VILNIAUS KOLEGIJA

ELEKTRONIKOS IR INFORMATIKOS FAKULTETAS

Informacinių sistemų katedra

Struktūrinio programavimo kursinis darbas

Kursinis darbas

INFORMACINĖS SISTEMOS (IS24 grupė)

STUDENTAI AUGUSTINAS JAZGEVIČIUS

2025-03-11

DĖSTYTOJA lekt. Airina SAVICKAITĖ

Vilnius
2025

TURINYS

1. ĮVADAS

Mano kodo biblioteka yra aktuali tuo, kad Odin kalboje dar yra tik vienas šią problemą bandantis išspręsti 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 C/C++ grafikos API, tam, kad mes galėtume naudoti greitą ciklišką grafinės sąsajos darymo metodą (išsivaizduokite pagrindinį „while(!glfwWindowShouldClose())“, kurį turite su GLFW, o ne „frame.repaint()“, kurį turite su Swing), Rust, mano nuomone, yra labai neergonomiška, C irgi reikalautų sudėtingo, sunkiai patikrinamo dėl virusų, projekto.

Šio darbo tikslas yra aprašyti mano programą, skirtą paversti HTML tekstą į Odin programavimo kalboje apibrėžtas duomenų struktūras ir sukurti lengvą ir patogų būdą pasiekti šiuos duomenis.

Darbo uždaviniai:
1. Aprašyti programą
2. Aprašyti ir parodyti kiekvieną reikalaujamą programos 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, pagrinde: atskiriamas 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.

Po to eina „supratimo“ (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, kad programa kompiliuotusi daug greičiau (šiuo metu, Odin naudoja LLVM siaubingai lėtą, milžinišką ir pasenusę kompiliatoriaus posistemę) aš parašiau mažą savo teksto manipuliacijos dalį.

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
    }
    // ...
```

Čia yra vienas iš mano dažniausiai naudojamų kodo fragmentų tokio tipo programose. Nors, iš pirmo žvilgsnio, atrodo, kaip resursų švaistymas, kodėl tiesiog nepridėti `skip` prie `i`? Na,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“ visiška beprasmį, nes „while“ tai tiesiog yra „for“ vidurys. Čia pasirinkau mažą algoritmėlį surasti tikrą simbolį JavaScript kode (JavaScript gali būti įterptas į HTML, 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, aš negaliu tiesiog praeiti pro visus simbolius ir surasti tą kurio noriu, užtat reikia naudoti šį mažą algoritmą, kurį, vėl, esu naudojęs jau, turbūt, apie dešimt kartų.

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

Čia dar viena iš mano mėgstamiausių funkcijų:

```
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
}
```

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.

3.5. Rekursyvi funkcija

Radau tik didžiulę rekursyvę funkciją:

```
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 == .TEXT:
                    append(&elem.text, peek().text)
                    bits.set(&elem.ordering, bits.len(&elem.ordering))
                    next()

                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 peek().type == .WHITESPACE:
                    txt := peek().text if pre else trim_ws(peek().text)
                    append(&elem.text, peek().text)
                    bits.set(&elem.ordering, bits.len(&elem.ordering), true)
                    next()

                case peek().type == .ELEMENT_END && ends_with(peek().text,
elem.type):
                    return elem

                case:
                    if peek().type == .ELEMENT_END do next()
                    else do break inner_for
                }
            }
        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
}//}}}
```

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.

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. *Čia turiu mažą rekursyvę funkciją, bet jau pabaigiau praeitą skyrių, tai paliksiu kaip yra.*

```
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] )
}
```

Funkcija yra skirta gauti pačioje pradžioje esančiam, neapdorotam, tekstui, kalbose su šiukšlių rinkėjais, ši funkcija būtų neįmanoma, bet, kadangi aš žinau, kad pradėjau su vientisu `string` ir mano `Element.text` masyvai neturi vėliau padarytų kopijų, ar ko nors panašaus, aš galiu 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`.

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