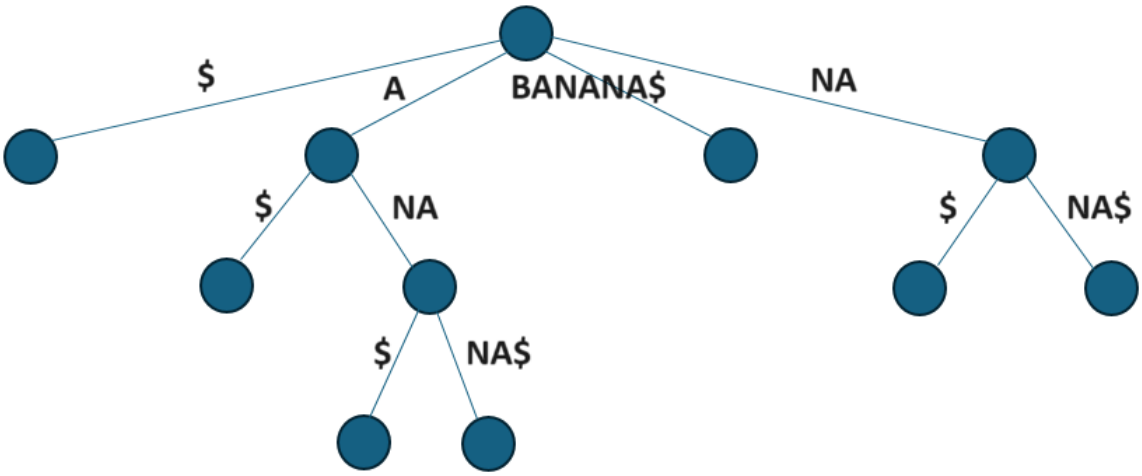
În prima etapă vă veți familiariza cu structura și reprezentarea arborilor de sufixe (ST) în Racket, veți implementa o mini-bibliotecă pentru tipul ST (în fișierul **suffix-tree.rkt**), apoi veți implementa câteva funcții utile construcției și manipulării arborilor de sufixe (în fișierul **etapa1.rkt**).
Un arbore de sufixe este un arbore care stochează toate sufixele unui text T, astfel:

- nodul rădăcină are un număr de fii egal cu dimensiunea alfabetului textului T (de exemplu, pentru un text care folosește toate literele mici ale alfabetul englez, nodul rădăcină va avea 26 de fii)
- muchia de la un nod tată la un nod fiu este etichetată cu prefixul comun al tuturor sufixelor stocate pe această cale
- procedeul se repetă: fiecare nod va avea un număr de fii egal cu numărul de simboluri distincte cu care încep sufixele stocate în nod
- în arborele final, fiecărui sufix al textului T îi corespunde o cale de la rădăcină la o frunză (iar sufixul se recompune prin concatenarea tuturor etichetelor de pe această cale)

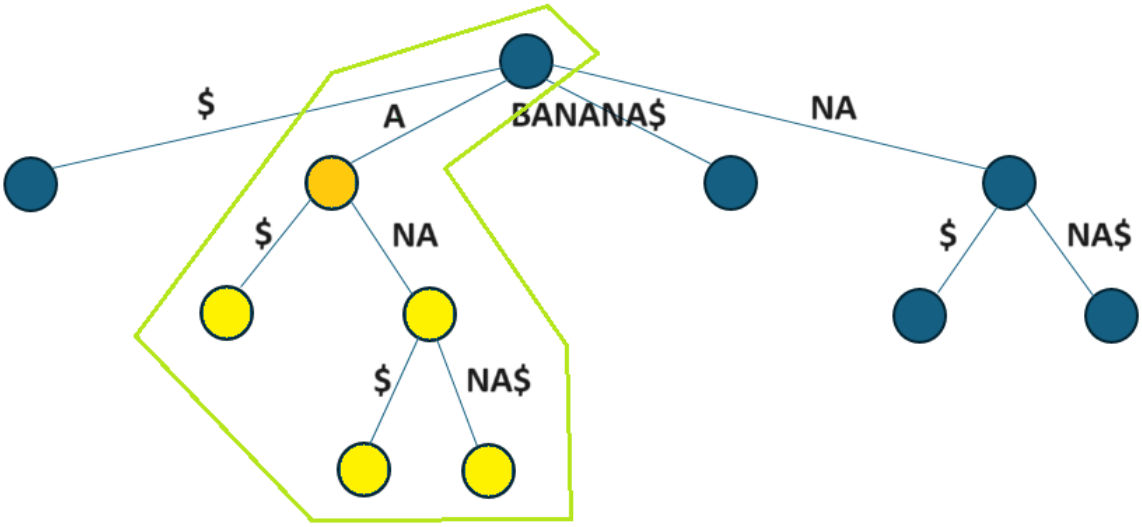
Convenim să terminăm fiecare sufix prin caracterul special `$`, asigurându-ne astfel că fiecărui sufix îi va corespunde o frunză în arbore. Pentru textul "BANANA", arborele de sufixe arată astfel:



Observăm că, fără convenția de a utiliza terminația `$`, sufixului `"ANA"` i-ar fi corespuns un nod intern în arborele de sufixe, nu o frunză.

- Când muchiile sunt etichetate cu cel mai lung prefix comun al sufixelor din subarborele respectiv, arborele de sufixe se numește **compact** (ca în figura de mai sus).
- Când muchiile sunt etichetate cu câte un singur caracter (care este tot un prefix comun al tuturor sufixelor "de mai jos", însă nu neapărat cel mai lung), arborele de sufixe se numește **atomic** (găsiți un exemplu în scheletul de cod).

Reprezentare în Racket



- Vom reprezenta un arbore de sufixe ca pe o **listă** de ramuri, unde fiecare ramură corespunde unui fiu al rădăcinii. Pe exemplul de mai sus, textul `"BANANA"` generează o listă de 4 ramuri (una cu eticheta `"$"`, una cu eticheta `"A"`, una cu eticheta `"BANANA$"` și una cu eticheta `"NA"`).
- Prin ramură, vom înțelege o **pereche** între o etichetă și subarborele cu rădăcina în nodul de sub etichetă. Pe exemplul din figură, cea de-a doua ramură (încadrată cu verde) are eticheta `"A"` și subarborele cu rădăcina portocalie și restul nodurilor galbene.
- Pentru a folosi funcțiile de lucru pe liste, fiecare etichetă va fi reprezentată nu ca string, ci ca **listă de caractere**. Astfel, eticheta `"BANANA$"` va fi de fapt reținută sub forma listei `'(#\B #\A #\N #\A #\N #\A #\$)` (acesta este modul de reprezentare a caracterelor în Racket).

Rezultă că reprezentarea completă în Racket a arborelui din figură este:

```
'( ((#\$))                                  ; prima ramură, de etichetă "$", corespunzătoare sufixului vid

   ((#\A)                                 ; a doua ramură, de etichetă "A"

        ((#\$))                           ; ramura 2.1, terminală, corespunzătoare sufixului "A"

        ((#\N #\A)                           ; ramura 2.2, de etichetă "NA"

             ((#\$))                           ; ramura corespunzătoare sufixului "ANA"

             ((#\N #\A #\$))))                 ; ramura corespunzătoare sufixului "ANANA"
```

```
   ((#\B #\A #\N #\A #\N #\A #\$))  ; a treia ramură, de etichetă "BANANA$"

   ((#\N #\A)                       ; a patra ramură, de etichetă "NA"

         ((#\$))                          ; ramura corespunzătoare sufixului
"NA"

         ((#\N #\A #\$)))   )             ; ramura corespunzătoare sufixului
"NANA"
```

Acești arbori vor fi definiți în checker, însă este necesar să le înțelegeți structura pentru a putea implementa funcțiile din cerință.

În etapa 1, veți exersa lucrul cu:

- liste și operatorii acestora (datorită modului de reprezentare a tipului ST și a etichetelor, precum și a funcțiilor care vă solicită să întoarceți ca rezultat liste cu un format specific)
- perechi și operatorii acestora (pentru că fiecare ramură este o pereche între o etichetă și un subarbore)
- funcții recursive pe stivă, respectiv pe coadă (observați tipul de recursivitate al fiecărei funcții implementate, și atenție la cazurile în care vi se solicită un anumit tip de implementare - chiar dacă obțineți punctaj pe checker, punctajul va fi anulat în cazul în care funcțiile nu sunt implementate conform cerințelor)
- operatori condiționali, operatori logici și valori boolene

În următoarele exemple, considerăm că reprezentarea arborelui de sufixe pentru textul `"BANANA"` este reținută în variabila `st-banana`.
Funcțiile principale pe care le veți implementa sunt:

```
(first-branch st)

(other-branches st)
```

- first-branch primește un arbore de sufixe (ST) și întoarce prima ramură a acestuia (o pereche etichetă-subarbore)
- other-branches primește un ST și întoarce ST-ul fără prima sa ramură (o listă de ramuri, așa cum era și ST-ul original)
- ex: `(first-branch st-banana)` ⇒ `'((#\$))`
    - rezultatul este o pereche între eticheta `"$"` (reprezentată ca `'(#\$)` - listă de caracterul `$`) și subarborele vid (reprezentat ca o listă vidă)
    - când construim o pereche între un element E și o listă L, ceea ce se întâmplă este că obținem o listă cu E urmat de toate elementele din L, de aceea rezultatul final este o

listă care conține doar eticheta `'(#\$)` (o listă care conține o altă listă)

- ex: `(other-branches st-banana)` ⇒ `'(((#\A) ((#\$)) ((#\N #\A) ((#\$)) ((#\N #\A #\$)))) ((#\B #\A #\N #\A #\N #\A #\$)) ((#\N #\A) ((#\$)) ((#\N #\A #\$))))`, adică o listă cu cele 3 ramuri în afară de prima (unde fiecare ramură este o pereche între un element (eticheta) și o listă (subarborele))

```
(get-branch-label branch)

(get-branch-subtree branch)
```

- get-branch-label primește o ramură a unui ST și întoarce eticheta acesteia
- get-branch-subtree primește o ramură a unui ST și întoarce subarborele de sub eticheta acesteia
- ținând cont că o ramură este o pereche între o etichetă și un subarbore, cele două funcții nu fac decât să extragă cele două componente
- ex: pentru branch definit ca `'((#\A) ((#\$)) ((#\N #\A) ((#\$)) ((#\N #\A #\$))))`:
  - `(get-branch-label branch)` ⇒ `'(#\A)`
  - `(get-branch-subtree branch)` ⇒ `'(((#\$)) ((#\N #\A) ((#\$)) ((#\N #\A #\$))))`

```
(get-ch-branch st ch)
```

- get-ch-branch primește un ST și un caracter ch și întoarce acea ramură a ST-ului a cărei etichetă începe cu caracterul ch, respectiv false în cazul în care nu există o asemenea ramură
- ex: `(get-ch-branch st-banana #\N)` ⇒ `'((#\N #\A) ((#\$)) ((#\N #\A #\$)))`
- ex: `(get-ch-branch st-banana #\Z)` ⇒ `#f`

```
(longest-common-prefix w1 w2)
```

- longest-common-prefix primește două cuvinte (liste de caractere) w1 și w2 și întoarce o listă formată din trei elemente: cel mai lung prefix comun al lui w1 și w2, restul lui w1 după eliminarea acestui prefix, restul lui w2 după eliminarea acestui prefix
- în cazul în care cele două cuvinte nu au un prefix comun, cel mai lung prefix comun este lista vidă

- ex: `(longest-common-prefix '(#\w #\h #\y) '(#\w #\h #\e #\n))` ⇒ `'((#\w #\h) (#\y) (#\e #\n))`

```
(longest-common-prefix-of-list words)
```

- longest-common-prefix-of-list primește o listă nevidă de cuvinte care încep cu același caracter și întoarce cel mai lung prefix comun al tuturor cuvintelor din listă
- având în vedere că toate cuvintele încep cu același caracter, rezultatul va fi mereu o listă nevidă de caractere
- ex: `(longest-common-prefix-of-list (list (string->list "when") (string->list "where") (string->list "why") (string->list "who")))` ⇒ `'(#\w #\h)`

```
(match-pattern-with-label st pattern)
```

- match-pattern-with-label se folosește pentru a căuta un șablon (un subșir) într-un text, folosind ST-ul asociat textului
- funcția primește un ST și un șablon (listă nevidă de caractere) și procedează astfel:
    - caută ramura din ST care s-ar putea potrivi cu șablonul (ramura care începe cu același caracter cu care începe șablonul)
    - dacă găsește o asemenea ramură, există 3 posibilități (iar formatul rezultatului diferă în funcție de cazul în care ne aflăm):
        1. dacă șablonul este conținut integral în etichetă, înseamnă că el este conținut în text, și atunci întoarcem true
        2. dacă eticheta este conținută integral în șablon, înseamnă că este în continuare posibil să găsim șablonul în text, și pentru asta va trebui să cercetăm subarborele de sub etichetă; în acest caz, întoarcem lista (etichetă, șablon nou, subarbore) care ne oferă informațiile despre ce s-a potrivit până acum (eticheta) și ce subșir (noul șablon) ne-a rămas de căutat în subarbore pentru a putea determina dacă șablonul inițial apărea în text (întoarcem și

subarborele pentru a ști unde să continuăm căutarea)

3. dacă șablonul și eticheta au un prefix comun dar nu se potrivesc până la final, putem conchide că șablonul nu apare în text, și atunci întoarcem lista (false, cel mai lung prefix comun între etichetă și șablon); prefixul comun nu ne folosește la căutare, dar ne folosește în rezolvarea altor aplicații din etapele următoare ale temei

- dacă nu găsește o asemenea ramură, atunci șablonul nu apare în text și întoarcem un rezultat de tipul celui de la cazul 3 anterior: lista (false, lista vidă) - pentru că practic cel mai lung prefix comun al șablonului cu orice etichetă este lista vidă

- ex: `(match-pattern-with-label st-banana (string->list "BABA"))` ⇒ `'(#f (#\B #\A))`, pentru că șablonul `"BABA"` s-a potrivit doar parțial cu eticheta `"BANANA$"`, deci ne încadrăm în cazul 3, iar cel mai lung prefix comun dintre șablon și etichetă este `"BA"`

```
(st-has-pattern? st pattern)
```

- st-has-pattern? primește un ST și un șablon și întoarce true dacă șablonul apare în ST, respectiv false dacă nu apare
- ex: `(st-has-pattern? st-banana (string->list "ANAN"))` ⇒ `#t`, pentru că șirul `"ANAN"` apare în textul `"BANANA"`
- acest lucru se determină astfel:
  - eticheta `"A"` este conținută în șablonul `"ANAN"`, așadar se va căuta noul șablon `"NAN"` în subarborele de sub eticheta `"A"`
  - eticheta `"NA"` este conținută în șablonul `"NAN"`, așadar se va căuta noul șablon `"N"` în subarborele de sub eticheta `"NA"`
  - șablonul `"N"` este conținut în eticheta `"NA$"`, așadar șablonul apare în text (și întoarcem true)