Rust i napredni tipski sustavi

6. Rust - Jednostavan interpreter kalkulator jezika

Ivan Radiček

24. travnja 2021.

Plan

- Do sad:
  - Osnove Rusta (let, funkcije, struct, enum, trait, impl, ...)
  - Memorija u Rustu (vlasništvo, posudba, lifetime, pametni pokazivači, ...)
- Dalje:
  - Implementacija jednostavnog interpretera za "kalkulator" jezik u Rustu
  - Kroz implementaciju ćemo učiti i još neke sitnice vezane uz Rust
    - Npr. danas ćemo malo o modulima, macroima, funkcijama

Moduli - crash course

- Svrha modula:
  1. Sakrivanje implementacijskih detalja (enkapsulacija, apstrakcija)
  2. Logička organizacija veće količine koda
- Modul možemo kreirati sa `mod Name` ili stavljajući kod u različite datoteke
  - Svaki modul može sadržavati i pod-module (hierarhija modula)
- Sadržaj modula je generalno **nedostupan/nevidljiv van modula**, osim ako se ne označi sa `pub`
- Nekom elementu (tip, varijabla, trait, ...) nekog modula pristupamo s `modul::podmodul1::podmodul2::Element`
  - Ako neki modul često koristimo možemo, njegove elemente možemo uvesti i sa `use m1::m2::X` (jedan element), `use m1::m2::{self, X1, X2}` (više elemeneata) ili `use m1::m2::*` (svi elementi)
- Knjiga: https://doc.rust-lang.org/book/ch07-00-managing-growing-projects-with-packages-crates-and-modu html

Primjer modula i korištenje

```
1   mod point {
2     #[derive(Debug)]
3     pub struct Point {
4       x : f64,
5       y : f64
6     }
7
8     impl Point {
9       pub fn new(x: f64, y : f64) -> Point {
10        Point { x, y }
11      }
12
13      pub fn get_x(&self) -> f64 {
14        self.x
15      }
16    }
17  }
18
19  fn main() {
20    use point::Point; // Bez ovoga bi morali pisati point::Point
21    // Ne funkcionira (osim ako su x i y pub): let p = Point { x : 1.0, y : 2.0 };
22    // Ovo funkcionira
23    let p = Point::new(1.0, 2.0);
24    // Ne funkcionira: let x = p.x;
25    let x = p.get_x();
26    println!("p = {:?}, x = {}", p, x)
27  }
```

- Zašto je sakrivanje/apstrakcija/enkapsulacija bitna? Skrivanje **detalja implementacije** i otkrivanje samo stabilnih javnih svojstava! Npr:
  - String je interno samo vektor, no to se može i promijeniti, a korisnik to ne mora ni znati jer je sakriveno
  - Slično i sa našim tipom racionalnih brojeva sa početka predavanja
  - Ako su komponente strukture `pub`, a struktura je `mut`, možemo mijenjati pojedine komponente (nekad i nije poželjno)

```
1   mod point {
2     #[derive(Debug)]
3     pub struct Point {
4       x : f64,
5       pub y : f64
6     }
7     /* ... */
8   }
9
10  fn main() {
11    use point::Point; // Bez ovoga bi morali pisati point::Point
12    let mut p = Point::new(1.0, 2.0);
13    p.y = 42.0; // p.x ne možemo promijeniti
14    println!("p = {:?}", p)
15  }
```

Crateovi i datoteke

- Najveća jedinica koda je `crate` (ekvivalent paketa u JS/Pythonu)
  - Postoje `bin` (izvršni) i `lib` (biblioteke)
  - Dobijemo ga sa `cargo new <crate_name>` (za `lib` treba ispred imena dodati `--lib`)
- Svaka `.rs` datoteka je modul, a direktorij pod-modul
  - `main.rs`, `lib.rs` i `mod.rs` su posebna imena za *top-level* programa (bin), biblioteke (lib) i pod-modula
- Crateovima, modulima i pod-modulima možemo navigirati sa:
  - `use crate::mod1::mod2` (apsolutna putanja - `/mod1/mod2`)
  - `use super::mod1::mod2` (relativno krenuvši od nad-modula - `../mod1/mod2`)
  - `use self::mod1::mod2` (relativno krenuvši od *sebe* - `./mod1/mod2`)
- Postoji još i **workspace** koji može kombinirati `bin` i `lib` - korisno za odvajanje osnovne logike i same aplikacije
- Nećemo u detalje jer bi potrošili previše vremena, već ćemo po potrebi proširivati - za domaću zadaću proučiti detalje ovih pojmova u *knjizi* :)

Podsjetnik : programski jezik

- Sastoji se od **sintakse** i **semantike**
- Sintaksa opisuje valjane programe
- A semantika im daje značenje
- U implementaciji: sintaksa će string pretvoriti u neku strukturu koja predstavlja program, a semantika je fukcija koja može *izvršiti* program u nešto konkretno
- Ajmo vidjeti na primjeru . . .

Jezik kalkulatora

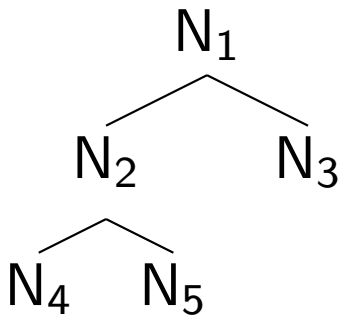
- Ovo su neki ispravni izrazi (nizovi znakova):
  - 42 + 1 * 3
  - (42 + 1) * 3
- Ovo su neki nespravni nizovi:
  - 3 * + 4
  - 1 2 3
  - 2) * -
- Ovo su značenja/izvršavanje *ispravnih* nizova:
  - 42 + 1 * 3 $\Rightarrow$ 45
  - (42 + 1) * 3 $\Rightarrow$ 129

Sintaksa kalkulatora

- Neki ispravan izraz e se može konstruirati na sljedeće načine:
  - $\overline{n}$ - niz znakova koji predstavljaju cijeli broj
    - Napomena: sa n ćemo označavati cijeli broj, a sa $\overline{n}$ znakovnu reprezentaciju)
    - Ili u programiranju: n (npr. 42) je broj, a $\overline{n}$ je string (npr. "42")
  - $e_1 + e_1$, $e_1 * e_2$, ... (gdje e_1 i e_2 predstavljaju neke pod-izraze - definicija je **rekurzivna**)
  - (e') (e' je isto pod-izraz)
- U literaturi programskih jezika se to obično piše na sljedeći način:
  $e := \overline{n} \mid e_1 + e_2 \mid e_1 * e_2 \mid (e')$

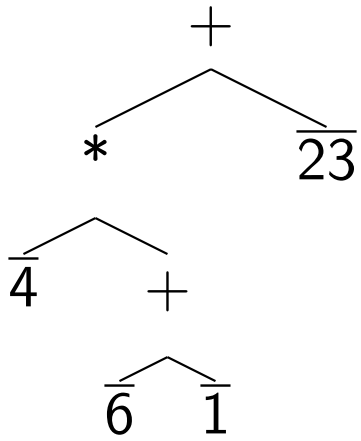
Stablo u matematici

- **Stablo** je matematička struktura, korisna u internoj reprezentaciji sintakse:
  - Struktura koja se sastoji od **čvora** (node), a svaki čvor može imati i pod-čvorove (*child node*)
  - Za čvor koji nema pod-čvorova kažemo da je list (*leaf node*)
  - U stablu ne postoje ciklusi, tj. slijedeći put od nekog čvora kroz pod-čvorove ne možemo doći natrag do čvora iz kojeg smo krenuli
  - Programski izrazi čine *konačna* stabla



(Apstraktno) sintaksno stablo

- Kada je neki izraz znakova ispravan, tada možemo konstrukirati apstraktno stablo izraza
- Primjer za izraz: $4 * (6 + 1) + 23$:



- Stablo se zove **apstraktno** jer izostavlja detalje niza znakova: formatiranje, praznine, zagrade (!), ...
  - Zagrade, kao i prvenstvo operatora je **implicitno** u stablu i smatra se implementacijskim detaljom
- Vidi više: https://en.wikipedia.org/wiki/Abstract_syntax_tree

Sintaksno stablo u Rustu

Vrlo jednostavna i prirodna implementacija

```
1   #[derive(Debug)]
2   pub enum Op {
3     Plus,
4     Times
5   }
6
7   #[derive(Debug)]
8   pub enum Expr {
9     Num(String),
10    BinOp(Op, Box<Expr>, Box<Expr>)
11  }
12
13  fn main() {
14    // Brojevi
15    let n42 = Expr::Num("42".to_string());
16    let n2 = Expr::Num("2".to_string());
17    let n1 = Expr::Num("1".to_string());
18    // 42 * 1 + 2
19    let e = Expr::BinOp(
20      Op::Plus,
21      Box::new(Expr::BinOp(Op::Times, Box::new(n42), Box::new(n1))),
22      Box::new(n2));
23    println!("e = {:?}", e)
24  }
```

Kako doći od string do Expr-a?

- Kako pretvoriti `"42 * 1 + 2"` u `BinOp(Plus, BinOp(Times, Num(42), Num(1)), Num(2))`?

Kako doći od string do `Expr`-a?

- Kako pretvoriti `"42 * 1 + 2"` u `BinOp(Plus, BinOp(Times, Num(42), Num(1)), Num(2))`?
- Ručna implementacija je *dosadna* i *lagano pogriješiti*!
- Postoje generatori parsera, kojima opišemo sintaksu i automatski dobijemo kod koji pretvara `string` u `Expr`

Kako doći od string do `Expr`-a?

- Kako pretvoriti `"42 * 1 + 2"` u `BinOp(Plus, BinOp(Times, Num(42), Num(1)), Num(2))`?
- Ručna implementacija je *dosadna* i *lagano pogriješiti*!
- Postoje generatori parsera, kojima opišemo sintaksu i automatski dobijemo kod koji pretvara `string` u `Expr`
- Proces se obično sastoji od 2 faze:
  - Razbijanje (**lexing**) stringa u minimalne jedinice (**token**e); u našem jeziku: `(`, `)`, $*$, $+$, $\overline{n}$ (broj)
  - **Parsiranje** niza tokena u sintaksno stablo; u našem primjeru: $e := \overline{n} \mid e + e \mid e * e \mid (e)$ (za svaku varijantu dodajemo i kod koji konstruira `Expr`)

Kako doći od string do `Expr`-a?

- Kako pretvoriti `"42 * 1 + 2"` u `BinOp(Plus, BinOp(Times, Num(42), Num(1)), Num(2))`?
- Ručna implementacija je *dosadna* i *lagano pogriješiti*!
- Postoje generatori parsera, kojima opišemo sintaksu i automatski dobijemo kod koji pretvara `string` u `Expr`
- Proces se obično sastoji od 2 faze:
  - Razbijanje (**lexing**) stringa u minimalne jedinice (**token**e); u našem jeziku: (,), $*$, +, $\overline{n}$ (broj)
  - **Parsiranje** niza tokena u sintaksno stablo; u našem primjeru: $e := \overline{n} \mid e + e \mid e * e \mid (e)$ (za svaku varijantu dodajemo i kod koji konstruira `Expr`)
- Kako opisati **lexer** i **parser**? - pomoću **meta-programiranja**/**makro**a!

Makroi i meta-programiranje

- Glavna ideja: *kod je samo vrsta podatka* (tip) i pomoću funkcija koje transformiraju te podatke (kod) možemo graditi nove apstrakcije (možemo graditi nove specijalizirane mini-jezike)
- Vrlo različito od makro definicija u C-u!
- Jezik poput Lispa je tu ideju doveo do ekstrema: i kod i podaci su samo liste (dopušta moćno proširivanje jezika)

Primjeri meta-programiranja u Rustu

- `#[derive]` za automatsko implementiranje nekih traitova
- `println!(...)` za formatirani ispis na standardni izlaz
- `vec![e1, ..., en]` kreira novi vektor sa elementima $e_1, ..., e_n$
  - Simplificirana verzija: igralište (vidi "Tools" $\rightarrow$ "Expand macros")
- Attributi na elementima kao funkcija; npr u web programiranju:

```
1  #[route(GET, "/")]
2  fn index() {
3      ...
4  }
```

- Kao funkcija, npr. za validaciju SQL upita:
  `let sql = sql!(SELECT * FROM posts WHERE id=1);`
- Knjiga: https://doc.rust-lang.org/book/ch19-06-macros.html

Definicija parsera pomoću meta-programiranja - `pomelo`

- `pomelo` (https://docs.rs/pomelo/0.1.5/pomelo/) definira makro `pomelo! { ... }` sa kojim možemo opisati naš jezik u Backus-Naur formi (BNF, više: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form)
- Opis *gramatike* jezika se tada automatski pretvara u parser (funkcija koja prima niz tokena i vraća AST - ili neki drugi proizvoljni rezultat)
- Napomena: u `Cargo.toml` treba pod `[dependencies]` dodati `"pomelo" = "*"` (`"*"` označava bilo koju verziju cratea `pomelo`)

BNF kao opis jezika

- Gramatika jezika sastoji se od definicija (simbola, non-terminals), od kojih je jedan simbol glavni (top), i tokena (završnih jedinica, terminals)
  - Za naš jednostavni jezik biti će dovoljan samo jedan simbol
- Svaka definicija sastoji se od opisa mogućih varijanti (tog simbola)
- Svaka varijanta je niz simbola (može i rekurzivno) i tokena (završnih jedinica)
- Primjer kalkulatora:
  - Jedini simbol: e
  - Tokeni (završne jedinice): (,), +, ∗, $\overline{n}$
  - Definicija sa 4 varijante: $e := \overline{n} \mid e + e \mid e * e \mid (e)$
- Kod definicije parsera, svakoj varijanti treba pridružiti i kod koji kreira AST (općenito proizvoljan kod koji se još naziva i "*akcija*")
- Vidi npr za Python (iako nije čisti BNF):
  https://docs.python.org/3/reference/grammar.html

BNF notacija u pomelo-u

- Svaka varijanta na posebnoj liniji u obliku
  `sym ::= X1(x1) ... XN(xn) { <action> }`
- `X` može biti *token* (terminal) ili *symbol* (non-terminal)
  - Razlikujemo ih po prvom slovu: token počinje velikim slovom, simbol počinje malim slovom
- Svaki simbol ima svoj tip (`Expr` u našem slučaju), te tako za neki `Xi(xi)` u kodu možemo koristiti varijablu `xi`
- Token može i ne mora imati vrijednost (pa samim tim i tip), npr:
  - token `(` nema vrijednost
  - token $\overline{n}$ ima vrijednost (string)
- Za svaki simbol i token koji ima vrijednost moramo definirati tip pomoću: `%type X T`

Početak definicije gramatike

```
1  use pomelo::pomelo;
2
3  pomelo! {
4    // Sa %include ubacujemo proizvoljan Rust kod
5    // Inače definicija nije vidljiva u makro-u
6    %include { use super::Expr; }
7
8    %type output Expr;
9    %type expr Expr;
10   %type Number String;
11
12   // Output ima samo jednu varijantu
13   output ::= expr(e) { e }
14
15   expr ::= Number(n) { Expr::Num(n) }
16   expr ::= LParen expr(e) RParen { e }
17 }
```

Ostatak primjera

- Makro `pomelo!` generira modul `parser` koji sadrži tipove `Parser` i `Token` (sve se može konfigurirati)
- Pomoćna funkcija `parse` koja pretvara vektor tokena u izraz (`Expr`)

```
1   use parser::{Parser, Token};
2
3   fn parse(toks : Vec<Token>) -> Result<Expr, ()> {
4     let mut p = Parser::new();
5     for tok in toks.into_iter() {
6       p.parse(tok)?;
7     }
8     p.end_of_input()
9   }
10
11  fn main() {
12    let toks = vec![
13      Token::LParen,
14      Token::Number("3.0".to_string()),
15      Token::RParen
16    ];
17    let e = parse(toks).unwrap();
18    println!("Expr: {:?}", e);
19  }
```