VILNIAUS KOLEGIJA

ELEKTRONIKOS IR INFORMATIKOS FAKULTETAS

Informacinių sistemų katedra

Struktūrinio programavimo kursinis darbas

Kursinis darbas

INFORMACINĖS SISTEMOS (IS24 grupė)

STUDENTAI AUGUSTINAS JAZGEVIČIUS

2025-03-16

DĖSTYTOJA lekt. Airina SAVICKAITĖ

Vilnius
2025

TURINYS

1. ĮVADAS

Ši kodo biblioteka yra aktuali tuo, kad Odin kalboje yra tik vienas šią problemą sprendžiantis projektas ir jis neveikia su tikro pasaulio puslapiais, būtent kuriems ir reikia tokios bibliotekos. Be to, nors kitos kalbos kaip: Rust, Java ir C jau turi HTML interpretatorius, jos turi kitų problemų, kurias Odin išsprendžia, pavyzdžiui: Java reikalauja projekto struktūros užteršimo norint naudoti C/C++ grafikos API, Rust yra ne tokia ergonomiška, C bibliotekų kompiliavimas Windows sistemoje yra sudėtingas.

Šio darbo tikslas yra sukurti ir aprašyti kodo biblioteką, skirtą perkelti HTML tekstą į masyvus ir struktūras programos atmintyje.

Darbo uždaviniai:
1. Sukurti kodo biblioteką
2. Aprašyti kodo biblioteką
3. Aprašyti ir parodyti kiekvieną esantį ir reikalaujamą kursinio darbo aspektą

2. PROGRAMOS APRAŠYMAS

Pagrindinis programos tikslas yra paversti paprastą Hypertext markup language dokumentą į šią duomenų struktūrą:

```
Element :: struct {
    type:       string,
    attrs:      map [string] string,
    text:       [dynamic] string,
    children:   [dynamic] ^Element,
    parent:     ^Element,
    ordering:   bits.Bit_Array,
}
```

Duomenų struktūroje yra apibrėžtas elemento tipas, pavadinimas, kaip, pavyzdžiui: „button“, arba „span“, attributai, tai yra rakto-reikšmės poros, kaip: „src=/next-image“, visas tekstas apsupantis dukterinius elementus (iš eilės), patys dukteriniai elementai, tėvinis elementas, ir teksto-elementų tvarka – bitų masyvas, kurio ilgis yra teksto gabalų ir vaikų kiekio suma, ir kur, jei bitas yra 1 tai reiškia, kad dabar eis kitas teksto gabalas, o jei 0, tai eis HTML elements (čia alternatyva sumos tipui, kitaip dar vadinamu: „union“)

Tačiau, prieš šios struktūros pildymą, kodo bibliotekoje dar yra du etapai: atskirimo į žetonus (Angl.: „tokenizer“) ir žetonų analizės (Angl.: „Lexer“) etapai. Pirmas suplėšo tektą į gabaliukus pagal tam tikras taisykles, atskiriami specialūs simboliai, tarpai, viskas kabutėse ir paprastas tekstas. Po to, leksinis analizatorius priskiria kiekvienam teksto gabaliukui kategoriją, kaip: „elemento pavadinimas“, „etiketės pradžia“, „elemento pabaiga“ ir t.t. ir gale duoda masyvą turintį teksto-tipo struktūras.

Kitas etapas yra „suvokimas“ (Angl.: „parsing“), anksčiau duotos struktūros užpildymo etapas. Čia yra viena pagrindinė rekursyvi funkcija, su ciklais, kurie eina per praeito etapo rezultatą ir pagal tipą ir dabartinę programos padėtį pildo jų dukterinį elementą. Pavyzdžiui, jei dabartinis elementas yra „pre“, reikia palikti visus tarpus ir tabuliacijos žymes.

Be to, biblioteka dar pateikia ir funkcijas daryti užklausas gauti norimus elementus, kaip „by_id“ ir „get_next_sibling“, čia irgi, tik dėl patogumo, automatiškai surenkami elementai į sąrašą. Ir dar, kadangi Odin kompiliatorius šiuo metu yra labai lėtas, naudoja LLVM, yra naudojamas tik vienas standartinės bibliotekos failas ir, vietoje to, maža teksto manipuliacijos dalis.

3. STRUKTŪRINIO PROGRAMAVIMO DALYS

3.1. Sąlygos sąkinys

„if“ teiginiai yra naudojami praktiškai visur programoje, štai pavyzdys iš „tokenizer“ dalies:

```
// ...
// skip: int
// for r, i in html {
    if skip > 0 {
        skip -= 1
        continue
    }
    // ...
```

Šio algoritmo reikia tam, nes Odin kalboje `string` tipas naudoja UTF-8 teksto formatą ir teksto simboliai čia nėra vienodo dydžio, todėl, per 1 `skip`, `i` gali pasistumėti per 1 baitą, bet ir per 2, ir per 3, ir per 4. Na ir, be to, Odin, iš tikro, neleidžia pakeisti `i` reikšmės.

Pats „skip“ mechanizmas yra tiesiog skirtas praleisti kelis simbolius, kai pavyzdžiui, surandama kabutė, tai iš kart surandama kita kabutė ir tekstas (imtinai) tarp kabučių yra pridedamas į rezultatą, tada šio teksto simbolių skaičius tiesiog, pridedamas prie skip ir kabučių tekstas praleidžiamas.

3.2. Ciklo sąkinys

Odin kalboje, iš tikro, nėra „while“ ciklo, yra tik „for“, bet moderni „for“ sintakė, padaro „while“ visiškai beprasmį, nes „while“ tai tiesiog yra „for“ vidurys. Čia pasirinktas mažas algoritmas surasti tikrą simbolį daugelį C šeimos programavimo kalbų kode (į HTML gali būti įterpta JavaScript, tačiau, mano atvejų, tai yra tiesiog tekstas, kuris gali turėti specialių simbolių):

```
escaped: bool; c: int // teksto simbolio indeksas
for r in a[skip:] {
    defer c += 1
         if escaped   do escaped = false
    else if r == '\\' do escaped = true
    else if r == b    do return c + skip
}
return len(a) - 1 + skip
```

JavaScript (ir, turbūt, CSS) sintaksėje, simbolis „\“ pasako, kad simbolis einantis po jo turėtų būti interpretuojamas kaip, tiesiog, tekstas, jis neturėtų turėti jokios įpatingos reikšmės (arba iš vis praleistas, arba pakeistas kitu), todėl, negalima tiesiog praeiti pro visus simbolius ir surasti reikiamą, užtat reikia naudoti anksčiau paminėtą kodą.

Beje `defer` raktažodis tiesiog pasako, kad po jo einantis teiginys eis paskutis šiame kodo bloke, šiuo atveju, `c` bus padidinta po **kiekvienos** iteracijos, net ir po `return`.

3.3. Masyvai

Pavyzdžiui, iš kitos MIT licensijuotos bibliotekos aš paimiau šį elementų sąrašą. Jam priklauso elementai, kuriems visai nereikia uždarančios etiketės ir, techniškai, turėti ją net gi būtų klaida, bet interneto naršyklės yra labai atlaidžios.

```
VOID_TAGS : [] string : {
    "meta", "link", "base", "frame", "img", "br",
    "wbr", "embed", "hr", "input", "keygen", "col", "command",
    "device", "area", "basefont", "bgsound", "menuitem", "param", "source", "track"
}
```

3.4. Paprasta funkcija

Funkcija, efektyviai, gauna 32 bitų sveikąjį skaičių, kuris Odin kalboje pavadintas „rune“, nors tai yra, šiaip, tik `char*` sąrašo dalies kopija. Ji duoda UTF-8 simbolio ilgį baitais.

```
rune_size :: proc(r: rune) -> int {
    assert(r >= 0)
    switch { // Dėl logikos viskas yra apversta, prasideda nuo 4 baitų, eina iki 1
    case r > 0x10ffff:  return 4 // 11110___  10______  10______  10______
    case r > 1 << 16 :  return 3 // 1110____  10______  10______  KITI SIMBOLIAI
    case r > 1 << 11 :  return 2 // 1100____  10______  KITI SIMBOLIAI
    } return 1                   // 0_______  KITI SIMBOLIAI
}
```

Ši funkcija yra paimta iš teksto sluoksnio, ji yra vidinė.

3.5. Rekursyvi funkcija

Čia yra pagrindinė „parser“ dalies funkcija, ji sukuria elementus ir, be to, ji, techniškai, yra procedūra, o ne funkcija, nes paima duomenis ir iš išorės („global scope“), ne tik argumentų sąrašo, o tie duomenys tai yra praeitos stadijos masyvas ir „dabartinio“ elemento indeksas, kas su pagalbinėmis funkcijomis `next()` ir `peek(amount: int)` išorėja praktiškai susidaro iteratoriaus duomenų struktūra, kurį `parse_elem` naudoja.

```
parse_elem :: proc(pre := false) -> ^Element {//{{{
    if peek().type != .ELEMENT do return nil

    elem := new(Element)
    elem.type = next().text

    pre := pre || any_of(elem.type, ..KEEP_WHITESPACE)
    is_inline := any_of(elem.type, ..INLINE_TAGS)

    for current < len(tokens) {
        #partial switch peek().type {
        case .A_KEY:
            if peek(1).type == .A_VALUE {
                elem.attrs[peek().text] = next(1).text
            } else {
                elem.attrs[peek().text] = "true"
                next()
```

```
            }
        case .TAG_END:
            next()

            if any_of(elem.type, ..VOID_TAGS) {
                return elem
            }

            has_closing_tag := is_closed(elem.type)
            inner_for: for current < len(tokens) {

                switch {
                    // ...

                case peek().type == .ELEMENT:
                    if any_of(peek().text, ..BLOCK_TAGS) {
                        if !has_closing_tag && eq(peek().text, elem.type) do return
elem
                        if is_inline do return elem
                    }

                    child := parse_elem(pre)
                    if child == nil do continue
                    append(&elem.children, child)
                    child.parent = elem
                    bits.set(&elem.ordering, bits.len(&elem.ordering), false)

                    // ...
                }
            }
        case .ELEMENT_END:
            if !ends_with(peek().text, elem.type) && peek().text != "/" do break
            if peek().type == .ELEMENT_END do next()
            if peek().type == .TAG_END do next()
            return elem

        case: current += 1
        } // switch
    } // for

    return elem
}//}}}
```

Pati funkcija pereina per žetonus ir ką daryti su kiekvienu iš jų. Ir, kai sutinka, `TAG_END`, elemento etiketės pabaigos žetoną („>“), kol elementas užsidaris, kiekvienam toliau einančiui `ELEMENT` tipo žetonui, vėl kreipiasi į save.

3.6. Rodyklės

Rodyklių naudojimą parodžiau jau pačioje pradžioje, duomenų struktūroje. Tačiau, čia, taip pat, naudojamos rodyklės gauti vidiniam HTML tekstui tarp dviejų elementų. Šią funkciją pateikti pačiai bibliotekai yra įpatingai svarbu, nes ji kliaujasi ant techninių, vidinių detalių, kurios gali

bet kada pasikeisti be pranešimo, kaip atminties vientisumo ir kopijų nedarymo. O kitaip, pačiam naudotojui, jos padaryti yra praktiškai neįmanoma (arba reikalautų didesnių struktūrų).

```
inner_html :: proc(elem: ^Element) -> string {

    // Čia nieko įpantigo, tiesiog addr() yra privati šiam blokui
    addr :: proc(ptr: [^]u8) -> u64 {
        return transmute(u64) ptr
    }

    get_last_text :: proc(elem: ^Element) -> string {
        is_last_text := bits.get(&elem.ordering, bits.len(&elem.ordering) - 1)
             if is_last_text do             return last(elem.text)^
        else if len(elem.children) > 0 do \
            return get_last_text(last(elem.children)^)
        if len(elem.text) > 0 do            return elem.text[0]
                                            return last(elem.parent.text)^
    }

    if len(elem.text) == 0 do return ""
    from := raw_data(elem.text[0])
    l: u64

    to := get_last_text(elem)
    l = addr(raw_data(to)) + u64(len(to)) - addr(from)

    return string( (transmute([^]u8) from)[:l] )
}
```

Žinoma, kad tikrai buvo pradėta su vientisu `string` ir kad `Element::text` teksto dalys neturi vėliau padarytų kopijų, taigi galima tiesiog suskaičiuoti, kur atmintyje prasideda pirmo elemento vaiko pirmas teksto gabalas ir kur pasibaigia paskutinio elemento vaiko paskutinis teksto gabalas ir taip gauti, nuo kur iki kur atmintyje yra originalus HMTL tekstas. Po to, viskas yra taip lengva kaip tiesiog prisimenant, kad Odin kalboje string yra tiesiog: `struct string { char* C_string; int len; }` ir naudoti idiomatišką sintaksę paversti `unsigned long long` į `char*`, į `char[]` ir tada į `string`.

IŠVADOS

1. Pabaigus pirmąją bibliotekos iteraciją, buvo pastebėta, kad HTML, nekaip XML ir daugelis kitų maketavimo kalbų, dokumento struktūra priklauso nuo elementų etikečių pavadinių, taigi reikia turėti ir kelis elementų tipų sąrašus ir kiekvienam elementui nustatyti kuriai grupiai jis priklauso.
2. Kadangi programa nedaro teksto alokacijų, galima gauti neapdoroto teksto dalį iš, tiesiog, dviejų atminties adresų.
3. Kadangi UTF-8 simboliai nėra vienodo dydžio, suskaičiuoti kiek simbolių yra baitų masyve, yra O(n) operacija ir dėl to, for cikle, yra gerai turėti trečią kintamąjį: „skip“, kad būtų galima praleisti daugiau kaip vieną UTF-8 simbolį.
4. Testavimo fazėje buvo atrasta, kad HTML parseris turi subegėti praleisti JavaScript kodą, nes `if(4<html>0) console.log("</script>") //<b>` yra galiojantis JavaScript kodas.
5. Aprašius kodo biblioteką, pastebėta kad yra tik kelios funkcijos, nepaisant labai reukursyvaus HTML formatas, tai reiškia, kad formatas yra pakankamai enkapsuliuotas, retai reikia suprasti toliau einančią informaciją.

LITERATŪROS SĄRAŠAS

1. Jonathan Hedley and jsoup contributors. (2025). *JSoup*. https://jsoup.org
2. The Web Hypertext Application Technology Working Group. (2025). *HTML Standard*. https://html.spec.whatwg.org/
3. Mozilla Developer Network. (n.d.). *HTML elements reference*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element

PRIEDAI

Bibliotekos kodas Github platformoje: https://github.com/Up05/ohtml