NEXT Sukkertoppen Gymnasium
Programmering B
Sebastian s2o

Årsprøve i Programmering B

Indledning

Wavefront .OBJ er et udbredt filformat til at gemme forskellige former for geometri. Det blev oprindeligt udviklet til Wavefront Technologies animations-pakke. Filformatet er open-source og er senere blevet adopteret af mange andre 3D-grafik og animations applikationer, hvilket gør det meget universelt, fordi alle understøtter det. Filformatet udmærker sig ved at være i læseligt format, altså man kan blot åbne filen og læse eller ændre i filen uden at skulle gøre det gennem et specielt program. Dog betyder dette, at filerne oftest fylder mere end, hvis de er serialiseret til binær.

Problemformulering

Hvordan kan man bruge Rusts abstraktioner (funktionel- og generisk programmering), til at skrive en Parser, der kan indlæse filer i OBJ-formatet.

Teori

Extended Backus-Naur Form (EBNF)

Extended Backus-Naur Form, forkortet EBNF, er et metasyntaks sprog, der er lavet til at udtrykke kontekstfri grammatik i eks. programmeringssprog eller filformater. EBNF er yders simpelt. Det består af blot 8 forskellige regler. Dog kan det let blive komplekst. Det smarte med EBNF er, at man kan opdele sin grammatik i små del-komponenter, som gør det lettere at parse senere[1].

Parsing

Parsing er, når man tager tokens som input og undersøger om inputtet følger en specificeret grammatik. Dette kaldes også for syntaktisk analyse[2].

Overvej følgende EBNF-kode:

```
1   faceElement = int32, [, ("/", int32 [, "/" int32])] | ("//", int32);
2   face = "f", whitespace, faceElement, {whitespace, faceElement};
```

Først defineres ”faceElement” som er enten 1, 2 eller 3 heltal, separeret af en skråstreg. Desuden kan den også bestå af et heltal efterfulgt af to skråstreger og et heltal mere. Altså må 3 følgende kombinationer valide: ”1”, ”1/2/3”, ”1//3”. Derefter defineres ”face” som starter med ”f” efterfulgt af en eller flere mellemrum og mindst et ”faceElement”, hvor alle elementerne er separeret af en eller flere mellemrum.

En parser til ”faceElement” ville starte med at undersøge om det første i inputtet bestod af et heltal. Derefter ville den kontrollere hvorvidt det næste input var en skråstreg efterfulgt af endnu et heltal, hvor der måske var endnu en skråstreg og heltal efter. Hvis den til gengæld finder to skråstreger, ville den blot forvente et enkelt heltal. Dette bliver hurtigt kompliceret med regler inden i regler og med valgfri segmenter, der måske eller måske ikke eksisterer.

[1] https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form
[2] https://en.wikipedia.org/wiki/Parsing#Computational_methods

Combinators

Parser combinators, ofte bare kaldet combinators, er en højere-orden funktion, der tager en parser og eventuelle andre argumenter og returnerer en ny parser, som så kan parse noget andet[3]. Eksempelvis følgende:

```rust
/// Parses a vertex normal in (x, y, z) format.
///
/// `vertexNormal = "vn", whitespace, f32, whitespace, f32, whitespace, f32;`
pub fn vertex_normal<'a>() -> impl Parser<'a, VertexNormal> {
    let coordinate = whitespace1().then(f32()).right();
    literal("v").then(coordinate.repeat::<3>()).right()
}
```

”.then” er en combinator, der tager en anden parser ind som argument og kører den i forlængelse af den anden. Desuden er ”.right” en combinator, der smider resultatet til venstre væk. Den konverterer altså en type af ”Left, Right” om til ”Right”. ”.repeat” er en helt tredje combinator, der gentager parseren ”N” gange og giver alle resultaterne ud som en array.

Functional Programming

Functional Programming eller funktionel programmering på dansk er et paradigme, hvor programmer består af funktioner der er kædede sammen. De laver så værdier om til andre værdier. I funktionel programmering kan variabler kan initialiseres til funktioner, funktioner kan bruges som argumenter til andre funktioner eller blive returneret fra en anden funktion. Det sidste tilfælde kaldes også en higher-order funktion. Funktionel programmering bliver ofte brugt som synonym for udelukkende ren funktionel programmering”, hvor ens program ikke kan have sideeffekter. En funktion uden sideeffekter returnerer altid det samme output for et givent input - Ligesom matematiske funktioner. Rust er et af de programmeringssprog, der har indbygget dele af funktionelle koncepter[4].

Generic Programming

Generic Programming eller Generisk Programmering, også bare kaldet Generics er en type af programmering, hvor algoritmerne og funktionerne er skrevet med ikke-kendte typer. Når funktionerne så bliver brugt, finder compileren ud af hvilke typer, der skal være der i stedet for. Overvej følgende Rust kode:

```rust
#[derive(Clone, Copy)]
3 implementations
pub struct Right<A, O, U>(A, PhantomData<(O, U)>);

impl<'a, O, U, A: Parser<'a, (O, U)>> Parser<'a, U> for Right<A, O, U> {
    fn parse(&self, input: &'a str) -> ParseResult<'a, U> {
        self.0.parse(input).map(|((_, right), rest)| (right, rest))
    }
}
```

[3] https://en.wikipedia.org/wiki/Parser_combinator
[4] https://en.wikipedia.org/wiki/Functional_programming

På linje 200 deklareres Right. Right er en struktur bestående af 3 generiske typer, der ikke er konkret specificeret; A, O og U. Nedenunder på linje 202 implementeres traitet Parser, hvis nogen betingelser er opfyldt. Hvad et trait er, snakkes om senere.

Parser<'a, U> implementeres for typen Right<A, O, U>, hvis A er en parser, der returnerer en to-komponents værdi, også kaldet en tuple. Dette betyder altså, at flere forskellige parsere kan kalde ".right()", så længe deres output er en tuple bestående af to komponenter. Bemærk, hvordan den nye parsers output er typen U og ikke (O, U).

Traits (Interfaces)

Traits er en form for interfaces, som man kender dem fra Java eller C#. Dog er der alligevel nogen forskellige, der ikke er relevante her[5]. Traits og Interfaces løser problemet i generisk kode, med hvordan skal man vide hvilke funktioner og metoder man kan kalde på en generisk type. Derudover også bare for at sørge for at noget funktionalitet ikke virker, hvis en generisk type ikke implementerer et interface. Når man implementer et interface indgår man en kontrakt med brugeren af interfacet.

```
trait Parser<'a, Output> {
    fn parse(&self) -> Result<(Output, &'a str), ParseError>;
}
```

"Parser" er et trait, som har en enkelt funktion, parse, der returner en værdi af typen "Output", hvis parseren lykkedes. Ellers retunerer den en "ParseError". Dette betyder altså, at man kan kalde ".parse" på alle typer, der implementerer "Parser" traitet.

Funktionsbeskrivelse

Programmet udviklet, er et bibliotek, der kan benyttes af andre rust-programmører til deres egne 3D-grafik applikationer eller biblioteker. Biblioteket har et meget simpelt interface. Man skal blot kalde "parse()" og give den en string som måske eller måske ikke er i OBJ-format. Den så parser det ud til en "Obj" struktur.

```
▶ Run | Debug
fn main() {
    let file = include_str!("../test.obj");
    println!("{:?}", parse(file))
}
```

Ovenstående program læser en .OBJ-fil in på compile-time, hvorefter "parse()" så kaldes med filens indhold som input. Derefter printes resultatet i terminalen. For at bygge programmet skal rustup være installeret og sat til nightly. Derefter skrives "cargo build" i terminalen, mens man er inde i topmappen, for at bygge biblioteket. Integrationstesten kan køres ved at skrive "cargo test".

[5] https://stackoverflow.com/questions/69477460/is-rust-trait-the-same-as-java-interface

Dokumentation

```rust
#[derive(Clone, Copy)]
3 implementations
pub struct Then<A, B>(A, B);

impl<'a, O, U, A: Parser<'a, O>, B: Parser<'a, U>> Parser<'a, (O, U)> for Then<A, B> {
    fn parse(&self, input: &'a str) -> ParseResult<'a, (O, U)> {
        match self.0.parse(input) {
            Ok((o, rest)) => self.1.parse(rest).map(|(u, rest)| ((o, u), rest)),
            Err(e) => Err(e),
        }
    }
}
```

Ovenstående koder viser hvordan ”.then()”-combinatoren er implementeret. Parser<’a, (O, U)> implementeres for Then<A, B>, hvis A implementerer Parser<’a, O> og B implementerer Parser<’a, U>. I ”parse” funktionen prøves der først at parses med A, hvorefter resten af inputtet parses med B. Hvis begge er succesfulde, så returneres (O, U). Ellers returneres en ”ParseError”.

```rust
#[derive(Clone, Copy)]
4 implementations
pub struct Repeat<A, const N: usize>(A);

1 implementation
trait RepeatInit<O> {
    const INIT: MaybeUninit<O> = MaybeUninit::uninit();
}

impl<'a, O, A: Parser<'a, O>, const N: usize> RepeatInit<O> for Repeat<A, N> {}

impl<'a, O, A: Parser<'a, O>, const N: usize> Parser<'a, [O; N]> for Repeat<A, N> {
    fn parse(&self, mut input: &'a str) -> ParseResult<'a, [O; N]> {
        let mut output: [MaybeUninit<O>; N] = [Self::INIT; N];
        for i in 0..N {
            let (o, rest) = self.0.parse(input)?;
            input = rest;
            output[i] = MaybeUninit::new(o);
        }

        Ok((output.map(|item| unsafe { item.assume_init() }), input))
    }
}
```

Ovenstående koder viser hvordan ”.repeat::<N>()”-combinatoren er inplementeret. ”.repeat()” returnerer en anden parser, hvor outputtet er en array af typen O med størrelsen N. Altså kan den allokere alle sine data på stakken og på den måde køre hurtigere, fordi den undgår dynamiske allokeringer.

På linje 144 initialisere en output array som alle de forskellige iterationers resultater skal gemmes i. Derefter loopes gennem 0 til N og hver iterations parses. På linje 151 returneres outputtet. Det forventes at alle pladserne i arrayet er initialiserede. Derfor kaldes ”.assume_init()” på hvert element.

```rust
#[derive(Clone, Debug)]
2 implementations
pub enum Value {
    Empty,
    Vertex(Vertex),
    TextureCoordinate(TextureCoordinate),
    VertexNormal(VertexNormal),
    VertexParameter(VertexParameter),
    Face(Face),
    Line(Line),
}

pub fn value<'a>() -> impl Parser<'a, (Value, Option<&'a str>)> {
    empty()
        .to(Value::Empty)
        .or(vertex().map(|v| Value::Vertex(v)))
        .or(texture_coordinate().map(|v| Value::TextureCoordinate(v)))
        .or(vertex_normal().map(|v| Value::VertexNormal(v)))
        .or(vertex_parameter().map(|v| Value::VertexParameter(v)))
        .or(face().map(|f| Value::Face(f)))
        .or(line().map(|l| Value::Line(l)))
        .then(comment().or_not())
        .or(comment().map(|s| (Value::Empty, Some(s))))
}
```

”value()” er den sidste parser, der parser hver af de forskellige .OBJ-strukturer. Efterfulgt af en valgfri kommentar.

```rust
pub fn parse(input: &str) -> Result<Obj, ParseError> {
    let mut obj = Obj::default();
    let mut values = Vec::default();

    for line in input.lines() {
        values.push(value().parse(line)?)
    }

    values.into_iter().for_each(|(v, _)| obj.add(v.0));
    Ok(obj)
}
```

.OBJ-formatet er opbygget således at hver form for struktur eller element er på en enkelt linje. Derfor itereres over hver linje og værdien gemmes i et array. Hvis der er en fejl i nogen af elementerne, så returnerer funktionen en fejl. Bemærk spørgsmålstegnet på linje 595.

Til sidst itereres over alle de fundne værdier og hver tilføjes til den samlede struktur, hvorefter denne returneres.

Testning

For at teste programmet, er der skrevet en integrationstest. Integrationstesten tester en .OBJ-fil og tjekker om parseren kan håndtere inputtet.

```
    Finished test [unoptimized + debuginfo] target(s) in 0.01s
     Running unittests src\lib.rs (target\debug\deps\obj-b11872bb859f6080.exe)

running 1 test
test tests::integration ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Parseren virker, fordi integrations-testen passerer

Perspektivering

Den resulterende parser er meget simpel og basal. Den kan ikke håndtere alle de forskellige strukturer i .OBJ-formatet. Dog er den bygget på en sådan måde, at man nemt kan integrere de manglende dele. Desuden er den resulterende struktur stadig meget basal og der kan forbedres mange ting som iteration over de forskellige faces.

Konklusion

Det kan konkluderes, at funktionel programmering sammen med generisk programmering og combinators er en god måde at bygge parsere på i rust til forskellige formål.