ROZDZIAŁ SZESNASTY: **DOPASOWANIE DO WZORCA**

Ostatni rozdział omawiał ciągi znaków i różne działania na tych ciągach. Typowy program odczytuje sekwencję ciągu od użytkownika i porównuje ciągi czy są dopasowane. Na przykład DOS'owski program COMMAND.COM odczytuje linię poleceń od użytkownika i porównuje ciąg użytkownika z wpisanym stałym ciągiem takim jak „COPY", „DEL", „RENAME" i tak dalej. Takie polecenia są łatwe do analizowania ponieważ zbiór dostępnych poleceń jest skończony i stały. Czasami jednak ciągi jakie chcemy przetestować nie są stałe; zamiast tego należą do (możliwie nieskończonego) zbioru różnych ciągów. Na przykład jeśli wykonujemy DOS'owe polecenie „DEL .BAK", MS-DOS nie próbuje usunąć pliku nazwanego „.BAK". Zamiast tego usuwa wszystkie pliki, do których pasuje ogólny wzorzec „.BAK". To oczywiście jest każdy plik, który zawiera cztery lub więcej znaków i kończą się „.BAK". W świecie MS-DOS, ciąg zawierający znaki takie jak „*" i „?" są nazywane symbolami wieloznacznymi; znaki symboli wieloznacznych po prostu dostarczają sposobu do określenia różnych poprzez wzorce. DOS'owe znaki symboli wieloznacznych maja bardzo ograniczoną postać, co jest znane jako wyrażenia regularne; wyrażenia regularne mają , generalnie, ograniczoną postać wzorca. Rozdział ten opisuje jak stworzyć wzorzec, który dopasowuje różne ciągi znaków i pisać podprogramy dopasowania do wzorca, aby zobaczyć czy szczególny ciąg dopasowany jest do danego wzorca.

16.1 WPROWADZENIE DO TEORII JĘZYKA FORMALNEGO (AUTOMATÓW)

Dopasowanie do wzorca jest ważnym tematem w informatyce. Istotnie, dopasowanie do wzorca jest głównym paradygmatem programistycznym w kilku językach programowania takich jak Prolog, SNOBOL i Icon Kilka programów używanych cały czas stosuje dopasowanie do wzorca jako ważną część ich pracy. MASM na przykład używa dopasowania do wzorca do określenia czy symbole są poprawnie sformułowane, wyrażenia są właściwe itd. Kompilatory języków wysokiego poziomu jak Pascal i C również używają dopasowania do wzorca dla analizy pliku źródłowego określając czy jest on syntaktycznie poprawny. Niespodziewanie dość, ważne wyrażenie znane jako Hipoteza Church'a sugeruje, że każda obliczalna funkcja może być zaprogramowana jako problem dopasowania do wzorca. Oczywiście, nie ma żadnej gwarancji, że to rozwiązanie będzie wydajne (zazwyczaj nie jest) ale możemy dojść do poprawnego rozwiązania. Prawdopodobnie nie będziemy musieli nic wiedzieć o maszynie Turinga (temat hipotezy Church'a) jeśli interesuje nas pisanie, powiedzmy obliczanie otrzymanych pakietów. Jednakże, jest wiele sytuacji gdzie możemy chcieć wprowadzić umiejętność dopasowania jakiegoś ogólnego wzorca; więc zrozumienie teorii dopasowania do wzorca jest ważna. Te obszar informatyki nazywa się teorią języka formalnego lub teorią automatów. Kursy z tego tematu są często mniej niż popularne ponieważ wprowadzają one dużo dowodów, matematyki i , cóż, teorii. Jednakże, pojęcia poza dowodami są całkiem proste i bardzo użyteczne. W rozdziale tym nie będziemy się zajmowali próbować dowodzić wszystkiego o dopasowaniu do wzorca. Zamiast tego będziemy akceptować fakt, że działa to w rzeczywistości i stosujemy to. Pomimo to musimy omówić pewne tematy z teorii automatów, więc bez dalszych wstępów.

16.1.1 MASZYNY KONTRA JĘZYKI

Znajdziemy odniesienie do terminu „maszyna" w całej literaturze teorii automatów. Termin ten nie odnosi się do jakichś określonych komputerów , na których wykonuje się program. Zamiast tego, jest to zazwyczaj jakaś funkcja, która odczytuje ciąg symboli na wejściu i tworzy jeden lub dwa wyjścia :dopasowanie lub niepowodzenie. Typowa maszyna (lub automat) dzieli wszystkie możliwe ciągi na dwa zbiory – te ciągi, która akceptuje (lub dopasowuje) i te ciągi które odrzuca. Język akceptowany przez tą maszynę jest zbiór wszystkich ciągów, które maszyna akceptuje. Zauważ, że ten język może być nieskończony, skończony lub zbiorem pustym (tj. maszyna odrzuca wszystkie ciągi wejściowe). Zauważ też, że język nieskończony nie wskazuje, ze maszyna akceptuje wszystkie ciągi. Jest całkiem możliwe, że maszyna akceptuje nieskończony liczbę ciągów a odrzuca

większą liczbę ciągów. Na przykład, bardzo łatwo jest zaprojektować funkcję, która akceptuje wszystkie ciągi których długość jest wielokrotnością trzech. Funkcja ta akceptuje nieskończoną liczbę ciągów (ponieważ jest nieskończona liczba ciągów , których długość jest wielokrotnością trzech) mimo to odrzuca dwa razy więcej ciągów niż akceptuje. Jest to bardzo łatwa funkcja do napisania. Rozważmy poniższy program 80x86, który akceptuje wszystkie ciągi o długości trzy (zakładając, że znak powrotu karetki kończy ciąg):

```
MatchLen3       proc  near
                getc                            ;pobiera znak #1
                cmp     al., cr                 ;znak zera jeśli EOLN
                je      Accept
                getc                            ;pobranie znaku #2
                cmp     al., cr
                je      Failure
                getc                            ;pobranie znaku #3
                cmp     al., cr
                jne     Match    Len3
Failure:        mov     ax, 0                   ;zwraca zero oznaczające niepowodzenie
                ret
Accept:         mov     ax, 1                   ;zwraca jeden oznaczające powodzenie
                ret
MatchLen3       endp
```

Przez śledzenie całego kodu powinniśmy łatwo przekonać się sami, że zwraca jeden w ax jeśli powiodło się (odczytano ciąg, którego długość jest wielokrotnością trzech) i zero w przeciwnym razie.
Maszyny są z natury rozpoznawaczami. Sama maszyna jest ucieleśnieniem wzorca. Rozpoznaje każdy ciąg wejściowy, który dopasowuje do wbudowanego wzorca. Dlatego też, kodyfikacja tych automatów jest podstawową pracą programisty, który chce dopasować jakieś wzorce.
Jest wiele różnych maszyn i języków, które one rozpoznają. Od prostych do złożonych, ważna klasyfikacją jest deterministyczny skończony stan automatów (który jest ekwiwalentem niedeterministycznego skończonego stanu automatów), deterministyczny automat stosowy, niedeterministyczny automat stosowy i maszyna Turinga. Każda kolejna maszyna na tej liście dostarcza nadzbioru zdolności maszyn pojawiających się przed nią. Jedynym powodem dla którego nie używamy maszyny Turinga dla wszystkiego, jest to, że jest dużo bardziej złożone zaprogramowanie niż, powiedzmy, deterministycznego skończonego stanu automatu. Jeśli możemy dopasować wzór używając deterministycznego skończonego stanu automatu, prawdopodobnie będziemy chcieli zakodować go w ten sposób niż jako maszynę Turinga.
Każda klasa maszyny ma klasę języka z nią powiązaną. Deterministyczne i niedeterministyczne skończone stany automatów rozpoznają język regularny. Niedeterministyczny automat stosowy rozpoznaje język bez kontekstowy. Maszyna Turinga może rozpoznać wszystkie rozpoznawalne języki. Będziemy omawiali każdy z tych zbiorów języków i ich właściwości po kolei.

16.1.2 JĘZYKI SKOŃCZONE

Języki skończone są najmniej złożonymi językami opisanymi w poprzedniej sekcji. Nie znaczy to, że są mniej użyteczne; faktycznie, wzorce oparte o wyrażenia skończone są prawdopodobnie bardziej popularne niż inne

16.1.2.1 WYRAŻENIA SKOŃCZONE

Najbardziej zwartym sposobem określenia ciągów, które należą do języka skończonego jest wyrażenie skończone. Zdefiniujemy wyrażenia skończone według następujących zasad :

- ∅ (zbiór pusty) jest jeżykiem skończonym i oznacza zbiór pusty
- ε jest wyrażeniem skończonym. Oznacza zbiór języków zawierających tylko pusty ciąg: {ε}.
- Każdy pojedynczy symbol ‚a ‚jest wyrażeniem skończonym (będziemy używali małych liter do oznaczania przypadkowych symboli). Ten pojedynczy symbol dopasowuje dokładnie jeden znak w ciągu wejściowym, który to znak musi być równy pojedynczemu symbolowi w wyrażeniu skończonym. Na przykład, wzorzec „m” dopasowuje pojedynczy znak „m” w ciągu wejściowym.

Zauważmy, że ∅ i ε nie są tym samym. Zbiór pusty jest skończonym językiem, który nie akceptuje żadnego ciągu, wliczając ciągi o długości zero. Jeśli język skończony jest oznaczony przez {ε} wtedy akceptuje dokładnie jeden ciąg, ciąg długości zero. Ten ostatni język skończony akceptuje coś, pierwszy nie.
Trzecia z powyższych zasad dostarcza nam podstaw dla definicji rekurencji. Teraz będziemy definiować wyrażenie skończone rekurencyjnie. W następujących definicjach, zakładamy, że r , s i t są poprawnymi wyrażeniami skończonymi.

- Konkatenacja. Jeśli r i s są wyrażeniami skończonymi, więc to rs. Wyrażenie skończone rs dopasowuje każdy ciąg, który zaczyna się ciągiem dopasowanym przez r i kończy ciągiem dopasowanym przez s
- Suma logiczna / Unia. Jeśli r i s są wyrażeniami skończonymi, więc r | s (czytamy to jako r **lub** s) Jest to odpowiednik dla r ∪ s (czytamy jako r unia s). To wyrażenie skończone dopasowuje każdy ciąg , który dopasowuje r lub s
- Iloczyn logiczny. Jeśli r i s są wyrażeniami skończonymi, więc r ∩ s. Jest to zbiór wszystkich ciągów, które dopasowują oba r **i** s.
- Kleene Star. Jeśli r jest wyrażeniem skończonym, więc r*. To wyrażenie skończone dopasowuje zero lub więcej wystąpień r. To znaczy, dopasowuje ε, r, rr, rrr, rrrr,.....
- Różnica. Jeśli r i s są wyrażeniami skończonymi, więc r – s. To oznacza zbiór ciągów dopasowanych przez r, które nie są dopasowane również przez s.
- Pierwszeństwo. Jeśli r jest wyrażeniem skończonym, więc (r). To dopasowuje każdy ciąg dopasowany przez samo r. Normalne algebraiczne prawa łączności i rozdzielności mają tu zastosowanie, więc (r | s) t jest odpowiednikiem rt | st.

Operatory te wykorzystują zwykłe prawa łączności i rozdzielności mają następujące pierwszeństwo;

Najwyższe:	(r)
	Kleene Star
	Konkatenacja
	Iloczyn logiczny
	Różnica
Najniższe:	Suma logiczna

Przykłady:

(r | s) t = rt | st
rs* = r(s*)
r ∪ t – s = r ∪ (t – s)
r ∩ t – s = (r ∩ t) – s

Generalnie, będziemy używali nawiasów okrągłych aby uniknąć niejasności.
Chociaż ta definicja jest wystarczająca dla klasy teorii automatów, są praktyczne aspekty tej definicji, która pozostawia trochę do życzenia. Na przykład, dla zdefiniowania wyrażenia skończonego, które dopasowuje pojedynczy znak alfabetu, będziemy musieli stworzyć coś takiego jak (a | b | c |.... | y | z). Trochę pisania jak na tak trywialny zbiór znaków. Dlatego też powinniśmy dodać jakąś notację aby uczynić łatwiejszym określanie wyrażeń skończonych.

- Zbiór Znaków. Każdy zbiór znaków otoczonych przez nawiasy kwadratowe np. [abcdefg] jest wyrażeniem skończonym i dopasowuje pojedynczy znak ze zbioru. Możemy określić zakres znaków używając myślnika tj. „[a – z]" oznaczający zbiór małych liter a to wyrażenie skończone dopasowuje pojedynczy znak małej litery
- Kleene Plus. Jeśli r jest wyrażeniem skończonym, więc r^+. To wyrażenie skończone dopasowuje jedno lub więcej wystąpień r. To znaczy, dopasowuje r, rr, rrr, rrrr,.. Pierwszeństwo Kleene Plus jest takie samo jak dla Kleene Star. Zauważmy, że $r^+ = rr^*$.
- Σ przedstawia dowolny pojedynczy znak z dostępnego zbioru znaków. Σ* przedstawia zbiór wszystkich możliwych ciągów. Wyrażenie skończone Σ* - r jest uzupełnieniem r – to znaczy, zbiór wszystkich ciągów, których r nie dopasowało

Nadszedł czas aby omówić jak w rzeczywistości używamy wyrażeń skończonych przy specyfikacji dopasowania do wzorca. Następujące przykłady powinny nam dać odpowiednie wprowadzenie

Identyfikatory: Większość języków programowania, takich jak Pascal lub C/C++ określa poprawne formy dla identyfikatorów używając wyrażeń skończonych. Używając angielskiej terminologii określamy je: „Identyfikator musi zaczynać się znakiem alfabetu i następuje po nim zero lub więcej znaków alfanumerycznych lub znaku podkreślenia.” Używając składni wyrażenia skończonego (WS) opisanej w tej sekcji, identyfikator to

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

Stałe Całkowite: Wyrażenie skończone dla stałych całkowitych jest relatywnie łatwe do zaprojektowania .Stałe całkowite składają się opcjonalnie z plus lub minusa i następujących po nich jednej lub więcej cyfr. WS to
(+ | - | ε |) $[0-9]^+$.
Zauważ, że użycie pustego ciągu (ε) czyni plus lub minus opcjonalnym

Stałe rzeczywiste: stałe rzeczywiste są trochę bardziej złożone, ale łatwe do określenia przy użyciu WS. Nasz definicja wzorca, która dla stałych rzeczywistych pojawia się w programie pascalowskim – opcjonalnie plus lub minus, po którym następuje jedna lub więcej cyfr; opcjonalnie następuje punkt dziesiętny i zero lub więcej cyfr; opcjonalnie następuje po „e” lub „E” z opcjonalnym znakiem i jedną lub więcej cyframi:
(+ | - | ε) $[0-9]^+$ („ .”[0-9]* | ε) (((e | E) (+ | - | ε)$[0-9]^+$) | ε)
Ponieważ WS jest relatywnie złożone, powinniśmy je rozłożyć kawałek po kawałku. Pierwszy człon w nawiasach daje nam opcjonalny znak. Jedna lub więcej cyfr są obowiązkowe przed punktem dziesiętnym, drugim dostarczonym członem. Trzeci człon pozwala ma punkt dziesiętny po którym następuje zero lub więcej cyfr. Ostatni człon dostarcza opcjonalnego wykładnika składającego się z „e” lub „E”, następujący po opcjonalnym znaku lub jednej lub więcej cyfrach.

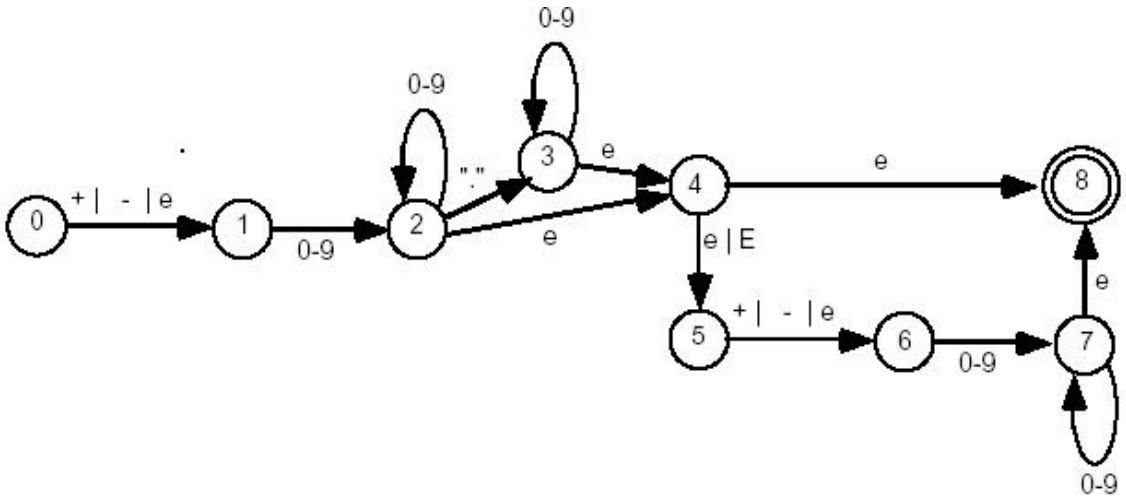
Słowa zarezerwowane: Jest bardzo łatwo dostarczyć wyrażenie skończone, które dopasowuje zbiór zarezerwowanych słów. Na przykład, jeśli chcemy stworzyć wyrażenie skończone, które dopasowuje słowa zarezerwowane MASM’a , możemy użyć WS podobnego do tego

(mov | add | and |... | mul)

Parzystość: Wyrażenie skończone (ΣΣ)* dopasowuje wszystkie ciągi, których długość jest wielokrotnością dwóch

Zdania: Wyrażenie skończone:

(Σ* „ „ *)* run („ „ $^+$(Σ* „ „$^+$ | ε)) fast („ „ Σ*)*



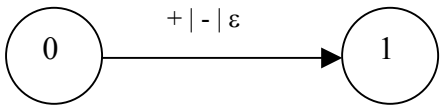
Rysunek 16.1 NFA dla Wyrażenia Skończonego (+|-|e)[0-9]+(„.”[0-9]|e)(((e|E)(+|-e)[0-9]+)|e

dopasowuje wszystkie ciągi, które zawierają oddzielne słowa „run” następujące po nim „fast” gdzieś w linii. To dopasowuje ciągi jakie „I want to run very fast” i „run as fast as you can” tak jak i „run fast”

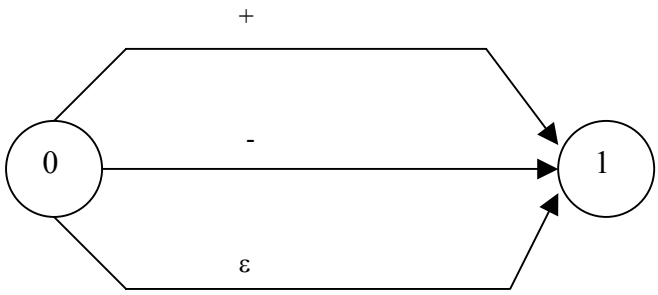
Podczas gdy WS są dogodne do określania wzorca jaki chcemy rozpoznać, nie są one szczególnie użyteczne tworzenia programów (tj. „maszyn"), które w rzeczywistości rozpoznają takie wzorce. Zamiast tego, powinniśmy najpierw skonwertować WS do niedeterministycznego skończonego stanu automatu, lub NFA. Jest bardzo łatwo skonwertować NFA do programu asemblerowego 80x86; jednakże takie programy rzadko są wydajne tak jak mogłyby być. Jeśli wydajność jest dużym zmartwieniem, możemy skonwertować NFA do deterministycznego skończonego stanu automatu (DFA) , który również jest łatwy do skonwertowania do kodu asemblowanego 80x86, ale konwersja jest dużo bardziej sprawna

16.1.2.2 NIEDERTMINISTYCZNE SKOŃCZONE STANY AUTOMATÓW (NFA)

NFA jest bezpośrednim wykresem z liczbą stanów powiązanych z każdym węzłem i znakiem lub ciągiem znaków powiązanych z każdym brzegiem wykresu. Stan wyróżniający się, stan startowy określa gdzie maszyna zaczyna próbę dopasowania ciągu wejściowego. Maszyna w stanie startowym porównuje znaki wprowadzone ze znakami lub ciągami na każdym brzegu wykresu. Jeśli zbiór znaków wejściowych jest dopasowany do jednego z brzegów, maszyna może zmienić stan z węzła na początku brzegu (ogon) do stanu na końcu brzegu (głowa)
Pewne inne stany, znane jako końcowy lub akceptowalny, są zazwyczaj również obecne Jeśli maszyna przeszła do stanu końcowego po wyczerpaniu wszystkich znaków wejściowych, wtedy maszyna ta akceptuje lub dopasowuje ten ciąg. Jeśli maszyna wyczerpała już wprowadzane znaki i przeszła do stanu, który nie jest stanem końcowym, wtedy maszyna ta odrzuca ten ciąg. Rysunek 16.1 pokazuje przykład NFA dla zmienno przecinkowego WS przedstawianego wcześniej.
Przez konwencję, zawsze będziemy zakładać, ze stan startowy to stan zero. Oznaczymy stany końcowe (których może być więcej niż jeden) przez użycie podwójnego okręgu dla tego stanu (powyżej stan osiem jest stanem końcowym).
NFA zawsze zaczyna się ciągiem wejściowym w stanie startowym (stan zero). Na każdym brzegu wychodzącym ze stanu jest albo ε, pojedynczy znak lub ciąg znaków. Pomoc przy nie zasłoniętym diagramie NFA, pozwoli na wyrażenia w postaci „xxx | yyy | zzz |..." gdzie xxx, yyy, zzz są to ε , pojedynczym znakiem lub ciągiem znaków. Odpowiada to wielokrotnym brzegom z jednego stanu do innego z pojedynczą pozycją na każdym brzegu. W powyższym przykładzie :



jest odpowiednikiem



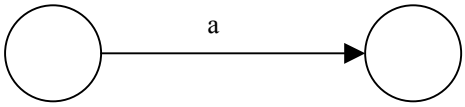
Podobnie jak pozwolimy zbiorowi znaków, określonemu przez ciąg w postaci x – y, oznaczyć wyrażeniem x | x+1 | x+2 | ... | y.
Zauważ, że NFA akceptuje ciąg jeśli jest jakaś ścieżka ze stanu startowego do stanu akceptowalnego, która wyczerpuje ciąg wejściowy. To mogą być wielokrotne ścieżki od stanu startowego do różnych stanów końcowych. Co więcej, to może być jakaś określona ścieżka ze stanu startowego do stanu nie akceptowalnego, która wyczerpała ciąg wejściowy. Niekoniecznie to znaczy, że NFA odrzuca ten ciąg; jeśli jest jakaś inna

ścieżka ze stanu startowego do stanu akceptowalnego, wtedy NFA akceptuje ten ciąg. NFA odrzuca ciąg tylko jeśli nie ma ścieżek ze stanu startowego do stanu akceptowalnego, które wyczerpują ten ciąg.
Przejście przez stan akceptowalny nie powoduje, że NFA akceptuje ciąg. Musimy dotrzeć do stanu końcowego i wyczerpać ciąg wejściowy.
Przetwarzanie ciągu wejściowego z NFA zaczyna się w stanie startowym. Brzeg wychodzący ze stanu startowego zawiera znak, ciąg lub ε, z nim powiązane. Jeśli wybierzemy przesunięcie z jednego stanu do innego wzdłuż brzegu z pojedynczym znakiem, wtedy usuwamy ten znak z ciągu wejściowego i przesuwamy do nowego stanu wzdłuż brzegu przemierzonego przez ten znak. Podobnie , jeśli wybieramy przesuniecie wzdłuż brzegu z ciągiem znaków, usuwamy ten ciąg znaków ciągu wejściowego i przełączamy do nowego stanu. Jeśli jest brzeg z pustym ciągiem ε, wtedy możemy wybrać przesunięcie do nowego stanu danego przez ten brzeg bez usuwania jakiegoś znaku z ciągu wejściowego.
Rozważmy ciąg „1.25e2" i NFA z rysunku 16.1. Z punktu startowego możemy przesunąć do stanu jeden używając ciągu ε (nie ma początkowego plus lub minus, więc tylko ε jest naszą opcją). Ze stanu jeden możemy przesunąć do stanu dwa przez dopasowanie „1" w naszym ciągu wejściowym ze zbiorem 0-9, to zjada „1" z naszego ciągu wejściowego pozostawiając".25e2". Ze stanu dwa przesuwamy do stanu trzy i zjada kropkę z ciągu wejściowego pozostawiając „25e2". Stan trzy jest zapętlony, wiec zjada znaki „2" i „5" z początku naszego ciągu wejściowego i wraca do stanu trzy z nowym ciągiem wejściowym „e2". Następnym znakiem wejściowym jest „e", ale nie ma wychodzącego brzegu ze stanu trzy z „e" na nim; jednakże mamy brzeg - ε, wiec możemy go użyć do przesunięcia do stanu cztery. To przesunięcie nie zmienia ciągu wejściowego. Ze stanu cztery możemy przesunąć do stanu pięć po znaku „e". Zjada to „e" i pozostawia nas z ciągiem „2". Ponieważ nie ma znaku plus lub minus ,musimy przesunąć ze stanu pięć do stanu sześć po brzegu ε. Przesuniecie ze stanu sześć do siedem zjada ostatni znak w naszym ciągu. Ponieważ ciąg jest pusty (i, w szczególności, nie zawiera żadnej cyfry), stan siedem nie może się zapętlić. Jesteśmy obecnie w stanie siedem (który nie jest stanem końcowym) a nasz ciąg wejściowy jest wyczerpany. Jednakże, możemy przesunąć do stanu ósmego 9stan akceptowalny) ponieważ przejściem pomiędzy stanem siedem a osiem jest brzeg ε.Ponieważ jesteśmy w stanie końcowym i wyczerpaliśmy ciąg wejściowy, NFA akceptuje ten ciąg wejściowy.

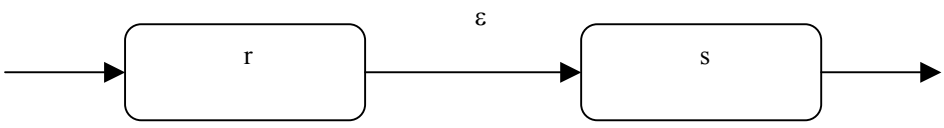
16.1.2.3 KONWERTOWANIE WYRAŻEŃ SKOŃCZONYCH DO NFA

Jeśli mamy wyrażenie skończone i chcemy zbudować maszynę, która rozpoznaje ciągi w języku skończonym określonym przez to wyrażenie, musimy skonwertować WS do NFA. Okazuje się łatwym do skonwertowania wyrażenia skończonego do NFA. Zrobimy to posługując się następującymi zasadami:

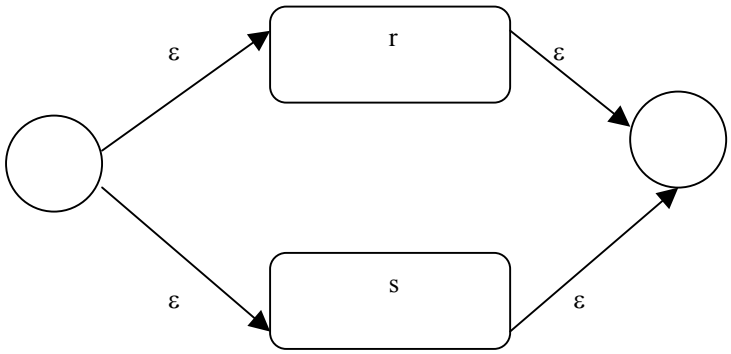
- NFA przedstawiające język skończony oznaczony przez wyrażenie skończone ∅ (zbiór pusty) jest pojedynczym, nie akceptowalnym stanem.
- Jeśli wyrażenie skończone zawiera ε, pojedynczy znak lub ciąg, tworzy dwa stany i rysuje łuk pomiędzy nimi z ε, pojedynczym znakiem lub ciągiem jako etykietą. Na przykład, WS „a" jest konwertowane do NFA jako



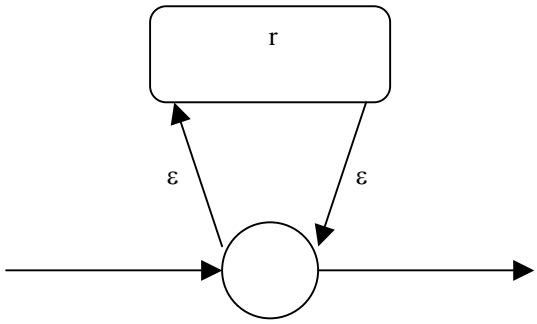
* Symbol  oznacza NFA , który rozpoznaje jakiś język skończony określony przez jakieś skończone wyrażenie r, s lub t. Jeśli wyrażenie skończone przybiera formę rs wtedy odpowiednie NFA to



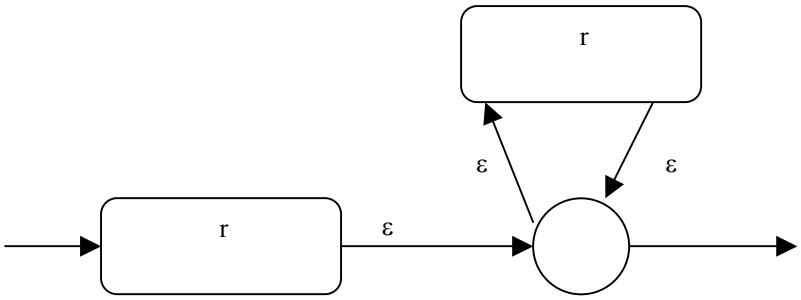
- Jeśli wyrażenie skończone przybiera postać r | s wtedy odpowiednie NFA to



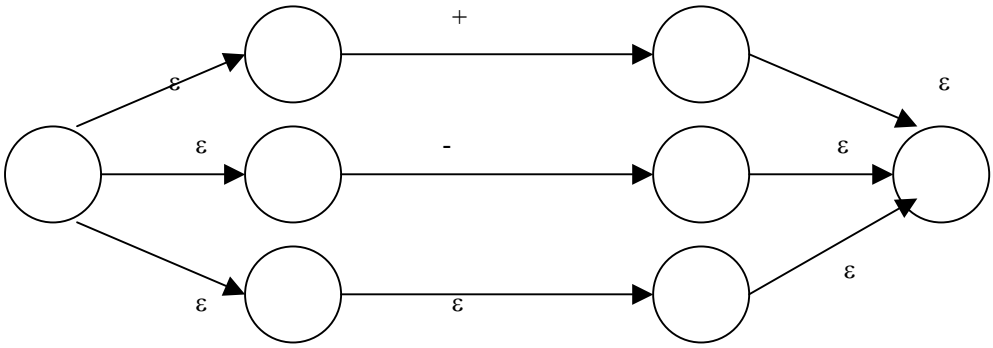
* Jeśli wyrażenie skończone przybiera postać r* wtedy odpowiednie NFA to



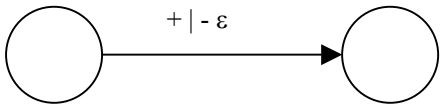
Wszystkie inne formy wyrażeń skończonych są łatwo syntetyzowane z tych, dlatego też konwertowanie tych innych postaci wyrażeń skończonych do NFA jest po prostu dwu krokowym procesem, konwertuje Ws do jednej z tych form, a potem konwertuje tą postać do NFA. Na przykład konwertując r^+ do NFA, najpierw skonwertujemy r^+ do rr*. To tworzy NFA:



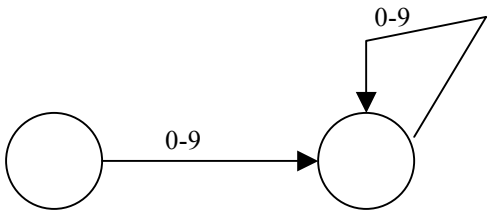
Następny przykład konwertuje wyrażenie skończone dla stałej całkowitej do NFA. Pierwszym krokiem jest stworzenie NFA dla wyrażenia skończonego (+ | - | ε). Kompletna konstrukcja



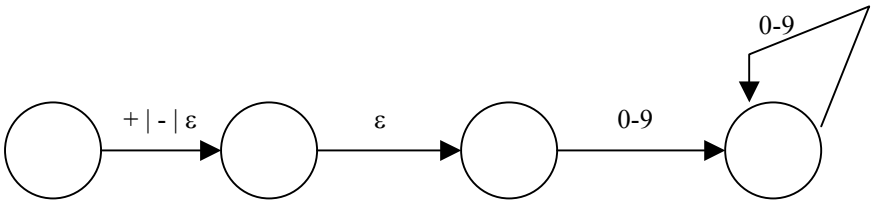
Chociaż możemy oczywiście zoptymalizować to tak



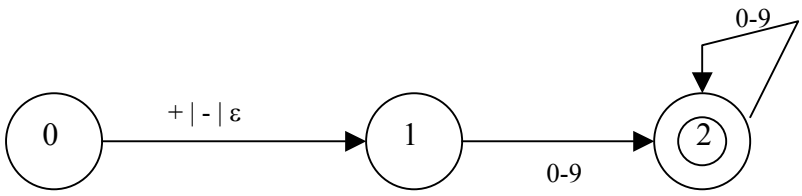
Następnym krokiem jest działanie na wyrażeniu skończonym [0-9]*; po optymalizacji daje to NFA



Teraz po prostu łączymy wszystko tworząc



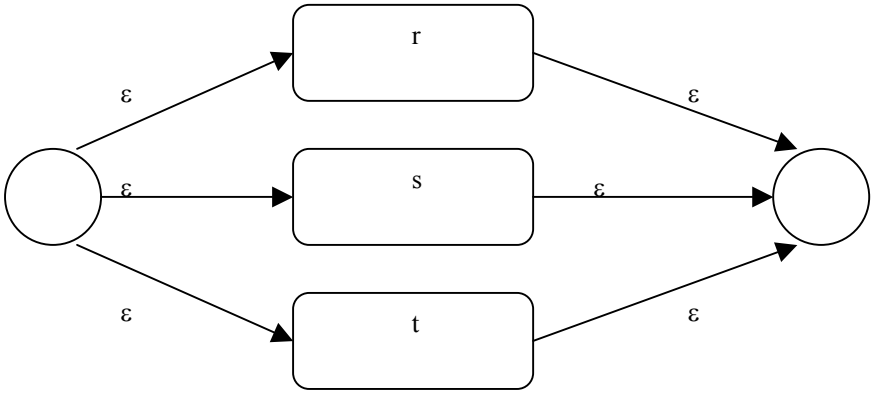
Wszystko co teraz trzeba do znaleźć stan startowy i stan końcowy. Stan startowy jest zawsze pierwszym stanem NFA tworzonym przez konwersję pozycji najbardziej na lewo w wyrażeniu skończonym. Stan końcowy jest zawsze ostatnim stanem NFA tworzonym przez konwersję pozycji najbardziej na prawo w wyrażeniu skończonym. Dlatego też, kompletne wyrażenie skończone dla stałych całkowitych (po optymalizacji powyższego środkowego brzegu, który nie służy żadnemu celowi) to
0-9



16.1.2.4 KONWERSJA NFA DO JĘZYKA ASEMBLERA

Jest tylko jeden ważny problem z konwersją NFA do właściwej funkcji dopasowującej – NFA jest nie deterministyczne. Jeśli jesteśmy w jakimś stanie i mamy jakiś znak wejściowy, powiedzmy „a”, nie ma gwarancji, że NFA powie nam co robić dalej. Na przykład, nie jest wymagane aby brzegi wychodzące ze stanu maiły unikalną etykietę. Możemy mieć dwa lub więcej brzegów wychodzących ze stanu, wszystkie prowadzące do różnych stanów pojedynczego znaku „a”. Jeśli NFA akceptuje ciąg, tylko gwarantuje, że jest jakaś ścieżka, która prowadzi do stanu akceptowalnego, nie gwarantuje, że ta ścieżka będzie łatwa do odnalezienia.
Podstawową techniką jaką będziemy stosować do rozwiązania nie deterministycznych zachowań NFA jest backtracing – sprawdzanie wsteczne. Funkcja , która próbuje dopasować wzorzec używając NFA zaczyna w stanie startowym i próbuje dopasować pierwszy znak(i) ciągu wejściowego do brzegu opuszczającego stan

startowy. Jeśli jest tylko jedno dopasowanie, kod musi następować po tym brzegu. Jednakże, jeśli są dwa możliwe brzegi , wtedy kod musi arbitralnie wybrać jeden z nich i również zapamiętać drugi, jako bieżący punkt w ciągu wejściowym. Później, jeśli okaże się , że algorytm wybrał niewłaściwy brzeg, może wrócić i próbować inną z alternatyw(tj. wraca i próbuje innej ścieżki). Jeśli algorytm wyczerpie wszystkie alternatywy bez przechodzenia do stanu końcowego (z pustym ciągiem wejściowym), wtedy NFA nie akceptuje ciągu.
Prawdopodobnie najłatwiejszy sposób implementacji backtracingu jest poprzez procedurę wywołującą. Zakładamy, że procedura dopasowująca zwraca ustawioną flagę przeniesienia jeśli powodzenie (tj. akceptuje ciąg) i zwraca wyzerowaną flagę przeniesienia jeśli niepowodzenie (tj. odrzucony ciąg) Jeśli NFA oferuje wielokrotny wybór, możemy zaimplementować tą część NFA jak następuje:




```
AltRST      proc    near
            push    ax              ;celem tych dwóch instrukcji jest zachowanie di w
            mov     ax, di          ;przypadku niepowodzenia
            call    r
            jc      Success
            mov     di, ax          ;Przywrócenie di (może być  modyfikowane przez r)
            call    s
            jc      Success
            mov     di, ax          ;Przywrócenie di (może być modyfikowany przez s)
            call    t
Success:    pop     ax              ;przywrócenie ax
            ret
AltRST:     endp
```

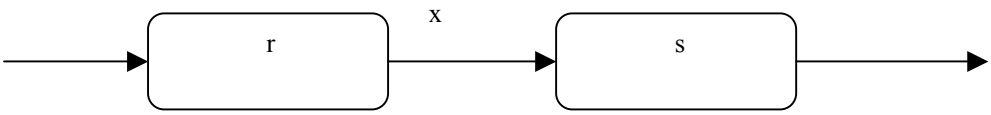
Jeśli procedura dopasowująca r zakończy się powodzeniem, nie ma potrzeby próbować s I t. Z drugiej strony jeśli r zakończyła się niepowodzeniem, wtedy musimy próbować s. Podobnie, jeśli r i s, oba są błędne, próbujemy t. AltRST zakończy się niepowodzeniem, tylko jeśli r,s i t wszystkie są błędne. Kod ten zakłada, ze es:di wskazuje ciąg wejściowy do dopasowania. Przy zwrocie, es:di wskazuje następny dostępny znak w ciągu po dopasowaniu lub wskazuje jakiś przypadkowy punkt jeśli dopasowanie skończyło się niepowodzeniem. Kod ten zakłada ,że r,s i t wszystkie zachowują rejestr ax, więc zachowują wskaźnik do bieżącego punktu w ciągu wejściowym w ax jeśli r lub s są błędne.
Działanie na pojedynczym NFA powiązanym z prostym wyrażeniem skończonym (tj. dopasowanie ε lub pojedynczego znaku) nie jest wcale trudne. Przypuśćmy, że funkcja dopasowująca r dopasowuje wyrażenie skończone (+ | - | ε). Kompletna procedura dla r to

```
r           proc    near
            cmp     byte   ptr es:[di] , ,+'
            je      r_matched
            cmp     byte  ptr  es:[di], ,-,
            jne     r_nomatch
r_matched:  inc     di
r_nomatch:  stc
            ret
r           endp
```

Zauważ, że nie ma wyraźnego testu dla ε. Jeśli ε jest jedną z alternatyw, funkcja próbuje dopasować najpierw jedną z alternatyw. Jeśli żadna z alternatyw nie zakończyła się powodzeniem, wtedy funkcja dopasowująca będzie poprawna zawsze, chociaż nie konsumuje żadnych znaków wejściowych (dlatego powyższy kod przeskakuje instrukcję inc di, jeśli nie dopasowuje „+" lub „-„). Dlatego też, każda funkcja dopasowująca , która ma ε jako alternatywę zawsze będzie kończyć się powodzeniem.
Oczywiście, nie wszystkie funkcje dopasowujące kończą się powodzeniem w każdym przypadku. Przypuśćmy, że funkcja dopasowująca s akceptuje pojedynczą dziesiętną cyfrę, kod dla s może wyglądać jak następuje:

```
s           proc    near
            cmp     byte  ptr es:[di], '0'
            jb      s_fails
            cmp     byte  ptr es:[di], ,9'
            ja      s_fails
            inc     di
            stc
            ret
s_fails:    clc
            ret
s           endp
```

Jeśli NFA przybiera postać



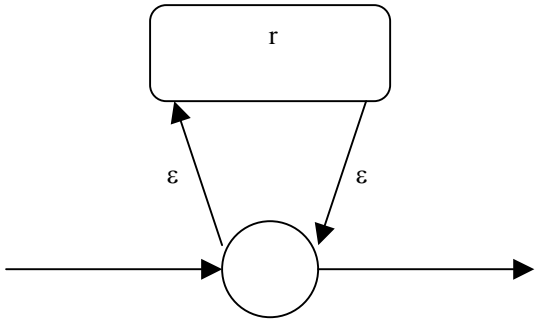
gdzie x jest przypadkowym znakiem lub ciągiem lub ε, odpowiedni kod asemblerowy dla tej procedury będzie

```
ConcatRxS   proc    near
            call    r
            jnc     CRxS_Fail                   ;jeśli  żadnego r, niepowodzenie
;Notka, jeśli x = ε wtedy po prostu usuwamy następujące trzy instrukcje. Jeśli x jest ciągiem zamiast
;pojedynczym znakiem, wkładamy dodatkowy kod dopasowujący wszystkie znaki w ciągu.
            cmp     byte  ptr es:[di], 'x'
            jne     CRxS_Fail
            inc     di

            call    s
            jnc     CRxS_Fail
            stc                                 ;Powodzenie!
            ret

CRxS_Fail:  clc
            ret
ConcatRxS   endp
```

Jeśli wyrażenie skończone jest w postaci r* a odpowiedni NFA ma postać



wtedy odpowiedni kod asemblerowy 80x86 może wyglądać jak coś takiego:

```
RStar       proc    near
            call    r
            jc      RStar
            stc
            ret
RStar       endp
```

Wyrażenie skończone oparte na Kleene Star zawsze kończy się powodzeniem ponieważ pozwala na zero lub więcej wystąpień. Jest tak dlatego, ze kod ten zawsze zwraca ustawioną flagę przeniesienia.
Operacja Kleene Plus jest tylko odrobinę bardziej złożone, odpowiedni (odrobinę zoptymalizowany) kod asemblerowy

```
Rplus       proc    near
            call    r
            jnc     Rplus_Fail
RplusLp:    call    r
            jc      RPlusLP
            stc
            ret

Rplus_Fail: clc
            ret
Rplus       endp
```

Odnotuj jak podprogram ten kończy się niepowodzeniem jeśli nie ma przynajmniej jednego wystąpienia r.

Ważnym problemem z backtracingiem jest to, że jest potencjalna niewydajność. Jest to bardzo łatwo stworzyć wyrażenie skończone, które, kiedy konwertujemy do NFA i kodu asemblerowego, generuje znaczne sprawdzenie wsteczne w pewnym ciąg wejściowym. Jest to później zaostrzone przez fakt, że podprogramy dopasowujące, jeśli są napisane jak opisano powyżej, są generalnie bardzo krótkie; tak krótkie, faktycznie, że procedura wywołująca i powrotna zajmują znaczną część czasu wykonania. Dlatego też, dopasowanie do wzorca w ten sposób, chociaż łatwe, może być wolniejsze niż może być.
To jest właśnie próba jak skonwertować WS do NFA do języka asemblera. Nie pójdziemy dalej po więcej szczegółów w tym rozdziale; nie dlatego że nie jest to interesujące, ale dlatego, że rzadko będziemy używali tej techniki w rzeczywistych programach. Jeśli potrzebujemy wysoko wydajnego dopasowania do wzorca, nie możemy używać technik niedeterministycznych , takich jak te. Jeśli chcemy łatwości programowania oferowanej przez konwersję NFA do asemblera nie używajmy tej techniki. Zamiast twego Biblioteka Standardowa UCR dostarcza bardzo silnych udogodnień dopasowania do wzorca (które przekraczają zdolności NFA),więc powinniśmy używać ich w zamian; ale więcej o tym później.

16.1.2.5 DETERMINISTYCZNE SKOŃCZONE STANY AUTOMATU (DFA)

Nie deterministyczny skończony stan automatu, kiedy konwertuje rzeczywisty kod programu, może cierpieć na problemy z wydajnością z powodu backtracingu, który wystąpi kiedy dopasujemy ciąg. Deterministyczny skończony stan automatu rozwiązuje ten problem przez porównanie różnych ciągów równolegle. Podczas gdy, w najgorszym przypadku, NFA może wymagać n porównań, gdzie n jest sumą długości wszystkich ciągów rozpoznawanych przez NFA, DFA wymaga tylko m porównań (najgorszy przypadek), gdzie m jest długością najdłuższego ciągu rozpoznawanego przez DFA.
Na przykład przypuśćmy, że mamy NFA, który dopasowuje następujące wyrażenia skończone (zbiór mnemoników trybu rzeczywistego 80x86, które zaczynają się na „A”):

(AAA | AAD | AAM | AAS | ADC | ADD | AND)

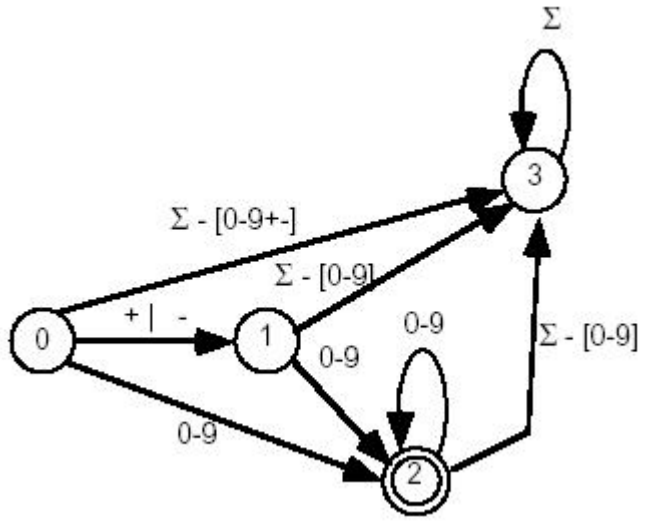
Typowa implementacja jako NFA może wyglądać następująco:

```
MatchAMnem: proc    near
```

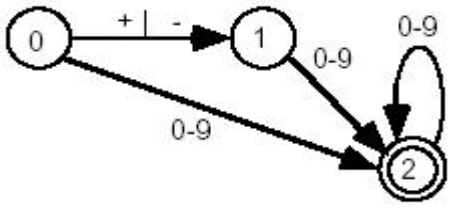
```
                strcmpl
                byte    „AAA”, 0
                je      matched
                strcmpl
                byte    „AAD”, 0
                je      matched
                strcmpl
                byte    „AAM”, 0
                je      matched
                strcmpl
                byte    „AAS”, 0
                je      matched
                strcmpl
                byte    „ADC”, 0
                je      matched
                strcmpl
                byte    „ADD”, 0
                je      matched
                strcmpl
                byte    „AND”, 0
                je      matched
                clc
                ret

matched:        add     di, 3
                stc
                ret
MatchAMnem      endp
```

Jeśli przekazujemy do NFA ciąg, który nie jest dopasowany np. „AAND” , musi wykonać siedem porównań ciągów, które +wykonuje około 18 porównań znaków (plus wszystkie koszty wywołania strcmpl). Faktycznie, DFA może określić, że nie dopasuje tego ciągu znaków przez porównywanie tylko trzech znaków



Rysunek 16.2 DFA dla Wyrażenia Skończonego (+ | - ε)$[0\text{-}9]^{+}$

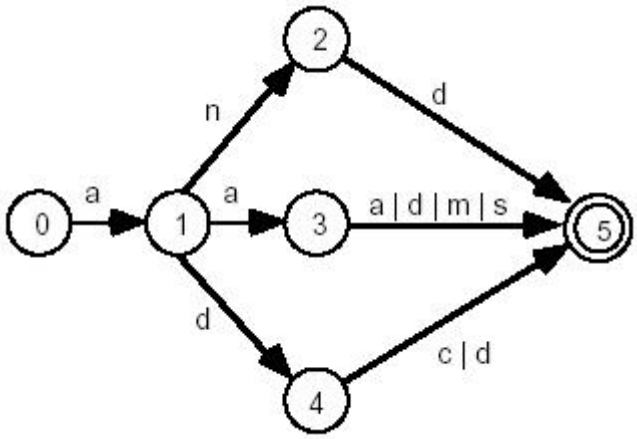


Rysunek 16.3 Uproszczone DFA dla Wyrażenia Skończonego (+ | - ε) $[0-9]^+$

DFA jest specjalną formą NFA z dwoma ograniczeniami. Po pierwsze, musi być dokładnie jeden brzeg wychodzący z każdego węzła dla każdego z możliwych znaków wejściowych; implikuje to , że musi jeden brzeg dla każdego symbolu wejściowego i nie może mieć dwóch brzegów z takimi samymi symbolami wejściowymi. Po drugie, nie możemy przesuwać z jednego stanu do innego w pustym ciągu ε. DFA jest deterministyczne ponieważ przy każdym stanie, następny wprowadzany symbol określa następny stan do jakiego będziemy wchodzili. Ponieważ każdy symbol wejściowy ma z nim powiązany brzeg, nigdy nie m przypadku, że DFA się 'zablokuje" ponieważ nie możemy opuścić stanu tego symbolu wejściowego. Podobnie, każdy nowy stan jaki wprowadzamy nie jest nigdy niejasny, ponieważ jest tylko jeden brzeg danego szczególnego stanu z bieżącym symbolem wejściowym na nim. Rysunek 16.2 pokazuje DFA, które działa na stałych całkowitych opisanych przez wyrażenie skończone

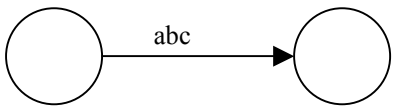
$$(+ \mid - \mid \varepsilon\ )\ [0-9]^+ .$$

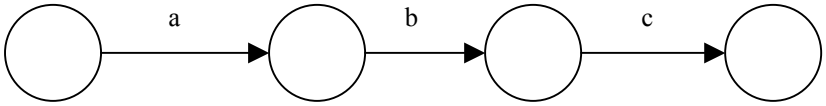
Zauważ, że wyrażenie w postaci „Σ - [0-9]" oznacza każdy znak z wyjątkiem cyfr, to znaczy kompletnego zbioru [0-9].
Stan trzy jest stanem niepowodzenia. To nie jest stan akceptowalny a DFA raz wchodzi w stan niepowodzenia, jest tam zablokowany (tj. konsumuje wszystkie dodatkowe znaki w ciągu wejściowym bez opuszczenia stanu niepowodzenia). Po wejściu do stanu niepowodzenia, DFA już odrzuciła ciąg wejściowy. Oczywiście, nie jest to jedyne sposób odrzucenia ciągu; powyższe DFA, na przykład, odrzuca pusty ciąg (ponieważ opuściliśmy stan zero) i odrzuca ciąg zawierający tylko znaki „+" lub „-„.
DFA generalnie zawiera więcej stanów niż porównywalne NFA. Pomocą w utrzymaniu rozmiaru DFA pod kontrolą, pozwolimy sobie na kilka skrótów, które, w żaden sposób, nie wpływają na działania DFA. Po pierwsze usuwamy ograniczenia, które były przy brzegach powiązanych z każdym możliwym symbolem wejściowym opuszczającym każdy stan. Większość brzegów opuszczających określony stan prowadzi do stanu niepowodzenia. Dlatego też naszym pierwszym uproszczeniem będzie zezwolenie DFA opuścić brzegi , które prowadza do stanu niepowodzenia. Jeśli symbol wejściowy nie jest reprezentowany na brzegu wychodzącym z jakiegoś stanu, założymy, że prowadzi on do stanu niepowodzenia. Powyższe DFA z tym uproszczeniem pojawia się na Rysunku 16.2



Rysunek 16.4 DFA, które rozpoznaje AND, AAA, AAD,AAM. AAS, AAD i ADC

Drugim skrótem, który jest obecny rzeczywiście w dwóch powyższych przykładach, jest zezwolenie zbiorowi znaków (lub symbolowi alternatywy) powiązać kilka znaków z pojedynczym brzegiem. W końcu, również pozwolimy ciągom przyłączyć się do brzegu. Jest to notacja skrótowa dla listy stanów, które rozpoznają pomyślnie każdy znak tj. dwa następujące DFA, są ekwiwalentami:





Wracając do wyrażenia skończonego, które rozpoznaje mnemoniki trybu rzeczywistego 80x86 zaczynające się na „A", możemy skonstruować DFA, które rozpoznaje takie ciągi jak pokazano na rysunku 16,4.
Jeśli przejdziemy przez to DFA z kilkom ciągami zaakceptowanymi i odrzuconymi , odkryjemy, ze wymaga on nie więcej niż sześć znaków porównania do określenia czy DFA powinna zaakceptować lub odrzucić ciąg wejściowy.
Chociaż nie będziemy omawiać tu specyfiki, okazuje się, że wyrażenia skończone. NFA i DFA są ekwiwalentami. To znaczy, możemy skonwertować coś z jednego do drugiego. W szczególności możemy zawsze skonwertować NFA do DFA. Chociaż konwersja nie jest całkowicie trywialna, zwłaszcza jeśli chcemy zoptymalizować DFA, zawsze jest to możliwe do zrobienia. Konwertowanie pomiędzy wszystkimi tymi formami oznaczałoby przekroczenie zakresu tego tekstu. Jeśli interesują cię szczegóły , każdy tekst o języku formalnym lub teorii automatów Ci ich dostarczy.

16.1.2.6 KONWERSJA DFA DO JĘZYKA ASSEMBLERA

Jest stosunkowo prosto skonwertować DFA do sekwencji instrukcji asemblerowych. Na przykład kod asemblerowy dla DFA, który akceptuje mnemoniki –A w poprzedniej sekcji

```
DFA_A_Mnem      proc    near
                cmp     byte  ptr  es:[di], 'A'
                jne     Fail
                cmp     byte  ptr  es:[di + 1], A'
                je      DoAA
                cmp     byte   ptr es:[di+1], ,D'
                je      DoAD
                cmp     byte   ptr  es:[di], ,N'
                je      DoAN
Fail:           clc
                ret

DoAN:           cmp     byte  ptr  es:[di+2], 'D'
                jne     Fail
Succeed:        add     di, 3
                stc
                ret
DoAD:           cmp     byte  ptr  es:[di+2], ,D'
                je      Succeed
                cmp     byte  ptr  es:d[di+2], ,C'
                je      Succeed
                clc                                     ;Zwracane niepowodzenie
                ret
DoAA:           cmp     byte  ptr  es:[di+2], 'A'
                je      Succeed
                cmp     byte  ptr  es:[di+2], ,D'
                je      Succeed
                cmp     byte  ptr  es:[di+2], ,M'
                je      Succeed
                cmp     byte  ptr  es:[di+2], ,S'
                je      Succeed
                clc
                ret
```

```
DFA_A_Mnem          endp
```

Chociaż ten schemat działa i jest znacznie bardziej wydajny niż kodowanie w schemacie NFA, napisanie tego kodu może być nużące, zwłaszcza kiedy konwertujemy duże DFA do kodu asemblerowego. Jest to technika, która czyni konwertowanie DFA do języka asemblera prawie trywialnym, chociaż może pochłonąć całkiem dużo miejsca – użyć stanów maszynowych. Prosty stan maszynowy jest dwu wymiarowa tablicą. Kolumny są indeksami możliwych znaków w ciągu wejściowym a wiersze są indeksami liczby stanów (tj. stanów w DFA).Każdy element tablicy jest nowa liczbą stanu. Algorytm dopasowujący dany ciąg używający stanu maszynowego jest trywialny:

```
state := 0;
while (inny znak wejściowy) do begin
ch := następny znak wejściowy;
state :=  StateTable [state] [ch];
end;
if (state in FinalStates) then accept
else  reject;
```

FinalStates jest zbiorem stanów akceptowalnych. Jeśli bieżąca liczba stanów jest w tym zbiorze po wyczerpaniu przez algorytm znaków w ciągu, wtedy stan maszynowy akceptuje ciąg, w przeciwnym razie odrzuca go.
Poniższa tablica stanów odpowiada DFA dla mnemoników „A” pojawiających się w poprzedniej sekcji:

Stan	A	C	D	M	N	S	Else
0	1	F	F	F	F	F	F
1	3	F	4	F	2	F	F
2	F	F	5	F	F	F	F
3	5	F	5	5	F	5	F
4	F	5	5	F	F	F	F
5	**F**	**F**	**F**	**F**	**F**	**F**	**F**
F	F	F	F	F	F	F	F

Tablica 62: Stany maszynowe dla Instrukcji “A” DFA 80x86

Stan pięć jedynie jest stanem akceptowalnym.
Jest tylko jeden ważny minus zastosowania tej tablicy schematów – tablica będzie całkiem duża. Nie jest to widoczne w powyższej tablicy ponieważ kolumna „Else;” ukrywa wiele szczegółów. W prawdziwej tablicy stanu będziemy potrzebowali jednej kolumny dla każdego możliwego znaku wejściowego, ponieważ jest 256 możliwych znaków wejściowych (lub przynajmniej 128 jeśli ograniczymy się do siedmiu bitów ASCII), powyższa tablica będzie miała 256 kolumn. Tylko jeden bajt na element, to daje około 2K dla tego małego stanu maszynowego. Duże stany maszynowe mogą generować bardzo duże tablice.
Jedyny sposób redukcji rozmiaru tablicy przy (bardzo) małej utracie szybkości wykonania jest klasyfikacja znaków przed użyciem ich jako indeksu w tablicy stanu. Przez użycie pojedynczej 256 bajtowej tablicy połączeń, łatwo jest zredukować stan maszynowy do powyższej tablicy. Rozważmy 256 bajtową tablicę połączeń, która zawiera:

- Jeden na pozycji Base +”a” i Base + „A”
- Dwa przy lokacji Base + ”c” i Base + „C”
- Trzy przy lokacji Base + „d” i Base + „D”
- Cztery przy lokacji Base + „m” i Base + „M”
- Pięć przy lokacji Base + „n” i Base + „N”
- Sześć przy lokacji Base + „s” i Base + „S”, i
- Zero wszędzie gdzie można

Teraz możemy zmodyfikować powyższą tabelę tworząc

Stan	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6

3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
6	6	6	6	6	6	6	6	6

Tabela 63 Tabela sklasyfikowanych stanów maszynowych dla instrukcji „A" DFA 80x86

Powyższa tabela zawiera dodatkową kolumnę „7", której nie będziemy używać. Powodem dodania dodatkowej kolumny jest uczynienie łatwiejszym indeksowanie w tej dwu wymiarowej tablicy (ponieważ ta dodatkowa kolumna pozwala nam mnożyć numer stanu przez osiem zamiast siedem).

Zakładając, że Classify jest nazwą tablicy połączeń, poniższy kod 80386 rozpoznaje ciągi określone przez to DFA:

```
DFA2_A_Mnem     proc    near
                push    ebx                     ;wskaźnik do Classify
                push    eax                     ;bieżący znak
                push    ecx                     ;bieżący stan
                xor     eax, eax                ;EAX := 0
                mov     ebx, eax                ;EBX := 0
                mov     ecx, eax                ;ECX (stan) := 0
                lea     bx, Classify
WhileNotEOS:    mov     al, es:[di]             ;pobranie następnego znaku wejściowego
                cmp     al., 0                  ;koniec ciągu?
                je      AtEOS
                xlat                            ;znak sklasyfikowany
                mov     cl, State_Tbl [eax+ecx*8] ;pobranie nowego stanu #
                inc     di                      ;przesuniecie na następny znak
                jmp     WhileNotEOS
AtEOS:          cmp     cl, 5                   ;czy w stanie akceptowalnym?
                stc                             ;zakładamy akceptację
                je      Accept
                clc
Accept:         pop     ecx
                pop     eax
                pop     ebx
                ret
DFA2_A_Mnem     endp
```

Chociaż używamy tabeli stanu w ten sposób upraszczając kodowanie asemblerowe, cierpimy z powodu dwóch wad. Po pierwsze, jak wspomniano wcześniej, jest to wolne. Technika ta musi wykonać wszystkie te instrukcje w całej pętli dla każdego znaku w dopasowaniu; a instrukcje te nie są szczególnie szybkie. Drugą wadą jest to ,że musimy stworzyć tablicę stanu dla stanu maszynowego; to przetwarzanie jest nużące i skłonne do błędów.
Jeśli potrzebujemy absolutnie wysokiej wydajności, możemy użyć technik stanu maszynowego opisanych w „Stany maszynowe i skoki pośrednie". Tu sztuczką jest przedstawianie każdego stanu jako krótkiego segmentu kodu i jego własną jedno wymiarową tablicą stanu. Każde wejście w tablicy jest adresem docelowym segmentu kodu przedstawiającego następny stan. Poniżej mamy przykład naszego stanu maszynowego „A Mnemonic" napisanego w ten sposób. Jedyną różnicą jest to, że bajt zero jest zaklasyfikowany jako wartość siódma (zero oznacza koniec ciągu, więc będziemy tego używać do określenia kiedy napotykamy koniec ciągu) Odpowiednia tablica stanu może być:

Stan	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6
3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**5**
6	6	6	6	6	6	6	6	6

Tabela 64 Inna tablica stanu maszynowego dla instrukcji „A” DFA 80x86

Kod 80x86 to

```
DFA3_A_Mnem     proc
                push    ebx
                push    eax
                push    ecx
                xor     eax, eax

                lea     ebx, Classify
State0:         mov     al., es:[di]
                xlat
                inc     di
                jmp     cseg: state0Tbl [eax*2]
State0Tbl       word    State6, State1, State6, State6
                word    State6, State6, State6, State6

State1:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg: State1Tbl [eax*2]
State1Tbl       word    State6, State3, State6, State4
                word    State6, State2, State6, State6
Statet2:        mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg: State2Tbl [eax*2]

State2Tbl       word    State6, State6.State6, State5
                word    State6, State6,State6, State6
State3:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg;State3Tbl [eax*2]
State3Tbl       word    Staet6,State5, State6, State5
                word    State5, State6, State5,State6
Staet4:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg: State4Tbl [eax*2]
State4Tbl       word    State6, State6, State5, State5
                word    State6, State6,State6,State6

State5:         mov  al, es:[di]
                cmp     al, 0
                jne     State6
                stc
                pop     ecx
                pop     eax
                pop     ebx
                ret
State6:         clc
                pop     ecx
                pop     eax
                pop     ebx
```

Są dwie ważne cechy które powinniśmy odnotować o tym kodzie. Po pierwsze wykonuje tylko cztery instrukcje na porównanie znaku (średnio mniej niż inne techniki). Po drugie, instancja DFA wykrywając niepowodzenia

zatrzymuje przetwarzanie znaków wejściowych. Inna tablica ukierunkowana przez technikę DFA na oślep przetwarza cały ciąg, nawet po tym jak jest oczywiste, że maszyna utknęła na stanie niepowodzenia.
Odnotujmy również, ze kod ten traktuje stany akceptowalne i niepowodzenia trochę inaczej niż ogólny kod tabeli stanu. Kod ten rozpoznaje fakt, że już jesteśmy w stanie pięć i albo zakończymy z powodzeniem (jeśli EOS jest następnym znakiem) lub niepowodzeniem. Podobnie w stanie sześć kod ten zna i nie próbuje szukać dalej.
Oczywiście ta technika nie jest łatwa do zmodyfikowania dla różnych DFA'ów jako prosta wersja tablicy stanów, ale jest trochę szybsza. Jeśli szukamy szybkości, jest to dobry powód do kodowania w DFA.

16.1.3 JĘZYK BEZKONTEKSTOWY
Języki bezkontekstowe dostarczają nadzbioru języków skończonych – jeśli możemy określić klasę wzorców z wyrażeniami skończonymi, możemy wyrazić taki sam język używając gramatyki bezkontekstowej. Dodatkowo możemy określić wiele języków, które nie są skończone używając gramatyki bezkontekstowej (CFG).
Przykłady języków, które są bezkontekstowe, ale nie skończone, zawierają zbiór wszystkich ciągów reprezentujących powszechne wyrażenia arytmetyczne, poprawne Pascalowe lub C pliki źródłowe i makra MASM. Języki bezkontekstowe są charakteryzowane przez zrównoważenie i zagnieżdżenie. Na przykład, wyrażenia arytmetyczne równoważy zbiór nawiasów okrągłych. Instrukcje języka wysokiego poziomu takie jak repeat ... until pozwalają na zagnieżdżanie i zawsze są zrównoważone (np. dla każdego repeat jest odpowiednia instrukcja until dalej w pliku źródłowym)
Jest tylko drobne rozszerzenie języka skończonego do działania na języku bezkontekstowym – wywołanie funkcji. W wyrażeniach skończonych, uznajemy tylko obiekty, które chcemy dopasować i określamy operatory WS takie jak „ | „„ „*", konkatenacje i tak dalej. Rozszerzając język skończony do języka bezkontekstowego potrzebujemy tylko dodać rekursywne funkcje wywołujące dla wyrażeń skończonych. Chociaż byłoby łatwe stworzenie składni pozwalającej na wywoływanie funkcji wewnątrz wyrażeń skończonych, informatyka używa zupełnie innej notacji dla języka bezkontekstowego – gramatykę bezkontekstową.
Gramatyka bezkontekstowa składa się z dwóch typów symboli: symboli terminalnych (kończących) i symboli nieterminalnych (pomocniczych). Symbole terminalne są pojedynczymi znakami i ciągami , które gramatyka bezkontekstowa dopasowuje plus ciąg pusty ε. Gramatyka bezkontekstowa używa symboli nieterminalnych dla wywołania funkcji i definicji. W naszej gramatyce bezkontekstowej używać będziemy kursywy do oznaczania symboli nieterminalnych i zwykłej czcionki do oznaczania symboli terminalnych.
Gramatyka bezkontekstowa składa się ze zbioru definicji funkcji znanych jako wyroby
Wyrób przybiera formę:

Nazwa_ funkcji $\rightarrow$ <lista terminalnych i nieterminalnych symboli>

Nazwa funkcji z lewej strony strzałki jest nazywana lewostronnym wyrobem. Treść funkcji, która jest listą symboli terminalnych i nieterminalnych, jest nazywana prawostronnym wyrobem. Poniżej mamy gramatykę dla prostych arytmetycznych wyrażeń:

expression $\rightarrow$ *expression + factor*
expression $\rightarrow$ *expression - factor*
expression $\rightarrow$ *factor*
factor $\rightarrow$ *factor * term*
factor $\rightarrow$ *factor / term*
factor $\rightarrow$ *term*
term $\rightarrow$ *IntegerConstant*
term $\rightarrow$ *(expression)*
IntegerConstant $\rightarrow$ *digit*
IntegerConstant $\rightarrow$ *digit IntegerConstant*
digit $\rightarrow$ *0*
digit $\rightarrow$ *1*
digit $\rightarrow$ *2*
digit $\rightarrow$ *3*
digit $\rightarrow$ *4*
digit $\rightarrow$ *5*
digit $\rightarrow$ *6*
digit $\rightarrow$ *7*
digit $\rightarrow$ *8*
digit $\rightarrow$ *9*

Zauważmy, że możemy mieć wielokrotne definicje dla tej samej funkcji. Gramatyka bezkontekstowa zachowuje się w trybie niedeterministycznym, tak jak NFA. Kiedy próbujemy dopasować ciąg używając gramatyki bezkontekstowej , ciąg jest dopasowany jeśli istnieje jakaś funkcja dopasowująca, która dopasowuje bieżący ciąg wejściowy. Ponieważ jest powszechne posiadanie wielu identycznych lewostronnych wyrobów, będziemy używali alternatywnych symboli z wyrażeniami skończonymi do redukcji liczby linii w gramatyce. Następujące dwie podgramatyki są identyczne:

expression → *expression* + *factor*
expression → *expression* - *factor*
expression → *factor*

Powyższe jest odpowiednikiem :

expression → *expression* + *factor* | *expression* → *expression* – *factor* | *factor*

Pełna gramatyka arytmetyczna, używająca tej notacji skrótowej to

expression → *expression* + *factor* | *expression* → *expression* - *factor* | *factor*
factor → *factor* * *term* | *factor* / *term* | *term*
term → *IntegerConstant* | *(expression)*
IntegerConstant → *digit* | *digit IntegerConstant*
digit → *0* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Jeden z nieterminalnych symboli, zazwyczaj pierwszy wyrób, jest symbolem startowym. Jest to mniej więcej odpowiednik stanu startowego w skończonym stanie automatu. Symbol startowy jest pierwszą funkcją dopasowującą jaką wywołujemy kiedy chcemy przetestować jakiś ciąg wejściowy aby zobaczyć czy jest składnikiem języka bezkontekstowego. W powyższym przykładzie expression jest symbolem startowym.

Podobnie jak NFA i DFA rozpoznaje ciągi w języku skończonym określonym przez wyrażenia skończone, niedeterministyczny automat stosowy i deterministyczny automat stosowy rozpoznają ciągi należące do języka bezkontekstowego określonego przez gramatykę bezkontekstową. Nie będziemy jednak wnikać w szczegóły automatów stosowych (lub PDA), ale musimy być świadomi ich obecności. Możemy dopasować ciągi bezpośrednio przez gramatykę. Na przykład rozważmy ciąg

7+5*(2+1)

Dopasowując ten ciąg, zaczniemy przez wywołanie funkcji symbolu startowego, expression, używając funkcji *expression* → *expression* + *factor*. Pierwszy znak plus sugeruje, że termin *expression* musi dopasować „7” a *factor* musi dopasować „5*|(2+1)”. Teraz musimy dopasować nasz ciąg wejściowy ze wzorcem *expression* + *factor* . Robimy to wywołując ponownie funkcję *expression*, tym razem używając wyrobu *expression* → *factor*. Daje to nam redukcję:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor*

Symbol ⇒ oznacza zastosowanie wywołania nieterminalnej funkcji (redukcji).

Następnie, wywołujemy funkcję *factor* używając wyrobu *factor* → *term* zwracającej redukcję:
expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor*

Kontynuując , wywołujemy funkcję *term* tworząc redukcję:
expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor*⇒ *IntegerConstant* + *factor*

Następnie wywołujemy funkcję *IntegerConstant* zwracając:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor*⇒ *term*+ *factor* ⇒ *IntegerConstant* + *factor*⇒ *7* + *factor*
W tym punkcie, pierwsze dwa symbole naszego generowanego ciągu dopasowują pierwsze dwa znaki ciągu wejściowego, więc możemy usunąć je z ciągu i skoncentrować na następnych pozycjach. Sukcesywnie, wywołujemy funkcję *factor* tworząc redukcję *7* + *factor** *term* a potem wywołujemy *factor, term* i *IntegerConstant* aby uzyskać 7+5 * *term*. W podobny sposób możemy zredukować termin „(*expression*)” i redukujemy wyrażenie „2+1”. Kompletne wyprowadzenie dla tego ciągu to

Expression ⇒ *expression* + *factor*

$\Rightarrow$ *factor + factor*
$\Rightarrow$ *term + factor*
$\Rightarrow$ *IntegerConstant + factor*
$\Rightarrow$ *7 + factor*
$\Rightarrow$ *7+factor * term*
$\Rightarrow$ *7 + term * term*
$\Rightarrow$ *7 + IntegerConstant 8 term*
$\Rightarrow$ *7 + 5 * term*
$\Rightarrow$ *7+5* (expression)*
$\Rightarrow$ *7+5* (expression + factor)*
$\Rightarrow$ *7+5*(factor + factor)*
$\Rightarrow$ *7 + 5* (IntegerConstant + factor)*
$\Rightarrow$ *7+5*(2 + factor)*
$\Rightarrow$ *7 + 5 *(2+ term)*
$\Rightarrow$ *7+5*(2 + IntegerConstant)*
$\Rightarrow$ *7+5*(2 +1)*

Kompletną końcową redukcję wyprowadzamy z naszego ciągu wejściowego , więc ciąg 7+5*(2+1) jest w języku określonym przez gramatykę bezkontekstową

16.1.4 ELIMINACJE LEWOSTRONNIE REKURENCYJNE I OPUSZCZANIE WSPÓŁCZYNNIKA CFG

W następnej sekcji będziemy omawiali jak skonwertować CFG do programu w języku asemblera. Jednakże, technika jakiej będziemy używali dla tej konwersji będzie wymagała zmodyfikowania pewnych gramatyk przed jej skonwertowaniem. Gramatyczne wyrażenia arytmetyczne w poprzedniej sekcji są dobrym przykładem takiej gramatyki – która jest lewostronnie rekurencyjna.

Gramatyka lewostronnie rekurencyjna stanowi dla nas problem ponieważ sposób w jaki będziemy zazwyczaj konwertować wyrób do kodu asemblowego, jest wywołanie funkcji zgodnej z nieterminalną i porównanie z symbolami terminalnymi. Jednakże przeciwdziałamy temu jeśli spróbujemy skonwertować wyrób używając tej techniki:

expression $\rightarrow$ *expression = factor*

Taka konwersja do kodu asemblerowego wyglądałaby podobnie do poniższego:

```
expression      proc    near
                call    expression
                jnc     fail
                cmp     byte  ptr  es:[di], '+'
                jne     fail
                inc     di
                call    factor
                jnc     fail
                stc
                ret
Fail:           clc
                Ret
expression      endp
```

Oczywisty problem z tym kodem jest taki, ze generuje pętlę nieskończoną. Na wejściu do funkcji expression kod ten bezpośrednio wywołuje expression rekurencyjnie, który bezpośrednio wywołuje expression rekurencyjnie, który bezpośrednio wywołuje expression rekurencyjnie, najwyraźniej musimy rozwiązać ten problem jeśli napiszemy rzeczywisty kod dopasowujący ten wyrób.

Sztuczka rozwiązująca rekurencję lewostronną jest tak, że jeśli jest wyrób , które cierpi z powodu rekurencji lewostronnej, musi być jakiś, taki sam lewostronny wyrób , który nie jest lewostronnie rekurencyjny. Wszystko co musimy zrobić to przepisać wywołanie lewostronnie rekurencyjne pod względem wyrobu, który nie ma żadnej rekurencji lewostronnej. To brzmi jak trudne zadanie, ale w rzeczywistości jest całkiem łatwe.

Zobaczmy jak wyeliminować rekurencję lewostronną, X_i i Y_i reprezentują jakiś zbiór symboli terminalnych lub nieterminalnych, które nie mają prawej strony zaczynającej się nieterminalnym A. Jeśli mamy jakieś wyroby w postaci:

$$A \rightarrow AX_1 \mid AX_2 \mid \ldots \mid AX_n \mid Y_1 \mid Y_2 \mid \ldots \mid Y_m .$$

Możemy przetłumaczyć to na odpowiednią gramatykę bez rekurencji lewostronnej przez zastąpienie każdego elementu w postaci $A \rightarrow Y_i$ przez $A \rightarrow Y_iA$ i każdy element w postaci $A \rightarrow AX_i$ przez $A' \rightarrow X_iA' \mid \varepsilon$. Na przykład, rozważmy trzy wyroby z gramatyki arytmetycznej:

expression $\rightarrow$ *expression + factor*
expression $\rightarrow$ *expression - factor*
expression $\rightarrow$ *factor*

W tym przykładzie A odpowiada *expression*, X_1 odpowiada „*+ factor"*, X_2 odpowiada *„- factor"* a Y_1 odpowiada „*factor"* .Odpowiednia gramatyka bez rekurencji lewostronnej

expression $\rightarrow$ *factor E'*
E' $\rightarrow$ *- factor E'*
E' $\rightarrow$ *+ factor E'*
E' $\rightarrow \varepsilon$

Kompletna gramatyka arytmetyczna z usuniętą rekurencją lewostronną to:

expression ' $\rightarrow$ *factor E'*
E' $\rightarrow$ *+ factor E'* | *- factor E'* | ε
factor $\rightarrow$ *term E'*
F' $\rightarrow$ ** term F'* | */ term F'* | ε
term $\rightarrow$ *IntegerConstant* | *(expression)*
IntegerConstant $\rightarrow$ *digit* | *digit IntegerConstant*
digit $\rightarrow$ *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

Inną użyteczną transformacją gramatyczną jest gramatyką opuszczania współczynnika**.** Może ona zredukować potrzebę backtracingu, poprawiając wydajność naszego kodu dopasowującego do wzorca. Rozważmy fragment następującego CFG:

stmt $\rightarrow$ *if expression then stmt endif*
stmt $\rightarrow$ *if expression then stmt else stmt endif*
Te dwa wyroby zaczynają się takim samym zbiorem symboli. I jeden i drugi wyrób będzie dopasowywał wszystkie znaki w instrukcji if do punktu dopasowującego napotkany algorytm else lub endif. Jeśli algorytm dopasowujący przetwarza pierwszą instrukcję do punktu z symbolem terminalnym endif i napotyka zamiast tego symbol terminalny else, musi wrócić do stanu początkowego całej drogi do symbolu if i zaczynać ponownie. Może to być strasznie nieefektywne z powodu rekurencyjnego wywołania stmt (wyobraź sobie 10000 lini programu, które mają pojedynczą instrukcję if we wszystkich 10000 linii, kompilator używając tej techniki dopasowania do wzorca będzie musiała zrekompilować cały program ze skreśleniem, jeśli używamy backtracingu w ten sposób). Jednakże poprzez gramatykę lewego współczynnika przed konwersją jej do kodu programu możemy wyeliminować potrzebę backtracingu.

W gramatyce lewego współczynnika zbieramy wszystkie wyroby, które maja taką samą lewą stronę i zaczynają z tymi samymi symbolami po prawej stronie. W tych dwóch powyższych wyrobach tymi symbolami są „if expression then stmt" . Łączymy ciągi w pojedynczy wyrób a potem dołączamy nowy symbol nieterminalny na koniec tego nowego wyrobu np.
stmt $\rightarrow$ *if expression then stmtNewNonTerm*

W końcu, tworzymy nowy zbiór wyrobów używając tego nowego nieterminala dla każdego z przyrostków wspólnego wyrobu:
NewNonTerm $\rightarrow$ *endif* | *else stmt endif*

Wyeliminuje to backtracing ponieważ algorytm dopasowania może przetwarzać if, expression, then I stmt przed tym nim wybierze endif I else.

16.1.5 KONWERSJA WS DO CFG

Ponieważ języki bezkonekstowe są nadzbiorem języków skończonych, nie powinno być niespodzianką ,ze jest możliwe konwertowanie wyrażeń skończonych do gramatyki bezkontekstowej. Jest to bardzo łatwy proces wymagający tylko kilku intuicyjnych zasad.

1) Jeśli wyrażenie skończone składa się z sekwencji znaków xyz, możemy łatwo stworzyć wyrób dla tego wyrażenia skończonego w postaci P $\rightarrow$ xyz. Odnosi się to równie do ciągu pustego ε.
2) Jeśli r i s są dwoma wyrażeniami skończonymi które konwertujemy do CFG tworząc wyroby R i S i mamy wyrażenie skończone rs które chcemy skonwertować do wyrobu, po prostu tworzymy nowy wyrób w postaci T $\rightarrow$ R S
3) Jeśli r i s są dwoma skończonymi wyrażeniami, które skonwertowaliśmy do CFG tworząc wyroby R i S, i mamy wyrażenie skończone r | s, które chcemy skonwertować do wyrobu, po prostu tworzymy nowy wyrób w postaci T $\rightarrow$ R | S
4) Jeśli r jest wyrażeniem skończonym , które skonwertowaliśmy tworząc wyrób R i chcemy stworzyć wyrób dla r* po prostu używamy wyrobu Rstar $\rightarrow$ R Rstar | ε
5) Jeśli r jest wyrażeniem skończonym, które skonwertowaliśmy tworząc wyrób R i chcemy stworzyć wyrób dla r^+, po używamy wyrób Rplus $\rightarrow$ R Rplus | R
6) Dla wyrażeń skończonych mamy operacje o różnych pierwszeństwach. Wyrażenia skończone również pozwalają na nawiasy okrągłe do przesłonięcia domyślnego pierwszeństwa. Ta notacja pierwszeństwa nie przenosi się na CFG. Zamiast tego musimy zakodować pierwszeństwo bezpośrednio w gramatyce. Na przykład kodując RS* prawdopodobnie użyjemy wyrobów w postaci:

   T $\rightarrow$ R SStar
   SStar $\rightarrow$ S SStar | ε

   Podobnie, działając na gramatyce w postaci (RS)* możemy użyć wyrobów w postaci:

   T $\rightarrow$ R S T | ε
   RS $\rightarrow$ R S

16.1.6 KONWERSJA CFG DO JĘZYKA ASEMBLERA

Jeśli mamy usuniętą lewostronną rekurencję i mamy gramatykę lewego współczynnika, łatwo jest skonwertować taką gramatykę do programu w języku asemblera, który rozpoznaje ciągi w języku bezkontekstowym.

Pierwszą konwencją jaką przyjmiemy jest to , że es:di zawsze wskazują początek ciągu jaki chcemy dopasować. Drugą konwencją jaką przyjmiemy jest stworzenie funkcji dla każdego nieterminala. Funkcja ta zwraca sukces (przeniesienie ustawione) jeśli dopasowała powiązany podwzorzec, zwraca niepowodzenie (przeniesienie wyzerowane) w przeciwnym razie. Jeśli wystąpiło powodzenie, pozostawia di wskazujące na następny znak, który jest startowy po dopasowaniu wzorca; jeśli mamy niepowodzenie, zachowuje wartość w di po wywołaniu funkcji.

Konwertując zbiór wyroby do odpowiedniego kodu asemblerowego, musimy zadziałać z czterema rzeczami ; symbolami terminalnymi, nieterminalnymi, sumą logiczną i pustym ciągiem. Najpierw rozpatrzymy proste funkcje (nieterminalne), które nie mają wielokrotnych wyrobów (tj. suma logiczna)

Jeśli wyrób przybiera postać T $\rightarrow$ ε i nie ma innych wyrobach powiązanych z T, wtedy ten wyrób zawsze kończy się powodzeniem. Odpowiedni kod asemblerowy:

```
T       proc    near
        stc
        ret
T       endp
```

Oczywiście nie ma rzeczywistej potrzeby zawsze wywoływać t i testować zwracanej wartości ponieważ wiemy,, że zawsze kończy się powodzeniem. Z drugiej strony, jeśli T jest stub, wtedy zamierzamy wprowadzić później, powinniśmy wywołać T

Jeśli wyrób przybiera postać T $\rightarrow$ xyz, gdzie xyz jest ciągiem z jednym lub więcej symboli terminalnych, wtedy funkcja zwraca powodzenie jeśli kilka następnych wprowadzanych znaków dopasowuje się do xyz, zwraca niepowodzenie w przeciwnym razie. Pamiętajmy, że jeśli przedrostek ciągu wejściowego dopasuje się do xyz, wtedy funkcja dopasowująca musi przesunąć di poza te znaki. Jeśli pierwsze znaki ciągu wejściowego nie są dopasowane do xyz, musi zachować di. Poniższy podprogram demonstruje dwa przypadki gdzie xyz jest pojedynczym znakiem i gdzie xyz jest ciągiem znaków:

```
T1          proc    near
            cmp     byte ptr es:[di], 'x'       ;pojedynczy znak
            je      Success
            clc                                 ;zwraca niepowodzenie
            ret
Success:    inc     di                          ;przeskok dopasowanego znaku
            stc                                 ;zwraca powodzenie
            ret
T1          endp

T2          proc    near
            call    MatchPrefix
            byte    'xyz' , 0
            ret
T2          endp
```

MatchPrefix jest podprogramem, który dopasowuje przedrostek ciągu wskazywanego przez es:di do ciągu następującego po wywołaniu w strumieniu kodu. Zwraca ustawione przeniesienie i modyfikuje di jeśli ciąg w strumieniu kodu jest przedrostkiem ciągu wejściowego, zwraca wyzerowaną flagę przeniesienia i zachowuje di jeśli ciąg literalny nie jest przedrostkiem wprowadzanym. Kod MatchPrefix jest następujący:

```
MatchPrefix     proc    far                 ;musi być far!
                push    bp
                mov     bp, sp
                push    ax
                push    ds
                push    si
                push    di

                lds     si, 2[bp]           ;pobranie adresu zwrotnego
CmpLoop:        mov     al., ds:[si]        ;pobranie ciągu do dopasowania
                cmp     al., 0              ;jeśli koniec przedrostka
                je      Success             ;mamy powodzenie
                cmp     al., es;[di]        ;zobacz czy dopasowany przedrostek
                jne     Failure             ;jeśli nie, bezpośrednio niepowodzenie
                inc     si
                inc     di
                jmp     CmpLoop
Success:        add     sp, 2               ;nie przywracaj di
                inc     si                  ;przeskakujemy bajt zakończony zerem
                mov     2[bp], si           ;zachowanie jako adresu zwrotnego
                pop     si
                pop     ds.
                pop     ax
                pop     bp
                stc                         ;zwraca powodzenie
                ret

Failure:        inc     si                  ;potrzeba skoku do bajtu zerowego
                cmp     byte ptr ds.:[si], 0
                jne     Failure
                inc     si
                mov     2[bp], si           ;zachowanie jako adres powrotu

                pop     di
                pop     si
                pop     ds.
                pop     ax
                pop     bp
                clc                         ;zwraca niepowodzenie
```

```
                ret
MatchPrefix     endp
```

Jeśli wyrób przybiera postać T → R gdzie R jest nieterminalne, wtedy funkcja T wywołuje R i zwraca jakikolwiek stan R zwraca np.

```
T       proc    near
        call    R
        ret
T       endp
```

Jeśli prawa strona wyrobu zawiera ciąg z symbolami terminalnymi i nieterminalnymi, odpowiedni kod asemblerowy sprawdza każdą pozycję po kolei . Jeśli każde sprawdzenie jest błędne, wtedy funkcja zwraca niepowodzenie. Jeśli wszystkie pozycje zakończyły się powodzeniem, wtedy funkcja zwraca powodzenie. Na przykład jeśli mamy wyrób w postaci T → R abc S możemy zaimplementować w języku asemblera

```
T       proc    near
        push    di                      ;jeśli błąd, musimy zachować di

        call    R
        jnc     Failure
        call    MatchPrefix
        byte    „abc”, 0
        jnc     Failure
        call    S
        jnc     Failure
        add     sp, 2                   ;nie zachowujemy di jeśli mamy powodzenie
        stc
        ret
Failure: pop    di
        clc
        ret
T       endp
```

Zobacz jak ten kod zachowuje di jeśli niepowodzenie, ale nie zachowuje jeśli powodzenie

Jeśli mamy wielokrotne wyroby takie same lewostronne (tj. suma logiczna), wtedy napisana właściwa funkcja dopasowującą dla wyrobów jest odrobinę bardziej złożona niż w przypadku pojedynczego wyrobu. Jeśli mamy wielokrotne wyroby powiązane z pojedynczym nieterminalem lewostronnym, wtedy tworzymy sekwencję kodu dopasowującą każdy pojedynczy wyrób. Połączymy je razem w pojedynczą funkcję dopasowującą, po prostu piszemy funkcję tak aby uzyskała powodzenie jeśli jedna z tych sekwencji kodu kończy się powodzeniem. Jeśli jeden z wyrobów jest w postaci T → e, wtedy testujemy drugi z warunków. Jeśli żaden z nich nie może być wybrany, funkcja kończy się powodzeniem. Na przykład rozważmy wyroby:

$$E' \rightarrow + \mathit{factor}\ E' \mid - \mathit{factor}\ E' \mid \varepsilon$$

Tłumaczymy go na następujący na kod asemblerowy

```
Eprime      proc    near
            push    di
            cmp     byte  ptr  es:[di]
            jne     TryMinus
            inc     di
            call    factor
            jnc     EP_Failed
            call    Eprime
            jnc     EP_Failed
Success:    add     sp, 2
            stc
            ret
TryMinus:   cmp     byte  ptr es:[di], ‘-‘
```

```
                jne     EP_Failed
                inc     di
                call    factor
                jnc     EP_Failed
                call    Eprime
                jnc     EP_Failed
                add     sp, 2
                stc
                ret
EP_Failed:      pop     di
                stc                             ;powodzenie ponieważ E' → ε
                ret
Eprime          endp
```

Ten podprogram zawsze kończy się powodzeniem ponieważ ma wyrób E' → ε. Jest tak dlatego, że instrukcja stc pojawia się po etykiecie EP_Failed

Wywołując funkcję dopasowania do wzorca, po prostu ładujemy es:di z adresem ciągu jaki chcemy przetestować i wywołujemy funkcję dopasowania do wzorca. Przy zwrocie, flaga przeniesienia będzie zawierała jeden jeśli dopasowano do wzorca ciąg do punktu zwracanego w di. Jeśli chcemy zobaczyć czy dopasowano cały ciąg do wzorca, po prostu sprawdzamy czy es:di wskazuje na bajt zero kiedy wracamy z funkcji wywołującej Jeśli chcemy zobaczyć czy ciąg należy do języka bezkontekstowego powinniśmy wywołać funkcję powiązaną z symbolem startowym dla danej gramatyki bezkontekstowej. Poniższy podprogram implementuje gramatykę arytmetyczną jakiej używaliśmy jako przykładów w kilku poprzednich sekcjach. Kompletna implementacja:

```
; ARTH.ASM
;
; Prosty rekurencyjny analizator dla gramatyki arytmetycznej

                .xlist
                include  stdlib.a
                include stdlib.lib
                .list

dseg            segment    para  public  'data'

; Gramatyka dla prostej gramatyki arytmetycznej ( wspiera  +, - , * , /):
;
; E → FE'
; E' → + F E' | - F E' | <ciąg pusty>
; F → TF'
; F' → * T F' | / T F' | <ciąg pusty>
; T → G | (E)
; G → H | H G
; H → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;

InputLine       byte      128  dup (0)

dseg            ends

cseg            segment   para   public   ,code'
                assume    cs: cseg, ds:dseg

; Funkcje dopasowujące dla gramatyki
; Funkcje te zwracają ustawioną flagę przeniesienia jeśli dopasowują swoje pozycje odpowiednio. Zwracają
; wyzerowaną flagę przeniesienia jeśli kończą się niepowodzeniem. Jeśli kończą się niepowodzeniem,
; zachowują di. Jeśli kończą się niepowodzeniem di wskazuje pierwszy znak po dopasowaniu.
```

```
; E → FE'

E               proc    near
                push    di
                call    F                       ;zobacz czy F, wtedy E' powodzenie
                jnc     E_Failed
                call    EPrime
                jnc     E_Failed
                add     sp, 2                   ,Powodzenie nie odtwarzamy di
                stc
                ret

E_Failed:       pop     di                              ;Niepowodzenie, musimy odtworzyć di
                clc
                ret
E               endp

; E' → F E' |  - F E' | ε

EPrime          proc    near
                push    di

; Próbujemy tu + F E'

                cmp     byte ptr es;[di], ,+'
                jne     TryMinus
                inc     di
                call    F
                jnc     EP_Failed
                call    EPrime
                jnc     EP_Failed
Success:        add     sp, 2
                stc
                ret

        ; Próbujemy tu  - F E'

TryMinus:       cmp     byte  ptr  es:[di], '-'
                jne     Success
                inc     di
                call    F
                jnc     EP_Failed
                call    EPrime
                jnc     EP_Failed
                add     sp, 2
                stc
                ret

; Jeśli żaden z powyższych nie zakończył się powodzeniem, zwraca sukces tak czy owak ponieważ mamy
; wyrób w postaci E' → ε

EP_Failed:      pop     di
                stc
                ret
EPrime          endp

; F → TF'

F               proc    near
                push    di
```

```
                call    T
                jnc     F_Failed
                call    FPrime
                jnc     F_Failed
                add     sp, 2                   ;powodzenie, nie przywracamy di
                stc
                ret

F_Failed:       pop     di
                clc
                ret
F               endp

; F → *T F' |  / T F' | ε

Fprime          proc    near
                push    di
                cmp     byte  ptr  es:[di], ,*'         ;zaczynamy z „*"?
                jne     TryDiv
                inc     di                              ;przeskakujemy „*"
                call    T
                jnc     FP_Failed
                call    Fprime
                jnc     FP_Failed
Success:        add     sp, 2
                stc
                ret

;Próbujemy  tu F → /T F'

TryDiv:         cmp     byte  ptr  es:[di], '/'         ;zaczynamy z „/" ?
                jne     Success                         ;powodzenie
                inc     di                              ;przeskakujemy „/"
                call    T
                jnc     FP_Failed
                call    FPrime
                jnc     FP_Failed
                add     sp, 2
                stc
                ret

; Jeśli powyższe oba są błędne, zwraca sukces ponieważ mamy wyrób w postaci F → ε

FP_Failed:      pop     di
                stc
                ret
Fprime          endp

; T → G | (E)

T               proc    near

;Próbujemy ty  T → G
                call    G
                jnc     TryParens
                ret

; Próbujemy tu T → (E)

Tryparens:      push    di                      ;zachowujemy jeśli błąd
```

```
                cmp     byte ptr  es:[di], '('       ;zaczynamy z „(„?
                jne     T_Failed                    ;błąd jeśli nie
                inc     di                          ;przeskakujemy znak „(„
                call    E
                jnc     T_Failed
                cmp     byte  ptr  es:[di], ')'     ;Koniec z „)"?
                jne     T_Failed                    ; błąd jeśli nie
                inc     di                          ;przeskakujemy „)"
                add     sp, 2                       ; nie przywracamy di  jeśli powodzenie
                stc
                ret

T_Failed:       pop     di
                clc
                ret
T               endp

; Poniżej jest swobodna translacja
;
; G → H | H G
; H→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;
; Podprogram ten sprawdza czy jest przynajmniej jedna cyfra. Jest błąd jeśli nie ma przynajmniej jeden cyfry;
; powodzenie i przeskok wszystkich cyfr jeśli jest jedna lub więcej cyfr

G               proc    near
                cmp     byte  ptr  es:[di], '0'     ;sprawdzamy obecność przynajmniej jednej cyfry
                jb      G_Failed
                cmp     byte  ptr  es:[di], '9'
                ja      G_Failed

DifitLoop:      inc     di                          ;przeskakujemy pozostałe znalezione cyfry
                cmp     byte  ptr  es:[di], '0'
                jb      G_Succeeds
                cmp     byte  ptr es:[di], '9'
                jbe     DigitLoop
G_Succeeds:     stc
                ret
G_Failed;       clc                                 ;błąd jeśli żadnych cyfr
                ret
G               endp

; Program główny  testuje powyższe funkcje dopasowujące  i demonstruje jak wywołać funkcje dopasowujące

Main            proc
                mov     ax, seg  dseg               ;ustawiany rejestr segmentowy
                mov     ds., ax
                mov     es, ax

                printf
                byte    „Wprowadź wyrażenie arytmetyczne: „, 0
                lesi    InputLine
                gets
                call    E
                jnc     BadExp

; Dobrze, ale czy jesteśmy na końcu  ciągu?
                cmp     byte  ptr  es:[di], '0
                jne     BadExp
;Okay, to naprawdę dobre wyrażenie w tym miejscu
```

```
                printf
                byte    „’%s’  jest poprawnym wyrażeniem”, cr, lf , 0
                dword   InputLine
                jmp     Quit

BadExp:         printf
                byte    „ ‘%s’ jest niepoprawnym wyrażeniem arytmetycznym”, cr, lf, 0
                dword   InputLine
Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack  ‘stack’
stk             byte     1024 dup (“stack”)
sseg            ends

zzzzzzseg       segment  para public  ‘zzzzzz’
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      main
```

16.1.7 KILKA KOŃCOWYCH UWAG NA TEMAT CFG

Techniki przedstawiane w tym rozdziale do konwersji CFG do kodu asemblerowego nie działa dla wszystkich CFG. Działają tylko na (dużych) nadzbiorach CFG znanych jako gramatyki LL(1). Kod który te techniki tworzy jest to rekurencyjne zmniejszanie składni predykcyjnej. Chociaż zbiór języków bezkontekstowych rozpoznawalnych przez gramatykę LL(1) jest nadzbiorem języków bezkontekstowych, jest bardzo duży nadzbiór i nie powinniśmy napotykać zbyt wiele różnic używając tej techniki.

Jedną ważna cechą analiz prognostycznych jest to ,że nie wymagają żadnego backtracingu. Jeśli jesteśmy zżyci z nieefektywnością związaną z backtracingu, łatwo można rozszerzyć rekurencyjne zmniejszanie składni do działania z każdym CFG. Zauważmy ,że kiedy używamy backtracingu , odchodzi przymiotnik predykcyjny, pozostajemy z systemem niedeterministycznym zamiast systemem deterministycznym (predykcyjny i deterministyczny są bardzo blisko znaczeniowo w tym przypadku)

Jest inny system CFG również LL(1). Tak zwany operator pierwszeństwa i LR(k) CFG ma dwa przykłady. Po więcej informacji o składni i gramatyce, skonsultuj z dobrym tekstem o teorii języka skończonego lub konstrukcji kompilatora.

16.18 POZA JĘZYKAMI BEZKONTEKSTOWYMI

Chociaż większość wzorców jakie będziemy chcieli prawdopodobnie przetwarzać będzie skończonych lub bezkontekstowych, może będzie czas kiedy musimy rozpoznać pewne typ wzorców, które są poza tymi dwoma (np. języki kontekstowe). Jak się okazuje, skończony stan automatu jest najprostszą maszyna; automat stosowy (który rozpoznaje języki bezkontekstowe) jest następnym krokiem. Po automacie stosowym, następnym krokiem jest maszyna Turinga. Jednakże maszyny Turinga mają odpowiedniki w silnym 80x86, więc dopasowanie do wzorca rozpoznawane przez maszyny Turinga nie różnią się od napisania zwykłego programu.

Kluczem do napisania funkcji, która rozpoznaje wzorce , które nie są bezkontekstowe jest zachowanie informacji w zmiennych i użycia tych zmiennych do decydowania który z kilku wyrobów chcemy użyć w danym czasie. Technika ta wprowadza kontekstowość. Takie techniki są bardzo użyteczne w programach sztucznej inteligencji (takie jak przetwarzanie języka naturalnego) gdzie niejasne rozwiązania zależą od przeszłej wiedzy lub bieżącego kontekstu operacji dopasowania do wzorca. Jednak stosowanie takich typów dopasowania do wzorca szybko wykroczy poza zakres tego tekstu o programowaniu w języku asemblera.

16.2 PODPROGRAMY DOPASOWANIA DO WZORCA STANDARDOWEJ BIBLIOTEKI UCR

Standardowa Biblioteka UCR dostarcza bardzo wyszukanego zbioru podprogramów dopasowania do wzorca Są one wzorowane na dopasowaniu do wzorca według SNOBOLA $, wspierającego CFG i dostarczającego w pełni automatycznego backtracingu , jeśli to konieczne. Co więcej , przez napisanie tylko pięciu instrukcji języka asemblera, możemy dopasować proste lub złożone wzorce.

Jest niewiele kodu asemblerowego kiedy używamy podprogramów dopasowania do wzorca Biblioteki Standardowej, ponieważ większość działań występuje w segmencie danych. Używając podprogramów dopasowania do wzorca, najpierw konstruujemy wzorcową strukturę danych w segmencie danych. Potem przekazujemy adres tego wzorca i ciągu jaki życzymy sobie przetestować do podprogramu Biblioteki Standardowej match. Podprogram match zwraca niepowodzenie lub powodzenie w zależności od stanu porównania. Nie jest to takie całkiem łatwe jak brzmi; nauczenie się jak konstruować wzorcowe struktury danych to prawie tak jak nauczyć się programowania w nowym języku .Na szczęście, jeśli przebrnęliśmy przez omówienie języków bezkontekstowych, nauczenie się tego nowego „języka" jest lekkie.

Wzorcowa struktura danych Biblioteki Standardowej przybiera następującą postać:

```
Pattern         struct
MatchFunction   dword   ?
MatchParm       dword   ?
MatchAlt        dword   ?
NextPattern     dword   ?
EndPattern      dword   ?
StartPattern    word    ?
StrSeg          word    ?
Pattern         ends
```

Pole MatchFunction zawiera adres podprogramu do wywołania, wykonującego jakąś część porównania. Powodzenie lub porażka tej funkcji określa czy dopasowano ciąg wejściowy. Na przykład Standardowa Biblioteka UCR dostarcza funkcji MatchStr, która porównuje jakiś n znakowy ciąg wejściowy z innym ciągiem znaków.

Pole MatchParm zawiera adres lub wartość parametru (jeśli odpowiedni) dla podprogramu MatchFunction. Na przykład, jeśli podprogramem MatchFunction jest MatchStr, wtedy pole MatchParm zawiera adres ciągu do porównania z wprowadzanymi znakami. Podobnie podprogram MatchChar porównuje kolejne znaki w ciągu z najmniej znaczącym bajtem pola MatchParm. Niektóre funkcje dopasowujące nie wymagają żadnych parametrów, będą ignorowały każdą wartość przypisaną do pola MatchParm. Przez konwencję większość programistów przechowuje zero w nieużywanych polach struktury Pattern.

Pole MatchAlt zawiera albo zero (NULL) albo adres jakiejś innej wzorcowej struktury danych. Jeśli aktualnie dopasowujemy znaki wejściowe, podprogramy dopasowania do wzorca ignorują to pole. Jednakże jeśli wzorzec jest błędnie dopasowany do ciągu wejściowego, wtedy podprogramy dopasowania do wzorca próbują dopasować wzorzec którego adres pojawia się w tym polu. Jeśli powiedzie się to alternatywne dopasowanie do wzorca, wtedy podprogram dopasowania do wzorca zwraca powodzenie do funkcji wywołującej, w przeciwnym razie zwraca niepowodzenie. Jeśli pole MatchAlt zawiera NULL, wtedy podprogram dopasowania do wzorca bezpośrednio zawodzi jeśli główny wzorzec jest nie dopasowany.

Struktura danych Pattern dopasowuje tylko jedną pozycję na przykład, może dopasować pojedynczy znak, pojedynczy ciąg lub znak ze zbioru znaków. Rzeczywisty wzorzec słowa będzie zawierał prawdopodobnie kilka mniejszych wzorców połączonych razem np. wzorzec dla identyfikatora Pascal składa się z pojedynczych znaków ze zbioru znaków alfabetycznych następującym po jednym lub więcej znaku ze zbioru [a-zA-Z0-9]. Pole NextPattern pozwala nam tworzyć łączny wzorzec jako połączenie dwóch pojedynczych wzorców. Dal takiego połączonego wzorca zwracającego powodzenie, bieżący wzorzec musi być dopasowany a potem wzorzec określony przez pole NextPattern również musi być dopasowany. Zauważmy, że możemy łączyć wiele wzorców razem jeśli używamy tego pola.

Ostatnie trzy pola EndPattern, StartPattern i StrSeg są do uzytku wewnętrznego podprogramu dopasowania do wzorca. Nie powinniśmy modyfikować ani analizować tych pól.

Jeśli już stworzyliśmy wzorzec, bardzo łatwo jest przetestować ciąg aby zobaczyć czy jest dopasowany do tego wzorca. Sekwencja wywołująca dla podprogramu Biblioteki Standardowej UCR match to

```
lesi    <Ciąg wejściowy do dopasowania >
ldxi    <Wzorzec dla dopasowywanego ciągu >
mov     cx, 0
match
jc      Success
```

Podprogram match Biblioteki Standardowej oczekuje wskaźnika do ciągu wejściowego w rejestrach es:di; oczekuje wskaźnika do wzorca jaki chcemy dopasować w parze rejestrów es:di. Rejestr cx powinien zawierać długość ciągu jaki chcemy przetestować. Jeśli cx zawiera, podprogram match będzie testował cały ciąg wejściowy. Jeśli cx zawiera wartość niezerową, podprogram match będzie tylko testował pierwsze znaki cx w

ciągu . Zauważmy ,że koniec ciągu (bajt zakończony zerem) nie może pojawić się w ciągu przed pozycją określoną w cx. Dla większości aplikacji, ładujemy cx zerem przed wywołaniem match jest najbardziej stosowną operacją.

Przy powrocie z podprogramu match, flaga przeniesienia oznacza powodzenie lub niepowodzenie. Jeśli flaga przeniesienia jest ustawiona, dopasowujemy ciąg do wzorca; jeśli flaga przeniesienia jest wyzerowana, wzorzec nie jest dopasowany do ciągu. W przeciwieństwie do przykładów podanych we wcześniejszych sekcjach, podprogram match nie modyfikuje rejestru di, nawet jeśli dopasowano pozytywnie. Zamiast tego zwraca pozycję niepowodzenie / powodzenie w rejestrze ax. Jest to pozycja pierwszego znaku po dopasowaniu jeśli match zwraca powodzenie, jest to pozycja pierwszego niedopasowanego znaku jeśli match zakończyło się niepowodzeniem.

16.3 FUNKCJE DOPASOWANIA DO WZORCA BIBLIOTEKI STANDARDOWEJ

Standardowa Biblioteka UCR dostarcza około 20 wbudowanych funkcji dopasowania do wzorca. Funkcje te są oparte na umiejętnościach dopasowania do wzorca dostarczanych przez język programowania SNOBOLA4, więc w rzeczywistości są bardzo silne! Prawdopodobnie odkryjemy , ze te podprogramy rozwiązują wszystkie nasze potrzeby dopasowania do wzorca, chociaż łatwo jest napisać własny podprogram dopasowania do wzorca (zobacz „Projektowanie Własnych Podprogramów Dopasowania Do Wzorca”) jeśli żaden z nich nie jest odpowiedni. Poniższe subsekcje opisują każdy z tych podprogramów dopasowania do wzorca szczegółowo.

Są dwie rzeczy jakie powinniśmy odnotować jeśli używamy pliku SHELL.ASM Biblioteki Standardowej kiedy tworzymy programy, które używają dopasowania do wzorca i zbiorów znaków. Po pierwsze jest linia na samym początku pliku SHELL.ASM, która zawiera instrukcję „matchfuncs”. Linia ta jest w rzeczywistości komentarzem ponieważ zawiera średnik w kolumnie jeden. Jeśli mamy zamiar używać zdolności dopasowania do wzorca Biblioteki Standardowej, musimy odkomentować tą linię przez usunięcie średnika z kolumny jeden. Jeśli będziemy chcieli skorzystać z zdolności zbioru znaków Biblioteki Standardowej UCR (bardzo popularne jeśli używamy udogodnień dopasowania do wzorca) możemy chcieć odkomentować linię zawierającą „include stdsets.a” w segmencie danych. Plik „stdsets.a” zawiera kilka popularnych zbiorów znaków, wliczając w to alfabetyczny, cyfrowy, alfanumeryczny, białych znaków i tak dalej.

16.3.1 SPANCSET

Podprogram spancset przeskakuje wszystkie znaki należące do zbioru znaków. Ten podprogram bezie dopasowywał zero lub więcej znaków w określonym zbiorze i ,dlatego też, zawsze kończy się powodzeniem. Pole MatchParm we wzorcowej strukturze danych musi wskazywać zmienną zbioru znaku Biblioteki Standardowej UCR.

Przykład:

```
SkipAlphas      pattern   {spanceset , alpha}
                -
                -
                -
                lesi      StringWAlphas
                ldxi      SkipAlphas
                xor       cx, cx
                match
```

16.3.2 BRKCSET

Brkcset jest przeciwne do spancset – dopasowuje zero lub więcej znaków w ciągu wejściowym, które nie są składnikami określonego zbioru znaków. Innymi słowy, brkcset będzie dopasowywał wszystkie znaki w ciągu wejściowym do znaku w określonym zbiorze znaków (lub końca ciągu). Pole matchparm zawiera adres zbioru znaków do dopasowania.

Przykład:

```
DoDigits        pattern  {brkcset, digits, 0 , DoDigits2}
DoDigits2       pattern   {spancset, digits}
                -
                -
                -
                lesi      StringWDigits
```

```
        ldxi    DoDigits
        xor     cx, cx
        match
        jnc     NoDigits
```

Powyższy kod dopasowuje jakiś ciąg, który zawiera ciąg z jedną lub więcej cyfr gdzieś w ciągu.

16.3.3 ANYCSET

Anycset dopasowuje pojedynczy znak w ciągu wejściowym ze zbioru znaków. Pole matchparm zawiera adres zmiennej zbioru znaków. Jeśli kolejny znak w ciągu wejściowym jest składnikiem tego ciągu, anycset ustawia akceptację ciągu i przeskakuje ten znak. Jeśli kolejny wprowadzany znak nie jest składnikiem tego zbioru, anycset zwraca niepowodzenie.
Przykład:

```
DoID    pattern  {anycset, alpha, 0, DOID2}
DoID    pattern  {spancset, alphanum}
        -
        -
        -
        lesi     StringWID
        ldxi     DoID
        xor      cx, cx
        match
        jnc      NoID
```

Ten fragment kodu sprawdza ciąg StirngWID aby zobaczyć czy zaczyna się identyfikatorem określonym przez wyrażenie skończone[a-zA-Z][a-zA-Z0-9]*. Pierwszy pod wzorzec z anycset upewnia się ,że jest znak alfabetyczny na początku ciągu (alpha jest ustawianą zmienną stdsets.a, która ma jako składniki wszystkie znaki alfabetu) Jeśli ciąg nie zaczyna się znakiem alfabetu, wzorzec DoID to niepowodzenie. Drugi podwzorzec DoID2 przeskakuje każdy kolejny znak alfanumeryczny używający funkcji dopasowującej spancset. Zauważmy, że spancset zawsze kończy się powodzeniem.
Powyższy kod po prostu nie dopasowuje ciągu, który jest identyfikatorem; dopasowuje ciąg, który zaczyna się poprawnym identyfikatorem. Na przykład, dopasowując „hisIsAnID” z „thisISAnID+SolThis-5”. Jeśli chcemy tylko dopasować pojedynczy identyfikator i nic więcej, musimy wyraźnie sprawdzić koniec ciągu w naszym wzorcu.

16.3.4 NOTANYCSET

Notanycset dostarcza uzupełnienia do anycset - dopasowuje pojedynczy znak w ciągu wejściowym , który nie jest składnikiem zbioru znaków. Pole matchparm, jak zwykle, zawiera adres zbioru znaków, którego składniki, nie muszą pojawiać się jako kolejne znaki w ciągu wejściowym. Jeśli notanycset pomyślnie dopasuje znak(to znaczy kolejny znak wprowadzony nie jest w wyznaczonym zbiorze znaków), funkcja przeskakuje znak i zwraca powodzenie; w przeciwnym razie zwraca niepowodzenie.
Przykład:

```
DoSpecial       pattern  {notanycset, digits, 0, DoSpecial2}
DoSpecial2      pattern  {spancset, alphanum}
                -
                -
                -
                lesi     StringWSpecial
                ldxi     DOSpecial
                xor      cx, cx
                match
                jnc      NoSpecial
```

Kod ten jest podobny do wzorca DoID w poprzednim przykładzie. Dopasowuje ciąg zawierający jakiś znak z wyjątkiem cyfr a potem dopasowuje ciąg znaków alfanumerycznych

---

16.3.5 MATCHSTR

Matchstr porównuje kolejne zbiór znaków wejściowych z ciągiem znaków. Pole matchparm zawiera adres ciągu zakończonego zerem do porównania. Jeśli matchparm kończy się powodzeniem, zwraca ustawioną flagę przeniesienia i przeskakuje znaki, które dopasowano; jeśli kończy niepowodzeniem, próbuje alternatywnej funkcji dopasowującej lub zwraca niepowodzenie jeśli nie ma alternatywy.
Przykład:

```
DoSting     pattern  {matchstr, MyStr}
MyStr       byte     “Match this!”, 0
            -
            -
            -
            lesi     String
            ldxi     DoString
            xor      cx, cx
            match
            jnc      NotMatchThis
```

Ten przykładowy kod dopasowuje ciąg, który zaczyna się znakami „Match This!”

16.3.6 MATCHISTR

Matchistr jest podobny do matchstr na tyle , że porównuje kolejnych kilka znaków z wartością ciągu zakończonego zerem. Jednakże, matchistr robi porównanie bez rozróżniania małych i dużych liter, Podczas porównania konwertuje znaki w ciągu wejściowym do dużych liter przed ich porównaniem za znakami, na które wskazuje pole matchparm. Dlatego też, ciąg wskazywany przez pole matchparm musi zawierać duże litery gdziekolwiek pojawiają się znaki alfabetu. Jeśli ciąg matchparm zawiera jakieś małe znaki, funkcja matchistr będzie zawsze błędne.
Przykłady:

```
DoString    pattern  {matchistr, Mystr}
MyStr       byte     “Match THIS!”, 0
            -
            -
            -
            lesi     String
            ldxi     DoString
            xor      cx,cx
            match
            jnc      NotMatchThis
```

Ten przykład jest identyczny do jednego z poprzednich sekcji z wyjątkiem tego, że bezie dopasowywał znaki “match this!” używając kombinacji dużych i małych liter.

16.3.7 MATCHTOSTR

Matchtostr dopasowuje wszystkie znaki w ciągu wejściowym w tym znaki określone przez parametr matchparm. Ten podprogram kończy się powodzeniem jeśli określony ciąg pojawia się gdzieś w ciągu wejściowym , kończy niepowodzeniem jeśli ciąg nie pojawia się w ciągu wejściowym. Ta funkcja wzorcowa jest całkiem użyteczna dla lokacji podciągu i ignorowania wszystkiego co przyszło przed podciągiem.
Przykład:

```
DoString    pattern  {matchtostr, MyStr}
MyStr       byte     :match this!”, 0
            -
            -
            -
            lesi     String
            ldxi     DoString
            xor      cx, cx
            match
            jnc      NotMatchThis
```

Podobnie jak poprzednie dwa przykłady, ten fragment kodu dopasowuje ciąg „Match This!”. Jednakże nie jest wymagane aby ciąg wejściowy (String) zaczynał się „Match this!”. Zamiast tego wymagane jest tylko, aby „Match this!” pojawiło się gdzieś w ciągu.

16.3.8 MATCHCHAR

Funkcja matchchar dopasowuje pojedynczy znak. Najmniej znaczący bajt pola matchchar zawiera znak jaki chcemy dopasować . Jeśli kolejny znak w ciągu wejściowym jest tym znakiem, wtedy ta funkcja kończy się powodzeniem, w przeciwnym razie kończy się niepowodzeniem.
Przykład:

```
DoSpace     pattern  {matchchar, ‘ ‘}
            -
            -
            -
            lesi     String
            ldxi     DoSpace
            xor      cx, cx
            match
            jnc      NoSpace
```

Ten fragment kodu dopasowuje każdy ciąg, który zaczyna się spacją. Zapamiętajmy, że podprogram match sprawdza tylko przedrostek ciągu. Jeśli chcielibyśmy zobaczyć czy ciąg zawierał tylko spacje (zamiast ciągu który zaczyna się spacją) będziemy musieli wyraźnie sprawdzić koniec ciągu po spacji. Oczywiście, byłoby dużo bardziej wydajne zastosowanie strcmp zamiast match do tego celu!

Zauważ, że w odróżnieniu od matchstr, kodujemy znak jaki chcemy dopasować bezpośrednio w polu matchparm. To pozwala nam określić znak jaki chcemy przetestować bezpośrednio w definicji wzorca.

16.3.9 MATCHTOCHAR

Podobnie jak matchtostr, matchtochar dopasowuje wszystkie znaki wliczając w to znak jaki określiliśmy. Jest podobna do brkcset z wyjątkiem tego, że musimy tworzyć zbioru znaków zawierającego pojedynczego składnika i brkcset skacze ale nie do wliczonego, określonego znaku(ów). Matchtochar kończy się niepowodzeniem jeśli nie można znaleźć określonego znaku w ciągu wejściowego
Przykład:

```
DoToSpace   pattern  {matchtochar, ‘ ‘)
            -
            -
            -
            lesi     String
            ldxi     DoSpace
            xor      cx, cx
            match
            jnc      NoSpace
```

To wywołanie match skończy się niepowodzeniem jeśli nie ma spacji w ciągu wejściowym. Jeśli są wywołanie matchtochar przeskoczy wszystkie znaki do pierwszej spacji. Jest to użyteczny wzorzec dla przeskakiwania nad słowami w ciągu.

16.3.10 MATCHCHARS

Matchchars pomija zero lub więcej wystąpień pojedynczego znaku w ciągu wejściowym. Jest to podobne do spancset z wyjątkiem tego, że możemy określić pojedynczy znak zamiast całego zbioru znaków z pojedynczym składnikiem. Podobnie jak matchchar, matchchars oczekuje pojedynczego znaku w najmniej znaczącym bajcie pola matchparm. Ponieważ ten podprogram dopasowuje zero lub więcej wystąpień tego znaku, zawsze kończy się powodzeniem.
Przykład:

```
Skip2NextWord       pattern  {matchchars, ‘ ‘ , 0 , SkipSpcs}
SkipSpcs            pattern  {matchchars, ‘ ‘ }
                    -
                    -
```

```
                -
                lesi    String
                ldxi    Skip2NextWord
                xor     cx, cx
                match
                jnc     NoWord
```

Ten fragment kodu skacze do początku następnego słowa w ciągu. Kończy się niepowodzeniem jeśli nie ma dodatkowego słowa w ciągu (tj. ciąg nie zawiera spacji)

16.3.11 MATCHTOPAT

Matchtopat dopasowuje wszystkie znaki w ciągu w tym podciągi dopasowywane przez jakieś inne wzorce. Jest to jeden z dwóch podprogramów dopasowania do wzorca Biblioteki Standardowej UCR dostarczonej aby pozwolić na implementację wywołania funkcji nieterminalnej. Ta funkcja dopasowująca kończy się pozwodzeniem jeśli znajduje dopasowywany ciąg określony wzorcem gdzieś w linii. Jeśli kończy się powodzeniem pomija znaki po ostatnim znaku dopasowanym przez parametr pattern. Jak można oczekiwać, pole matchparm zawiera adres wzorca do dopasowania
Przykład:

```
; Zakładamy, że jest wzorzec „expression”, który dopasowuje wyrażenia arytmetyczne. Poniższy wzorzec
;określa czy  jest takie wyrażenie po którym następuje średnik

FindExp         pattern   {matchtopat,  expression, 0 , matchSemi}
MatchSemi       pattern   {matchchar, ‘;’}
                -
                -
                -
                lesi      String
                ldxi      FindExp
                xor       cx, cx
                match
                jnc       NoExp
```

13.3.12 EOS

EOS dopasowuje wzorzec końca ciągu. Ten wzorzec , który musi oczywiście pojawić się na końcu listy wzorca, jeśli pojawia się w ogóle., sprawdza bajt zakończony zerem. Ponieważ podprogramy Biblioteki Standardowej dopasowują tylko przedrostki, powinniśmy wstawić ten wzorzec na koniec listy jeśli chcemy zapewnić, że wzorzec dokładnie dopasowuje ciąg bez żadnych resztek znaków na końcu. EOS kończy się powodzeniem jeśli dopasowuje bajt zakończony zerem, niepowodzeniem w przeciwnym razie.

Przykład:

```
SkipNumber      pattern   {anycset, digits, 0, SkipDigits}
SkipDigits      pattern   {spancset, digits, 0 , EOSPat}
EOSPat          pattern   {EOS}
                -
                -
                -
                lesi      String
                ldxi      SkipNumber
                xor       cx, cx
                match
                jnc       NoNumber
```

SkipNumber dopasowuje ciąg wzorcowy, który zawiera tylko cyfry dziesiętne (od początku dopasowania do końca ciągu) Zauważ, że EOS nie wymaga parametrów, nawet parametru matchparm.

16.3.13 ARB

ARB dopasowuje liczbę dowolnych znaków. Ta funkcja dopasowania do wzorca jest odpowiednikiem Σ*. Zauważmy, że ARB jest bardzo niewydajnym podprogramem w użyciu. Działa przy założeniu, że można dopasować wszystkie pozostałe znaki w ciągu a potem próbować dopasować wzorzec określony przez pole nextpattern. Jeśli pozycja nextpattern kończy się niepowodzeniem , ARB wraca jeden znak i próbuje dopasować nextpattern ponownie. Jest to kontynuowane dopóki wzorzec określony przez nextpattern nie zakończy się sukcesem lub ARB wróci do swojej początkowej pozycji startowej. ABC kończy się powodzeniem, jeśli wzorzec określony przez nextpattern kończy się powodzeniem, niepowodzeniem, jeśli wraca do swojego punktu startowego.

Daje to ogromna ilość backtracingu, który może wystąpić z ARB (zwłaszcza przy długich ciągach), więc powinniśmy próbować unikać takich wzorców jeśli to możliwe. Funkcje matchtostr, matchtochar i matchtopat realizują więcej niż może zrealizować ARB, działają one w przód zamiast w tył w ciągu źródłowym i mogą być bardziej wydajne. ARB jest użyteczna głównie jeśli jesteśmy pewni, że kolejny wzorzec pojawi się później w ciągu jaki dopasowujemy lub jeśli ciąg jaki chcemy dopasować wystąpi kilka razy i chcemy dopasować ostatnie wystąpienie (matchtostr, matchtochar i matchtopat zawsze dopasowują pierwszy wystąpienie jakie znajdą).
Przykład:

```
SkipNumber      pattern   {ARB,0, 0, SkipDigit}
SkipDigit       pattern   {anycset, digits, 0 , SkipDigits}
SkipDigits      pattern   {spancset, digits}
                -
                -
                -
                lesi      String
                ldxi      SkipNumber
                xor       cx, cx
                match
                jnc       NoNumber
```

Ten przykładowy kod dopasowuje ostatnią liczbę, która pojawia się w linii wejściowej. Zauważmy, że ARB nie używa pola matchparm, więc powinniśmy go ustawić domyślnie na zero.

16.3.14 ARBNUM

ARBNUM dopasowuje dowolną liczbę (zero lub więcej) wzorców, które występują w ciągu wejściowym. Jeśli R przedstawia jakąś nieterminalną liczbę (funkcja dopasowania do wzorca) wtedy ARBNUMR jest odpowiednikiem wyrobu ARBNUM $\rightarrow$ R ARBNUM | ε.

Pole matchparm zawiera adres wzorca, który ARBNUM próbuje dopasować.
Przykład:

```
SkipNumbers     pattern   {ARBNUM, SkipNumber}
SkipNumber      pattern   {anycset, digits, 0 ,SkipDigits}
SkipDigits      pattern   {spancset, digits, 0 , EndDigits}
EndDigits       pattern   {matchchars, , , , EndString}
EndString       pattern   {EOS}
                -
                -
                -
                lesi      String
                ldxi      SkipNumbers
                xor       cx, cx
                match
                jnc       IllegalNumbers
```

Kod ten akceptuje ciąg wejściowy jeśli składa się z sekwencji zera lub więcej liczb oddzielonych spacjami i zakończonych wzorcem EOS. Odnotujmy użycie pola matchalt we wzorcu EndDigits do wyboru EOS zamiast spacji dla ostatniej liczby w ciągu.

16.3.15 SKIP

Skip dopasowuje n dowolnych znaków w ciągu wejściowym .Pole matchparm jest wartością całkowitą zawierającą liczbę znaków do przeskoczenia. Chociaż pole matchparm jest podwójnym słowem , podprogram ten ogranicza liczbę znaków do przeskoczenia do 16 bitów (65,535 znaków); to znaczy, n jest najmniej znaczącym słowem w polu matchparm. Powinno to udowodnić swoją przydatność w wielu potrzebach.

Skip kończy się powodzeniem, jeśli jest przynajmniej n znaków pominiętych w ciągu wejściowym; niepowodzeniem jeśli jest mniej niż n znaków pominiętych w ciągu wejściowym.

Przykład:

```
Skiplst16       pattern   {skip, 6, 0, SkipNumber}
SkipNumber      pattern   {anycset, digits, 0 , SkipDigits}
SkipDIgits      pattern   {spancset, digits, 0, EndDigits}
EndDigits       pattern   {EOS}
                -
                -
                -
                lesi      String
                ldxi      Skiplst6
                xor       cx, cx
                match
                jnc       IllegalItem
```

To przykład dopasowania ciągu zawierającego sześć dowolnych znaków po których następuje jedna lub więcej cyfr i bajt zakończony zerem.

16.3.16 POS

Pos kończy się powodzeniem jeśli funkcje dopasowujące są obecnie przy n-tym znaku w ciągu, gdzie n jest wartością w najmniej znaczącym słowie pola matchparm. Pos kończy się niepowodzeniem jeśli funkcja dopasowująca nie jest obecnie na pozycji n w ciągu. W odróżnieniu od innych funkcji dopasowujących, pos nie pochłania znaków wejściowych. Zauważmy, że ciąg zaczyna się od pozycji zero. Więc kiedy używamy funkcji pos , kończy się powodzeniem jeśli dopasowaliśmy n znaków w tym punkcie.

Przykład:

```
SkipNumber      pattern   {anycset, digits, 0, SkipDigits}
SkipDigits      pattern   {spancset, digits, 0 , EndDigits}
EndDigits       pattern   {pos, 4}
                -
                -
                -
                lesi      String
                ldxi      SkipNumber
                xor       cx, cx
                match
                jnc       IllegalItem
```

Kod ten dopasowuje ciąg, który zaczyna się dokładnie 4 cyframi dziesiętnymi .

16.3.7 RPOS

Rpos działa podobnie jak funkcja pos z wyjątkiem tego, że kończy się powodzeniem jeśli bieżąca pozycja jest pozycją n znaku z końca ciągu. Podobnie jak w pos, n jest 16, najmniej znaczącymi bitami pola matchparm. Również jak w pos, rpos nie pochłania znaków wejściowych
Przykład:

```
SkipNumber      pattern   {anycset, digits, 0 , SkipDigits}
SkipDigits      pattern   {spancset, digits, 0 , EndDigits}
EndDigits       pattern   {rpos, 4}
                -
```

```
                -
                -
                lesi    String
                ldxi    SkipNumber
                xor     cx, cx
                match
                jnc     IllegalItem
```

Kod ten dopasowuje jakiś ciąg, który jest cały z cyfr dziesiętnych, z wyjątkiem ostatnich czterech znaków ciągu. Ciąg musi być długi przynajmniej na pięć znaków , aby powyższe dopasowanie do wzorca zakończyło się powodzeniem.

16.3.18 GOTOPOS

Gotopos skacze ponad znakami w ciągu dopóki nie osiągnie pozycji znaku n w ciągu. Funkcja ta zawodzi jeśli wzorzec jest już poza pozycją n w ciągu. Najmniej znaczące słowo pola matchparm zawiera wartość dla n.

Przykład:

```
SkipNumber      pattern {gotopos, 10, 0, MatchNmbr}
MatchNmbr       pattern {anycset, digits, 0, SkipDigits}
SkipDigits      pattern {spancset, digits,0 , EndDigits}
EndDigits       pattern {rpos, 4}
                -
                -
                -
                lesi    String
                ldxi    SkipNumber
                xor     cx, cx
                match
                jnc     IllegalItem
```

Ten przykładowy kod skacze do pozycji 10 w ciągu i próbuje dopasować ciąg cyfr zaczynając od znaku jedenastego. Ten wzorzec kończy się powodzeniem jeśli pozostały cztery znaki po przetworzeniu wszystkich cyfr.

16.3.19 RGOTOPOS

Rgotopos działa podobnie jak gotopos z wyjątkiem tego, że idzie do pozycji określonej na końcu ciągu. Rgotopos kończy się niepowodzeniem jeśli podprogram dopasowujący jest już poza pozycją n z końca ciągu. Podobnie jak przy gotopos, najmniej znaczące słowo pola matchparm zawiera wartość dla n

Przykład:

```
SkipNumber      pattern {rgotopos, 10, 0, MatchNmbr}
MatchNmbr       pattern {anycset, digits, 0 , SkipDigits}
SkipDigits      pattern {spancset, digits}
                -
                -
                -
                lesi    String
                ldxi    SkipNumber
                xor     cx,  cx
                match
                jnc     IllegalItem
```

Ten przykład skacze do dziesiątego znaku z końca ciągu a potem próbuje dopasować jedną lub więcej cyfr startując z tego punktu. Kończy się niepowodzeniem jeśli nie ma przynajmniej 11 znaków w ciągu lub ostatnie 10 znaków nie zaczyna się ciągiem z jedną lub więcej cyframi

---

16.3.20 SL_MATCH2

Podprogram sl_match2 jest niczym więcej niż rekurencyjnym wywołaniem dopasowania. Pole matchparm zawiera adres wzorca do dopasowania. Jest to całkiem użyteczne dla udawania nawiasów okrągłych wokół wzorca w wyrażeniu wzorcowym. Jeśli chodzi o poniższe ciągi dopasowywane pattern1 i pattern2, są one odpowiednikami:

```
Pattern1        pattern   {sl_match2, Pattern1}
Pattern2        pattern   {matchchar, ‚a'}
```

Jedyna różnica między wywołaniem wzorca bezpośrednio i wywołaniem go z sl_match2 jest taka, że sl_match2 pociąga kilka wewnętrznych zmiennych śledząc pozycję dopasowania wewnątrz ciągu wejściowego. Później możemy wyciągnąć ciąg znaków dopasowanych przez sl_match2 używając podprogramu patgrab.

16.4 PROJEKTOWANIE WŁASNEGO PRDROGRAMU DOPASOWANIA DO W ZORCA

Chociaż Biblioteka Standardowa UCR dostarcza szerokiej gamy funkcji dopasowujących, nie ma sposobu aby przewidzieć potrzeby dal wszystkich aplikacji. Dlatego też, prawdopodobnie odkryjemy, że biblioteka nie wspiera pewnych funkcji drapowania do wzorca jakich potrzebujemy . Na szczęście, bardzo łatwo stworzymy swoje własne funkcje dopasowujące zwiększając ich dostępność w Bibliotece Standardowej UCR. Kiedy określimy nazwę funkcji dopasowującej we wzorcowej strukturze danych, podprogram dopasowujący wywoła określony adres używając dalekiego wywołania i przekaże następujące parametry:

es:di - Wskazuje kolejny znak w ciągu wejściowym. Nie powinniśmy patrzeć na znaki przed tym adresem. Co więcej, nigdy nie powinniśmy zaglądać poza koniec ciągu (zobacz poniżej cx)

ds:si- Zawiera cztero bajtowy parametr z pola matchparm

cx - Zawiera ostatnią pozycję, plus jeden, w ciągu wejściowym, pozwalając się nam przypatrzeć. Zauważmy, że nasz podprogram nie powinien wychodzić poza lokację es:cx lub bajt zakończony zerem; którykolwiek nadejdzie jako pierwszy.

Przy powrocie z funkcji , ax musi zawierać offset do ciągu (wartość di) ostatniego znaku dopasowanego plus jeden, jeśli nasza funkcja dopasowująca zakończyła się powodzeniem. Musi również ustawić flagę przeniesienia oznaczającą sukces. Po naszym dopasowaniu do wzorca, podprogram dopasowujący może wywołać inną funkcję dopasowującą (jedynie określoną przez kolejne pole pattern) a ta funkcja zaczyna dopasowanie spod lokacji es:ax.

Jeśli dopasowanie wypadło niepomyślnie, wtedy musimy zwrócić oryginalna wartość di w rejestrze ax i zwrócić wyzerowaną flagę przeniesienia. Zauważmy, że nasza funkcja dopasowująca musi zachować wszystkie inne rejestry.

Jest jeden ważny szczegół, o którym nigdy nie możemy zapomnieć pisząc własne podprogramy dopasowania do wzorca – ds nie wskazuje naszego segmentu danych, zawiera najbardziej znaczące słowo parametru matchparm. Dlatego też, jeśli mamy zamiar uzyskać dostęp do zmiennych globalnych w naszym segmencie danych będziemy musieli odłożyć ds., załadować go adresem dseg i zdjąć ds. przed opuszczeniem. Kilka przykładów w tym rozdziale demonstruje jak to zrobić.

Jest kilka oczywistych przeoczeń w (bieżącej wersji) zakresie Biblioteki Standardowej UCR. Na przykład powinny być prawdopodobnie funkcje matchtostr, matchichar i matchtoichar Poniższy przykładowy kod demonstruje jak dodać podprogram matchtoistr 9doapsowanie do ciągu, wykonuje porównanie bez rozróżniania małych i dużych liter)

```
                .xlist

                include         stdlib.a
                includelib      stdlib.lib
                matchfuncs
                .list

dseg            segment para public 'data'

TestString      byte    "This is the string 'xyz' in it", cr, lf, 0

TestPat         pattern {matchtoistr, xyz}
xyz             byte    "XYZ", 0
```

```
dseg                    ends

cseg                    segment   para public  'code'
                        assume  cs:cseg. Ds:dseg

;MatchToiStr-           Dopasowuje wszystkie znaki w ciągu w górę, i wliczając, określone
;                       parametry ciągu. Parametr ciągu musi składać się z dużych znaków. Dopasowuje
;                       ciąg używając porównania bez rozróżniania małych i dużych liter.
;
;inputs:
;                       es:ds.- ciąg źródłowy
;                       ds.:si – Ciąg do dopasowania
;                       cx – maksymalna pozycja dopasowania
;outputs:
;                       ax – wskazuje pierwszy znak poza końcem dopasowywanego
;                       ciągu jeśli sukces, zawiera początkową wartość DI jeśli wystąpi
;                       niepowodzenie
;                       carry – 0 jeśli niepowodzenie, 1 jeśli sukces

MatchToiStr             proc    far
                        pushf
                        push    di
                        push    si
                        cld

;Sprawdzamy aby zobaczyć czy już jesteśmy poza punktem , które pozwoli nam przeszukać w ciąg
;wejściowy
                        cmp     di, cx
                        jae     MtiSFailure

;Jeśli ciąg wzorcowy jest ciągiem pustym, zawsze dopasowanym

                        cmp     byte  ptr  ds.:[si', 0
                        je      MTSuccess

;Następująca pętla przeszukuje cały ciąg wejściowy szukając pierwszego znaku w ciągu
;dopasowywanym

ScanLoop:               push    si
                        lodsb                   ;Pobranie pierwszy znak ciągu

                        dec     di
FinfdFirst:             inc     di              ;przesuwamy na następny (lub ostatni) znak
                        cmp     di, cx          ;jeśli przy cx wtedy musimy mieć niepowodzenie
                        jae     CantFindlst

                        mov     ah, es:[di]     ;pobiera wprowadzany znak
                        cmp     ah, 'a'         ;konwertuje wprowadzany znak do
                        jb      DoCmp           ;dużego znaku jeśli jest to mały znak
                        cmp     ah, 'z'
                        ja      DoCmp
                        and     ah, 5fh
DoCmp:                  cmp     al, ah          ;Porównanie znaku wprowadzonego
                        jne     FindFirst       ;ciągu wzorcowego

;W tym punkcie, umiejscawiamy pierwszy znak w ciągu wejściowym, który dopasowuje pierwszy znak ciągu
;wzorcowego Zobacz czy ciągi są równe

                        push    di              ;zachowanie punktu restartu
```

```
CmpLoop:        cmp     di, cx          ;zobacz czy idziemy poza ostatnią
                jae     StrNotThere     ;dostępną pozycję
                lodsb                   ;pobranie kolejnego wprowadzanego znaku
                cmp     al., 0          ;Czy koniec parametru ciągu? Jeśli tak, powodzenie
                je      MTSsuccess2

                inc     di
                mov     ah, es:[di]     ;pobranie kolejnego wprowadzanego znaku
                cmp     ah, 'a'         ;konwertuje znak wprowadzany do dużego znaku jeśli
                jb      DoCmp2          ;jest to mały znak
                cmp     ah, 'z'
                ja      DoCmp2
                and     ah, 5fh
DoCmp2:         cmp     al, ah          ;porównuje znak wejściowy
                je      CmpLoop
                pop     di
                pop     si
                jmp     Scanloop

StrnotThere:    add     sp, 2           ;usuwa di ze stosu
CanFindlst:     add     sp, 2           ;usuwa si ze stosu
MtiSFailure:    pop     si
                pop     di
                mov     ax, di          ;zwraca błędną pozycję w AX
                popf

                clc                     ;zwraca niepowodzenie
                ret

MTSSuccess2:    add     sp, 2           ;usuwa wartość DI ze stosu
MTSSuccess:     add     sp, 2           ;usuwa wartość SI ze stosu
                mov     ax, di          ;zwraca kolejną pozycję w AX
                pop     si
                pop     di
                popf
                stc                     ;zwraca powodzenie
                ret
MatchToiStr     endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                lesi    TestString
                ldxi    TestPat
                xor     cx, cx
                match
                jnc     NoMatch
                print
                byte    "Matched", cr, lf, 0
                jmp     Quit

NoMatch:        print
                byte    "Did not match", cr,lf, 0

Quit:           ExitPgm
Main            endp
```

```
cseg            ends
sseg            segment  para stack ' stack'
stk             db       1024  dup ("stack")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16  dup(?)
zzzzzzseg       ends
                end      Main
```

16.5 WYCIAGANIE PODCIĄGÓW Z DOPASOWYWANEGO WZORCA

Często, po prostu określamy, że dopasowywanie ciągu do danego wzorca jest niewystarczające Możemy chcieć wykonać różne operacje, które zależą do aktualnej informacji w ciągu. Jednakże, udogodnienia dopasowania do wzorca opisane do tej pory nie dostarczały mechanizmu dla testowania pojedynczych składników ciągu wejściowego. W tej sekcji zobaczymy jak wyciągnąć część wzorca dla dalszego przetwarzania.

Być może przykład może pomóc wyjaśnić potrzebę wyekstrahowania części ciągu. Przypuśćmy, że piszemy program kupna / sprzedaży giełdowej i chcemy go przetworzyć poleceniami opisanymi przez następujące wyrażenie skończone:
(buy | sell) [0-9]* shares of (ibm | apple | hp | dec)

Podczas gdy łatwo jest wynaleźć wzór Biblioteki Standardowej, który rozpozna ciągi w tej postaci wywołując podprogram match, powie on nam tylko, że mamy poprawne polecenie kupna sprzedaży. Nie powie nam czy kupujemy czy sprzedajemy, kto kupuje lub sprzedaje lub jak dużo akcji kupujemy lub sprzedajemy. Oczywiście możemy wziąć różne produkty z (buy | sell |) z (ibm | apple | hp | dec) i wygenerować osiem różnych wyrażeń skończonych, które w unikalny sposób określą czy kupujemy czy sprzedajemy i czyimi akcjami handlujemy, ale nie możemy przetworzyć wartości całkowitych w ten sposób (chyba że mamy miliony wyrażeń skończonych). Lepszym rozwiązaniem byłoby wyodrębnienie podciągu ze wzorca i przetwarzać te podciągi po tym jak zweryfikujemy, że mamy poprawne polecenie kupna lub sprzedaży. Na przykład, możemy wyodrębnić kupno lub sprzedaż z jednego ciągu , cyfry z drugiego i nazwę firmy z trzeciego. Po weryfikacji składni polecenia, możemy przetwarzać pojedyncze wyekstrahowane ciągi. Podprogram Biblioteki Standardowej UCR patgrab dostarcza takiej właściwości.

Zwykle wywołujemy patgrab po wywołaniu match i weryfikacji, że dopasowujemy ciąg wejściowy. Patgrab oczekuje pojedynczego parametru – wskaźnika do wzorca ostatnio przetwarzanego przez match. Patgrab tworzy ciąg na stercie składający się ze znaków dopasowanych przez dany wzorzec i zwraca wskaźnik do tego ciągu w es:di. Zauważmy, że patgrab zwraca tylko ciąg powiązany z pojedynczym wzorcową strukturą danych, nie łańcuchem wzorcowych struktur danych. Rozważmy następujący wzorzec:

```
PatToGrab       pattern  {matchstr, str1, 0, Pat2}
Pat2            pattern  {matchstr, str2}
str1            byte     „Hello", 0
str2            byte     " there", 0
```

Wywołując match dla PatToGrab będziemy dopasowywać ciąg „Hello there". Jednak, jeśli po wywołaniu match wywołamy patgrab i przekażemy mu adres PatToGrab, patgrab zwróci wskaźnik do ciągu „Hello"

Oczywiście, możemy chcieć odebrać ciąg, który jest połączeniem kilku ciągów dopasowywanych wewnątrz naszego wzorca (tj. części listy wzorców). Rozważmy poniższy wzorzec:

```
Numbers         pattern  {sl_match2, FirstNumber}
FirstNumber     pattern  {anycset, digits,0, OtherDigs}
OtherDigs       pattern  {spancset, digits}
```

To dopaoswuje do wzorca ciagi takie same jak

```
Numbers         pattern  {anycset, digits, 0 ,OtherDigs}
OtherDigs       pattern  {spancset, digits}
```

Więc dlaczego zawracamy sobie głowę dodatkowym wzorcem, który wywołuje sl_match2? Cóż, jak się okazuje funkcja dopasowująca sl_match2 pozwala nam tworzyć wzorce nawiasowe. Wzorzec nawiasowy jest listą wzorców, które podprogramy dopasowania do wzorca (zwłaszcza patgrab) traktują jako pojedynczy wzorzec.

Chociaż podprogram match będzie dopasowywał takie same ciągi bez względu na to jaką wersję Numbers użyjemy, patgrab stworzy dwa całkowicie różne ciągi w zależności od wybrania jednego z powyższych wzorców. Jeśli użyjemy drugiej wersji patgrab zwróci tylko pierwszą cyfrę liczby. Jeśli użyjemy pierwszej wersji (z wywołaniem sl_match2), wtedy patgrab zwróci cały ciąg dopasowany przez sl_match2 a to wyłącza cały ciąg cyfr.

Następujący program przykładowy demonstruje jak używać wzorców nawiasowych do wyodrębniania adekwatnych informacji z poleceń giełdowych przedstawionych wcześniej. Używa wzorców nawiasowych dla poleceń kupno / sprzedaż, liczby akcji i nazwy firmy

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

dseg            segment   para public  'data'
;Zmienne używane do przechowania liczby akcji sprzedanych / kupionych , wskaźnik do
;ciągu zawierającego polecenie kup / sprzedaj i wskaźnik do ciągu zawierającego nazwę
;firmy

Count           word    0
CmdPtr          dword   ?
CompPtr         dword   ?

;Jakieś ciągi testowy do wypróbowania:

Cmd1            byte    „Buy 25 shares of apple stock”, 0
Cmd2            byte    “Sell 50 shares of hp stock”, 0
Cmd3            byte    “Buy 123 shares of dec stock”, 0
Cmd4            byte    “Sell 15 shares of ibm stock”, 0
BadCmd0         byte    “This is not buy/sell command”, 0

;Wzorce  dla polecenia kupno / sprzedaż:
;
;StkCmd dopasowuje kupno lub  sprzedaż i tworzy wzorzec nawiasowy, który zawiera
;ciąg „buy’ lub „sell”

StkCmd          pattern {sl_match2, buyPat, 0 , skipspcs1}

buyPat          pattern {matchistr, buystr, sellpat}
buystr          byte    “BUY”, 0

sellpat         pattern {matchistr, sellstr}
sellstr         byte    „SELL“, 0

;Przeskakujemy zero lub więcej białych znaków po poleceniu kupuj

skipspcs1       pattern {spancset, whitespace, 0, CountPat}

;CountPat jest wzorcem nawiasowym, który dopasowuje jeden lub więcej znaków

CountPat        pattern {sl_match2, Numbers, 0, skipspcs2}
Numbers         pattern {anycset, digits,0, RestOfNum}
RestOfNum       pattern {spancset, digits}

;następujące wzorce dopasowują „ shares of „ pozwalając na białe znaki pomiędzy słowami

skipspcs2       pattern {spancset, whitespace, 0, sharesPat}
sharesPat       byte    “SHARES”, 0
```

```
skipspcs3       pattern  {spancset, whitesopace, 0, ofPat}

ofPat           pattern  {matchistr, ofStr, 0, skipspcs4}
ofStr           byte     "OF", 0

skipspcs4       pattern  {spancset, whitespace, 0, CompanyPat}

;Poniżysz wzorzec nawiasowy dopasowuje nazwę firmy. Dostępny ciąg patgrab będzie
;zawierał nazwę firmy

CompanyPat      pattern  {sl_match, ibmpat}

ibmpat          pattern  {matchistr, ibm, applePat}
ibm             byte     "IBM", 0

applePat        pattern  {matchistr, apple, hpPat}
apple           byte     "APPLE", 0

hpPat           pattern  {matchistr, hp, decPat}
hp              byte     "HP", 0

decPat          pattern  {matchistr, decstr}
decstr          byte     "DEC", 0

                include  stdsets.a

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

;DoBuySell -    Podprogram ten przetwarza polecenia giełdy kup / sprzedaj
;               Po dopasowaniu polecenia, przechwytuje składowe polecenia i wyprowadza je
;               jako właściwe. Podprogram demonstruje jak używać patgrab do wyodrębniania
;               podciągów z ciągu wzorcowego
;
;               Na wejściu, es:di musi wskazywać polecenia kup / sprzedaj jakie chcemy
;               przetworzyć.

DoBuySell       proc     near
                ldxi     StkCmd
                xor      cx, cx
                match
                jnc      NoMatch

                lesi     StkCmd
                patgrab
                mov      word  ptr  CmdPtr, di
                mov      word  ptr  CmdPtr+2, es
                lesi     CountPat
                patgrab
                atoi                          ;konwertuje cyfry na liczby całkowite
                mov      Count, ax
                free                          ; zwraca pamięć ze sterty

                lesi     ComapnyPat
                patgrab
                mov      word  ptr  CompPtr, di
                mov      word  ptr  CompPtr+2, es
```

```
                printf
                byte    "Stock command: %^s\n"
                byte    "Numbers of shares:  %d\n"
                byte    "Company  to trade:  %^s\n\n", 0
                dword   CmdPtr, Cout, CompPtr

                les     di, CmdPttr
                free
                les     di, CompPtr
                free
                ret

NoMatch:        print
                byte    "Illegal buy/sell command", cr, lf, 0
                ret
DoBuySell       endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit

                lesi    Cmd1
                call    DoBuzSell
                lesi    Cmd2
                call    DoBuzSell
                lesi    Cmd3
                call    DoButSell
                lesi    Cmd4
                call    DoBuzSell
                lesi    BadCmd10
                call    DoBuzSell

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para  stack 'stack'
stk             db      1024 dup ("stack")
sseg            ends

zzzzzzseg       segemnt  para  public  'zzzzzz'
LastBytes       db      16  dup (?)
Zzzzzzseg       ends
                end     Main
```

Przykład danych wyjściowych:

```
Stock command: Buy
Number of shares: 25
Company to trade: apple

Stock command: Sell
Number of shares: 50
Company to trade: hp

Stock command: Buy
```

Number of shares: 123
Company to trade: dec

Stock command: Sell
Number of shares: 15
Company to trade: ibm

Illegal buy/sell command

16.6 ZASADY SEMATYCZNE I AKCJE

Teoria automatów jest głównie interesuje się czy lub nie dopasowano ciąg danym wzorcem. Podobnie jak wiele nauk teoretycznych , praktyka teorii automatów jest tylko skoncentrowana na tym czy coś jest możliwe, praktyczne aplikacje nie są ważne. Przy rzeczywistych programach jednakże chcemy wykonać pewne działania jeśli dopasowujemy ciąg lub wykonujemy jeden ze zbiorów operacji w zależności od tego jak dopasowujemy ciąg.

Zasada semantyczna lub akcja semantyczna jest działaniem jakie wykonujemy w oparciu o typ wzorca jaki dopasowujemy. To znaczy, jest to kawałek kodu wykonywany kiedy jesteśmy zadowoleni z zachowania dopasowania do wzorca. Na przykład, wywołanie patgrab w poprzedniej sekcji jest przykładem akcji semantycznej.

Normalnie wykonujemy kod powiązany z zasadą semantyczną po powrocie z wywołania match. Z pewnością kiedy przetwarzamy wyrażenie skończone, nie ma potrzeby przetwarzania akcji semantycznej w środku operacji dopasowania do wzorca. Jednakże nie jest to przypadek dla gramatyki bezkontekstowej. Gramatyki bezkontekstowe często wymagają rekurencji lub możemy użyć takiego samego wzorca kilka razy kiedy dopasowujemy pojedynczy ciąg (to znaczy, możemy się odnosić do takiego samego nieterminala kilka razy podczas dopasowywania wzorca).Struktura danych dopasowania do wzorca tylko utrzymuje wskaźniki (EndPattern, StartPattern i StrSeg) do ostatniego podciągu dopasowywanego przez dany wzorzec. Dlatego też jeśli używamy ponownie podwzorca przy dopasowywaniu ciągu i musimy wykonać zasadę semantyczną powiązaną z tym podwzorcem, będziemy musieli wykonać zasadę semantyczną w środku operacji dopasowywania do wzorca, zanim odniesiemy się to tego podciągu ponownie.

Okazuje się bardzo łatwe wprowadzanie zasad semantycznych w środku operacji dopasowania do wzorca. Wszystko co musimy zrobić to napisanie funkcji dopasowania do wzorca, która zawsze kończy się powodzeniem (tj. zwraca wyzerowaną flagę przeniesienia) . Wewnątrz ciała naszego podprogramu dopasowania do wzorca możemy wybrać zignorowanie ciągu dopasowywanego kodu, i przeprowadzenia testowania i wykonania innych akcji jakie sobie życzymy.

Nasz podprogram akcji semantycznej, przy zwrocie, musi ustawić flagę przeniesienia i musi skopiować oryginalną zawartość di do ax. Musi zachować wszystkie inne rejestry. Nasza akcja semantyczna nie może wywołać podprogramu match (zamiast tego wywołujemy sl_match). Match nie pozwala na rekurencję (nie jest współbieżny) i wywołując match wewnątrz podprogramu akcji semantycznej spaskudzimy dopasowanie do wzorca w toku.

Następujący przykład dostarcza kilka przykładów podprogramów akcji semantycznej wewnątrz programu. Program ten konwertuje wyrażenia arytmetyczne w postaci (algebraicznej) wzrostkowej do postaci odwróconej notacji polskiej

```
; INFIX.ASM
;
;Prosty program który demonstruje podprogram dopasowania do wzorca w bibliotece UCR. Program akceptuje
; wyrażenia arytmetyczne w linii poleceń ( nie jest dozwolone żadne przeplatanie miejsca, to znaczy, musi być
;tylko jeden parametr w linii poleceń0 i konwertuje go z notacji wzrostkowej do notacji odwrotnej (rpn)

                        .xlist
                        include     stdlib.a
                        includelib stdlib.lib
                        .list

dseg                    segment  para public  'data'
```

;Gramatyka dla prostej operacji translacji infix -> postfix (akcje semantyczne są otoczone nawiasami ;klamrowymi):
;
; E → FE'
; E → +F {output '+'} E' | -F {output ' – '} E' | <pusty ciąg>
; F → TF'
; F → *T {output '*'} F' | /T {output '/'} F' | <pusty ciąg>
;T → -T {output 'neg'} | S
; S → <stała> {output stała} | (E)
;
;Wzorzec Biblioteki Standardowej UCR , który działa na powyższej gramatyce:
; Wyrażenie składa się z pozycji „E" po której następuje koniec ciągu:

```
infix2rpn       pattern   {sl_Match2, E, EndOfString}
EndOfString     pattern   {EOS}
```

; pozycja "E" składa się z pozycji "F" opcjonalnie, po której następuje "+" lub "-" i inna
;pozycja „E":

```
E               pattern   {sl+maych2, F, , Eprime}
Eprime          pattern   {MatchChar, '+', Eprome2, epf}
epf             pattern   {sl_match2, F,,epPlus}
epPlus          pattern   {OutputPlus,,,Eprime}          ;zasada semantyczna

Eprome2         pattern   {MatchChar, '-', Succeed, emf}
emf             pattern   {sl_match2, F,,epMinus}
epMinus         pattern   {OutputMinus,,,Eprime}   ;zasada semantyczna
```

;Pozycja "F" składa się z pozycji "T" opcjonalnie po której następuje „*" lub „/", po którym
;następuje inna pozycja „T":

```
F               pattern   {sl_match2, T, Fprime}
Fprime          pattern   {MatchChar, '*' , Fprime2, fmf}
fmf             pattern   {sl_match2, T, 0, pMul}
pMul            pattern   {OutputMul,,,Fprime}           ;zasada semantyczna

Fprime2         pattern   {MatchChar, '/', Succeed, fdf}
fdf             pattern   {sl_match2, T, 0, pDiv}
pDiv            pattern   {OutputDiv, 0,0, Fprime}       ;zasada semantyczna
```

;Pozycja „T" składa się z pozycji „S" lub „-„ po których następuje inna pozycja „T":

```
T               pattern   {MatchChar, '-', S, TT}
TT              pattern   {sl_match2, T, 0, tpn}
tpn             pattern   {OutputNeg}              ;zasada semantyczna
```

;Pozycja „S" jest albo ciągiem z jedną lub więcej cyfr albo „(„ po którym następuje i pozycja „E"
; po której następuje „)":

```
Const           pattern   {sl_Match2, DoDigits, 0, spd}
spd             pattern   {OutputDigits}

DoDigits        pattern   {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern   {Spancset, Digits}

S               pattern   {MatchChar, '(', Const, IntE}
IntE            pattern   {sl_Match2, E,0, CloseParen}
CloseParen      pattern   {MatchChar, ')'}

Succeed         pattern   {DoSucceed}
```

```
                        Include  stdsets.a

dseg                    ends

cseg                    segment  para  public  'code'
                        assume  cs:cseg, ds:dseg

;DoSucceed dopasowuje pusty ciąg. Innymi słowy, dopasowuje cokolwiek i zawsze zwraca powodzenie
;bez zjadania jakiegoś znaku z ciągu wejściowego

DoSucceed               proc    near
                        mov     ax, di
                        stc
                        ret
DoSucceed               endp

;OutputPlus jest zasadą semantyczną , która wyprowadza operator „+" po  analizie poprawności operatora
;dodawania w ciągu wzrostkowym

OutputPlus              proc    far
                        print
                        byte    „ +", 0
                        mov     ax, di                          ;wymagane przez sl_Match
                        stc
                        ret
OutputPlus              endp

;OutputMinus jest zasadą semantyczną , która wyprowadza operator „-„ po analizie poprawności operatora
;odejmowania w ciągu wzrostkowym

OutputMinus             proc    far
                        print
                        byte    „ –„ , 0
                        mov     ax, di                  ;wymagane przez sl_match
                        stc
                        ret
OutputMinus             endp

;OutputMul jest zasadą semantyczną , która wyprowadza operator „*" po analizie poprawności operatora
;mnożenia w ciągu wzrostkowym

OutputMul               proc    far
                        print
                        byte    „ *", 0
                        mov     ax, di                          ;wymagane przez sl_match
                        stc
                        ret
OutputMul               endp

;OutputDiv jest zasadą semantyczna która wyprowadza operator „/" po analizie poprawności operatora dzielenia
; w ciągu wzrostkowym

OutputDiv               proc    far
                        print
                        byte    „ /", 0
                        mov     ax, di                          ;wymagane przez sl_Match
                        stc
                        ret
OuyputDiv               endp
```

```
;OutputNeg      jest zasadą semantyczną która wyprowadza jednoargumentowy operator „-„ po analizie
poprawności operatora negacji w ciągu wzrostkowym

OutputNeg       proc    far
                print
                byte    „ neg”, 0
                mov     ax, di                  ;wymagane przez sl_match
                stc
                ret
OutputNeg       endp

;OutputDigits wyprowadza wartość numeryczną kiedy napotyka poprawną wartość całkowitą w ciągu
;wejściowym

OutputDigits    proc    far
                push    es
                push    di
                mov     al., ‘ ‘
                putc
                lesi    stała
                patgrab
                puts
                free
                stc
                pop     di
                mov     ax, di
                pop     es
                ret
OutputDigits    endp

;Okay, tu mamy program główny, który pobiera parametr z linii poleceń i analizuje go

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit                         ;pamięć na stercie

                print
                byte    „Enter an arithemtic expression: „, 0
                getsm
                print
                byte    “Expression in postfix form: “,0
                ldxi    infix2rpn
                xor     cx, cx
                match
                jc      Succeeded

                print
                byte    “Syntax error”, 0
Succeeded:      putcr

Quit:           ExitPgm
Main            endp

cseg            ends

;Alokacja rozsądnej ilości miejsca na stosie (8k)
```

```
sseg            segment  para stack 'stack'
stk             db       1024   dup ("stack")
sseg            ends

;zzzzzzseg musi być ostatnim segmentem ładowanym do pamięci!

Zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
Zzzzzzseg       ends
                end      Main
```

16.7 KONSTRUOWANIE WZORCÓW DLA PODPROGRAMU MATCH

Głównym tematem jaki omówimy jest to ,jak konwertować wyrażenia skończone i gramatyki bezkontekstowe do wzorców odpowiednich dla podprogramów dopasowania do wzorca Biblioteki Standardowej UCR. Większość przykładów pojawiających się do tego punktu używało doraźnych schematów translacji; teraz jest najwyższy czas dostarczyć algorytmu do wykonania tego zadania.

Poniższy algorytm konwertuje gramatykę bezkontekstową do wzorcowej struktury danych Biblioteki Standardowej UCR. Jeśli chcemy skonwertować wyrażenie skończone do wzorca, najpierw konwertujemy wyrażenie skończone do gramatyki bezkontekstowej (zobacz „Konwertowanie WS do CFG"). Oczywiście, łatwo jest skonwertować wiele postaci wyrażeń skończonych bezpośrednio do wzorca, kiedy takie konwersje są oczywiste możemy ominąć następujący algorytm; na przykład powinno być oczywiste, że możemy użyć spancset do dopasowania wyrażenia skończonego takiego jak [0-9]*.

W pierwszym kroku musimy zawsze wyeliminować lewostronną rekurencję z gramatyki. Wygenerujemy pętlę nieskończoną (i krach maszyny) jeśli próbujemy kodować gramatykę zawierającą lewostronną rekurencję we wzorcowej strukturze danych. Po informacji o eliminowaniu rekurencji lewostronnej, zobacz „Eliminowanie Rekurencji Lewostronnej I opuszczanie współczynnika CFG" Możemy również chcieć opuszczenia współczynnika gramatyki podczas eliminacji rekurencji lewostronnej Podprogramy Biblioteki Standardowej w pełni wspierają backtracing, więc opuszczanie współczynnika nie jest wyłącznie konieczne, jednak podprogramy dopasowujące będą wykonywały się szybciej jeśli nie będzie backtracku.

Jeśli gramatyka wyrobu przybiera formę A $\rightarrow$ B C gdzie A, B i C są nieterminalnymi symbolami, stworzymy poniższy wzór:

```
A               pattern  {sl_match2, B, 0, C}
```

Ten wzorzec opisany dla A sprawdza wystąpienia wzorca B po którym następuje wzorzec C.

Jeśli B jest relatywnie prostym wyrobem (to znaczy możemy skonwertować go do pojedynczej wzorcowej struktury danych),możemy zoptymalizować to do:

```
A               pattern  {B's Matching Function, B's parametr, 0, C}
```

Pozostałe przykłady zawsze będą wywoływały sl_match2. Jednakże tak długo jak te nieterminalne są po prostu wywoływane, możemy je zagiąć do wzorca A''

Jeśli gramatyka wyrobu przybiera postać A $\rightarrow$ B | C gdzie A,B i C są nieterminalnymi symbolami możemy stworzyć następujący wzór:

```
A               pattern  {sl_match2, B, C}
```

Ten wzór próbuje dopasować B. Jeśli kończy się powodzeniem, kończy się powodzeniem A; jeśli kończy się niepowodzeniem, próbuje dopasować C. W tym punkcie A'' kończy się sukcesem lub niepowodzeniem , sukcesem lub niepowodzeniem kończy się C.

Działanie z symbolami terminalnymi jest kolejną rzeczą do rozważenia. To jest całkiem łatwe – wszystko co musimy zrobić to użycie właściwej funkcji dopasowującej dostarczanej przez Bibliotekę Standardową np. matchstr lub matchchar. Na przykład jeśli mamy wyrób w postaci A $\rightarrow$ abc | y skonwertujemy do następującego wzorca:

```
A               pattern  {matchstr, abc, ypat}
ac              byte     "abc", 0
ypat            pattern  {matchstr, 'y'}
```

Jedynym pozostałym szczegółem do rozpatrzenia jest ciąg pusty. Jeśli mamy wyrób w postaci A $\rightarrow$ ε wtedy musimy napisać funkcję dopasowania do wzorca która zawsze kończy się powodzeniem. Eleganckim sposobem zrobienia tego jest napisanie zwykłej funkcji dopasowania do wzorca. Ta funkcja to

```
succeed     proc    far
            mov     ax, di          ;wymagane przez sl_match
            stc                     ;zawsze powodzenie
            ret
succeed     endp
```

Innym , podstępnym, sposobem do osiągnięcia sukcesu jest użycie matchstr i przekazanie pustego ciągu do dopasowania np.

```
succees         pattern {matchstr, emptustr}
emptystrbyte    0
```

Pusty ciąg zawsze dopasowuje ciąg wejściowy, bez względu co zawiera ciąg wejściowy.

Jeśli mamy wyrób z kilkoma alternatywami a ε jest jedna z nich, musimy przetworzyć ostatnie ε. Na przykład, jeśli mamy wyrób A $\rightarrow$ abc | y | BC | ε użyjemy poniższego wzorca:

```
A           pattern {matchstr, abc, tryY}
abc         byte    "abc", 0
tryY        pattern {matchchar, 'y', tryBC}
tryBC       pattern {sl_match2, B, DoSuccess, c}
DoSucceess  pattern {succeed}
```

Technika opisana powyżej pozwala nam skonwertować każdą CFG do wzorca, który może przetworzyć Biblioteka Standardowa, co z pewnością nie wykorzystuje udogodnień Biblioteki Standardowej, nie tworząc szczególnie wydajnego wzorca. Na przykład rozważmy wyrób:

Digits $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Konwertują to do wzorca używając tej techniki opisanej powyżej dostajemy wzorzec:

```
Digits      pattern {matchchar, '0', try1}
try1        pattern {matchchar, '1',try2}
try2        attern  {matchchar, '2,try3}
try3        pattern {matchchar, '3',try4}
try4        pattern {matchchar, '4',try5}
try5        pattern {matchchar, '5',try6}
try6        pattern {matchchar, '6',try7]
try7        pattern {matchchar, '7',try8}
try8        pattern {matchchar, '8',try9]
try9        pattern {matchchar, '9'}
```

Oczywiście nie jest to bardzo dobre rozwiązanie ponieważ możemy dopasować ten sam wzór w pojedynczej instrukcji:

```
Digits      pattern {anycset, digits}
```

Jeśli nasz wzorzec jest łatwy do określenia przy użyciu wyrażeń skończonych, powinniśmy spróbować zakodować go przy użyciu wbudowanych funkcji dopasowania do wzorca i powrócić do powyższego algorytmu ponieważ działamy na wzorcach niskiego poziomu jak najlepiej można . Z doświadczenia możemy wybrać właściwą równowagę pomiędzy algorytmem w tej sekcji a doraźnymi metodami odkrytymi przez nas.

16.11 PODSUMOWANIE

Z pewnością był to długi rozdział. Generalnie tematowi dopasowania do wzorca jest poświęcono niewystarczająca ilość uwagi w różnych tekstach. Faktycznie, rzadko widać więcej niż dwanaście stron

dedykowanych teorii automatów, kompilatorów lub językom dopasowania do wzorca takich jak Icon lub SNOBOL4. Jest to jeden z głównych powodów dla którego ten rozdział jest rozległy, pomagając pokryć niedostatki dostępnej gdzie indziej. Jednakże, jest inny powód dla długości tego rozdziału a zwłaszcza liczby linii kodu pojawiającego się w tym rozdziale – demonstruje jak łatwo jest odkryć pewną klasę programów używających technik dopasowania do wzorca. Czy możesz sobie wyobrazić napisanie programu takiego jak Madventure używając standardowego C lub technik programistycznych Pascala? Program wynikowy byłby prawdopodobnie dłuższy niż wersja asemblerowa pojawiająca się w tym rozdziale! Jeśli nie jesteś pod wrażeniem siły dopasowania do wzorca, być może powinieneś jeszcze raz przeczytać ten rozdział. Jest bardzo zaskakujące jak mało programistów naprawdę rozumie teorię dopasowania do wzorca , zwłaszcza rozważając jak wiele programów używa , lub może korzystać z technik dopasowania do wzorca.

Rozdział zaczyna się omówieniem teorii poza dopasowaniem do wzorca. Omawia proste wzorce, znane jako języki skończone i opisuje jak zaprojektować niedeterministyczne i deterministyczne skończone stany automatów – funkcje, które dopasowują wzorce opisane przez wyrażenia skończone. Rozdział ten opisuje również jak skonwertować NFA i DFA do programów asemblerowych.

* ”Wprowadzenie do teorii języka formalnego (automatów)
* ”Maszyny kontra Języki’
* „Języki skończone”
* „Wyrażenia skończone”
* „Niedeterministyczne Skończone stany automatów (NFA)
* „Konwertowanie Wyrażeń skończonych do NFA”
* „Konwertowanie NFA do języka asemblera”
* „Deterministyczne skończone stany automatów (DFA)
* „Konwertowanie DFA do języka asemblera”

Chociaż języki skończone są prawdopodobnie najpowszechniej przetwarzanymi wzorcami w nowoczesnych programach dopasowania do wzorca, są one również tylko małym podzbiorem możliwych typów wzorców jakie możemy przetwarzać w programie. Języki bezkontekstowe ,wliczając wszystkie języki skończone jako podzbiór, wprowadzają wiele typów wzorców które nie są skończone. Do przedstawienia języka bezkontekstowego często używamy gramatyki bezkontekstowej. CFG zawiera zbiór wyrażeń znanych jako wyroby. Ten zbiór wyrobów, zbiór nieterminalnych symboli, zbiór symboli terminalnych i specjalnego nieterminala, symbolu startowego, dostarcza podstaw do konwersji wzorców do języka programowania.

W tym rozdziale posługujemy się specjalnym zbiorem gramatyk bezkontekstowych znanych jako gramatyka LL(1). Aby właściwie zakodować CFG do asemblera musimy najpierw skonwertować gramatykę do gramatyki LL(1). Kodowanie to daje nam rekurencyjne zmniejszenie analizy predykcyjnej. Dwoma pierwszymi krokami wymaganymi przed konwersją gramatyki do programu który rozpoznaje ciągi w języku bezkontekstowym jest eliminacja lewostronnej rekurencji z gramatyki i opuszczanie współczynnika gramatyki. Po tych dwóch krokach jest relatywnie łatwo skonwertować CFG do asemblera

*”Języki bezkontekstowe”
*”Eliminacja rekurencji lewostronnej i opuszczanie współćzynnika CFG’
*”Konwersja CFG do języka asemblera”
*”Końcowe uwagi na temat CFG”

Czasami łatwiej jest działać z wyrażeniami skończonymi zamiast gramatykach bezkontekstowych. Ponieważ CFG są bardziej mocniejsze niż wyrażenia skończone, ten tekst generalnie adoptuje gramatyki gdzie tylko to możliwe. Jednakże wyrażenia skończone są generalnie łatwiejsze do pracy (dla prostych wzorców) , zwłaszcza we wczesnych etapach projektowania Wcześniej czy później możemy potrzebować skonwertować wyrażenie skończone do CFG, wiec połączymy go z innym składnikiem gramatyki. Jest to bardzo łatwe do zrobienia i mamy prosty algorytm do konwersji WS do CFG.

*”Konwersja WS do CFG”

Chociaż konwersja CFG do asemblera jest prostym procesem, jest bardzo nużące. Biblioteka Standardowa UCR wprowadza zbiór podprogramów dopasowania do wzorca, które w zupełności eliminują to znużenie i dostarczają dodatkowych udogodnień (takich jak automatyczny backtracing, pozwalający kodować gramatyki, które nie są LL(1)) Pakiet dopasowania do wzorca w Bibliotece Standardowej jest prawdopodobnie najnowocześniejszym i silnym zbiorem dostępnych podprogramów. Powinniśmy zdecydowanie zbadać zastosowanie tych podprogramów, co może zabrać sporo czasu.

*”Podprogramy dopasowania do wzorca Biblioteki Standardowej UCR”
*”Funkcje dopasowania do wzorca Biblioteki Standardowej”

Jedna z cech dostarczaną przez Bibliotekę Standardową jest nasza zdolność do pisania przerobionych funkcji dopasowani do wzorca. Dodatkowo te funkcje dopasowania do wzorca pozwalają nam dodać zasady semantyczne do naszej gramatyki.

*"Projektowanie własnych podprogramów dopasowania do wzorca"
*"Wyodrębnianie podciągów z dopasowywanego wzorca"
*"zasady semantyczne i akcje"

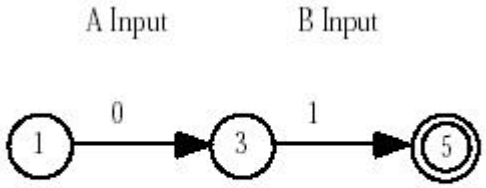
Chociaż Biblioteka Standardowa UCR dostarcza silnego zbioru podprogramów dopasowania do wzorca, ich bogactwo może być ich podstawową wadą. Ci, którzy napotkali podprogramy dopasowania do wzorca Biblioteki Standardowej po raz pierwszy mogą czuć się przytłoczeni, zwłaszcza kiedy próbują pogodzić materiał z sekcji o gramatyce bezkontekstowej z wzorcami Biblioteki Standardowej. Na szczęście jest prosty, choć niewydajny, sposób przetłumaczenia CFG na wzorzec Biblioteki Standardowej.

*"Konstruowanie wzorców dla podprogramu MATCH"

Chociaż dopasowanie do wzorca jest silnym paradygmatem, z którym większość programistów powinna się zapoznać, większość ludzi ma kłopoty z aplikacjami, kiedy pierwszy raz napotykają dopasowanie do wzorca.

16.12 PYTANIA

1) Załóżmy, że mamy dwa wejścia, które są albo zerem albo jedynką. Stwórz DFA implementujące poniższe funkcje logiczne (zakładamy, że przejście do stanu końcowego jest odpowiednikiem prawdy, jeśli działamy w nieakceptowanym stanie, zwracamy fałsz)
   a) OR b) XOR c) NAND d) NOR e) Equals (XNOR) f) AND



Example, A<B

2) Jeśli r, s I t są wyrażeniami skończonymi, jaki ciąg dopasujemy dla następujących wyrażeń skończonych?
   a) r* b) r s c) r^+ d) r | s

3) Dostarcz wyrażenia skończonego dla liczb całkowitych, które pozwala na przecinki co trzy cyfry, jak w składani US (np. dla każdych trzech cyfr od prawej strony musi być dokładnie jeden przecinek). Nie pozwolono na złe umieszczenie przecinków
4) Pascalowska stała rzeczywista ma przynajmniej jedna cyfrę przed punktem dziesiętnym. Dostarcz wyrażenia skończonego dla stałej rzeczywistej FORTRAN'a, która nie ma takiego ograniczenia.
5) W wielu systemach języków(np. FORTRAN lub C) są dwa typy liczb zmienno przecinkowych o pojedynczej i podwójnej precyzji. Dostarcz wyrażenia skończonego dla liczb rzeczywistych, które pozwala na wprowadzenie liczb zmienno przecinkowych przy użyciu znaków [dDeE] jako symbol wykładnika (d/D) oznaczający podwójną precyzję.
6) Dostarcz NFA, które rozpoznaje mnemoniki dla zbioru instrukcji 886
7) Skonwertuj powyższe NFA do języka asemblera. Nie używaj podprogramów dopasowania do wzorca Biblioteki Standardowej.
8) Powtórz pytanie (7) przy użyciu podprogramów dopasowania do wzorca Biblioteki Standardowej
9) Stwórz DFA dla identyfikatorów Pascala
10) Skonwertuj powyższe DFA do kodu asemblerowego używając prostych instrukcji asemblera
11) Skonwertuj powyższe DFA do kodu asemblera używając tablicy stanu z sklasyfikowanymi wejściami. Opisz dane w swojej sklasyfikowanej tablicy.
12) Wyeliminuj lewostronną rekurencję w poniższej gramatyce

```
Stmt    →    if expression then Stmt endif
         |   if expression then Stmt else Stmt endif
         |   Stmt ; Stmt
         |   ε
```

13) Opuść współczynnik gramatyki stworzony w problemie 12
14) Skonwertuj wynik z pytania (13) do języka asemblera bez używania podprogramów dopasowania do wzorca Biblioteki Standardowej
15) Skonwertuj wynik z pytania (13) do języka asemblera używając podprogramów dopasowania do wzorca Biblioteki Standardowej
16) Skonwertuj wyrażenie skończone uzyskane w pytaniu (3) do zbioru wyrobów dla gramatyki bezkontekstowej
17) Dlaczego funkcja dopasowująca ARB jest niewydajna? Opisz jak wzorzec (ARB „hello" ARB) można dopasować do ciągu „hello there"
18) Spancset dopasowuje zero lub więcej wystąpień jakichś znaków w zbiorze znaków. Napisz funkcję dopasowania do wzorca , wywoływanej jako pierwsze pole wzorcowego typu danych, która dopasowuje jedno lub więcej wystąpień jakiegoś znaku (zerknij do źródeł spancset)
19) Napisz funkcję dopasowania do wzorca matchichar, która dopasowuje pojedynczy znak bez względu na wielkość (zerknij do źródeł matchchar)
20) Wyjaśnij jak użyć funkcji dopasowania do wzorca do implementacji zasady semantycznej
21) Jak wyodrębnić podciąg z dopasowywanego wzorca?
22) Co to są wzorce nawiasowe? Jak je tworzymy?