# Analiza leksykalna: problem dopasowywania wzorca, budowanie lekserów:

## Wyszukiwanie wzorca

W praktycznych zastosowaniach teorii języków formalnych nie sposób nie wspomnieć o problemie wyszukiwania wzorca. Zajmiemy się najprostszą formą, w jakiej można sformułować ten problem – jego wersją dla tekstów. Jest ona szczególnie ważna w erze rozwoju Internetu, a wydajne implementacje algorytmów rozwiązaujących ten problem wykorzystwane są powszechnie. Codziennie przeszukujemy zawartość sieci, naszego dysku, skrzynki pocztowej czy edytowanego plikut tekstowego za pomocą narzędzi dostarczanych przez serwisy internetowe, systemy operacyjne, klientów pocztowych czy edytory dokuemntów tekstowych.

## Definicja

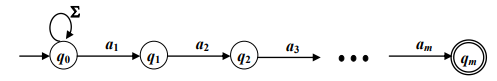
Wzorzec x =a1a2…am oraz tekst y =b1b2…bn to dowlne skończone ciągi znaków nad ustalonym alfabetem Σ. Zazwyczaj zakłada się, że m<=n ( m jest istotnie mniejsze niż n).  
Problem (wyszukiwania wzorca x w tekście y):  
Znajdź wszystkie i, takie że bi...bi+m-1 = a1 … m.

## Przykład:

W tekście nigdy nie ufaj komputerowi, którego nie możesz wyrzucić przez okno (Steve Wozniak) wzorzec nie występuje na pozycjach 1, 7, 37 oraz 78,   
zaś w tekście  
abaababaabaababaababaabaababaabaababaababaabaababaababaabaababaabaababaababaab

Wzorzec aabab występuje na pozycjach 3, 11, 16, 24, 32, 37, 45, 50, 58, 66 oraz 71,  
natomiast wzorzec babaabaa na pozycjach 5, 18, 26, 39, 52 oraz 60.

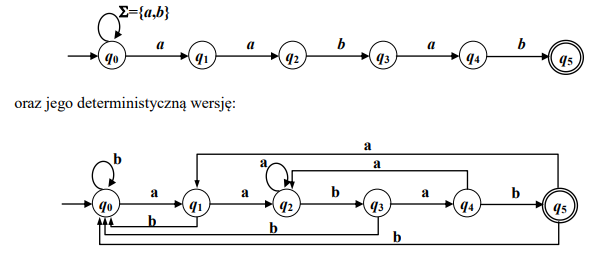
Spróbujmy spojrzeć na rozwiązania problemu wyszukiwania wzorca z punktu widzenia teorii automatów. Dla każdego wzorca x bardzo łatwo skonstruować niedeterministyczny automat skończony, który będzie akceptował wszystkie prefiksy tekstu y, które kończą się wystąpeiniem wzorca x.  
Konstrukcja przebiega jak następuje. Zbiór stanów ma moc m+1, stan początkowy i po jednym stanie dla każdego znaku wzorca, stanem akceptującym jest am. Przejścia między stanami etykietowane są kolejnymi literami wzorca, to jest δ(qi-1,ai)=qi. Ponadto dodajemy przejścia prowadzące ze stanu q0 do stanu q0 oraz ze stanu qm do stanu qm dla każej litery alfabetu. Zauważmy, że bardzo podobne automaty rozważaliśmy w wykładzie 3.



Zaproponowany automat nie jest deterministyczny, przez co nie nadaje się do bezpośredniej imoplementacji jako procedura służąca do wyszukiwania wzorców. Z wcześniejszych wykładów znamy jednak metodę determinizacji automatów. Jej główną wadą jest możliwośc wykładniczego wzrostu liczby stanów. Okazuje się jednak, że w przypadku automatów tego typu, podczas determinizacji liczba stanów nie zmienia się.

### Zadanie” Udowodnij ten fakt przy założeniu, że ai != aj dla i!=j.

**Przykład:** Skonstruujmy automat niedeterministyczny dla wzorca aabab, przyjmując, że alfabet Σ={a,b}.

****Dodatkową zaletą takiego podejścia jest możliwość prostego rozszerzenia go na wyszukiwanie wielu wzorców jednocześnie. Konstruujemy wówczas osobny automat dla każdego wzorca i utożsamiamy ze sobą stany początkowe wszystkich powstałych automatów. W takim przypadku mamy do czynienia z podobną własnością, jak w przypadku wzorca dla pojedynczego wzorca. Tym razem jednak, liczba stanów wynikowego automatu deterministycznego może ulec zmniejszeniu. Z praktycznego punktu widzenia najważniejsze jest, że nie ma możliwości aby uległa zwiększeniu.

### Cwiczenie: Skonstruuj zgodnie z podanym przepisem automat niedeterministyczny rozpoznający jednocześnie wzorce aabab oraz babaabaa oraz zdeterminizuj go.

### Ćwizcenie: Zaproponuj metodę budowania automatu rozwiązującego problem wyszukiwania wzorca x=a1…am, w którym:

-dla stanu początkowego mamy przejścia δ(q0,x)=q0 dla x!=a1, oraz δ(q0, a1)=q1  
-dla pozostałych stanów mamy dokładnie po jednym przejściu postaci δ(qi,x)=qj oraz δ(qi,ε)=qk, gdzie xϵΣ.

Opis jakiego klasycznego algorytmu uzyskałeś w ten sposób?

Wyszukiwanie pojedynczego wzorca, a nawet jednego ze skończonego (i niewielkiego) zbioru wzorców, bywa niewystarczające. Wyobraźmy sobie sytuację, gdy chcielibyśmy wyszukać wszystkie komentarze zawierające słowo int w obszernym kodzie programu napisanego w C. Poszukujemy wówczas napisów postaci /\*y\*/, gdzie w miejsce y jest tekstem zawierającym słowo int. Wiemy już, że język takich napisów łatwo zapisać w postaci wyrażenia regularnego, a do opisanego zadania można wykorzystać opisany wcześniej schemat wyszukiwania wzorca i użyć automatu rozpoznającego ten język regularny. Dokładnie tak działa grep.

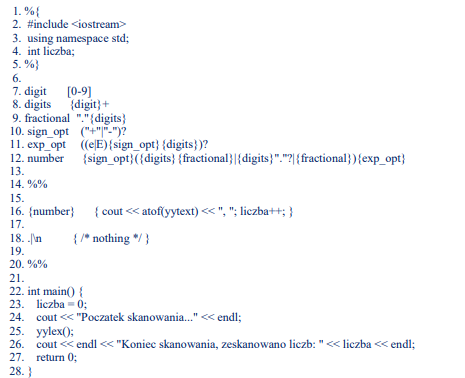
Uwaga: Wyrażenia regularne powłoki UNIX (wykorzystwane między innymi przez program grep) w sposób istotny rozszerzają znane CI wyrażenia regularne. Szczegóły sprawdź w podręczniku systemowym (strony man) dla polecenia grep.

Nie będziemy przytaczać dokładnej składni wyrażeń regularnych, która jest dostępna w systemie UNIX. Podamy tylko prostą translację znanego Ci z jednego z pierwszych wykładów sposobu zapsu wyrażeń regularnych:

* Język pusty Ø – nie podaje się go wprost
* Język jednoelementowy – symbol odpowiedni dla danego znaku,  
  uwaga: znaki specjalne muszą być poprzedzane ‘\’, znaki specjalne to nawiasy, białe znaki czy symbole relacji.  
  przykład: a oznacza język {a}, zaś \. Język {.}
* Nawiasy ustalające kolejność działań – oznaczany dokładnie tak samo, wykorzystywane symbole to `(oraz `)`
* Język złożony ze słowa pustego ε – nie podaje się go wprost, jest natomiast możliwość zapisu języka r⋃ε, służy do tego operator ?  
  przykład: a? oznacza {a}⋃{ε}
* Suma dwóch języków – znakiem sumy jest ‘|`  
  uwaga: sumy języków jednoelementowych można zapisać wymieniając znaki w nawiasach kwadratowych, w przypadku ciągu znaków o kolejnych kodach ASCII możemy korzystając ze znaku `-` użyć zakresu  
  przykład: język {a,b,c} opisany wrażenie regularnym a⋃b⋃c można zapisać a|b|c| lub [abc] lub [a-c]
* Konkatenacja dwóch języków – oznaczamy dokładnie tak samo pisząc 2 języki jeden za drugim.
* Domknięcie Kleene’ego – oznaczane znakiem ‘\*’

Innym, ważnym zadaniem, które można powierzyć narzędziom korzystającym z teorii automatów jest analiza leksykalna. Zależy na wówczas nie tyle na odnalezieniu konkretnych fragmentów w podanym tekście, ile na pocięciu go na fragmenty zgodnie z ustalonymi zasadami. Sytuacja taka ma miejsce w przypadku bazodanowych plików tekstowy, jak /etc/passwd. Dane dotyczące każdego użytkownika systemu zapisane są tam w osobnych wierszach i rozdzielone znakami ‘:’. W rozwiązaniu tego typu prostych problemów analizy leksykalnej możemy się posiłkować innym programem powłoki UNIX- cut, czy też klasą StringTokenizer standardowej biblioteki języka Java.  
Często struktura pliku nie jest tak regularna, a mimo to chcielibyśmy podzielić zawarty w nim tekst na fragmenty o określonej składni. Taką pracę wykonuje każdy kompilator, który musi zrozumieć kod programu, to jest w pierwszym etapie podzielić go na elementarne składniki zwane leksemami. Pozwala to określić, które fragmenty kodu są nazwami zmiennych, które słowami kluczowymi, a który typami danych. W kolejnym etapie kompilator sprawdza, czy tak wydzielone leksemy ułożone są zgodnie z zasadami obowiązującymi w danym języki programowania. Ale tym zajmiemy w czasie jednego z kolejnych wykładów.

Wróćmy do podziału tekstu na leksemy w naszym przykładzie kodu programu. Każdy leksem opisany być może za pomocą wyrażenia regularnego. Słowa kluczowe języka są listą zastrzeżonych ciągów znaków. Nazwy zmiennych to ciągi znaków alfanuerycznych rozpoczynający się literą (zazwyczaj może też zawierać, a nawet zaczynać się znakiem ‘\_’) i nie będących słowami kluczowymi. W podobny sposób opisać można inne elementy z których składają się kody programów napisane w zadanym języku programowania.

Page 4 skip: 

Analiza specyifkacji skanera:

Wierszse 14 i 20 (%%) rozgraniczają logiczne części specyfikacji skanera.

W wierszach 1-5 umieszczone są deklaracje globalne, dołączona jest biblioteka iostream, ustawiona zostaje standardowa przestrzeń nazw oraz zadeklarowana zmienna globalna typu całkowitego.

Wiersze 7-12 zawierająca specyfikację wykorzystywanych w regułach języków regularnych. Zauważ, że wykorzystano tylko wyrażenie zdefiniowane jako number (wiersz 16), pozostałe posłużyły do jego definicji.

W wierszu 18 zdefiniowana (jako puste) zachowanie w przypadku dopasowania pojedynczego znaku, pozwoli to pominąć całą zawartość pliku tekstowego nie pasującą do zdefiniowanego wzorca zapobiegając wydrukowaniu jej na standardowym wyjściu. Zauważ, że oprócz znaku ‘.’ Dopasowującego się do dowolnego symbolu wyrażenie zawiera znak specjalny nowego wiersza. Przypomnij sobie działanie programu grep. Bada on tekst wiersz po wierszu drukująć tylko te wiersze, których fragment pasuje do wzorca. Wyrażenia regularne w systemie UNIX nie dopasowują znaku nowego wiersza jako zwykłego znaku. Do obsługi początku/końca wiersza mamy symbole specjalne \n (symbol nowego wiersza) oraz ^i $, odpowiednio początek i koiec wiersza.

Wiersze 22-28 to implementacja funkcji main. Domyślnie (gdybysmy nie podali własnej implementacji funkcji main) lex wygenerowałby wiersze 22, 25, 27 oraz 28.

Kompilacja programów korzystających ze specyfikacji skanera przebiega dwuetapowo. Najpierw, korzystając z programu lex (lub flex) generujemy kod programu w języku C lub C++. Podajemy jako jedyny argument programu lex nazwę pliku ze specyfikacją skanera. Wygenerowany plik będzie domyślnie nosił nazwę lex.yy.c. Kompilujemy go z użyciem programu gcc lub g++ do pliku wykonywalnego z opcją -lfl, domyślnie wynik zapisany zostanie w pliku a.out.

Kompilacja pliku lekser.l na maszynie ultra60:

flex lekser.l  
g++ lex.yy.c -lfl

### Ćwiczenie: Zmodyfikuje specyfikację skanera lekser.l tak, aby drukowana była suma wszystkich zeskanowanych liczb zamiast ich ilości.

## Page 6 skip…

Zanim przejdziemy do kwestii technicznych, przyjrzyjmy się nieco bliżej innemu zagadnieniu bezpośrednio związanemu z analizą składniową. U podstaw technologii związanych z prezentacją treści w Internecie leżą języki znaczników z językami HTML (HyperText MarkupLanguage) i XML (Extensible Markup Language) na czele. Języki znaczników pozwalają opatrzyć tekst dodatkowymi informacjami dotyczącymi zarówno sposobu jego prezentacji, jak i znaczena semantycznego poszczególnych fragmentów tekstu. Znaczniki otoczone są nawiasami < oraz >, które stają się tym samym znakami szcególnymi w dokumentach napisanych w językach znacznikowych (zauważ, że wyświetlenie znaku mniejszości na stronie internetowej wymaga zapisania go jak &lh; ). Przykładami znaczników określających sposób wyświetlenia fragmentu tekstu na stronie są para <i></i> wskazująca, że otoczony nimi tekst powinien zostać pochylony czy znacznik <br/> łamiący wiersz. Typowymi znacznikami semantycznymi jest para <title></title> wskazująca na tytuł dokmuentu. Nie jest on wyświetlany jako integralny element samej storny, ale na pasku tytułu w przeglądarce czy jako nazwa strony w liście z wynikami wyszukiwania.  
Z naszego punktu widzenia o wiele ciekawszym językiem jest XML. Nie ma on na celu opisania sposobu wyświetlania dokumentu, ale skupia się na znaczeniu semantycznym jego elementów, Pozwala to nadać dokumentowi sturkturę i przygotować go do wykorzystania zarówno przez aplikacje którym ułatwia analizę składniową jak i człowieka, który jest w stanie domyślić się znacznia poszczególnych elementów na podstawie nadanych im nazw i narzuconych przez strukturę relacji między nimi.

Wymiana informacji za pomocą plików XML wymaga narzucenia ograniczeń na ich strukturę. Zagnieżdzenie jednej pary wskaźników w drugiej oznacza, że element wewnętrznej pary jest częścia składową lub atrybutem elementu zewnętrznego. W przypadku dokumentu XML, be zproblemu zrozumiemym zagnieżdżenie pary znaczników <imie></imie> wewnątrz pary <osoba></osoba> jako przypisanie konkretnej osobie atrybutu jakim jest jej imię. Nie znaczy to jednak, że zagnieżdżenie odwrotnie nie ma sensu. Można wyobrazić sobie dokument będący słownikiem imion, które przypisuje każdemu imieniu jego etymologię i podające listę sławnych osobistości to imię noszących.

Za narzucenie dokumentowi XML odpowiedniej struktury, innymi słowy zdefiniowanie dokumentów XML, które są zgodne z pewnym standardem opisu, odpowiedzialne sa pliki DTD (Document-Type Definition). Znajdziemy w nich opis gramatyki definiującej pewien język bezkontekstowy dokumentów XML. Rzeczywista specyfikacja zamieszczona w dokumentach DTD jest nieco bardziej złożona, my przeanalizujemy tylko jej podstawowe elementy (określane zreszą słowem kluczowym ELEMENT). Każdy taki element ma swoją unikatową nazwę, jak symbole nieterminalne gramatyki, którym odpowiadają elementy. Ponadto w nawiasach podane są wywrażenia określające listę produkcji, w których ten symbol występuje po lewej stronie. Przecinki czytamy jako konkatenacje symboli, znaki ‘|’, podobnie jak w przypadku przyjętej przez nas konwencji zapisu gramatyk, jako skróconą formę zapisu kilku produkcji (alternatywę), ‘?’ jako zero lub jedno wystąpienie danego symbolu, zaś ‘\*’ oraz ‘+’ jako listę i listę niepustą symboli, zaś #PCDATA oznacza dowolny cią znaków, czyli symboli terminalnych. Za symbol startowy przyjmuje się pierwszy symbol występujący w opisie. Przekształcenie takiego zapisu do przyjętej przez nas umowy może wymagać wprowadzenia nowych symboli nieterminalnych. Szczególnie w przypadku rozwijania list oraz konkatenacji bardziej złożonych elementów.

# PAGE 8-skip

# …

Pokazaliśmy w jaki sposób języków bezkontekstowyhc można użyć do opisu struktury kodu źródłowego programu napisanego w ustalonym języku programowania czy sposobu konstruowania plików XML zgodnych z wyspecyfikowanym formatem. Możliwość zastosowania języków bezkontekstowych do tych zadań zawdzięczamy w dużej mierze gramatykom opisującym te języki. A właściwie ich własności, która pozwala w sposób jednoznaczny określić strukturę wyprowadzenia ciągu symboli terminalnych z symbolu startowego.

Przypomnijmy, że gramatykę G=<Σ,V,P,S> nazywamy jednoznaczną, jeśli dla każdego słowa wϵL(G) istnieje dokładnie jedno drzewo wywodu tego słowa z symbolu startowego S.

### DEFINICJA

Język bezkontekstowy L nazywamy jednoznacznym, jeśli istnieje przynajmniej jedna gramatyka bezkontekstowa G, taka, że L=L(G).

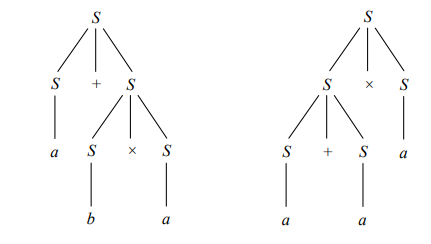
### PRZYKŁAD

Rozważmy język bezkontekstowy L opisujący poprawne wyrażenia arytmetyczne w notacji infiksowej, gdzie dopuszczalnymi działaniami są suma i iloczyn oraz opisującą go gramatykę Gn=<{a,b,+,x}, {S}, {S->S+S|S x S|a|b},S>

Zauważ, że wyrażenie a+b x a można w niej wywieśćz symbolu S na wiele sposobó:

* S->S+S->S+SxS->a+SxS->a+b x S-> a+b x a.
* S->S+S->a+S-> a+SxS->a+S x a->a+b x a.
* S->SxS->S+S x S-> a+S x S->a+b x S->a+b x a.

Dwa pierwsze wywodu mają takie same drzew , podczas kiedy drzeweo trzeciego wywodu jest inne:



Pokazuje to, że przedstawiona gramatyka jest niejednoznaczna. Co więcej, przyjmując, że a=2 zaś b=3 i wykonując działania zgodnie z kolejnością sugerowaną przez pierwsze drzewo otrzymamy 2+3x2=8, zaś w drugim przypadku 2+3x2=10. Uwzględniając ogólnie przyjęte priorytety wykonywania działań, uznamy pierwsze wyprowadzenie jako zdecydowanie lepsze.

Pokażemy na przykładzie języka L jak można radzić sobie niejednoznacznością gramatyki. Innymi słowy, przekształcimy niejednoznaczne przedstawienie języka jednoznacznego w przedstawienie jednoznaczne. W języku L występują dwa zasadnicze powody występowania niejednoznaczności. Pierwszym jest wspomniany brak priorytetów ustalających kolejność wykonywania działań, a jak wiemy, zgodnie z przyjętą umową mnożenie należy wykonywać przed dodawaniem. Osiągniemy to przez rozbicie produkcji S->S+S|S x S|a|b na S->S+S|X oraz X-> X x X|a|b. Dzieki temu najpierw produkujemy wszystkie sumy a następnie ich składniki zamieniamy na iloczyny jednego lub więcej symboli terminalnych. Pozostaje jeszcze kwestia jednozanczności w przypadku kilkakrotnego wykorzystania tej samej produkjci dublującej S lub X. Podobnie jak w przedstawionym powyżej przykładzie możemy uzyskać dwa drzewa wywodów dla wyrażenia a+b+a. Ze względu na łączność sumy nie ma co prawda znaczenia, czy skorzystamy z pierwszego czy z drugiego otrzymanego drzewa, w obu przypadkach uzyskamy wynik 2+3+2=7. Nie wszystkie działania są jednak łączne, a orzymana gramatyka jest niejednoznaczna. Ogólnie przyjęta konwencja mówi, że działania o tym samym priorytecie wykonujemy od lewej do prawej, spróbujmy uwzględnić to w gramatyce opisującej język L. Uzyskamy to wprowadzająć niewielkie zmiany do naszego zbioru produkcji uzyskując S->S+X|X oraz X->X x a|X x b|a|b.

## Zadanie: Uzasadnij, że otrzymana w ten sposób gramatyka jest jednoznaczna.

Okazuje się niestety, żę istnieją języki bezkontekstowe, których nie da się opisać za pomocą gramatyki jednoznacznej. Problemem tym zajmiemy się dokładniej w ramach jednego z kolejnych wykładów. Na potrzeby analizy składniowej wystarczyć nam muszą języki jednoznaczne, a właściwie jednoznaczne przedstawienia języków jednoznacznych. Narzędzia wspomagające oferują mechanizmy pozwalające im, po odpowiednim poinstruowaniu przez użytkownika, automatycznie usuwać proste i częste powody niejednoznaczności gramatyk. W przypadku języków niejednoznacznych są one jednak bezsilne.  
Dla lepszego zrozumienia mechanizmów rządzących parserami wygenerowanymi na podstawie specyifkacji przygotowanej dla yacca (narzędzia podobnego do poznanego przez nas wczęsniej lexa), wprowadzimy jeszcze pojęcie gramatyk S-atrybutywnych.

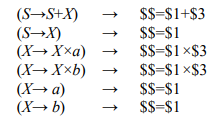
### Definicja

Gramatyką S-atrybutywną nazywamy trójkę AG=<G,A,R> gdzie G=<Σ, V, P,S> jest jednoznaczną gramatyką bezkontekstową, A – skończonym zbiorem atrybutów, zaś R zbiorem reguł, które dla każdej produkcji gramatyki G określają, w jaki sposób z wartości atrybutów symboli znajdujących się po prawej stronie produkcji uzyskać wartości atrybutów symbolu nieterminalnego znajdującego się po jej lewej stronie. W dalszej części przyjmować będziemy, że zbiór A jest jednoelementowy i jedyny atrybut symboli nieterminalnych będziemy nazywać ich wartością/

### Przykład

Przyjmemimy zapis reguł wyznaczania wartości instancji symboli nieterminalnych zapożyczony z yacca. Przez $$ oznaczać będziemy wartość instancji symbolu nieterminalnego znajdująceho się po lewej stronie produkcji, zaś przez $1, $2, … wartości instancji kolejnych symboli znajdujących się po jej lewej stronie (każda instancja tego samego symbolu terminalnego ma oczywiście tę samą wartość). Przeanalizujemy gramatykę atrybutywną skonstruowaną na bazie jednoznacznej gramatyki bezkontekstowej dla języka L wyrażeń arytmetycznych, analizowanego wcześniej.

Przypomnijmy, że G=<{a,b,+,x}, {S}, {S->S+X|X, X->X x a| X x b|a|b }, S> oraz A={x}.  
Reguły określające sposób wyznaczania wartości (zgodnie z ogólnie przyjętym znaczeniem wykorzystywanych symboli terminalnych) wyglądają następująco:



Zauważmy, żę do wyznaczenia wartości instancji symboli nieterminalnych bioraćych udział w wywodzie danego wyrażenia musimy jeszcze okreś,ić wartości dla symboli terminalnych.  
W analizowanym przez nas przypadku dotyczy to tylko symboli a oraz b. Wartości symboli + oraz x nie wpływają na wynik. Zauważ, że dokładnie w ten sposó (to jest przypisując wartości symbolom terminalnym i wyznaczając wartości instancji symboli nieterminalnych wykorzystanych w wywodzie) postąpiliśmy porównując wyprowadzenia wyrażenia arytmetycznego a+b x a w gramatyce Gn.

W dalszej części tego wykładu omówimy program generujący kod źródłowy parsera w języku C lub C++. Programem tym jest yacc (yet another compiler compiler). Mimo, że języki regularne są podzbiorami języków bezkontekstowych, przez co można przy pomocy gramatyk dzielić tekst na leksemy, lepiej wykorzystać w tym celu doskonale się do tego nadające wyrażenia regularne i wydzielone skanery. Dlatego ważnym elementem składowym każdej specyfikacji naisanej dla yacca jest właśnie skaner, któRy możemy stworzyć w języku C implementując funkcję yylex. Jak się łatwo domyślić widząc nazwę funkcji skanera, możemy też wykorzystaćw tym celu znany nam z poprzednich wykładów program lex. Tak też zrobimy w analizowanym na koniec tego wykładu przykładzie.  
Przyjrzyjmy się teraz składni specyfikacji parsera. Podobnie jak w przypadku specyfikacji skanera, składa się ona z trzech części rozdzielonych wierszami %%. Są to:

* Deklaracje
* Definicja gramatyki S-atrybutywnej
* Procedury

W części pierwszej umieszczamy ujęte w nawiasy % { oraz %{ wszystkie globalne deklaracje konieczne przy kompilacji – nasze wynikowego programu, w szczególności dyrektywy dołączające kod bibliotek (include). Ponadto, deklarujemy tam nazwy używanych przez nas symboli terminalnych jako rozdzieloną spacjami listę napisów zainicjowaną przez słowo kluczowe %token. Jeśli do zbudowania skanera będziemy chcieli użyć lexa, musimy zapewnić, że program wygenerowany na podstawie specyfikacji skanera będzie znał te nazwy. Osiągniemy to za pomocą wygenerowania pliku nagłówkowego dla programu parsera.

W częśći drugiej znajdują się produkcjie i reguły gramatyki S-atrybutywnej. Produkcje zapisujemy używając zadeklarowanych wcześniej nazw symboli terminalnych oraz stałych znakowych, rodzielając lewą stronę produkcji od prawej za pomocą dwukropka. Dalej zapisujemy kod w języku C/C++, który może być wzbogacany o instrukcje odpowiadające regułom wyznaczania wartości dla przetwarzanej instancji symbolu nieterminalnego, zgodnie z przyjętą przez nas wcześniej konwencją zapisu tych reguł. Pamiętaj, że gramtyka z przyjętą przez nas wcześniej konwencją zapisu tych reguł. Pamiętaj, że gramatyka zdeifniowana w tej częśći musi być jednoznaczna. Yacc posiada specjalne mechanizmy pozwalające określić priorytety wykonywania działań czy sposób ich wiązania i z pewnością ostrzeże Cie, jeśli napotka jakikolwiek powód dwuznaczności. Zostanie wówczas wydurkowane ostrzeżenie o konflikcie (shift/reduce lub reduce/reduce)

W ostatniej, trzeciej częsci, zawieramy definicje procedur z których chcemy skorzystać w parserze. W szczególności możemy tam umieścićfunkcje main, yylex zawierającą naszą własną ….