT; }
SET   { return CMD_SET;   }
{ID}  {
        yylval=dict_find(yytext);
        return ID;
      }
{INT} {
        yylval=atoi(yytext);
        return NUM;
      }
[ \t] ;
[\n]|. { return yytext[0]; }

%%
```

Parser pamięta w tablicy wartości zmiennych, oblicza wartości wyrażeń i przypisuje zmiennym wartości. Kod źródłowy specyfikacji parsera znajduje się w pliku `calc.y`.

```
%{
#include <stdio.h>

// Tablica wartości zmiennych
int d_value[1000];

// Odszukanie wartości zmiennej
int dict_value(int key) {
    return d_value[key];
}

// Przypisanie wartości zmiennej
int dict_set(int key,int value) {
    d_value[key]=value;
}

%}

%token NUM CMD_PRINT CMD_SET ID

%%

input: /* nic */
    | input line
    ;

line: '\n'
    | CMD_SET ID '=' exp '\n' { dict_set($2,$4); }
    | CMD_PRINT exp '\n' { printf("%d\n",$2); }
    ;

exp : exp '+' skl  { $$ = $1 + $3 }
    | exp '-' skl  { $$ = $1 - $3 }
    | skl
    ;

skl : skl '*' czy  { $$ = $1 * $3 }
    | skl '/' czy  { $$ = $1 / $3 }
    | czy
    ;
```

```
czy : NUM
    | ID           { $$ = dict_value($1); }
    | '(' exp ')'  { $$ = $2 }
    ;

%%

main()
{
  // Wyzerowanie wartości zmiennych
  bzero(&d_value,sizeof(d_value));

  yyparse();
}

yyerror(s)
char *s;
{
    fprintf(stderr, "%s\n", s);
}
```

Przykład: [[Przykład pokazujący przekazywanie stringów jako atrybutów]]

[[Opisać i dodać przykład z atrybutami różnych typów `%union`, `%token` z typem i `%type`.]]]
[[[JavaCup i JFlex]]]

13.5 Kompilacja programów generowanych przez Flex'a i Bison'a

Zodnie z przyjętą konwencją, specyfikacje dla Flex'a to pliki z rozszerzeniem `.l`, a specyfikacje dla Bison'a to pliki z rozszerzeniem `.y`. Załóżmy, że takie specyfikacje mamy w plikach `spec.l` i `spec.y`. Oba narzędzia, Flex i Bison, są tak zaprojektowane, żeby ze sobą współpracowały. Tak więc kod wygenerowany przez Bison'a oczekuje dokładnie takiej funkcjonalności, jakiej dostarcza kod wygenerowany przez Lex'a.

Sposób kompilowania skanerów generowanych przez Flex'a został opisany w p. kompilacja-flex. Jest jednak jeden element, zdefiniowany w kodzie wygenerowanym przez Bison'a, z którego korzysta kod wygenerowany przez Lex'a. Są to definicje żetonów. Problem ten rozwiązuje się w ten sposób, że Bison może również wygenerować plik nagłówkowy zawierający definicje stałych reprezentujących żetony, a w kodzie skanera powinniśmy dołączyć go (dyrektywą `#include`). Tak więc najpierw powinniśmy wygenerować kod parsera (wraz z plikiem nagłówkowym zawierającym definicje żetonów), potem wygenerować kod skanera, a na koniec skompilować cały program.

Najprostsze wywołanie Bisona'a wygląda tak:

```
bison spec.y
```

W jego wyniku Bison tworzy plik z kodem źródłowym parsera, o takiej samej nazwie jak specyfikacja, ale z rozszerzeniem `.tab.c`, czyli w naszym przypadku `spec.tab.c`. Jeśli chcemy, żeby Bison dodatkowo wygenerował plik nagłówkowy z definicjami żetonów, to musimy podać opcję `-d`:

```
bison -d spec.y
```

Plik nagłówkowy ma taką samą nazwę jak specyfikacja, ale z rozszerzeniem `.tab.h`, czyli w naszym przypadku `spec.tab.h`.

Kompilując program powinniśmy skompilować razem wygenerowany kod skanera i parsera, wraz z ew. innymi plikami. Pamiętajmy przy tym o dołączeniu biblioteki `fl` wykorzystywanej przez skaner:

```
gcc ... spec.yy.c spec.tab.c -lfl.
```

W przypadku programów (w C), których cały kod składa się tylko z kodu skanera i ew. parsera można skorzystać z prostego pliku `Makefile`. Wystarczy wówczas wydać tylko polecenie `make`, a ono już się samo zajmie wygenerowaniem skanera, parsera i kompilacją programu.

13.6 Podsumowanie

W tym wykładzie poznaliśmy generator analizatorów składniowych Bison oraz związane z nim pojęcie gramatyk S-atrybutywnych. Umiejętność korzystania z Bison'a będzie jeszcze przedmiotem ćwiczeń laboratoryjnych.

13.7 Skorowidz

- **Analiza składniowa** polega na sprawdzeniu poprawności składniowej i odtworzeniu (obejścia) drzewa wyprowadzenia dla wczytywanych danych.
- **Bison** to jeden z generatorów analizatorów składniowych.
- **Generator analizatorów składniowych** to program generujący, na podstawie specyfikacji, kod analizatora składniowego (parsera).
- **Gramatyki S-atrybutywne** to gramatyki atrybutywne, w których wszystkie atrybuty są syntezowane, tzn. węzła w drzewie wyprowadzenia jest obliczany na podstawie atrybutów jego synów.
- **Parser** to inaczej analizator składniowy, moduł zajmujący się analizą składniową.

13.8 Praca domowa

Odwrotna notacja polska (ONP) to sposób zapisu wyrażeń, w którym najpierw podajemy argumenty, a potem operację. Jeżeli wiadomo ile argumentów mają operacje (a w przypadku operacji arytmetycznych tak jest — mają one po dwa argumenty), to w ONP nie są potrzebne nawiasy. Na przykład, wyrażenie $2*(3+5)$ zapisane w ONP ma postać $2\ 3\ 5\ +\ *$. Nota bene, wyrażenie zapisane w ONP to gotowy program dla maszyny stosowej, patrz Ćwiczenia.

Napisz specyfikację (dla Lex'a i Bison'a) analizatora, który wczyta wyrażenie arytmetyczne i wypisze je w ONP.

13.9 Ćwiczenia

1. Napisz specyfikację (dla Lex'a i Bison'a) analizatora, który wczyta wyrażenie arytmetyczne i wyznaczy wielkość stosu potrzebnego do obliczenia wyrażenia przez maszynę stosową.

   Maszyna stosowa działa na następującej zasadzie: włóż liczbę na stos, zdejmij argumenty operacji arytmetycznej z wierzchołka stosu i włóż jej wynik na stos. Na przykład, obliczenie wyrażenia $2 + 3 * 5$ wymaga stosu wielkości 3. Obliczenie to przebiega następująco: włóż 2 na stos, włóż 3 na stos, włóż 5 na stos, wykonaj mnożenie, wykonaj dodawanie. Obliczenie wyrażenia $3 * 5 + 2$ wymaga stosu wielkości 2. Obliczenie to przebiega następująco: włóż 3 na stos, włóż 5 na stos, wykonaj mnożenie, włóż 2 na stos, wykonaj dodawanie.

2. Napisz specyfikację (dla Lex'a i Bison'a) analizatora, który wczyta wyrażenie arytmetyczne i wyznaczy maksymalną głębokość wyrażenia. Na przykład dla $3+3*(5-1)$ wynikiem powinno być 3, a dla $(2-1)+(1-2)$ wynikiem powinno być 2.

3. Rozszerz przykład kalkulatora z wykładu o możliwość definiowania nazwanych wyrażeń. Wartość takiego wyrażenia zależy od wartości zmiennych i jest obliczana za każdym razem, gdy takie wyrażenie jest używane.

4. Rozważamy następujący rodzaj kompresji słów nad alfabetem $\{0,1\}$. Mając dane słowo s (o długości będącej potęgą dwójki, $|s| = 2^i$), jeśli:

   - s składa się z samych znaków 0, $s = 0^{2^i}$, to jego kodem jest 0,
   - s składa się z samych znaków 1, $s = 1^{2^i}$, to jego kodem jest 1,
   - w przeciwnym przypadku napis s jest dzielony na równe części s_1 i s_2 długości 2^{i-1}, $s = s_1 s_2$, a jego kod jest postaci: $9kod(s_1)kod(s_2)$.

   Na przykład:

   - $kod(1) = 1$,
   - $kod(0111) = 99011$,

- $kod(01001111) = 9990101$,
- $kod(00110101) = 99019901901$.

Napisz specyfikacje dla Lex'a i Bison'a dające program, który: wczyta zestaw zakodowanych słów, po jednym w wierszu i dla każdego kodu wypisze jego odkodowaną postać.

Wykład 14. Maszyny Turinga i obliczalność

14.1 Wstęp

W tym wykładzie zajmiemy się pojęciem obliczalności. Poznamy maszyny Turinga, model teoretyczny opisują pojęcie obliczalności, równy pod względem siły wyrazu wszelkiego rodzaju językom programowania i komputerom.

Pojęcie obliczalności było badane zanim narodziła się informatyka, a nawet zanim powstały pierwsze komputery, bo było to już w latach 30-tych ubiegłego wieku. W ramach badań nad formalizacją podstaw matematyki, badano jakie zbiory formuł można uzyskać na drodze mechanicznego przepisywania napisów zgodnego z określonymi zasadami. Prowadziło to do sformalizowania intuicji dotyczącej tego co można „automatycznie obliczyć”. Pojawiło się wiele modeli formalizujących takie intuicyjne pojęcie obliczalności. W śród nich można wymienić maszyny Turinga, rachunek λ, czy systemy Posta. W tym wykładzie zajmiemy się maszynami Turinga, gdyż spośród tych modeli są one najbliższe komputerom.

Jak się okazało, wszystkie te formalizmy są sobie równoważne — w każdym z nich można symulować każdy z pozostałych formalizmów. Zostało to ujęte w tzw. hipotezie Church'a, mówiącej, że wszystkie te modele opisują to samo intuicyjne pojęcie *obliczalności*. Oczywiście nie jest to formalne twierdzenie, które można by udowodnić, gdyż dotyczy ona intuicji, ale jak dotąd nic nie podważyło słuszności tej hipotezy.

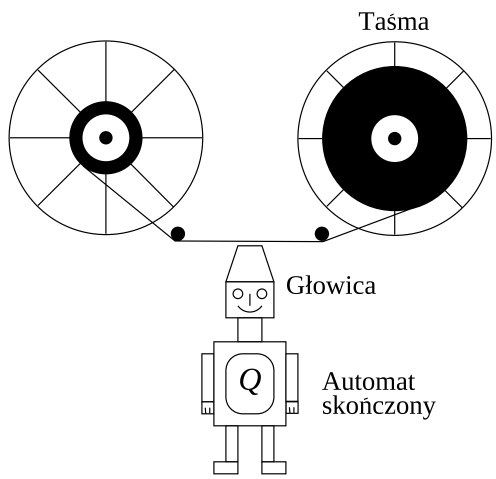
14.2 Uniwersalność

Formalizmy modelujące pojęcie obliczalności nie tylko są sobie nawzajem równoważne, ale też w każdym z nich można stworzyć model *uniwersalny* — taki, który potrafi symulować wszystkie inne modele danego rodzaju (na podstawie ich opisu). W szczególności istnieje uniwersalna maszyna Turinga, która potrafi symulować działanie dowolnej innej maszyny Turinga na podstawie jej opisu.

W pierwszym momencie, pojęcie uniwersalności może być niezrozumiałe lub zadziwiające. Natomiast dzisiaj jest ono prawie oczywiste. Zamiast mówić o maszynach Turinga wybierzmy dowolny język programowania. Uniwersalny program to nic innego jak interpreter danego języka programowania napisany w nim samym. (Jeśli czytelnik zna język Scheme, to zapewne zetknął się z interpreterem Scheme'a napisanym w Scheme'ie.) Interpretery może nie są tak popularne jak kompilatory. Natomiast np. kompilator C++ napisany w C++ jest czymś naturalnym, a w pewnym sensie jest to uniwersalny program w C++. Za jego pomocą można uruchomić każdy inny program w C++.

14.3 Maszyny Turinga

Maszyna Turinga przypomina skrzyżowanie automatu skończonego z magnetofonem szpulowym. Tak jak automat stosowy to automat skończony wyposażony w dodatkową pamięć w postaci stosu, tak maszyna Turinga to automat skończony wyposażony w dodatkową pamięć w postaci nieskończonej taśmy, na której można zapisywać i odczytywać informacje.



Taśma ta jest podzielona na klatki. W każdej klatce można zapisać jeden symbol z ustalonego alfabetu. Nad taśmą przesuwa się głowica, którą steruje automat skończony. W jednym kroku automat odczytuje znak zapisany pod głowicą, może w tym miejscu zapisać inny znak, po czym może przesunąć głowicę w lewo lub prawo. Oczywiście automat sterujący głowicą w każdym kroku zmienia również swój stan.

Taśma ma początek, lecz nie ma końca — jest nieskończona. W pierwszej klatce taśmy jest zapisany specjalny znak, tzw. *lewy ogranicznik*. Jeżeli głowica znajduje się nad lewym ogranicznikiem, to nie może go zamazać ani przesunąć się na lewo od niego. Na początku, zaraz za lewym ogranicznikiem, zapisane jest słowo, które stanowi dane wejściowe dla maszyny Turinga. Oczywiście słowo to jest skończone. Za tym słowem taśma wypełniona jest w nieskończoność specjalnymi pustymi symbolami, tzw. *balnk'ami*. Blank'i będziemy oznaczać przez $\sqcup$.

Będziemy tutaj rozważać jedynie deterministyczne maszyny Turinga, tzn. takie, w których automat sterujący głowicą jest deterministyczny. W momencie uruchomienia maszyny Turinga, głowica znajduje się nad pierwszą klatką taśmy (nad lewym ogranicznikiem) i jest w stanie początkowym. Maszyna ma dwa wyróżnione stany: akceptujący i odrzucający. Działa ona tak długo, aż znajdzie się w jednym z tych dwóch stanów. Jeśli jest to stan akceptujący, to słowo początkowo zapisane na taśmie zostało zaakceptowane, a w przeciwnym przypadku zostało odrzucone. Oczywiście jest możliwe, że maszyna Turinga nigdy nie znajdzie się w żadnym z tych stanów. Wówczas jej obliczenie trwa w nieskończoność i mówimy, że się *zapętliło*. Tak więc dla danego słowa maszyna Turinga może zrobić jedną z trzech rzeczy: zaakceptować to słowo, odrzucić je lub zapętlić się.

Nieznacznie zmieniając opisany model maszyny Turinga możemy go użyć do modelowania dowolnych obliczeń, a nie tylko akceptowania języków, czy problemów decyzyjnych. W tym celu wystarczy, że zamiast dwóch stanów: akceptującego i odrzucającego, mamy jeden stan końcowy. Wówczas to co jest zapisane na taśmie (z wyjątkiem lewego ogranicznika i blank'ów) w momencie gdy maszyna osiągnie stan końcowy, stanowi wynik obliczeń.

Def. 46. Maszyna Turinga to dowolna taka dziewiątka $M = \langle Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r \rangle$, że:

- Q, to skończony zbiór stanów,

- $\Sigma \subseteq \Gamma$, to alfabet wejściowy,
- Γ, to alfabet taśmowy,
- $\vdash \in \Gamma$, to lewy ogranicznik,
- $\sqcup \in \Gamma$, to symbol pusty, blank,
- $\delta : (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{L, P\}$, to funkcja przejścia; wymagamy, aby dla dowolnego $q \in Q$ istniał taki $q' \in Q$, że $\delta(q, \vdash) = (q', \vdash, P)$, czyli żeby $\vdash$ nie był zamazywany, a głowica nie mogła przesunąć się na lewo od niego
- $s \in Q$, to stan początkowy,
- $t \in Q$, to stan akceptujący,
- $r \in Q, r \neq t$, to stan odrzucający.

$\Box$

Def. 47. Niech $M = \langle Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r \rangle$ będzie ustaloną maszyną Turinga. *Konfiguracja* maszyny M, to dowolna taka trójka:

$$\langle q, \vdash \alpha, k \rangle \in Q \times \Gamma^* \times \mathcal{N}$$

że $|\alpha| \geq k$. W konfiguracji $\langle q, \alpha, k \rangle$ q to stan maszyny, $\vdash \alpha$ reprezentuje zawartość taśmy (uzupełnioną w nieskończoność blank'ami), a k to pozycja głowicy.

Konfiguracja początkowa maszyny M dla słowa $x \in \Sigma^*$ to $\langle s, \vdash x, 0 \rangle$. $\Box$

Def. 48. Niech $M = \langle Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r \rangle$ będzie daną maszyną Turinga. Na konfiguracjach maszyny Turinga definiujemy relację $\to \subseteq (Q \times \Gamma^* \times \mathcal{N})^2$. Relacja ta opisuje przejścia między konfiguracjami odpowiadające pojedynczym krokom w obliczeniach M. Jest to najmniejsza relacja, która spełnia podane poniżej warunki.

Niech $\langle q, \vdash \alpha, k \rangle$ będzie konfiguracją i niech $q \neq t$, $q \neq r$, $\vdash \alpha = a_0 a_1 a_2 \dots a_n$, $n \geq k$. Niech $\delta(q, a_k) = \langle () q', a', x)$. Jeśli $x = L$ (czyli głowica przesuwa się w lewo), to mamy:

$$\langle q, \vdash \alpha, k \rangle \to \langle q', a_0 a_1 \dots a_{k-1} a' a_{k+1} \dots a_n, k - 1 \rangle$$

Jeśli $x = P$ i $k < n$ (czyli głowica przesuwa się w prawo, ale w obrębie $\vdash \alpha$), to mamy:

$$\langle q, \vdash \alpha, k \rangle \to \langle q', a_0 a_1 \dots a_{k-1} a' a_{k+1} \dots a_n, k + 1 \rangle$$

Jeśli zaś $x = P$ i $k = n$ (czyli głowica jest na prawym końcu $\vdash \alpha$ i przesuwa się w prawo), to mamy:

$$\langle q, \vdash \alpha, k \rangle \to \langle q', a_0 a_1 \dots a_{k-1} a' a_{k+1} \dots a_n \sqcup, k + 1 \rangle$$

Przez $\to^*$ oznaczamy domknięcie zwrotno-przechodnie relacji $\to$. $\Box$

Relacja $\to^*$ opisuje do jakich konfiguracji możemy dojść w wyniku obliczenia. Konfiguracje akceptujące, to te, które zawierają stan t, a odrzucające to te, które zawierają stan r.

Def. 49. Niech $M = \langle Q, \Sigma, \Gamma, \vdash, \sqcup, \delta, s, t, r\rangle$ będzie daną maszyną Turinga. *Język akceptowany* przez M składa się z tych słów, dla których obliczenie maszyny prowadzi do konfiguracji akceptującej:

$$L(M) = \{x \in \Sigma^* : \exists_{\alpha\in\Gamma^*, k\in\mathbb{N}} \langle s, \vdash x, 0\rangle \to^* \langle t, \vdash \alpha, k\rangle\}$$

□

14.4 Języki częściowo obliczalne i obliczalne

Def. 50. Powiemy, że język $A \subseteq \Sigma^*$ jest *częściowo obliczalny*, jeżeli istnieje taka maszyna Turinga M, że $A = L(M)$. □

Wobec języków częściowo obliczalnych używa się też określeń częściowo rozstrzygalny i rekurencyjnie przeliczalny. Wszystkie te określenia oznaczają to samo pojęcie.

Zauważmy, że żeby maszyna Turinga M akceptowała język A, dla wszystkich słów z tego języka musi zatrzymywać się w stanie akceptującym, natomiast dla słów spoza języka A nie musi zatrzymywać się w stanie odrzucającym — może również się zapętlać. Intuicyjnie odpowiada to istnieniu programu komputerowego, który dla słów z języka A potrafi potwierdzić ich przynależność do języka, natomiast dla słów spoza tego języka wcale nie musi potrafić zaprzeczyć ich przynależności do języka, lecz może się zapętlić. Stąd słowo „częściowo" w nazwie.

Istnieje równoważna definicja języków częściowo obliczalnych, która mówi, że język A jest częściowo obliczalny, gdy istnieje taki program komputerowy, który wypisuje (w pewnej kolejności) wszystkie słowa należące do A. Jeżeli A jest nieskończony, to oznacza to, że każde słowo należące do A kiedyś zostanie wypisane.

Def. 51. Powiemy, że język $A \subseteq \Sigma^*$ jest *obliczalny*, jeżeli istnieje taka maszyna Turinga M, że $A = L(M)$ i dla każdego $x \in \Sigma^*$ maszyna M albo akceptuję x, albo go odrzuca. □

Fakt 17. *Każdy język bezkontekstowy jest obliczalny.*

Fakt ten wynika stąd, że algorytm CYK (dla danej gramatyki bezkontekstowej) można zaimplementować w postaci maszyny Turinga. Ponieważ algorytm ten zawsze się zatrzymuje, więc i taka maszyna nie będzie się zapętlać.

Przykład: Pokazaliśmy poprzednio, że język $\{a^n b^n c^n : n \geq 0\}$ nie jest bezkontekstowy. Zapewne dla większości Czytelników napisanie programu, który wczytuje napis i sprawdza czy jest on postaci $a^n b^n c^n$ nie sprawiłoby kłopotu. Pokażemy, jak skonstruować maszynę

Turinga, która to robi. Nie podamy jej formalnej definicji (gdyż byłoby to nudne), ale opiszemy jej sposób działania. Oto przykładowa początkowa konfiguracja:

$\vdash$	a	a	a	b	b	b	c	c	c	$\sqcup$	$\sqcup$	$\ldots$
$\uparrow$												

1. Nasza maszyna najpierw czyta słowo podane na wejściu i sprawdza czy pasuje ono do wzorca $a^*b^*c^*$, przesuwając głowicę w prawo, aż do napotkania pierwszego blanka. Jeżeli słowo na wejściu nie pasuje do wzorca, to odrzucamy je. Dodatkowo, jeżeli na wejściu jest dane słowo puste, to od razu je akceptujemy.

$\vdash$	a	a	a	b	b	b	c	c	c	$\sqcup$	$\sqcup$	$\ldots$
										$\uparrow$		

2. W miejsce pierwszego blank'a wstawiamy prawy ogranicznik.

$\vdash$	a	a	a	b	b	b	c	c	c	$\dashv$	$\sqcup$	$\ldots$
										$\uparrow$		

3. Następnie przejeżdżamy głowicą w lewo zamieniając pierwszy napotkany znak a, b i c na blanki. Jeżeli któregoś ze znaków zabrakło, to odrzucamy. Oznacza to, że znaków a, b i c nie było po równo.

$\vdash$	a	a	$\sqcup$	b	b	$\sqcup$	c	c	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
$\uparrow$												

4. Przesuwamy głowicę w prawo, aż do prawego ogranicznika, sprawdzając czy między ogranicznikami są jakieś znaki inne niż blanki.

$\vdash$	a	a	$\sqcup$	b	b	$\sqcup$	c	c	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
										$\uparrow$		

W takim przypadku skaczemy do kroku 3 i powtarzamy dwa ostatnie kroki tak długo, aż między ogranicznikami pozostaną same blank'i.

$\vdash$	a	$\sqcup$	$\sqcup$	b	$\sqcup$	$\sqcup$	c	$\sqcup$	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
$\uparrow$												

$\vdash$	a	$\sqcup$	$\sqcup$	b	$\sqcup$	$\sqcup$	c	$\sqcup$	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
										$\uparrow$		

$\vdash$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
$\uparrow$												

5. Gdy w końcu między ogranicznikami zostaną same blank'i, akceptujemy.

$\vdash$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\sqcup$	$\dashv$	$\sqcup$	$\ldots$
										$\uparrow$		

[[Prezentacja multimedialna: animacja powyższej maszyny.]]

Pokazaliśmy właśnie, że język $\{a^n b^n c^n : n \geq 0\}$ jest obliczalny, choć wiemy, że nie jest bezkontekstowy. Wynika stąd następujący fakt:

Fakt 18. *Klasa języków obliczalnych zawiera ściśle klasę języków bezkontekstowych.*

Czy każdy język jest częściowo obliczalny? Zdecydowanie nie! Wynika to stąd, że (dla ustalonego alfabetu wejściowego Σ) maszyn Turinga (z dokładnością do izomorfizmu) jest przeliczalnie wiele. Być może bardziej intuicyjne jest równoważne stwierdzenie, że w dowolnym języku programowania istnieje przeliczalnie wiele programów. Wszak każdy program to tylko słowo, a słów (nad ustalonym alfabetem Σ) jest przeliczalnie wiele. Natomiast języki to zbiory słów, czyli wszystkich możliwych języków (nad ustalonym alfabetem Σ) jest $2^{|\mathbb{N}|} > |\mathbb{N}|$. Z drugiej strony, każda maszyna Turinga czy program komputerowy akceptuje jeden określony język. Tak więc maszyn Turinga czy programów komputerowych jest **za mało**, żeby każdy język był częściowo obliczalny.

Powyższe rozumowanie jest poprawne, ale niekonstruktywne — nie dostarcza nam żadnego przykładu języka, który nie byłby częściowo obliczalny czy obliczalny. Dalej poznamy przykład takiego języka.

14.5 Wariacje na temat maszyn Turinga

W literaturze można spotkać różne warianty maszyn Turinga: maszyny z taśmą nieskończoną w obydwie strony, maszyny w taśmą wielościeżkową, maszyny z wieloma taśmami i głowicami, czy maszyny z dwuwymiarową płaszczyzną zamiast taśmy. Dalej w naszych rozważaniach czasem będzie nam wygodnie założyć, że maszyny Turinga posiadają któreś z tych rozszerzeń. Wszystkie te modle są równoważne przedstawionym tutaj maszynom Turinga. Nie będziemy tego formalnie dowodzić, ale krótko naszkicujemy odpowiednie konstrukcje.

Powiedzmy, że chcielibyśmy mieć maszynę wyposażoną w k ścieżek, przy czym na i-tej ścieżce są zapisane znaki z Γ_i. Głowica będąc w jednym położeniu może odczytać i zapisać równocześnie znaki na wszystkich ścieżkach. Tak naprawdę jest to równoważne klasycznej maszynie Turinga z alfabetem ścieżkowym $\Gamma_1 \times \Gamma_1 \times \cdots \times \Gamma_k$.

Jeżeli chcemy, aby taśma była nieskończona w obie strony, to wystarczy, że złożymy ją (w miejscu rozpoczęcia wejścia) na pół i skleimy w taśmę dwuścieżkową. Taka maszyna będzie w każdym kroku uwzględniać i modyfikować tylko jedną z dwóch ścieżek. Dodatkowo więc automat sterujący głowicą musi pamiętać nad którą ścieżką znajduje się głowica.

Jeśli chcielibyśmy mieć wiele taśm i głowic, to symulujemy to na maszynie z jedną głowicą i wieloma ścieżkami. Potrzebujemy taśmy o dwukrotnie większej liczbie ścieżek.

Połowa z nich będzie nam służyć do przechowywania zawartości taśmy symulowanej maszyny, a połowa będzie przeznaczona na znaczniki wskazujące położenia odpowiednich głowic. Oczywiście symulacja jednego kroku takiej maszyny może wymagać przejrzenia większego fragmentu taśmy i trwać odpowiednio długo. Niemniej jest to wykonalne.

14.6 Podsumowanie

W tym wykładzie poznaliśmy maszyny Turinga — model obliczeniowy równoważny programom komputerowym. Poznaliśmy też definicje klas języków obliczalnych i częściowo obliczalnych, które zbadamy bliżej w kolejnym wykładzie.

14.7 Skorowidz

- **Częściowa obliczalność** — język jest obliczalny, gdy istnieje maszyna Turinga akceptująca go. Nie musi ona zatrzymywać się dla słów spoza języka.
- **Maszyna Turinga** to teoretyczny model oddający pojęcie obliczalności.
- **Obliczalność** — język jest obliczalny, gdy istnieje taka maszyna Turinga akceptująca go, która zawsze się zatrzymuje.
- **Uniwersalna maszyna Turinga** to maszyna Turinga, która może symulować każdą inną maszynę Turinga na podstawie jej opisu oraz przetwarzanych przez nią danych.

14.8 Praca domowa

1. Opisz maszynę Turinga akceptującą język $\{ww : w \in \{a,b\}^*\}$. Maszyna ta nie może zapętlać się. Zasymuluj jej działanie dla słowa *abbabb*.

Wykład 15. Języki obliczalne, częściowo obliczalne i nieobliczalne

W poprzednim wykładzie poznaliśmy definicje języków obliczalnych i częściowo obliczalnych. Jak dotąd, wszystkie przykłady języków, jakie rozważaliśmy były obliczalne. Poznamy teraz przykłady języków, dla których problem przynależności słów do języka nie jest obliczalny, a nawet nie jest częściowo obliczalny.

Poznamy też hierarchię Chomsky'ego, hierarchię klas języków, która podsumowuje większość poznanych przez nas klas języków i formalizmów.

15.1 Problem stopu

Mówiliśmy wcześniej o uniwersalnej maszynie Turinga. Sprecyzujmy jej postać. Ustalamy alfabet wejściowy $\Sigma = \{0, 1, \$\}$. Uniwersalna maszyna Turinga symuluje działanie danej innej maszyny Turinga na podstawie jej opisu oraz jej wejścia. Zarówno opis symulowanej maszyny, jak i jej wejście maszyna uniwersalna otrzymuje na swoim wejściu w postaci ciągów zer i jedynek. Wymaga to zakodowania opisu symulowanej maszyny Turinga jako (ciągu bitów) słowa nad alfabetem $\{0, 1\}$. Nie opisujemy tutaj takiego kodowania, tylko zakładamy, że takie kodowanie jest ustalone. (Jest to problem czysto techniczny.) Podobnie, symulowana maszyna może operować na słowach nad innym alfabetem niż $\{0, 1\}$. Zakładamy że słowa te są, w miarę potrzeby, niejawnie kodowane — to że znaki dowolnego alfabetu można zakodować jako ciągi bitów odpowiedniej długości jest oczywiste. Uniwersalną maszynę Turinga będziemy oznaczać przez U. Przyjmujemy, że akceptuje ona następujący język:

$$L(U) = \{\text{kod } M\$x : x \in L(M)\}$$

Wynik działania maszyny uniwersalnej jest dokładnie taki sam, jak wynik działania symulowanej maszyny M dla danego słowa x.

Problem stopu to problem decyzyjny polegający na stwierdzeniu, czy dana maszyna Turinga M zatrzyma się dla danego słowa x. Problem ten możemy zdefiniować w formie języka:

$$STOP = \{\text{kod } M\$x : M \text{ zatrzymuje się dla słowa } x\}$$

Fakt 19. *Język STOP jest częściowo obliczalny.*

Dowód: Maszyna Turinga M_{STOP} akceptująca język $STOP$ to prosta przeróbka maszyny uniwersalnej. M_{STOP} działa tak samo jak U, z tą różnicą, że gdy maszyna symulowana odrzuca (a więc zatrzymuje się), to M_{STOP} akceptuje. □

Maszyna M_{STOP} nigdy nie odrzuca, natomiast zapętla się gdy symulowana maszyna się zapętla.

Tw. 6. *Język STOP nie jest obliczalny.*

Dowód: Dowód przebiega nie wprost. Załóżmy, że istnieje maszyna Turinga M', która akceptuje język $L(M') = STOP$ i zawsze się zatrzymuje. Niech maszyna M'' będzie wynikiem przeróbki M': M'' dostaje na wejściu kod maszyny M i działa tak jak M' dla słowa kod M\$kod M, przy czym jeśli M' odrzuca, to M'' akceptuje, a jeśli M' akceptuje, to M'' zapętla się. Tak więc

$$L(M'') = \{\text{kod } M : M \text{ zapętla się dla kodu } M\}$$

Ponieważ M'' nie odrzuca, więc zatrzymuje się dla kodu M, wtw., gdy M nie zatrzymuje się dla własnego kodu. Czy M'' zatrzyma się dla własnego kodu? Jeśli się zatrzymuje i akceptuje, to tym samym powinna się zapętlić. I na odwrót, jeżeli się zapętla, to powinna zaakceptować swój własny kod. Sprzeczność. Tak więc założenie, że język $STOP$ jest obliczalny jest fałszywe. $\square$

15.2 Własności klasy języków obliczalnych i częściowo obliczalnych

Poznaliśmy przykład języka, który nie jest obliczalny. Dalej poznamy przykład języka, który nie jest nawet częściowo obliczalny. Wcześniej jednak poznajmy trochę własności języków obliczalnych i częściowo obliczalnych.

Fakt 20. *Jeśli A i $\overline{A}$ są częściowo obliczalne, to A jest obliczalny.*

Dowód: Powiedzmy, że mamy dwie maszyny Turinga, M_1 i M_2, akceptujące odpowiednio A i $\overline{A}$. Możemy skonstruować maszynę M (z dwoma ścieżkami i głowicami), która będzie symulować równocześnie działanie M_1 i M_2. Maszyna ta patrzy, która z maszyn M_1 i M_2 zaakceptuje słowo i odpowiednio akceptuje lub odrzuca. Ponieważ któraś z maszyn M_1, M_2 musi zaakceptować, więc M nie zapętla się. $\square$

Fakt 21. *Język $\overline{STOP}$ nie jest częściowo obliczalny.*

Dowód: Wiemy, że $STOP$ jest częściowo obliczalny. Gdyby $\overline{STOP}$ był częściowo obliczalny, to $STOP$ byłby obliczalny, a wiemy, że taki nie jest. $\square$

15.3 Gramatyki ogólne (typu 0)

Jak dotąd, dla każdej poznawanej klasy języków mieliśmy dwa równoważne modele — jeden nastawiony na opisywanie składni i jeden na modelowanie obliczeń. Czy istnieje formalizm nastawiony na opisywanie składni równoważny maszynom Turinga? Tak. Są to Gramatyki ogólne (tzw. typu 0). Ich definicja jest analogiczna do definicji gramatyk bezkontekstowych, ale po obu stronach produkcji mogą stać słowa.

Def. 52. *Gramatyka ogólna (typu 0)*, to dowolna taka czwórka $G = \langle N, \Sigma, P, S\rangle$, gdzie:

- N to (skończony) alfabet symboli *nieterminalnych*,
- Σ to (skończony) alfabet symboli *terminalnych*, $\Sigma \cap N = \emptyset$,
- P to skończony zbiór *produkcji*, reguł postaci $\alpha \to \beta$ dla $\alpha, \beta \in (N \cup \Sigma)^*$,
- $S \in N$ to *aksjomat* wyróżniony symbol nieterminalny.

□

Produkcja $\alpha \to \beta$ pozwala w trakcie wyprowadzenia zamienić dowolne podsłowo α na β. Definicje wyprowadzenia i języka generowanego są analogiczne do tych dla języków bezkontekstowych. Jednak siła wyrazu gramatyk ogólnych jest dużo większa.

Tw. 7. *Niech $A \subseteq \Sigma^*$ będzie językiem. Następujące dwa stwierdzenia są równoważne:*

- *$A = L(M)$ dla pewnej maszyny Turinga M, czyli A jest częściowo obliczalny,*
- *$A = L(G)$ dla pewnej gramatyki ogólnej (typu 0).*

Szkic dowodu: Nie będziemy tutaj szczegółowo dowodzić. Dowód przebiega przez pokazanie implikacji w obie strony.

- Mając daną gramatykę ogólną możemy skonstruować maszynę Turinga, która będzie konstruować wszystkie możliwe wyprowadzenia, w kolejności wg. rosnącej ich długości. Jeśli słowo dane na wejściu ma wyprowadzenie, to zostanie ono w końcu skonstruowane i maszyna wówczas zaakceptuje to słowo. Jeżeli nie, a wszystkich możliwych wyprowadzeń jest nieskończenie wiele, maszyna zapętli się. Niemniej będzie ona akceptować język generowany przez daną gramatykę.
- Mając daną maszynę Turinga możemy skonstruować gramatykę, która będzie symulować działanie maszyny wspak. Słowo pojawiające się w wyprowadzeniu będzie reprezentować zawartość taśmy, z wstawionym w odpowiednim miejscu znacznikiem reprezentującym położenie głowicy i stan maszyny. Dla każdego możliwego ruchu maszyny mamy produkcję cofającą go. W tworzonej gramatyce możemy wyprowadzić z aksjomatu wszystkie słowa reprezentujące konfiguracje akceptujące (jest to język regularny), następnie możemy stosować produkcje cofające nas w obliczeniach. Na koniec, jeśli możemy dojść do konfiguracji początkowej maszyny, to mamy produkcje usuwające znaczniki i pozostawiające słowo dane na wejściu.

  Tak więc możemy wyprowadzić dokładnie te słowa, dla których istnieją obliczenia akceptujące je. □

Zauważmy jednak, że język generowany przez gramatykę ogólną może być nierozstrzygalny. Jest tak dlatego, że dla każdego języka częściowo obliczalnego mamy generującą go gramatykę ogólną, ale nie każdy język częściowo obliczalny jest obliczalny.

15.4 Gramatyki kontekstowe

Gramatyki kontekstowe to takie gramatyki ogólne, w których prawe strony produkcji nie są krótsze od lewych.

Def. 53. *Gramatyka kontekstowa*, to dowolna taka czwórka $G = \langle N, \Sigma, P, S\rangle$, gdzie:

- N to (skończony) alfabet symboli *nieterminalnych*,
- Σ to (skończony) alfabet symboli *terminalnych*, $\Sigma \cap N = \emptyset$,
- P to skończony zbiór *produkcji*, reguł postaci $\alpha \to \beta$ dla $\alpha, \beta \in (N \cup \Sigma)^*$, $|\beta| \geq |\alpha|$ i $\alpha \neq \varepsilon$,
- $S \in N$ to *aksjomat* wyróżniony symbol nieterminalny.

□

Def. 54. Powiemy, że język A jest kontekstowy, jeżeli istnieje generująca go gramatyka kontekstowa. □

Klasie języków kontekstowych odpowiadają maszyny Turinga ograniczone liniowo i gramatyki kontekstowe. Maszyny Turinga ograniczone liniowo to takie maszyny, których głowica nie wyjedzie dalej niż (o jeden znak za) słowo dane na wejściu. Inaczej mówiąc, są to maszyny, które dysponują pamięcią proporcjonalną do długości słowa danego na wejściu.

Fakt 22. *Dla każdego języka bezkontekstowego istnieje generująca go (z dokładnością do ε) gramatyka kontekstowa. Inaczej mówiąc, klasa języków kontekstowych zawiera w sobie klasę języków bezkontekstowych.*

Dowód: Zauważmy, że gramatyki bezkontekstowe w postaci normalnej Chomsky'ego są równocześnie gramatykami kontekstowymi. □

Przykład: Język $\{a^n b^n c^n : n \geq 1\}$ jest nie tylko obliczalny, ale i kontekstowy. Oto generująca go gramatyka:

[[Poprawić, żeby nie można było za wcześnie zjeść B.]]

$$
\begin{aligned}
S &\to abc \mid XaAbBC \\
Xa &\to aX \\
Xb &\to bX \\
Xc &\to cX \\
XA &\to aAX \mid aX \\
XB &\to bBX \mid bX \\
XC &\to cYC \mid cc \\
cY &\to Yc \\
BY &\to YB \\
bY &\to Yb \\
AY &\to XA
\end{aligned}
$$

Oto przykładowe wyprowadzenie słowa $aaabbbccc$:

$$
\begin{array}{llllllll}
S & \to & XaAbBC & \to & aXAbBC & \to & aaAXbBC & \to \\
 & \to & aaAbXBC & \to & aaAbbBXC & \to & aaAbbBcYC & \to \\
 & \to & aaAbbBYcC & \to & aaAbbYBcC & \to & aaAbYbBcC & \to \\
 & \to & aaAYbbBcC & \to & aaXAbbBcC & \to & aaaXbbBcC & \to \\
 & \to & aaabXbBcC & \to & aaabbXBcC & \to & aaabbbXcC & \to \\
 & \to & aaabbbcXC & \to & aaabbbccc & & &
\end{array}
$$

Fakt 23. *Języki generowane przez gramatyki kontekstowe są obliczalne.*

Dowód: Dla gramatyki liniowej długość słowa w wyprowadzeniu nie może maleć. Można więc prześledzić wszystkie możliwe do wyprowadzenia słowa o długości nie przekraczającej długości danego na wejściu słowa. $\square$

15.5 Hierarchia Chomsky'ego

Hierarchia Chomsky'ego to hierarchia czterech poznanych przez nas klas języków. Została ona stworzona przez Noama Chomsky'ego w ramach jego badań lingwistycznych. Są to języki: regularne, bezkontekstowe, kontekstowe i częściowo obliczalne. Każdej z tych klas odpowiada jeden rodzaj gramatyk. Odpowiadają im gramatyki: liniowe, bezkontekstowe kontekstowe i ogólne. Równocześnie tym samym klasom odpowiadają cztery różne modele obliczeniowe: automaty skończone, automaty stosowe oraz maszyny Turinga (ograniczone liniowo i dowolne). To, że niezależnie stworzone modele do opisu języków odpowiadają tym samym klasom wskazuje na ich głębsze znaczenie.

Zwyczajowo klasy języków tworzące hierarchię Chomsky'ego są oznaczane liczbami od 0 do 3. Następująca tabelka podsumowuje hierarchię Chomsky'ego oraz pozostałe klasy języków, które poznaliśmy w trakcie tego kursu. Są one podane w kolejności od klasy najszerszej do najwęższej.

Nr	Gramatyki	Maszyny	Języki
0	ogólne	maszyny Turinga	częściowo obliczalne
—	—	maszyny Turinga z własnością stopu	obliczalne
1	kontekstowe	maszyny Turinga ograniczone liniowo	kontekstowe
2	bezkontekstowe	automaty stosowe	bezkontekstowe
3	liniowe oraz wzorce	automaty skończone	regularne

15.6 Podsumowanie

W tym wykładzie poznaliśmy przykłady języków, które nie są, odpowiednio, obliczalne i częściowo obliczalne. Jest to problem stopu i jego dopełnienie. Poznaliśmy też gramatyki ogólne i kontekstowe. Gramatyki ogólne są równoważne maszynom Turinga i opisują dokładnie klasę języków częściowo obliczalnych. Poznane przez nas w trakcie tego kursu gramatyki i klasy języków (liniowe/regularne, bezkontekstowe, kontekstowe i częściowo obliczalne/ogólne) tworzą hierarchię Chomsky'ego. Hierarchia ta podsumowuje większość poznanych przez nas klas języków, z wyjątkiem klasy języków obliczalnych.

15.7 Skorowidz

- **Gramatyki kontekstowe** to takie gramatyki ogólne, w których prawe strony produkcji nie są krótsze od lewych.
- **Gramatyki ogólne (typu 0)** to gramatyki, w których po obu stronach produkcji mogą stać dowolne słowa.
- **Hierarchia Chomsky'ego** to hierarchia czterech klas języków: regularnych, bezkontekstowych, kontekstowych i częściowo obliczalnych, którym odpowiadają cztery rodzaje gramatyk: liniowe, bezkontekstowe, kontekstowe i ogólne. Zobacz także artykuł w Wikipedii.
- **Problem stopu** to język (lub równoważnie problem decyzyjny) złożony z tych kodów maszyn Turinga M i słów x, że M zatrzymuje się dla x. Język ten jest częściowo obliczalny, ale nie obliczalny. Jego dopełnienie nie jest nawet częściowo obliczalne.

15.8 Praca domowa

Napisz program, który wypisuje swój własny kod źródłowy. Uwaga:

- Program ma wypisywać swój kod źródłowy po skompilowaniu, usunięciu pliku źródłowego i uruchomieniu.

- Dopuszczalny jest dowolny, powszechnie dostępny, kompilowany język programowania (np. C, C++, Java, Pascal). Nie może to być jednak język, w którym kod skompilowany zawiera literalnie kod źródłowy.
- Pusty plik nie jest dobrym rozwiązaniem.