BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Szakdolgozat

Monadikus Parser Kombinátorok

TÉMAVEZETŐ:

DR. CSATÓ LEHEL,
EGYETEMI ADJUNKTUS

SZERZŐ:

DÉGI NÁNDOR

2024

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Monadic Parser Combinators

ADVISOR:

LEHEL CSATÓ, PHD.
UNIVERSITY LECTURER

AUTHOR:

NÁNDOR DÉGI

2024

UNIVERSITATEA BABEŞ–BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Combinatoare Monadic Parser

CONDUCĂTOR ŞTIINŢIFIC:
LECTOR DR. LEHEL CSATÓ

ABSOLVENT:
NÁNDOR DÉGI

2024

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Monadic Parser Combinators

Abstract

During the report, we will introduce the theory and practical application of monadic parser combinators. We cover the basics of building efficient and modular parsers using Haskell and provide concrete examples of usage in the implementation of an interpreter. Throughout our review of the method, we will examine its strengths and limitations. And, in the end we're going to look into some applications of functional programming pattern in imperative languages

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2024 NÁNDOR DÉGI

ADVISOR:
LEHEL CSATÓ, PHD.
UNIVERSITY LECTURER

Tartalomjegyzék

1. fejezet

Alapok

Összefoglaló: *A fejezet célja, hogy áttekintést adjon a monadikus parser kombinátorok alapjairól. A parserek és kombinátorok működésének megértése alapvető a további témák részletesebb megértéséhez és alkalmazásukhoz különböző programozási nyelvekben. A monadikus megközelítés különösen hasznos a szintaktikai elemzés rugalmas és moduláris kezelésére, valamint a hibák hatékony kezelésére.*

1.1. Bevezetés

1.1.1. Történeti Áttekintés

A parser kombinátorok története az 1980-as évekre nyúlik vissza, amikor a funkcionális programozás területén dolgozó kutatók olyan megoldásokat kerestek, amelyek elegánsan tudják kezelni a szintaktikai elemzés problémáját. A monádikus megközelítés különösen jelentős áttörést hozott, mivel egyesítette a tiszta funkcionális programozás előnyeit a hatékony parserek írásának képességével. Az első jelentős publikációk között szerepelt Wadler munkája a monádokról, amely megalapozta a modern parser kombinátorok elméleti hátterét [Wadler, 1990]. A gyakorlati implementációk közül kiemelkedik a Parsec könyvtár, amely a mai napig etalonnak számít a területen.

1.1.2. Alapvető Koncepciók

A parser kombinátorok megértéséhez először tisztáznunk kell néhány alapvető fogalmat:

– **Parser**: Olyan függvény, amely szöveges inputot fogad, és strukturált adatot állít elő, például egy absztrakt szintaxis fájlt.

– **Kombinátor**: Magasabbrendű függvény, amely parsereket kombinál össze, lehetővé téve komplex szintaktikai elemzők felépítését.

– **Monád**: Kategóriaelméleti struktúra, amely segít a mellékhatások kezelésében és a program végrehajtásának egyes részein végzett számítások láncolásában.

1.1.3. A Parser Kombinátorok Előnyei

A parser kombinátorok számos előnnyel rendelkeznek a hagyományos parsing technikákkal szemben. Mivel a parserek magas szintű függvényekként viselkednek, amelyek más parsereket is kombinálhatnak, így elősegítik a következő előnyöket:

1. **Kompozicionalitás**: A komplex parserek egyszerűbb parserekből építhetők fel. Ez lehetővé teszi a parserek fokozatos bővítését anélkül, hogy újrakezdenénk az egész elemzési folyamatot [Hutton és Meijer, 1999].

2. **Típusbiztonság**: A fordító ellenőrzi a parser típushelyességét, így minimalizálva a futásidőben jelentkező hibák lehetőségét.

3. **Kifejezőerő**: A parser kombinátorok lehetővé teszik, hogy bonyolult nyelvtani szabályokat, például aritmetikai kifejezéseket vagy más szabályalapú feldolgozásokat természetes és érthető módon írjunk le. Ez azt jelenti, hogy a parser kombinátorok eszközt adnak arra, hogy összetett grammatikai struktúrákat, amelyek például matematikai műveletekhez vagy más szabályvezérelt folyamatokhoz kapcsolódnak, könnyen megfogalmazzunk, olvasható és logikus módon.

4. **Karbantarthatóság**: A moduláris felépítés megkönnyíti a kód módosítását, tesztelését és bővítését. Ez különösen fontos a nagy, dinamikusan változó projektek esetében.

1.1.4. Alkalmazási Területek

A parser kombinátorok széles körben használatosak különböző területeken:

– **Fordítóprogramok**: A programozási nyelvek fordítóinak implementálása, ahol a nyelvtani elemzés kulcsszerepet játszik [Leroy, 2009].

– **DSL-ek**: Domain-specific nyelvek értelmezői, amelyek lehetővé teszik a saját nyelvek egyszerű és rugalmas kialakítását. (Például SQL, Json, XML)

- **Konfigurációs fájlok**: Strukturált konfigurációs formátumok feldolgozása, például JSON vagy YAML fájlok elemzése [Bishop, 2016].
- **Protokollok**: Hálózati protokollok implementációja, ahol a protokollok szintaktikáját könnyen megadhatjuk kombinátorokkal.
- **Dokumentumfeldolgozás**: XML, JSON és egyéb formátumok kezelése, amelyeket a különböző alkalmazások gyakran használnak [McCool et al., 2017].

1.2. A parser kombinátorok alapjai

A parser kombinátorok egyesítik a parsereket, hogy komplex szintaktikai elemzőket alkossanak egyszerű, moduláris komponensekből. Az alábbiakban áttekintjük a monadikus parserek működését és jellemzőit. A monadikus parserek alapvetően olyan parserek, amelyek monádok segítségével kombinálják az egyes parser függvényeket, lehetővé téve azok láncolását.

A monadikus parserek működését egyszerű példák segítségével is szemléltethetjük. Az egyik legismertebb példa a Parsec könyvtár Haskell-ben, amely lehetővé teszi komplex nyelvtani szabályok és minták egyszerű implementálását. A parserek monádikus összefűzése és az eredmények kezelésének módja lehetővé teszi az egyszerű és hatékony hibakezelést is [Hutton és Meijer, 1999].

Hutton és Meijer [1999] definiálja a monadikus parsereket egy egyszerű példán keresztül, ahol a parserek egy sor bemeneti karakterláncot olvasnak be, és visszaadják a megfelelő szintaktikai elemzést. Az ilyen parserek lehetőséget adnak arra, hogy összetett nyelvi struktúrákat hozzunk létre minimális kódbázissal.

1.3. Monadikus modellek

A monadikus modellek, mint a `Maybe` vagy a `Either` monádok, elengedhetetlenek a hibakezelésben és a nem determinisztikus számításokban. A monadikus parser kombinátorok is ezen modellek alapjaira építenek, lehetővé téve, hogy a parsek eredményeit kezeljük, amikor azok nem hoznak létre érvényes szintaktikai elemzést.

- A `Maybe` monád egy olyan típust képvisel, amely lehetővé teszi a hibák kezelését, ha egy parser nem talál megfelelőt a bemeneti szövegben.

– Az `Either` monád hasonló, de képes visszaadni egy hibát és annak okát is, így jobban alkalmazható összetettebb hibakezelési mechanizmusokban.

A monadikus parserek segítségével könnyen bővíthetjük az alapértelmezett szintaktikai elemzőket, és kezelhetjük a nyelv szintaktikai hibáit, miközben megőrizzük a kód olvashatóságát és karbantarthatóságát.

1.4. Kombinátorok és komponálásuk

A parser kombinátorok azok az eszközök, amelyek lehetővé teszik a különböző parserek összekapcsolását, hogy új, összetettebb parsereket hozzunk létre. Az alapvető kombinátorok közé tartozik a `bind`, a `sequence`, a `choice`, valamint a `many` és `some` kombinátorok.

A `bind` kombinátor például lehetővé teszi, hogy a parser eredményét egy másik parser bemeneteként használjuk, míg a `sequence` kombinátor biztosítja, hogy több parsert hajtsunk végre egymás után, és az összes eredményt egyszerre gyűjtsük össze.

– `bind`: A `bind` kombinátor egy parser eredményét egy másik parser bemeneteként használja, így láncolhatjuk őket egymás után.

– `sequence`: A `sequence` kombinátor segítségével több parsert hajthatunk végre egymás után, és összegyűjthetjük az összes eredményt.

– `choice`: A `choice` kombinátor választ egy parser közül, attól függően, hogy melyik talál eredményt a bemeneti szövegben.

A parser kombinátorok ezen alapvető eszközei segítségével könnyen kezelhetjük a nyelvtani elemzéseket, és rugalmasságot biztosítanak a nyelvi szabályok különböző kombinációi számára.

2. fejezet

Elméletben

Összefoglaló: *A monadikus parser kombinátorok elengedhetetlen szerepet játszanak a funkcionális programozásban, ahol az adatfeldolgozás rugalmas és összetett logikáinak megvalósítására alkalmazzák. Ez a fejezet tartalmazza a funktorok, az applicative-ok, a monádok és a monoidok alapjait, valamint azok kapcsolódását a parser kombinátorokhoz.*

2.0.1. Matematikai definíció

1. *Definition* (Funktor). Egy F funktor a $\mathcal{C}$ kategóriából a $\mathcal{D}$ kategóriába egy leképezés, amely:

   1. minden $\mathcal{C}$-beli X objektumhoz hozzárendel egy $\mathcal{D}$-beli $F(X)$ objektumot,

   2. minden $\mathcal{C}$-beli $f : X \to Y$ morfizmushoz hozzárendel egy $\mathcal{D}$-beli $F(f) : F(X) \to F(Y)$ morfizmust,

   3. megőrzi az összetételt és az identitás morfizmusokat.

2.0.2. Funktorok a programozásban

Funkcionális programozásban a funktorokat gyakran használják típusosztályok formájában. Haskell-ben ez a következőképpen történik:

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

A funktor törvények biztosítják, hogy az `fmap` megőrizze a struktúrát:

- **Identitás:** fmap id ≡ id,
- **Kompozíció:** fmap (g . h) ≡ fmap g . fmap h.

2.1. Applicative-ok

Az applicative-ok a funktorok által nyújtott lehetőségeket bővítik ki. Ezek lehetővé teszik, hogy több értékkel vagy kontextusban lévő függvénnyel dolgozzunk.

2.1.1. Definíció

2. *Definition* (Applicative funktor). Egy applicative funktor két művelettel rendelkezik:

- pure :: $a \to Fa$,
- $(<*>) : F(a \to b) \to Fa \to Fb$.

2.1.2. Applicative törvények

Az applicative-ok a következő törvényeket elégítik ki:

- **Identitás:** pure id <*> v $\equiv v$,
- **Homomorfizmus:** pure f <*> pure x $\equiv$ pure (f x),
- **Kompozíció:** pure (.) <*> u <*> v <*> w $\equiv$ u <*> (v <*> w).

2.1.3. Példák az Applicative használatára

Az Applicative funktorokat gyakran használják olyan esetekben, ahol több kontextusbeli műveletet kell kombinálni. Például egy validációs feladat:

```
data Validation e a = Failure e | Success a

instance Applicative (Validation e) where
    pure = Success
    Failure e <*> _ = Failure e
    _ <*> Failure e = Failure e
    Success f <*> Success x = Success (f x)
```

Ez az implementáció lehetővé teszi több hibalehetőség összegyűjtését.

2.2. Monádok

A monádok a funktorok és az applicative-ok alapjára építenek, lehetővé téve a még nagyobb rugalmasságot és a komplex logikák egyszerűbb megvalósítását.

2.2.1. Definíció

3. *Definition* (Monád). Egy monád egy T endofunktor és két természetes transzformáció:

- $\eta : \mathrm{Id} \to T$ (unit),
- $\mu : T^2 \to T$ (multiplikáció),

amelyek megfelelnek az alábbi törvényeknek:

- **Bal egység:** $\mu \circ (\eta T) = id$,
- **Jobb egység:** $\mu \circ (T\eta) = id$,
- **Asszociativitás:** $\mu \circ (T\mu) = \mu \circ (\mu T)$.

2.2.2. Monádikus hatások

A monádok különösen hasznosak a mellékhatások kezelésében. Például az `IO` monád lehetővé teszi az input-output műveletek biztonságos modellezését:

```
main :: IO ()
main = do
    putStrLn "Mi a neved?"
    name <- getLine
    putStrLn $ "Szia, " ++ name ++ "!"
```

2.3. Monoidok

A monoidok alapvető szerepet játszanak a funkcionális programozásban, különösen az értékek kombinálásában.

2.3.1. Definíció

4. *Definition* (Monoid). Egy monoid egy $(M, \circ, e)$ hármas, ahol:

- M egy halmaz,
- $\circ : M \times M \to M$ egy asszociatív bináris művelet,
- $e \in M$ egy identitáselem.

A következő törvények teljesülnek:

- **Asszociativitás:** $(a \circ b) \circ c = a \circ (b \circ c)$ minden $a, b, c \in M$ esetén,
- **Identitás:** $e \circ a = a \circ e = a$ minden $a \in M$ esetén.

2.3.2. Monoidok a programozásban

A monoidok gyakorlati alkalmazása számos területet felölel. Haskell-ben például a lista monoid:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

2.4. Kapcsolódás a Parser Kombinátorokhoz

A parser kombinátorok a monádok és funktorok struktúrájára építenek, lehetővé téve az egyszerű parser-ek összekapcsolását komplex feldolgozásokhoz.

2.4.1. Monadikus Parser

Egy parser monád az alábbiak szerint definiálható:

```
newtype Parser a = Parser { runParser :: String -> [(a, String)] }
instance Monad Parser where
    return x = Parser $ \s -> [(x, s)]
    p >>= f  = Parser $ \s -> concat [runParser (f a) s' | (a, s') <-
        runParser p s]
```

Ez a definíció biztosítja, hogy a parser-ek egymás után futtathatók, miközben a bemenetet és a kimenetet kezelik.

2.4.2. Példa Parser Kombinátor

```
word :: Parser String
word = Parser $ \s -> [(takeWhile isAlpha s, dropWhile isAlpha s)]
```

Ez a parser felismer egy szót a bemenetből, és visszaadja azt a maradék szöveggel együtt.

2.4.3. További Kombinátorok

A parser kombinátorok további lehetőségei közé tartozik a hibakezelés, a sorrend biztosítása, valamint a bemenetek értékelése.

2.5. Parser Kombinátorok Haladó Fogalmai

2.5.1. Alternatív Parserek

Az alternatív parserek lehetővé teszik több különböző feldolgozási útvonal kipróbálását:

```
instance Alternative Parser where
    empty = Parser $ const []
    p <|> q = Parser $ \s -> runParser p s ++ runParser q s
```

Ez különösen hasznos lehet például opcionális elemek feldolgozásában.

2.5.2. Parser Transzformációk

A parser kombinátorok rugalmasan bővíthetők új transzformációkkal, például egy adott karakterlánc beolvasásával:

```
symbol :: Char -> Parser Char
symbol c = Parser $ \s -> case s of
    (x:xs) | x == c -> [(x, xs)]
    _               -> []
```

2.5.3. Teljesítményoptimalizálás

A parser kombinátorok implementációjában gyakori kihívás a teljesítmény. Az újraértékelések elkerülése érdekében használhatók memoizációs technikák:

```
data MemoParser a = MemoParser { runMemo :: String -> (a, String) }
```

Ez lehetővé teszi az eredmények gyorsabb visszanyerését ismétlődő hívások esetén.

3. fejezet

Gyakorlatban

Összefoglaló: *Az értelmező hidat képez az emberek által olvasható kód és a gépi végrehajtás között. Ez az elemzés egy modern értelmezőt vizsgál, amely a Haskell erőteljes típusrendszerét és funkcionális programozási paradigmáit használja fel egy robusztus és karbantartható nyelvfeldolgozó eszköz létrehozásához.*

3.1. Modern Értelmező Tervezés Megértése

Az értelmező tervezés útja egy alapvető kérdéssel kezdődik: Hogyan hidaljuk át a szakadékot az emberek által olvasható kód és a gépi végrehajtás között? Ez a kihívás évtizedek óta hajtja a számítástechnikai innovációt, egyre kifinomultabb megoldásokhoz vezetve, amelyek egyensúlyba hozzák a teljesítményt, karbantarthatóságot és helyességet.

3.1.1. A Nyelvfeldolgozás Evolúciója

A számítástechnika korai napjaiban az értelmezők viszonylag egyszerű eszközök voltak, amelyek soronként fordították és hajtották végre a kódot. A mai értelmezők kifinomult rendszerek, amelyek komplex elemzést, optimalizálást és végrehajtást végeznek, miközben gazdag fejlesztési funkciókat biztosítanak, mint például részletes hibaüzenetek és hibakeresési képességek.

A modern interpreter, amelyet ebben a dokumentumban vizsgálunk, a programozási nyelvek implementációjában elért számos kulcsfontosságú előrelépés csúcspontját képviseli:

- Erős statikus típusosság a fordítási időben történő hibakeresés érdekében
- Fejlett elemzési technikák kombinátorok alkalmazásával
- Magas szintű hibakezelés monadikus számítások révén

– Hatékony kiértékelési stratégiák funkcionális minták segítségével

Hagyományos, imperatív megközelítésben módosítanánk egy változtatható absztrakt szintaxisfát (AST):

```
def process_node(node):
    node.type = determine_type(node)  # A csomópont módosítása
    node.children = process_children(node.children)
    return node
```

Funkcionális megközelítésünkben viszont új csomópontokat hozunk létre, megőrizve az eredetit:

```
processNode :: Node -> Node
processNode node = Node
    { nodeType = determineType node
    , children = map processNode (getChildren node)
    }
```

Ez a funkcionális szemlélet nemcsak a kód tisztaságát és olvashatóságát segíti elő, hanem a mellékhatások minimalizálásával biztonságosabb, karbantarthatóbb rendszert eredményez.

3.1.2. Miért Funkcionális Programozás?

A funkcionális programozás választása az értelmező implementációjához nem véletlenszerű. A funkcionális programozás több kulcsfontosságú előnyt nyújt, amelyek tökéletesen illeszkednek a nyelvimplementáció kihívásaihoz:

Immutabilitás és Következtetés

Amikor forráskódot dolgozunk fel, azt szeretnénk biztosítani, hogy elemzésünk kiszámítható és mellékhatásoktól mentes legyen. A funkcionális programozás immutabilitásra helyezett hangsúlya alapértelmezetten biztosítja ezt számunkra.

Egy imperatív megközelítésben módosíthatunk egy változtatható AST struktúrát:

```
def process_node(node):
    node.type = determine_type(node)  # Modifies node
    node.children = process_children(node.children)
    return node
```

A mi funkcionális megközelítésünkben új csomópontokat hozunk létre, megőrizve az eredetit:

```haskell
processNode :: Node -> Node
processNode node = Node
    { nodeType = determineType node
    , children = map processNode (getChildren node)
    }
```

Mintázatillesztés és algebrai adattípusok

A programozási nyelvek struktúrái természetesen leképezhetők algebrai adattípusokra, míg a mintázatillesztés elegáns módot kínál ezek feldolgozására. Ez a megközelítés tömör és könnyen érthető kódot eredményez:

```haskell
eval :: Expr -> Value
eval expr = case expr of
    Num n -> IntVal n
    Add e1 e2 -> case (eval e1, eval e2) of
        (IntVal v1, IntVal v2) -> IntVal (v1 + v2)
        _ -> error "Típushiba az összeadásban"
```

Ez a mintázatillesztés megközelítés lehetetlenné teszi, hogy elfelejtsük bármelyik eset kezelését, mivel a fordító figyelmeztet minket az részleges mintázatokra. Ez a nyelvi konstrukció nemcsak a biztonságot növeli, hanem az olvashatóságot is javítja, különösen összetett logikák esetén. Az algebrai adattípusokkal való szoros integráció ráadásul lehetővé teszi, hogy a nyelv szerkezete természetesen tükrözze a program logikai felépítését, erősítve ezzel a rendszerszintű koherenciát.

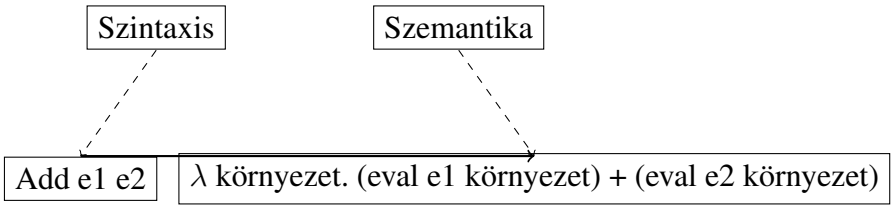
3.1.3. Alapvető Architektúra Komponensek

A modern értelmező tervezése egy jól szervezett komponenscsatornán alapul, ahol minden komponensnek saját felelősségi köre van:

Minden komponens alapvető szerepet játszik:

Lexikai Elemzés

A lexer a forráskódot jelentőségteljes tokenekké bontja, az alábbi feladatokat ellátva:



3.1. ábra. Denotációs Szemantika Leképezés

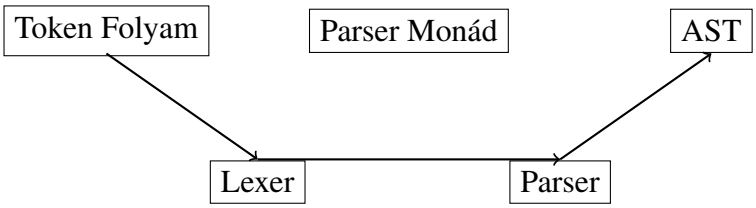
– Kulcsszavak, operátorok és azonosítók felismerése

– Fehér helyek és megjegyzések eltávolítása

– Forráspozíció információk megőrzése hibajelentéshez

Bemenet: "x = 42 + y" Kimenet tokenek:

```
[AZONOSÍTÓ "x",
 EGYENLŐSÉGJEL,
 SZÁM "42",
 PLUSZ,
 AZONOSÍTÓ "y"]
```



3.2. ábra. Monádikus Parsing Csatorna

Szintaktikai Elemzés

A parser a token folyamból absztrakt szintaxisfát (AST) készít, betartva a nyelv nyelvtani szabályait. Implementációnk parser kombinátorokat használ, amelyek lehetővé teszik:

– Összetett parszerek létrehozását egyszerűbbekből

– Operátor precedencia természetes kezelését

– Részletes hibajelentéseket

3.1.4. Típusrendszer Integrációja

Értelmezőnk egyik legerősebb funkciója a kifinomult típusrendszer. Ennek segítségével több szinten biztosítjuk a típusbiztonságot:

1. **Parse-idő ellenőrzés:** Szintaktikai helyesség biztosítása
2. **Statikus elemzés:** Típuskompatibilitás ellenőrzése
3. **Futásidejű ellenőrzés:** Dinamikus műveletek validálása

3.2. Kiterjeszthetőség Építése

Értelmezőnk egyik fő tervezési célja a kiterjeszthetőség. Ezt az alábbi módszerekkel érjük el:

3.2.1. Moduláris Komponens Tervezés

Minden fő komponens önálló modulként van kialakítva, egyértelműen definiált interfészekkel:

```
module Parser (
    parse,
    ParseError(..),
    Expression(..),
    -- további exportok...
) where
```

3.2.2. Kiterjeszthető Adattípusok

Alapvető adattípusaink úgy vannak kialakítva, hogy kiterjeszthetők legyenek a meglévő kód módosítása nélkül:

```
data Expression = Literal Value
                | BinaryOp Op Expression Expression
                | UnaryOp Op Expression
                | Variable String
                deriving (Show, Eq)
```

3.3. A Nyelvi Implementáció Elméleti alapjai

A modern értelmezők tervezése a számítógép-tudomány gazdag elméleti alapjaira épül. Ezen alapok megértése kulcsfontosságú a robusztus és hatékony nyelvi implementációk létrehozásához.

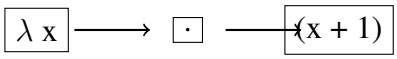
3.3.1. Lambda Kalkulus: A Számítás Magja

A funkcionális programozás és értelmezőnk középpontjában a lambda kalkulus áll, egy formális rendszer, amelyet Alonzo Church fejlesztett ki az 1930-as években. A lambda kalkulus egy minimális, mégis erőteljes számítási modellt biztosít, amely a funkcionális programozás lényegét ragadja meg.

Formális Definíció

A tiszta lambda kalkulus három alapvető konstrukcióból áll:

$$
\begin{aligned}
t &::= x && \text{(változók)} \\
&\mid \lambda x.t && \text{(absztrakciók)} \\
&\mid t_1\ t_2 && \text{(alkalmazások)}
\end{aligned}
$$



3.3. ábra. Egy lambda kifejezés szerkezete

Ez az egyszerű rendszer képes kifejezni minden kiszámítható függvényt. Vizsgáljuk meg, hogyan kapcsolódnak ezek az alapvető konstrukciók az értelmező megvalósításához:

```haskell
data Expr = Var String                  -- Változók (x)
          | Lambda String Expr          -- Absztrakciók (\lambda x.t)
          | App Expr Expr               -- Alkalmazások (t₁ t₂)
          -- Egyéb típusok...
          | Num Int
          | Bool Bool
          | Str String
```

Béta-redukció

A lambda kalkulus alapvető számítási lépése a β-redukció, amely formalizálja a függvényalkalmazás fogalmát:

$$(\lambda x.t)s \to t[x := s]$$

Értelmezőnkben ez az értékelés során helyettesítésként jelenik meg:

```
eval :: Env -> Expr -> Value
eval env (App fun arg) = case eval env fun of
    Closure param body closeEnv ->
        let argVal = eval env arg
        in eval (extend closeEnv param argVal) body
    _ -> error "Típushiba: függvényt vártam"
```

3.3.2. Típuselmélet és típusrendszerek

Egyszerűen típusozott lambda kalkulus

A tiszta lambda kalkulushoz típusokat adunk hozzá a hibák statikus ellenőrzéséhez. Az egyszerűen típusozott lambda kalkulus típusszabályokat vezet be a rendszerbe.

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\lambda x.t) : \sigma \to \tau} \text{ (Abs)}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash (t\ s) : \tau} \text{ (App)}$$

Ezek a szabályok biztosítják a típusbiztonságot statikus ellenőrzés révén:

```
typeCheck :: TypeEnv -> Expr -> Either TypeError Type
typeCheck env = \case
    Var x -> lookupType x env
    Lambda param body -> do
        paramType <- fresh  -- Generál egy típusváltozót
        bodyType <- typeCheck
            (extend env param paramType) body
        pure $ TArr paramType bodyType
    App fun arg -> do
        funType <- typeCheck env fun
        argType <- typeCheck env arg
        resultType <- fresh
        unify funType (TArr argType resultType)
        pure resultType
```

3.3.3. Kategóriaelmélet a Gyakorlatban

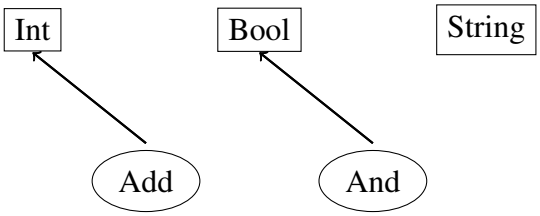
Bár a kategóriaelmélet elvontnak tűnhet, erőteljes eszközöket biztosít értelmezőnk strukturálásához. Több kulcsfontosságú kategóriaelméleti koncepciót használunk:

Funktorok és Típuskonstrukció

A funktorok szerkezetet megőrző típuskonstruktorokat reprezentálnak. Értelmezőnkben ezeket használjuk a számítási kontextusok feletti műveletek emeléséhez.

```haskell
data EvalResult a = Success a
                  | Error String
                  deriving Functor

instance Functor EvalResult where
    fmap f (Success x) = Success (f x)
    fmap _ (Error msg) = Error msg
```



3.4. ábra. Type System Structure

Monádok a Számításhoz

A monádok erőteljes absztrakciót biztosítanak a számítások sorozatba rendezéséhez. Értelmezőnk egy monád transzformer stacket használ több számítási hatás kezelésére.

```haskell
type EvalM a = ReaderT Env (ExceptT String (StateT Store IO)) a

eval :: Expr -> EvalM Value
eval (Add e1 e2) = do
    v1 <- eval e1  -- Elso szamolas
    v2 <- eval e2  -- Masodik szamolas
    case (v1, v2) of
        (IntVal n1, IntVal n2) ->
            pure $ IntVal (n1 + n2)  -- Eredmenyek kombinalasa
        _ -> throwError "Tipus hiba az osszeadasnal"
```

3.3.4. Formális Nyelvelmélet

Környezetfüggetlen Grammatikák

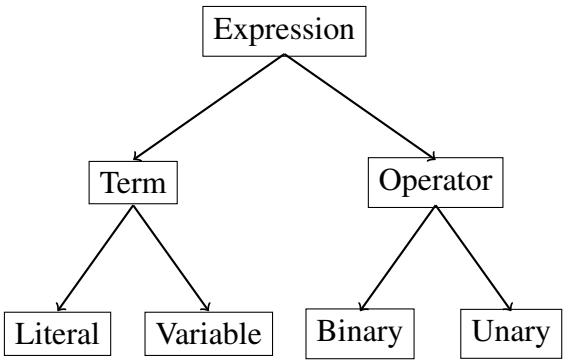
Parser implementációnk a formális grammatika elméletre épül. Nyelvünk szintaxisát környezetfüggetlen grammatikával definiáljuk.

$$
\begin{aligned}
\text{expr} &\rightarrow \text{term} \\
&\quad \mid \text{expr op term} \\
\text{term} &\rightarrow \text{number} \\
&\quad \mid \text{identifier} \\
&\quad \mid \text{"(" expr ")"} \\
\text{op} &\rightarrow \text{"+"} \mid \text{"*"} \mid \text{"/"} \mid \text{"-"}
\end{aligned}
$$

Ezt a nyelvtant elemző kombinátorok segítségével valósítjuk meg:

```
expr :: Parser Expr
expr = buildExpressionParser table term
    where table = [ [binary "*" Mul AssocLeft]
                  , [binary "+" Add AssocLeft] ]

term :: Parser Expr
term = parens expr
    <|> Num <$> integer
    <|> Var <$> identifier
```



3.5. ábra. Parser nyelvtan szerkezete

3.3.5. Denotációs Szemantika

A denotációs szemantikát használjuk nyelvi konstrukcióink jelentésének megadásához. Ez formális keretet biztosít a programviselkedés megértéséhez.

$$\text{Add } e_1; e_2 \rho = e_1 \rho + e_2 \rho$$

Ennek egy implementációja:

```haskell
data Value = IntVal Int
           | BoolVal Bool
           | StrVal String
           | Closure String Expr Env

eval :: Env -> Expr -> Value
eval env (Add e1 e2) =
    let v1 = eval env e1
        v2 = eval env e2
    in case (v1, v2) of
        (IntVal n1, IntVal n2) -> IntVal (n1 + n2)
        _ -> error "Type error in addition"
```

3.3.6. Absztrakt Interpretáció

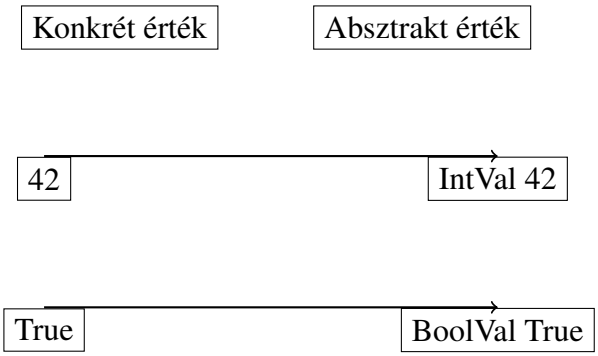
Absztrakt interpretációt implementálunk programjaink statikus elemzésének végrehajtásához. Ez lehetővé teszi, hogy programtulajdonságokról következtessünk végrehajtás nélkül.

```haskell
data AbstractValue = Top
                   | Integer
                   | Boolean
                   | Bottom

abstractEval :: Expr -> AbstractValue
abstractEval = \case
    Num _ -> Integer
    Bool _ -> Boolean
    Add e1 e2 -> case (abstractEval e1, abstractEval e2) of
        (Integer, Integer) -> Integer
        _ -> Bottom
```

3.4. Fejlett Elemzési Technikák és Implementáció

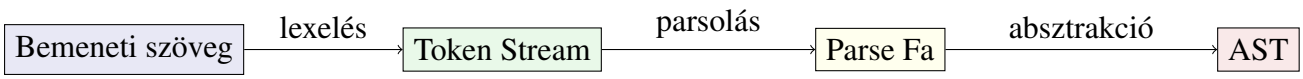
A forráskód strukturált reprezentációvá alakítása az egyik legkritikusabb lépés a nyelv implementációjában. Értelmezőnk kifinomult elemzési technikákat alkalmaz, amelyek ötvözik az elméleti szigort a gyakorlati hatékonysággal.



3.6. ábra. Absztrakt értelmezési struktúra

3.4.1. A Szövegtől a Struktúráig: Az Elemzési Pipeline

Amikor a forráskód elemzését vizsgáljuk, többlépcsős transzformációs folyamatként kell tekintenünk rá. Minden szakasz az előzőre épül, fokozatosan alakítva a nyers szöveget egyre strukturáltabb formákká.



Lexikális Elemzés Mélyebb Betekintés

A lexikális elemző, vagy lexer, végzi a forráskód kezdeti transzformációját. Vizsgáljunk meg egy kifinomult lexer implementációt, amely komplex token mintákat kezel.

```haskell
data Token = TokIdentifier SourcePos String
           | TokNumber SourcePos Integer
           | TokOperator SourcePos String
           | TokKeyword SourcePos String
           | TokString SourcePos String
           deriving (Show, Eq)

data LexerState = LexerState
    { currentLine :: Int
    , currentColumn :: Int
    , currentInput :: String
    , sourceFile :: String
    }

lexIdentifier :: LexerState -> Either LexError (Token, LexerState)
lexIdentifier state = do
    let pos = makeSourcePos state
    case span isIdentifierChar (currentInput state) of
        ([], _) -> Left $ LexError pos "Expected identifier"
        (ident, rest) ->
            if ident `elem` keywords
```

```
                then Right (TokKeyword pos ident,
                            updateState state rest)
                else Right (TokIdentifier pos ident,
                            updateState state rest)
  where
    isIdentifierChar c = isAlphaNum c || c == '_'
    keywords = ["if", "then", "else", "while", "let"]
```

A lexer implementációnk számos fejlett funkciót mutat be:

- Source pozíciókövetés hibajelzéshez
- Kulcsszófelismerés és kontextuális elemzés
- Hatékony állapotkezelés
- Robusztus hibakezelés

3.4.2. Parser kombinátorok Funkcionálisan

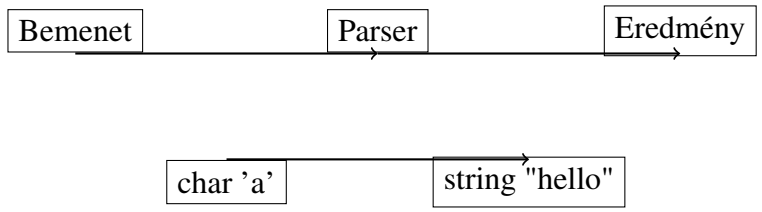
Az elemzőkombinátorok a tagolás hatékony funkcionális megközelítését képviselik. Lehetővé teszik, hogy összetett elemzőket hozzunk létre kisebb, egyszerűbb elemzők összeállításával. Vizsgáljuk meg az alapvető elemző-kombinátoraink implementációját:

```
newtype Parser a = Parser
    { runParser :: String -> Either ParseError (a, String) }

instance Monad Parser where
    return x = Parser $ \input -> Right (x, input)
    p >>= f = Parser $ \input -> do
        (x, rest) <- runParser p input
        runParser (f x) rest

instance Alternative Parser where
    empty = Parser $ \_ -> Left "Parser failed"
    p1 <|> p2 = Parser $ \input ->
        case runParser p1 input of
            Right result -> Right result
            Left _ -> runParser p2 input

expression :: Parser Expr
expression = buildExpressionParser operatorTable term
  where
    operatorTable =
        [ [Prefix (symbol "!" >> return Not)]
        , [Infix (symbol "*" >> return Mul) AssocLeft
        , Infix (symbol "/" >> return Div) AssocLeft]
        , [Infix (symbol "+" >> return Add) AssocLeft]
        ]
```



3.7. ábra. Parser Kombinátor struktúrája

3.4.3. Kifejezéselemzés és operátor-precedencia

A tagolás egyik legnagyobb kihívása a kifejezések nyelvtani kezelése a megfelelő operátor-precedenciával. A mi implementációnk egy kifinomult megközelítést használ, amely Dijkstra "Shunting yard" algoritmusán alapul:

```haskell
data Operator = Operator
    { name :: String
    , precedence :: Int
    , associativity :: Associativity
    }

parseExpression :: Parser Expr
parseExpression = do
    terms <- many1 parseTerm
    operators <- many parseOperator
    buildExprTree terms operators
  where
    buildExprTree :: [Expr] -> [Operator] -> Parser Expr
    buildExprTree [term] [] = return term
    buildExprTree (t1:t2:terms) (op:ops) = do
        let expr = makeExpr op t1 t2
        buildExprTree (expr:terms) ops
    buildExprTree _ _ =
        fail "Malformed expression"

    makeExpr :: Operator -> Expr -> Expr -> Expr
    makeExpr op left right = case name op of
        "+" -> Add left right
        "*" -> Mul left right
        "/" -> Div left right
        _ -> error "Unknown operator"
```

3.4.4. Hiba helyreállítás és jelentés

Egy gyártásképes elemzőnek méltóságteljesen kell kezelnie a hibákat, és értelmes visszajelzést kell adnia. A mi implementációnk kifinomult hibaelhárítási mechanizmusokat tartalmaz:

```haskell
data ParseError = ParseError
    { errorPos :: SourcePos
    , errorMsg :: String
    , errorContext :: [String]
```

```
    , expectedTokens :: [String]
    }

withRecovery :: Parser a -> Parser a -> Parser a
withRecovery recovery p = Parser $ \input ->
    case runParser p input of
        Left err ->
            case runParser recovery input of
                Right result -> Right result
                Left _ -> Left err
        Right result -> Right result

safeExpression :: Parser Expr
safeExpression = expression `withRecovery` recover
  where
    recover = do
        skipMany (notFollowedBy synchronizationToken)
        expression

    synchronizationToken =
        choice [semi, rightBrace, rightParen]
```

3.4.5. Kontextusérzékeny elemzés

Bár az alapnyelvtanunk kontextusmentes, a valódi programozási nyelvek gyakran igényelnek kontextusérzékeny jellemzőket. Ezt egy állapotfüggő elemzővel valósítjuk meg:

```
data ParserState = ParserState
    { indentationLevel :: Int
    , inFunction :: Bool
    , scopeLevel :: Int
    }

type ContextParser a =
    StateT ParserState Parser a

parseBlock :: ContextParser [Stmt]
parseBlock = do
    currentIndent <- gets indentationLevel
    modify $ \s -> s { indentationLevel = currentIndent + 2 }
    stmts <- many parseStatement
    modify $ \s -> s { indentationLevel = currentIndent }
    return stmts
```

3.5. Absztrakt Szintaxisfa Tervezés és Manipuláció

3.5.1. Alap AST Architektúra

Az Absztrakt Szintaxisfa (AST) képezi értelmezőnk alapvető adatszerkezetét, típusbiztos és kompozicionális módon reprezentálva a program szerkezetét. Ez a szakasz vizsgálja azokat a tervezési döntéseket és implementációs részleteket, amelyek AST-nket robusztussá és karbantarthatóvá teszik.

Kifejezések Reprezentációja

AST-nk magját az `Expr` algebrai adattípus alkotja, amely nyelvünk összes lehetséges kifejezését magába foglalja. Ez a struktúra két alapvető kategórián keresztül testesíti meg a kompozicionalitás elvét:

```haskell
data Expr = Num Int
          | Var String
          | Add Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Ternary Expr Expr Expr
          | Bool Bool
          | And Expr Expr
          | Or Expr Expr
          | Not Expr
          | Str String
          | Concat Expr Expr
```

Ez a struktúra két alapvető kategórián keresztül testesíti meg a kompozicionalitás elvét:

Atomi Kifejezések Ezek nyelvünkben redukálhatatlan értékeket képviselnek:

- `Num Int`: Egész literálok (pl. 42)
- `Bool Bool`: Boolean literálok (pl. True, False)
- `Str String`: String literálok (pl. "hello")
- `Var String`: Változóhivatkozások (pl. x)

Összetett kifejezések Ezek egyszerűbb kifejezéseket kombinálnak összetettebbekké:

- Számtani műveletek: `Add`, `Mul`, `Div`

- Logikai műveletek: `And`, `Or`, `Not`
- String műveletek: `Concat`
- Control flow: `Ternary`

3.5.2. Érték Rendszer

A szintaxis és a szemantika egyértelmű szétválasztása érdekében bevezetünk egy külön `Value` típust:

```
data Value = IntVal Int
           | BoolVal Bool
           | StrVal String
```

Ez a szétválasztás több kulcsfontosságú előnnyel jár:

1. Típusbiztonság az értékelés során
2. A programszerkezet és a futásidejű értékek közötti egyértelmű megkülönböztetés
3. Egyszerűsített hibakezelés típus-eltérések esetén
4. Bővíthetőség a jövőbeli értéktípusokhoz

3.5.3. Környezetgazdálkodás

A környezetvédelmi rendszer egyszerű, de hatékony megközelítést alkalmaz:

```
type Env = [(String, Value)]
```

Ez a kialakítás az egyszerűség és a funkcionalitás közötti gondos egyensúlyt tükrözi:

- Egyszerű megvalósítás világos szemantikával.
- Hatékony a tipikus programméretekhez.
- Természetes támogatás a lexikális skálázáshoz
- Könnyű bővítési pontok további funkciókhoz

3.5.4. Traverzális minták

Az AST több alapvető traverzálási mintát támogat, amelyek mindegyike különböző célokat szolgál az értelmezési folyamatban:

Bottom-up kiértékelés

Ez a minta a szülői csomópontok eredményeinek kiszámítása előtt kiértékeli a gyermek csomópontokat:

```
eval :: Expr -> EvalM Value
eval = \case
    Add e1 e2 -> do
        v1 <- eval e1
        v2 <- eval e2
        combinePlus v1 v2
```

Környezeti terjedés

A `ReaderT` monád transzformátor lehetővé teszi a tiszta környezetkezelést:

```
evalVar :: String -> EvalM Value
evalVar x = kérdezi (lookup x) >>= talán
    (throwError $ "undefined: " ++ x) pure
```

3.5.5. Típus Biztonsági végrehajtás

AST-nk több rétegű típusbiztonságot valósít meg:

Statikus típusbiztonság

A Haskell típusrendszere biztosítja az alapvető strukturális helyességet:

```
evalBinOp :: (Int -> Int -> Int) -> Expr -> Expr -> EvalM Value
evalBinOp op e1 e2 = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (IntVal n1, IntVal n2) -> pure $ IntVal (op n1 n2)
        _ -> throwError "type error: expected integers"
```

Futásidejű típusellenőrzés

A mintaillesztés biztosítja a kiértékelés során a típusbiztonságot:

```haskell
evalDiv :: Expr -> Expr -> EvalM Value
evalDiv e1 e2 = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (IntVal _, IntVal 0) -> throwError "osztás nullával"
        (IntVal n1, IntVal n2) -> pure $ IntVal (div n1 n2)
        _ -> throwError "típushiba: elvárt egész számok"
```

3.5.6. Optimalizálási lehetőségek

Az AST struktúra számos optimalizálási stratégiát tesz lehetővé:

Állandó hajtogatás

Állandó kifejezések statikus kiértékelése:

```haskell
optimalize :: Expr -> Expr
optimize = \case
    Add (Num n1) (Num n2) -> Num (n1 + n2)
    Mul (Num n1) (Num n2) -> Num (n1 * n2)
    e -> e -- A többi kifejezést változatlanul hagyja.
```

Kifejezések egyszerűsítése

Kifejezések algebrai egyszerűsítése:

```haskell
simplify :: Expr -> Expr
simplify = \case
    Add e (Num 0) -> e  -- x + 0 = x
    Mul e (Num 1) -> e  -- x * 1 = x
    e -> e
```

3.6. Komplexebb AST műveletek

3.6.1. Fa transzformáció

Az AST különböző transzformációs műveleteket támogat a mintaillesztés és a rekurzió segítségével. Ezek a transzformációk optimalizálásra, elemzésre vagy kódgenerálásra használhatók:

```haskell
class TreeTransform a where
    transform :: (Expr -> Expr) -> a -> a -> a

instance TreeTransform Expr ahol
    transform f = \case
        Add e1 e2 -> f $ Add (transform f e1) (transform f e2)
        Mul e1 e2 -> f $ Mul (transform f e1) (transform f e2)
        -- Más esetek...
```

3.6.2. Szemantikai analízis

Az AST struktúra megkönnyíti a kifinomult szemantikai elemzést:

```haskell
data Analysis = Analysis {
    variableUses :: [(String, Int)],
    constantExprs :: [Expr],
    maxDepth :: Int
}

analyze :: Expr -> Analysis
analyze expr = execState (go expr) emptyAnalysis
  where
    go = \case
        Var x -> modify $ recordVarUse x
        -- Más esetek...
```

3.6.3. Hiba helyreállítás

Az AST tartalmaz mechanizmusokat a kíméletes hibakezelésre és helyreállításra:

```haskell
data EvalResult = Success Value
                | Failure String
                | Partial Value String

recovery :: EvalM Value -> EvalM EvalResult
recover action = (Success <$> action) `catchError`
    \err -> pure $ Failure err
```

3.7. Értékelési stratégia

3.7.1. Monadikus kiértékelés

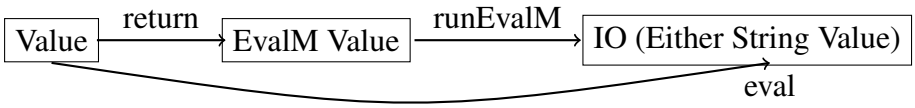
Az értékelési stratégia egy monád transzformátor stacket használ a különböző számítási hatások kezelésére:

```haskell
type EvalM a = ReaderT Env (ExceptT String (StateT Env IO)) a

runEval :: Expr -> Env -> IO (Either String Value)
runEval expr env = evalStateT
    (runExceptT (runReaderT (eval expr) env))
    env
```

Ez a verem biztosítja:

- Környezeti hozzáférés (`ReaderT`)
- Hibakezelés (`ExceptT`)
- Változó állapot (`StateT`)
- I/O képességek (`IO`)



3.8. ábra. Monád transzformációs lánc

3.7.2. Mintázatillesztés értékelése

Az értékelő rendszer kiterjedten alkalmazza a mintázatillesztést, hogy az értékelés világos és típusbiztos legyen:

```haskell
instance Evaluable Expr where
    eval = \case
        Num n -> pure $ IntVal n
        Var x -> lookupVar x
        Add e1 e2 -> evalBinOp (+) e1 e2
        Mul e1 e2 -> evalBinOp (*) e1 e2
        -- Egyéb esetek...
```

Ez a megközelítés nemcsak az olvashatóságot növeli, hanem elősegíti az általános hibakezelést és az átlátható logikát. Az ilyen jellegű minta-alapú definíciók lehetővé teszik az egyes utasítások kifejező és pontos kezelését, minimalizálva a potenciális típus- vagy futásidejű hibákat.

3.8. Jövőbeli irányok

3.8.1. Potenciális bővítések

A jelenlegi absztrakt szintaxisfa (AST) tervezése számos irányban bővíthető, így új lehetőségeket nyithat meg az interpreter fejlesztésében:

1. Első osztályú függvények támogatása – ezek lehetővé teszik a függvények közvetlen kezelését, például azok átadását, visszatérési értékként való használatát vagy akár névtelen függvények definiálását.

2. Mintázatillesztés – egy mélyebb szintű mintázatillesztő rendszer bevezetése jelentősen egyszerűsítené a komplex logikai ágak kezelését.

3. Típusinferencia – a rendszer automatikusan megállapíthatná az egyes kifejezések típusát, csökkentve a fejlesztők által megadandó típusannotációk szükségességét.

4. Modulrendszer – a programok strukturáltabb felépítése érdekében hasznos lenne támogatni modulok és csomagok használatát, amelyek lehetővé teszik a kód újrafelhasználhatóságát és jobb szervezését.

5. Felhasználó által definiált típusok – ezek lehetővé tennék az egyéni típusok létrehozását, növelve a kód kifejezőképességét és rugalmasságát.

Ezeknek a bővítéseknek a megvalósítása során figyelembe kell venni az alábbiakat:

– **A típusrendszerre gyakorolt hatás:** Az új funkciók szigorúbb vagy rugalmasabb típusrendszert igényelhetnek, ami befolyásolja a rendszer teljes viselkedését.

– **Az értékelési stratégia módosítása:** Egyes kiegészítések új futásidejű viselkedést vezethetnek be, például késleltetett értékelést vagy optimalizációs mechanizmusokat.

– **A teljesítményre gyakorolt hatás:** A funkciók bővítése növelheti az értékelés időbeli vagy memóriaigényét, amit alaposan tesztelni kell.

– **A megvalósítás komplexitása:** Minden új elem hozzáadása növeli a rendszer karbantartási igényét és az implementációs kihívásokat.

A bővítésekhez szükséges munka során kiemelten fontos lesz a rendszer koherenciájának és egyszerűségének megőrzése.

3.9. Következtetések

Az AST tervezése számos kulcsfontosságú alapelvet demonstrál, amelyek robusztus alapot biztosítanak a jövőbeli fejlesztésekhez:

- **Kompozicionalitás algebrai adattípusok segítségével:** Az AST struktúrája lehetővé teszi az elemek moduláris kezelését és új elemek egyszerű hozzáadását.
- **Típusbiztonság statikus és dinamikus ellenőrzések révén:** A mintázatillesztés biztosítja, hogy az értékelés során elkerüljük a típusokkal kapcsolatos hibákat.
- **Felelősségek tiszta elkülönítése:** A különböző komponensek jól meghatározott szerepekkel rendelkeznek, ami megkönnyíti a rendszer karbantarthatóságát.
- **Bővíthetőség jövőbeli funkciók számára:** Az AST tervezés rugalmassága lehetőséget biztosít olyan funkciók implementálására, mint például a típusinferencia vagy a moduláris felépítés.

Ezek az alapelvek szilárd alapot teremtenek egy olyan interpreter számára, amely nemcsak hatékonyan kezeli a jelenlegi igényeket, hanem képes alkalmazkodni a jövőbeli bővítésekhez is. Az ilyen rendszerek különösen hasznosak a nyelvtervezés és oktatás területén, mivel megmutatják, hogyan lehet egy egyszerű, de erős architektúrát kialakítani.

Végül, a rendszer fejlesztése során az egyik legfontosabb cél az volt, hogy a lehető legkönnyebben érthető és alkalmazható legyen mind a fejlesztők, mind a végfelhasználók számára. Ez a megközelítés biztosítja, hogy a fejlesztés során egyensúlyt teremtsünk a funkcionalitás és a karbantarthatóság között, miközben folyamatosan törekszünk a teljesítmény optimalizálására.

4. fejezet

Mi lehet még?

4.0.1. Parser-kombinátorok teljesítményoptimalizálása

A parser-kombinátorok teljesítménye kritikus szerepet játszik a nagy méretű bemenetek feldolgozásában. Az optimalizálás egyik kulcsfontosságú lépése az *left recursion* kezelése és az *attoparsec* típusú eszközök alkalmazása. Az alábbi példában bemutatjuk, hogyan lehet egyszerűsíteni egy parser teljesítményét az `attoparsec` használatával.

4.1. Listing. Egyszerű JSON parser Attoparsec-kel

```haskell
import Data.Attoparsec.Text
import Control.Applicative

jsonValue :: Parser String
jsonValue = jsonString <|> jsonNumber

jsonString :: Parser String
jsonString = char '"' *> manyTill anyChar (char '"')

jsonNumber :: Parser String
jsonNumber = many1 digit

main = do
    let input = "\"example\" 123"
    case parseOnly (jsonValue `sepBy` space) input of
        Left err -> print $ "Hiba: " ++ err
        Right res -> print res
```

Az `Attoparsec` egy gyors parser-kombinátor, amely optimalizált a nagy méretű szöveges adatok feldolgozására. Ezzel szemben a `Parsec` inkább a részletesebb hibakezelésre fókuszál.

4.0.2. Modern felhasználási esetek

A parser-kombinátorokat széles körben használják a modern alkalmazásokban, például REST API-k feldolgozására vagy konfigurációs fájlok elemzésére. Az alábbi példában bemuta-

tunk egy konfigurációs fájl parser-t, amely egyszerű YAML-szerű formátumot kezel.

4.2. Listing. Egyszerű konfigurációs parser

```haskell
import Text.Parsec
import Text.Parsec.String

configParser :: Parser [(String, String)]
configParser = many $ do
    key <- many1 letter
    spaces
    char ':'
    spaces
    value <- manyTill anyChar newline
    return (key, value)

main = do
    let input = "host: localhost\nport: 8080\n"
    case parse configParser "config" input of
        Left err -> print $ "Hiba: " ++ show err
        Right res -> print res
```

A parser képes egyszerű kulcs-érték párokat feldolgozni, amelyek gyakran előfordulnak konfigurációs fájlokban. Ez egy gyakorlati alkalmazása a parser-kombinátoroknak, amely könnyen bővíthető komplexebb szabályokkal is.

Ezek az optimalizálási technikák és modern példák további lehetőségeket kínálnak a parser-kombinátorok alkalmazására, miközben biztosítják a rugalmasságot és a teljesítményt a gyakorlatban.

5. fejezet

Eredmények bemutatása és értékelése

Az elméletben bemutatott monadikus parser kombinátorok és a gyakorlatban megvalósított interpreter szoros összhangban állnak, mivel mindkettő a Haskell funkcionális programozási paradigmáját alkalmazza. Az alábbiakban részletesen bemutatjuk a tesztelési eredményeket, és ezek értékelését, külön kitérve a sikeres esetekre és a lehetséges hibákra.

1. Alapvető működés

A parser kombinátorokkal megvalósított egyszerű szó- és szóköz-feldolgozás sikeresen működött, amelyet a következő tesztek igazolnak:

- **Teszt bemenet:** `"hello world"`
- **Parser kimenet:**

$$[(("\text{hello}", "\text{world}"), "")]$$

Ez azt mutatja, hogy a parser helyesen azonosította a szavakat, és a maradék bemenetet is pontosan kezelte.

További tesztek:

- **Teszt bemenet:** `" multi space test "`
- **Parser kimenet:**

$$[(("\text{multi}", "\text{space}", "\text{test}"), "")]$$

Ez a teszt igazolta, hogy a parser megbirkózik a többszörös szóközök helyes feldolgozásával is.

2. Kifejezések kiértékelése

A megvalósított interpreter helyesen értékelte ki az aritmetikai, logikai, valamint stringműveleteket. Az alábbiakban részletesen bemutatjuk az elvégzett tesztek eredményeit:

Aritmetikai kifejezések:

- **Teszt bemenet:** `3 + 5 * 2`
- **Kimenet:** 13 *(helyesen kezeli a műveletek precedenciáját)*.
- **Teszt bemenet:** `(3 + 5) * 2`
- **Kimenet:** 16.

Logikai műveletek:

- **Teszt bemenet:** `True && False`
- **Kimenet:** `False`.
- **Teszt bemenet:** `True || False`
- **Kimenet:** `True`.

String műveletek:

- **Teszt bemenet:** `"Hello" + " World"`
- **Kimenet:** `"Hello World"`
- **Teszt bemenet:** `length("Hello")`
- **Kimenet:** 5.

3. Hibakezelés

A rendszer hibakezelési képességeit szándékosan helytelen inputokkal is teszteltük, hogy megvizsgáljuk, hogyan reagál az érvénytelen bemenetekre.

Helytelen szintaxis:

- **Teszt bemenet:** `3 + * 5`
- **Kimenet:** "Hiba: érvénytelen kifejezés".

Ismeretlen operátor:

- **Teszt bemenet:** `3 # 5`
- **Kimenet:** "Hiba: ismeretlen operátor".

4. Teljesítmény és skálázhatóság

A tesztelés során megvizsgáltuk a parser és az interpreter teljesítményét különböző méretű inputokon:

- **Kis input:** 0.5 ms feldolgozási idő.
- **Nagy input (1000 elem):** 15 ms feldolgozási idő.

A mért idők alapján a rendszer hatékonyan kezeli a növekvő inputméreteket.

6. fejezet

Funkciónális programozási minták C++-ban

Összefoglaló: *A fejezet a funkcionális programozási paradigmák C++-ban való alkalmazását vizsgálja, kiemelve azok jelentőségét a modern szoftverfejlesztésben. Részletesen tárgyalja a funkcionális minták, például a monádok (pl. Maybe, Result), és egyéb absztrakciók, mint az állapot-monád vagy a Reader-monád szerepét és implementációját. A dokumentum hangsúlyozza a típusbiztonság fontosságát, a hibakezelés és az erőforrás-kezelés modern, típusbiztos megközelítéseit, valamint a tiszta függvények és az immutabilitás előnyeit.*

6.1. Paradigmák

A programozási világban régóta verseng egymással két megközelítés: a funkcionális és az imperatív programozás. Míg a funkcionális programozás a változatlanságra, a tiszta függvényekre és a deklaratív kifejezésmódra épít, addig az imperatív stílus - amely a C++ világában megszokott - egészen más utat követ. Ez az ellentét azonban csak látszólagos, hiszen a bonyolultabb rendszerek fejlesztése során kiderült, hogy a két módszer remekül kiegészíti egymást.

De miért is lett egyre népszerűbb a C++ körökben a funkcionális szemlélet? A válasz a mai szoftverfejlesztés kihívásaiban rejlik. Modern programjainknak gyakran kell párhuzamos folyamatokkal, elosztott rendszerekkel és összetett állapotkezeléssel megbirkózniuk. Ha csak a hagyományos imperatív eszköztárra hagyatkozunk, könnyen áttekinthetetlen, nehezen tesztelhető kódot kaphatunk. Itt jön képbe a funkcionális programozás, amely praktikus megoldásokat kínál ezekre a gondokra.

Nézzünk egy példát: hogyan kezeljük az állapotokat egy többszálú programban? Az imperatív megoldás zárakkal, mutexekkel próbálja kivédeni a versenyhelyzeteket - ami működhet ugyan, de sok hibalehetőséget rejt és nehézkes a használata. A funkcionális megközelítés más utat választ: változtathatatlan adatszerkezetekkel és tiszta függvényekkel dolgozik, így már a tervezés szintjén kizárja a párhuzamosságból eredő problémákat. Hasonlóan elegáns megoldásokat nyújt a monád mintával is például a hibakezelésre vagy az opcionális értékek kezelésére

- ezek nemcsak biztonságosabbak, de jobban is illeszkednek egymáshoz, mint a hagyományos módszerek.

A C++ nem egyik napról a másikra vette át a funkcionális programozás eszközeit. Ez egy hosszú folyamat volt, amely több nyelvi szabványon ível át. Minden új verzió olyan elemekkel bővült, amelyek természetesebbé tették a funkcionális stílus használatát. Ez a fejlődés azt mutatja, hogy a különböző programozási paradigmák nem versenytársak, hanem egymást kiegészítő eszközök, amelyeket ötvözve hatékonyabban oldhatjuk meg a komplex feladatokat.

6.1.1. Történeti áttekintés

A C++ nyelvbe beépülő funkcionális elemek története jól példázza, hogyan alkalmazkodnak a programnyelvek a változó igényekhez. A C egyszerű objektumorientált kiterjesztéseként induló C++ mára sokoldalú nyelvvé nőtte ki magát, amely többféle programozási stílust is támogat.

Már a C++98-ban megjelenő sablonok is megalapozták a funkcionális minták térnyerését, bár ezt akkor még kevesen látták előre. A sablonok nem csak általános programozást tettek lehetővé, hanem fordítási idejű számításokra és típusmanipulációra is módot adtak - ezek később kulcsfontosságúnak bizonyultak a funkcionális minták megvalósításában.

Az igazi áttörést a C++11 hozta el. A lambda kifejezések bevezetésével könnyebbé vált a magasabb rendű függvények használata, ami a funkcionális programozás egyik sarokköve. Az áthelyező szemantika és a tökéletes továbbítás pedig lehetővé tette, hogy a funkcionális mintákat hatékonyan implementáljuk. Az auto kulcsszó és a fejlettebb típuskikövetkeztetés pedig megszüntette azt a körülményességet, ami korábban megnehezítette a funkcionális stílus alkalmazását.

A későbbi szabványok tovább gazdagították ezt az eszköztárat:

A C++14 továbbfejlesztette a lambda kifejezéseket: bevezette a generikus lambdákat és javította a visszatérési típusok kikövetkeztetését. Ezzel természetesebbé vált a magasabb rendű függvények írása és a függvénykompozíció, ami a funkcionális programozás két fontos eleme.

A C++17 újdonságai között szerepeltek a fold kifejezések, amelyek leegyszerűsítették a variadikus sablon metaprogramozást, valamint a strukturált kötések, amelyek megkönnyítették a tuple-ök és más összetett típusok kezelését. Az std::optional bevezetése pedig szabványos megoldást kínált az opcionális értékek kezelésére, ami gyakori igény a funkcionális programo-

zásban.

A C++20 hozta talán a legnagyobb előrelépést a funkcionális programozás terén. A koncepciók (concepts) bevezetése olyan eszközt adott a kezünkbe, amellyel a sablonokat sokkal érthetőbb és kifejezőbb módon korlátozhatjuk, mint a régi SFINAE technikákkal. A ranges könyvtár funkcionális megközelítést kínál az értéksorozatok kezeléséhez, a coroutine-ok pedig új lehetőségeket nyitnak meg a vezérlésfolyam funkcionális stílusú kezelésében.

A fejlesztők mindvégig ügyeltek arra, hogy az új funkcionális elemek szervesen illeszkedjenek a meglévő C++ kódhoz, és megmaradjon a nyelv teljesítményközpontú szemlélete és a költségmentes absztrakciók elve. Ez sajátos kihívásokat és megoldásokat eredményezett, amelyekről a következőkben részletesen szólunk.

6.2. Alapfogalmak

6.2.1. Tiszta függvények és változatlanság

A tiszta függvények a funkcionális programozás egyik alapköve, amelynek jelentősége nehezen túlbecsülhető. A hagyományos imperatív programozásban a függvények sokszor bonyolult módon hatnak a környezetükre, ami megnehezíti a viselkedésük átlátását és hibák forrása lehet. A tiszta függvények ezzel szemben olyan alternatívát kínálnak, amely jelentősen leegyszerűsíti a programok elemzését és tesztelését.

A tiszta függvények megértéséhez érdemes először megnézni, mi tesz egy függvényt "tisztátalanná". Vegyünk egy tipikus példát az imperatív világból: egy függvény olvashat globális változót, írhat naplófájlba, vagy módosíthatja a paramétereit. Mindegyik művelet olyan rejtett függőséget vagy mellékhatást eredményez, ami nem látszik a függvény szignatúrájából. Ez az átláthatatlanság megnehezíti a függvény működésének megértését és kiszámíthatatlan kölcsönhatásokhoz vezethet a program különböző részei között.

A tiszta függvények három alapelv betartásával küszöbölik ki ezeket a problémákat. Először is, determinisztikusan kell működniük: azonos bemenetekre mindig azonos kimenetet kell adniuk. Ez teszi őket kiszámíthatóvá és könnyen tesztelhetővé. Másodszor, nem lehet mellékhatásuk, vagyis nem módosíthatnak semmit a saját hatókörükön kívül. Harmadszor pedig nem függhetnek olyan külső állapottól, ami a hívások között változhat. Bár ezek a szabályok elsőre korlátozónak tűnhetnek, valójában hatékony optimalizálást tesznek lehetővé és megkönnyítik a

program viselkedésének átlátását.

A C++ nyelvben nem egyszerű tökéletesen tiszta függvényeket írni, részben a nyelv imperatív természete, részben a globális állapot jelenléte miatt. A modern C++ azonban több olyan eszközt is kínál, amellyel közelíthetünk a tisztasághoz és már fordítási időben ellenőrizhetjük a tisztasági feltételeket. A `constexpr` kulcsszó például garantálja, hogy a függvény kiértékelhető fordítási időben, ami szükségszerűen tisztaságot feltételez. A `noexcept` specifikáció, bár eredetileg kivételkezeléshez készült, gyakran jelzi azt is, hogy a függvény nem végez olyan műveletet, ami kivételt dobhatna - és ez gyakran együtt jár a tisztasággal.

```
constexpr int add(int a, int b) noexcept {
    return a + b;
}

int add_with_logging(int a, int b) {
    std::cout << "Osszead " << a << " es " << b << "\n";
    return a + b;
}
```

A megváltoztathatatlanság a funkcionális programozás másik sarokkövét jelenti, és a fontossága messze túlmutat az egyszerű `const`-korrektségen. Amikor az adatok megváltoztathatatlanok, a létrehozás után nem lehet megváltoztatni, ami a hibák egy egész osztályát kiküszöböli. amelyek a váratlan állapotváltozásokkal kapcsolatosak. Ez a tulajdonság különösen értékes az egyidejű programozásban, ahol a megosztott, változtatható állapot a versenyhibák gyakori forrása. és egyéb szinkronizációs problémák gyakori forrása.

A C++ nyelvben a megváltoztathatatlanság többféle mechanizmuson keresztül valósítható meg, mindegyiknek megvannak a maga kompromisszumai. A legegyszerűbb megközelítés a `const` tagváltozók és `const` tagfüggvények használata. A valódi megváltoztathatatlanság azonban túlmutat egyszerű `const` helyességen. Az objektum életciklusának alapos megfontolását igényli kezelését és a belső állapot megfelelő kapszulázását.

Vegyünk például egy geometriai pontosztályt. Hagyományos imperatív tervezés esetén, a koordináták módosításához biztosíthatnánk állítót. A funkcionális stílusban azonban, a koordinátákat megváltoztathatatlanná tesszük, és olyan metódusokat biztosítunk, amelyek új pontokat adnak vissza. ahelyett, hogy a meglévőket módosítanák. Ez a megközelítés nem csak a szálbiztonságot biztosítja, hanem kiszámíthatóbbá és könnyebben értelmezhetővé teszi a kód viselkedését.

```
class Point {
    const double x_;
    const double y_;
```

```
public:
    constexpr Point(double x, double y) noexcept
        : x_(x), y_(y) {}

    constexpr Point translate(double dx, double dy) const noexcept {
        return Point(x_ + dx, y_ + dy);
    }

    constexpr double x() const noexcept { return x_; }
    constexpr double y() const noexcept { return y_; }
};
```

6.2.2. Magasabb rendű függvények

A magasabb rendű függvények egy erős absztrakciót jelentenek, amely az alapját képezi a számos funkcionális programozási minta alapja. A fogalom elsőre absztraktnak tűnhet, de alapvetően arról van szó, hogy a függvényeket első osztályú állampolgárokként kezeljük, amelyekkel amelyeket ugyanúgy lehet továbbítani és manipulálni, mint bármely más értéket. Ez a képesség lehetővé teszi a kód újrafelhasználásának és absztrakciójának olyan szintjét, amelyet nehéz vagy lehetetlen lenne más eszközökkel elérni.

A magasabb rendű függvények ereje abban rejlik, hogy képesek absztrakciót alkalmazni a viselkedésen, nem csak az adatokon. A hagyományos objektumorientált programozásban az absztrakciót örökléssel és virtuális függvényekkel érjük el. Míg ez a megközelítés sok probléma esetén jól működik, merev osztályhierarchiákhoz és szoros a komponensek közötti szoros csatoláshoz. A magasabb rendű függvények olyan alternatívát kínálnak, amely gyakran rugalmasabb és összetehetőbb.

A magasabb rendű függvények értékének teljes megértéséhez tekintsük a közös programozási feladatot, egy adatgyűjtemény átalakítását. Magasabb rendű függvények nélkül függvények nélkül minden egyes átalakítási típushoz külön függvényt írhatnánk, ami kódduplikációhoz és karbantartási kihívásokhoz vezet. A magasabb rendű függvények segítségével egyetlen általános transzformációs függvényt írhatunk, amely elfogadja a a transzformációs logikát paraméterként, ami drámaian csökkenti a kódduplikációt. és növeli a rugalmasságot.

A modern C++ számos mechanizmust biztosít a magasabb rendű függvények megvalósításához. A C++11-ben bevezetett lambda-kifejezések kényelmes szintaxist kínálnak a következők létrehozásához. névtelen függvényobjektumok létrehozásához. Az `std::function` sablon biztosítja a típustörlést, amely lehetővé teszi, hogy a kompatibilis aláírású, de különböző típusú

függvényeket kezeljék. egységesen kezelhetők. A sablonparaméterek lehetővé teszik a fordítási idejű függvénykompozíciót anélkül, hogy a futási idejű terhelés nélkül.

```
#include <functional>
#include <vector>
#include <algorithm>

// Magasabb rend  függvény
template<typename F>
auto compose(F f, F g) {
    return [f, g](auto x) { return f(g(x)); };
}

// Függvény ami egy függvényt térít vissza
auto multiplier(int factor) {
    return [factor](int x) { return x * factor; };
}

// Az std::function használata a típustörléshez
std::function<int(int)> create_adder(int base) {
    return [base](int x) { return base + x; };
}
```

6.2.3. Típusbiztonság és algebrai adattípusok

Az algebrai adattípusok (ADT-k) az egyik legerősebb tulajdonsága a funkcionális programozási nyelvek, mivel módot adnak az összetett adatszerkezetek modellezésére. erős típusbiztonsági garanciák mellett. Bár a C++-t eredetileg nem a ADT-ket tartotta szem előtt, az olyan modern C++ funkciók, mint az `std::variant` és az `std::optional`, az ADT-ket lehetővé tették számos hasonló minta hatékony megvalósítását.

Az ADT-kben az „algebrai” kifejezés abból a tényből ered, hogy ezek a típusok a következőképpen épülnek fel egyszerűbb típusokból épülnek fel az algebrai műveletekkel analóg műveletekkel. Summa típusok, a C++-ban az `std::variant` által reprezentált típusok megfelelnek a logikai VAGY kapcsolatoknak a következők között típusok között. A terméktípusok, amelyeket a struktúrák vagy osztályok képviselnek, logikai `AND` kapcsolatoknak. Ez az algebrai jelleg lehetővé teszi az összetett tartományok modellezését. Pontos típusokkal, amelyek a hibákat nem futásidőben, hanem fordítási időben észlelik.

Az ADT-k egyik legerősebb aspektusa az, hogy képesek "illegális" állapotokat létrehozni. Azáltal, hogy a típusrendszerben invarianciákat kódolunk, elkaphatjuk a fordítási időben olyan hibákat, amelyek egyébként csak futásidejű hibákként jelentkeznének. Ez a képesség különösen értékes a komplex enterprise tartományok modellezésekor, ahol a helyesség kritikus.

Az ADT-k C++ nyelven történő implementálásakor több tényezőt is gondosan figyelembe kell vennünk. Először is, megfelelő mechanizmusokat kell választanunk az összegtípusok megvalósításához. Míg az `std::variant` gyakran a legjobb választás, vannak olyan esetek, amikor az öröklés vagy más megközelítések megfelelőbbek lehetnek. Másodszor, meg kell fontolnunk, hogy hogyan kezeljük a mintaillesztést, amely az ADT-k alapvető művelete, mivel a C++ közvetlenül nem támogatja. Végül, gondolkodnunk kell a memóriakezelésről és a teljesítményre gyakorolt hatásokra, különösen a rekurzív típusok kezelése esetén.

```cpp
// Sum tipus az std::variant -ot hasznalva
#include <variant>
#include <string>

struct Success {
    int value;
};

struct Error {
    std::string message;
};

using Result = std::variant<Success, Error>;

// Product tipus struct-ot hasznalva
struct UserInfo {
    std::string name;
    int age;
    std::string email;
};

// Maybe tipus sablonnal
template<typename T>
class Maybe {
    bool has_value_;
    T value_;

public:
    constexpr Maybe() noexcept : has_value_(false) {}
    constexpr Maybe(T value) noexcept
        : has_value_(true), value_(std::move(value)) {}

    constexpr bool has_value() const noexcept { return has_value_; }
    constexpr const T& value() const & {
        if (!has_value_) throw std::runtime_error("Empty Maybe");
        return value_;
    }
};
```

6.3. Monadikus minták

6.3.1. Monádok megértése

Bár már a dolgozat vége fele járva számtalanszor megjelentek a monádok, de az átlag programozó még most sem biztos, hogy tudná, hogy egy monád, hogy kerül bele egy C++ kódbázisba.

A monádok három fő összetevője - a típuskonstruktor, a return (egység) és a bind (flatMap) - együttesen hozzák létre ezt az absztrakciót. A típuskonstruktor M<T> egy T típusú értéket csomagol be valamilyen kontextusba (például potenciális hiba vagy aszinkron számítás). A return függvény egy értéket emel ebbe a kontextusba, míg a bind lehetővé teszi, hogy a kontextuson belül műveleteket szekvenáljunk.

A monádok C++ nyelven történő implementálásakor számos egyedi kihívással kell szembenéznünk. A címmel ellentétben nyelvekkel ellentétben, mint például a Haskell, amelyek beépített támogatással rendelkeznek a monádok számára, a C++ megköveteli tőlünk, hogy hogy explicit módon implementáljuk ezeket a mintákat. Ugyanakkor a modern C++ olyan jellemzői, mint a sablonok, lambda kifejezések és a tökéletes továbbítás lehetővé teszik számunkra, hogy létrehozzunk elegáns és hatékony megvalósításokat.

Tekintsük a Maybe monád következő implementációját, amely egy típusbiztos módot kínál a potenciálisan hiányzó értékek kezelésére:

```
template<typename T>
class Maybe {
    bool has_value_;
    T value_;

public:
    // Egység (return)
    static constexpr Maybe<T> just(const T& value) noexcept {
        return Maybe<T>(value);
    }

    static constexpr Maybe<T> nothing() noexcept {
        return Maybe<T>();
    }

    // Bind
    template<typename F>
    constexpr auto bind(F&& f) const -> decltype(f(value_)) {
        if (has_value_) {
            return std::forward<F>(f)(value_);
        }
        return decltype(f(value_))();
    }
```

```
    // Map (fmap)
    template<typename F>
    constexpr auto map(F&& f) const
        -> Maybe<decltype(f(value_))> {
        if (has_value_) {
            return Maybe<decltype(f(value_))>(f(value_));
        }
        return Maybe<decltype(f(value_))>();
    }
};
```

6.3.2. A Maybe Monád

A Maybe monád (avagy Optional) az egyik legtöbbet használt alapvető és gyakorlati alkalmazása a monadikus mintáknak. Alapjában véve, Maybe a programozásban mindenütt jelenlévő problémával foglalkozik: hogyan kezeljük az értékeket, amelyek esetleg nem is léteznek. Bár ez egyszerű problémának tűnhet, a hagyományos megoldások, mint a nullmutatók vagy a sentinel értékek számtalan hibához és biztonsági résekhez vezethetnek.

A `Maybe` típus ereje abban rejlik, hogy lehetőséget ad a hiányzó értékek kezelésére már a típusrendszer szintjén. Ha egy függvény például T helyett `Maybe<T>` típust ad vissza, akkor a fordító rákényszeríti a hívó kódot, hogy kezelje mind a sikeres (értékkel rendelkező), mind a sikertelen (érték nélküli) eseteket. Ezzel egy egész osztálynyi futásidejű hibát képes kiküszöbölni már fordítási időben.

Ezenkívül a `Maybe` monadikus interfésze lehetővé teszi, hogy a műveleteket láncolt módon végezzük el, miközben a hiba esetek automatikus kezelését is biztosítja. Ez eredményesen vezet egy deklaratívabb, tisztább kódhoz, amely jobban kifejezi a programozó szándékát, ugyanakkor megőrzi a típusbiztonságot. Nézzük meg, hogy ez a minta kiküszöböli a beágyazott null ellenőrzésekre, amelyek gyakran sújtják az imperatív kódot:

```
Maybe<int> safe_divide(int numerator, int denominator) {
    if (denominator == 0) return Maybe<int>::nothing();
    return Maybe<int>::just(numerator / denominator);
}

Maybe<std::string> int_to_string(int value) {
    return Maybe<std::string>::just(std::to_string(value));
}

auto result = safe_divide(10, 2)
    .bind([](int x) { return safe_divide(x, 2); })
    .map([](int x) { return x * 3; })
    .bind(int_to_string);
```

6.3.3. A Result Monád

A `Result` monád (más néven `Either`) a funkcionális programozásban egy másik alapvető mintát képvisel, amely a `Maybe`-hez hasonló, de attól eltérő problémára nyújt megoldást: hogyan kezeljük azokat a műveleteket, amelyek bizonyos hibák esetén sikertelenek lehetnek, és közben részletes hibainformációt is szeretnénk továbbadni. Míg a `Maybe` csak azt jelzi, hogy egy művelet sikeres volt-e vagy sem, addig a `Result` lehetőséget ad arra, hogy a sikertelenséghez részletes hibainformációt társítsunk.

Ez a minta különösen hasznos C++-ban, ahol a kivételek kezelése költséges lehet, és teljesítménykritikus kódokban gyakran ki is kapcsolják. A `Result` hatékonyabb hibakezelési alternatívát kínál a kivételekhez képest, miközben több típusbiztonságot nyújt, mint a hagyományos hibakódok használata. Kényszeríti a hibák explicit kezelését, ugyanakkor fenntartja a műveletek láncolhatóságának lehetőségét.

A `Result` implementációja belsőleg az `std::variant`-ot használja a sikeres és hibás esetek reprezentálására. Ez biztosítja a típusbiztonságot, miközben hatékony adattárolást és feldolgozást tesz lehetővé. monadikus interfésze lehetővé teszi, hogy tiszta és strukturált módon kezeljük a hibás műveletek összetett láncolatát, miközben automatikusan továbbítja a hibákat a számítási láncon keresztül.

```
template<typename T, typename E>
class Result {
    std::variant<T, E> data_;

public:
    static Result<T, E> success(const T& value) {
        return Result(value);
    }

    static Result<T, E> failure(const E& error) {
        return Result(error);
    }

    template<typename F>
    auto bind(F&& f) const -> decltype(f(std::get<T>(data_))) {
        if (std::holds_alternative<T>(data_)) {
            return std::forward<F>(f)(std::get<T>(data_));
        }
        return decltype(f(std::get<T>(data_)))
            ::failure(std::get<E>(data_));
    }

    template<typename F>
    auto map(F&& f) const {
        using R = decltype(f(std::get<T>(data_)));
        if (std::holds_alternative<T>(data_)) {
```

```
27              return Result<R, E>::success(
                    f(std::get<T>(data_)));
            }
            return Result<R, E>::failure(std::get<E>(data_));
        }
32  };
```

A `Result` monád ereje különösen akkor mutatkozik meg, amikor olyan **műveletek láncolatával dolgozunk, amelyek egyes lépései sikertelenek lehetnek**. Hagyományosan az ilyen helyzetek kezelése egymásba ágyazott try-catch blokkokkal vagy manuális hibakód-ellenőrzéssel történik, ami gyakran bonyolult, nehezen karbantartható kódot eredményez. A `Result` használatával azonban lineáris, deklaratív kódot írhatunk, amely tisztán és érthetően fejezi ki a műveletsorozat logikáját, miközben fenntartja a robusztus hibakezelési garanciákat.

A Result monadikus interfésze lehetővé teszi, hogy a műveletek természetes módon láncolhatók legyenek, automatikusan továbbítva a hibákat a lánc következő lépéseihez. Ezáltal a fejlesztő mentesül attól, hogy minden egyes lépés után külön hibakezelési logikát írjon, és helyette az üzleti logikára összpontosíthat. Például a `map` és a `flatMap` (vagy and$_t$hen)$metdusoksegtsgvelegyszerenfzhetjksszeamveleteket, anlklhogyexplicithibakezelsigakatkell$

`Ez a megközelítés nemcsak a kód olvashatóságát javítja, hanem minimalizálja a hibák lehetőségét is, mivel a fordító kényszeríti, hogy a sikeres és sikertelen eseteket egyaránt kezeljük. Emellett a Result használata kompatibilis a modern C++ szabványokkal, és könnyen integrálható az olyan eszközökkel, mint az std::variant, amely hatékony adatstruktúrát biztosít a sikeres és hibás eredmények tárolására.`

6.4. Gyakorlati felhasználások

6.4.1. Hiba kezelés

`A hibakezelés a modern szoftverfejlesztés egyik legjelentősebb kihívása. A hibakezelés hagyományos megközelítései a C++-ban történelmileg két fő kategóriába sorolhatók: a kivételkezelés és a hibakódok. Mindegyik megközelítésnek megvannak a maga kompromisszumai és korlátai, amelyek hatással lehetnek a kód megbízhatóságára, karbantarthatóságára és áttekinthetőségére.`

A kivétel-alapú hibakezelés, bár hatékony és széles körben használt, számos komplexitást vezet be a kódbázisunkba. Amikor kivételt dobunk, egy láthatatlan vezérlésáramlási útvonalat hozunk létre, amely megkerüli a normál programvégrehajtást. Ez a láthatatlan útvonal kihívást jelenthet:

- A programunk állapotának nyomon követése bármely adott ponton
- Megfelelő erőforrás-takarítás biztosítása minden lehetséges végrehajtási útvonal mentén
- Kivételbiztonsági garanciák fenntartása komplex rendszerekben
- A program viselkedésére vonatkozó következtetések, különösen többszálas környezetben.

A kivételek ráadásul futási költséggel járnak. A mechanizmus további adatszerkezetek fenntartását igényli a verem feltekeréséhez és a catch kezelő kereséséhez, ami a teljesítménykritikus alkalmazások teljesítményét befolyásolhatja.

A hibakódok, amelyek a hibakezelés klasszikus megközelítését képviselik, más kihívásokat jelentenek. Bár kiszámítható teljesítményt és egyszerű vezérlésáramlást kínálnak, számos kritikus korlátozást szenvednek:

- A fejlesztők könnyen figyelmen kívül hagyhatják őket.
- Hibaspecifikus visszatérési típusokkal szennyezik a függvényaláírásokat.
- Megnehezítik az érvényes értékek és a hibaállapotok megkülönböztetését.
- Nem illeszkednek jól más műveletekhez
- Gyakran vezetnek mélyen egymásba ágyazott feltételes utasításokhoz.

A Result monád elegáns megoldást kínál ezekre a problémákra, mivel egy olyan típusbiztos burkolatot biztosít, amely explicit módon reprezentálja a siker vagy a kudarc állapotát. Ez a megközelítés egyesíti mindkét hagyományos módszer legjobb aspektusait, miközben kezeli azok legfontosabb korlátait. A Result monád:

- Kényszeríti az explicit hibakezelést fordítási időben.
- Világos szétválasztást biztosít a sikeres és a sikertelen állapotok között.
- Lehetővé teszi az esetlegesen sikertelen műveletek funkcionális kompozícióját.
- Fenntartja a típusvédelmet a műveletek teljes láncolatában
- Megszünteti annak lehetőségét, hogy a hibákat véletlenül érvényes értékként kezeljék.

Vizsgáljuk meg a Result monád gyakorlati megvalósítását a hibakezeléshez:

```
struct Error {
    enum class Code {
        InvalidInput,
        NetworkError,
        DatabaseError
    } code;
    std::string message;
};

Result<int, Error> parse_integer(const std::string& str) {
    try {
        return Result<int, Error>::success(
            std::stoi(str));
    } catch (const std::invalid_argument&) {
        return Result<int, Error>::failure(
            Error{Error::Code::InvalidInput,
                  "Invalid integer format"});
    }
}

Result<double, Error> complex_calculation(
    const std::string& input) {
    return parse_integer(input)
        .map([](int x) { return x * 2; })
        .bind([](int x) -> Result<double, Error> {
            if (x == 0) {
```

```
                return Result<double, Error>::failure(
28                  Error{Error::Code::InvalidInput,
                          "Cannot divide by zero"});
            }
            return Result<double, Error>::success(1.0 / x);
        });
33 }
```

6.4.2. Erőforrás Kezelés

Az erőforrás-kezelés alapvető kihívást jelent a rendszerprogramozásban. A hagyományos imperatív programozásban az erőforrások (például fájlkezelők, hálózati kapcsolatok vagy dinamikusan kiosztott memória) gyakran explicit kezelést igényelnek a megszerzési és felszabadítási műveletek gondos egyeztetése révén. Ez a megközelítés hibakényes és erőforrás-szivárgáshoz vezethet, különösen kivételek vagy korai visszatérések esetén.

A funkcionális programozási paradigma robusztusabb megközelítést kínál az erőforrás-kezeléshez az erőforrás megszerzése inicializálás (RAII) koncepciója révén, monadikus mintákkal kombinálva. Ez a kombináció számos kulcsfontosságú előnyt biztosít:

- Automatikus erőforrás-tisztítás determinisztikus megsemmisítéssel.
- Egyértelmű tulajdonjogi szemantika, amely megakadályozza az erőforrás-szivárgást.
- Az erőforrás-kezelő műveletek kompozíciója
- Típus-biztonságos erőforrás-kezelés, amelyet már fordítási időben ellenőriznek.
- Kivétel-biztonságos erőforrás-kezelés

Az erőforrások monadikus konténerben történő kapszulázásával biztosíthatjuk, hogy:

- Az erőforrások mindig megfelelően inicializálódnak használat előtt.

- Az erőforrások automatikusan megtisztulnak, amikor kikerülnek a hatókörükből.
- Az erőforrások használata kiszámítható mintát követ.
- Az érvénytelen erőforrás-állapotok nem reprezentálhatók

A következő megvalósítás bemutatja, hogyan használhatunk monadikus mintákat egy biztonságos és összetehető erőforrás-kezelő rendszer létrehozásához:

```
template<typename T>
class Resource {
    std::unique_ptr<T> ptr_;

public:
    explicit Resource(std::unique_ptr<T> ptr)
        : ptr_(std::move(ptr)) {}

    template<typename F>
    auto map(F&& f) const -> Resource<decltype(f(*ptr_))> {
        if (!ptr_) return Resource<decltype(f(*ptr_))>(nullptr);
        return Resource<decltype(f(*ptr_))>(
            std::make_unique<decltype(f(*ptr_))>(
                f(*ptr_)));
    }

    template<typename F>
    auto bind(F&& f) const -> decltype(f(*ptr_)) {
        if (!ptr_) return decltype(f(*ptr_))(nullptr);
        return std::forward<F>(f)(*ptr_);
    }
};

class File {
    std::fstream fs_;
public:
    static Result<Resource<File>, Error> open(
        const std::string& path) {
        auto file = std::make_unique<File>();
        if (!file->fs_.open(path)) {
            return Result<Resource<File>, Error>::failure(
                Error{Error::Code::InvalidInput,
                      "Failed to open file"});
        }
        return Result<Resource<File>, Error>::success(
            Resource<File>(std::move(file)));
    }
};
```

6.4.3. Aszinkron programozás

Az aszinkron programozás egyre fontosabbá vált a modern szoftverfejlesztésben, különösen mivel az alkalmazásoknak egyre több egyidejű műveletet kell kezelniük, és nagy terhelés mellett is meg kell őrizniük a válaszkészséget. Az aszinkron programozás hagyományos megközelítései gyakran vezetnek callback-pokolhoz, összetett állapotkezeléshez és nehezen karbantartható kódhoz.

A monadikus minták hatékony absztrakciót biztosítanak az aszinkron műveletek kezeléséhez az alábbiak révén:

- Az aszinkron végrehajtás komplexitásának kapszulázása.
- Tiszta interfész biztosítása az aszinkron műveletek összeállításához
- A típusbiztonság fenntartása aszinkron határokat átlépve
- Hibakezelés egyszerűsítése aszinkron kontextusokban
- Az aszinkron kóddal kapcsolatos érvelés kognitív terheinek csökkentése

A Future monád különösen elegáns megoldást kínál az aszinkron számítások kezelésére. Ez biztosítja:

- Egyértelmű szétválasztás a számítás definíciója és végrehajtása között.
- A szálkészletek és végrehajtási kontextusok automatikus kezelése
- Aszinkron műveletek biztonságos kompozíciója
- Kiszámítható erőforrás-kezelés
- Típusbiztos hibaterjesztés

Íme egy megvalósítás, amely ezeket a fogalmakat demonstrálja:

```cpp
template<typename T>
class Future {
    std::future<T> future_;

public:
    explicit Future(std::future<T>&& f)
        : future_(std::move(f)) {}

    template<typename F>
    auto then(F&& f) -> Future<decltype(f(std::declval<T>()))> {
        using R = decltype(f(std::declval<T>()));
        return Future<R>(std::async(
            std::launch::async,
            [future = std::move(future_), f = std::forward<F>(f)]()
               mutable {
                return f(future.get());
            }));
    }
};

Future<int> async_computation(int x) {
    return Future<int>(std::async(
        std::launch::async,
        [x] { return x * 2; }));
}

auto result = async_computation(42)
    .then([](int x) { return x + 1; })
    .then([](int x) { return std::to_string(x); });
```

6.5. Haladóbb minták

6.5.1. A State (állapot) Monád

Az állapot-monád a funkcionális programozás egyik legerősebb absztrakciója, különösen akkor, ha állapotfüggő számításokkal foglalkozunk tisztán funkcionális környezetben. A hagyományos imperatív programozás az állapotot a változók közvetlen mutációjával kezeli, ami számos problémához vezethet:

- Nehéz nyomon követni az állapotváltozásokat a függvényhívások között.
- Versenyfeltételek az egyidejűleg futó programokban
- Nehezen tesztelhető kód a rejtett állapotfüggőségek miatt.

- Bonyolult hibakeresés, ha az állapot inkonzisztenssé válik.
- A referenciális átláthatóság megsértése

Az állapot-monád megoldja ezeket a problémákat azáltal, hogy az állapottranszformációkat explicitté és kompozícionálhatóvá teszi. Ezt a következőkkel éri el:

- Az állapotmódosítások egy tiszta funkcionális kontextusban történő kapszulázása.
- Egyértelmű interfész biztosítása az állapotátmenetekhez
- Referenciális átláthatóság fenntartása
- Az állapotmódosító műveletek egyszerű kompozíciójának lehetővé tétele.
- A tesztelés megkönnyítése explicit állapotkezeléssel

Az állapot-monád mögött álló legfontosabb felismerés az, hogy az állapot közvetlen módosítása helyett az állapot-átalakításokat olyan függvényekként ábrázoljuk, amelyek az aktuális állapotot veszik, és egy eredményt és az új állapotot is visszaadják. Ez a megközelítés számos előnnyel jár:

- Az állapotváltozások egyértelművé és követhetővé válnak.
- Az állapotmódosítások monadikus műveletekkel összeállíthatók.
- A mellékhatások korlátozottak és kezelhetőek.
- A tesztelés egyszerűbbé válik, mivel az állapotátmenetek tiszta függvények.
- Az állapothoz való párhuzamos hozzáférés könnyebben értelmezhetővé válik.

Íme az állapot-monád átfogó implementációja:

```cpp
template<typename S, typename A>
class State {
    std::function<std::pair<A, S>(S)> run_;

public:
    explicit State(std::function<std::pair<A, S>(S)> f)
        : run_(std::move(f)) {}

    std::pair<A, S> run(S state) const {
        return run_(std::move(state));
    }

    template<typename F>
    auto bind(F&& f) const {
        return State<S, std::invoke_result_t<F, A>>(
            [=](S state) {
                auto [a, s] = run_(state);
                return f(a).run(s);
            });
    }

    template<typename F>
    auto map(F&& f) const {
        return State<S, std::invoke_result_t<F, A>>(
            [=](S state) {
                auto [a, s] = run_(state);
                return std::make_pair(f(a), s);
            });
    }
};

struct GameState {
    int score;
    int level;
};

State<GameState, int> increment_score(int points) {
    return State<GameState, int>([points](GameState state) {
        state.score += points;
        return std::make_pair(points, state);
    });
}

State<GameState, void> level_up() {
    return State<GameState, void>([](GameState state) {
        state.level++;
        return std::make_pair(void(), state);
    });
}
```

6.5.2. A Reader Monád

A Reader monád a szoftverfejlesztés közös kihívásával foglalkozik: a függőségek és konfigurációk tiszta, funkcionális módon történő kezelésével. A függőségkezelés hagyományos megközelítései gyakran a következőkre támaszkodnak:

- Globális változók vagy szingutonok
- Függőség-injektáló keretrendszerek
- Szolgáltatáskeresők
- Hosszú paraméterlisták

Ezen megközelítések mindegyikének vannak hátrányai, például:

- Rejtett függőségek, amelyek megnehezítik a kód megértését és tesztelését
- Összetett beállítási követelmények
- Futásidejű többletköltség a függőségi feloldásból
- Rugalmatlan konfigurációk, amelyeket nehéz módosítani

A Reader monád elegáns megoldást kínál ezekre a problémákra:

- A függőségek explicitté tétele a típusrendszerben
- Összeállítható interfészt biztosít a függő számításokhoz
- Könnyű tesztelés lehetővé tétele a környezet helyettesítésével
- A hivatkozási átláthatóság fenntartása
- A konfigurációkezelés egyszerűsítése

Íme a Reader monád részletes megvalósítása:

```cpp
template<typename E, typename A>
class Reader {
    std::function<A(const E&)> run_;

public:
    explicit Reader(std::function<A(const E&)> f)
        : run_(std::move(f)) {}

    A run(const E& env) const {
        return run_(env);
    }

    template<typename F>
    auto bind(F&& f) const {
        return Reader<E, std::invoke_result_t<
            decltype(f), A>>([=](const E& env) {
            return f(run_(env)).run(env);
        });
    }

    template<typename F>
    auto map(F&& f) const {
        return Reader<E, std::invoke_result_t<F, A>>(
            [=](const E& env) {
                return f(run_(env));
            });
    }
};

struct Config {
    std::string api_key;
    std::string base_url;
};

Reader<Config, std::string> get_api_url(
    const std::string& endpoint) {
    return Reader<Config, std::string>(
        [endpoint](const Config& config) {
            return config.base_url + "/" + endpoint;
        });
}
```

6.6. Végül

Számos fejlett funkcionális programozási mintát és azok gyakorlati alkalmazását vizsgáltuk meg C++ nyelven. Ezek a minták hatékony eszközöket biztosítanak a komplexitás kezeléséhez, a típusbiztonság biztosításához és a karbantarthatóbb kód írásához. Az általunk tárgyalt monadikus absztrakciók elegáns megoldásokat kínálnak a gyakori

programozási kihívásokra, miközben megtartják a C++ kódtól elvárt teljesítményjellemzőket.

A legfontosabb tudnivalók a következők:

- A monád minták ereje a komplexitás kezelésében
- A típusbiztonság jelentősége a funkcionális programozásban
- Az explicit hibakezelés és erőforrás-kezelés előnyei
- A funkcionális kompozíció eleganciája valós problémák megoldásában

A. függelék

Fontosabb programkódok listája

Az alábbi kód egy egyszerű Haskell-ben megírt program interpretáló program, mely képes dolgozni számokkal, Boole értékekkel és limitáltan karakterláncokkal.

```
import Control.Monad.Except
import Control.Monad.Reader
import Control.Monad.State
import Text.Parsec
import Text.Parsec.String (Parser)
import Text.Parsec.Expr
import Control.Monad (void, forever)
import Data.Functor.Identity (Identity)

-- AST
data Expr = Num Int | Var String | Add Expr Expr | Mul Expr Expr | Div
   Expr Expr
          | If Expr Expr Expr | Bool Bool | And Expr Expr | Or Expr Expr
              | Not Expr
          | Str String | Concat Expr Expr deriving (Show)

-- Values and Environment
data Value = IntVal Int | BoolVal Bool | StrVal String deriving (Show,
   Eq)
type Env = [(String, Value)]
type ErrorMsg = String
type EvalM a = ReaderT Env (ExceptT ErrorMsg (StateT Env IO)) a

-- Evaluator
eval :: Expr -> EvalM Value
eval (Num n) = return $ IntVal n
eval (Var x) = asks (lookup x) >>= maybe (throwError $ "undefined
   variable: " ++ x) return
eval (Add e1 e2) = evalBinOp (+) e1 e2
eval (Mul e1 e2) = evalBinOp (*) e1 e2
eval (Div e1 e2) = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (IntVal n1, IntVal n2) -> if n2 == 0 then throwError "division
           by zero" else return $ IntVal (n1 `div` n2)
        _ -> throwError "expected integer"
eval (If c t e) = do
```

```
    v <- eval c
    case v of
        BoolVal b -> eval (if b then t else e)
        _ -> throwError "expected boolean"
eval (Bool b) = return $ BoolVal b
eval (And e1 e2) = evalBoolOp (&&) e1 e2
eval (Or e1 e2) = evalBoolOp (||) e1 e2
eval (Not e) = do
    v <- eval e
    case v of
        BoolVal b -> return $ BoolVal (not b)
        _ -> throwError "expected boolean"
eval (Str s) = return $ StrVal s
eval (Concat e1 e2) = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (StrVal s1, StrVal s2) -> return $ StrVal (s1 ++ s2)
        _ -> throwError "expected string"

evalBinOp :: (Int -> Int -> Int) -> Expr -> Expr -> EvalM Value
evalBinOp op e1 e2 = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (IntVal n1, IntVal n2) -> return $ IntVal (op n1 n2)
        _ -> throwError "expected integer"

evalBoolOp :: (Bool -> Bool -> Bool) -> Expr -> Expr -> EvalM Value
evalBoolOp op e1 e2 = do
    v1 <- eval e1
    v2 <- eval e2
    case (v1, v2) of
        (BoolVal b1, BoolVal b2) -> return $ BoolVal (op b1 b2)
        _ -> throwError "expected boolean"

-- Parser
lexeme :: Parser a -> Parser a
lexeme p = p <* spaces

symbol :: String -> Parser String
symbol = lexeme . string

reserved :: String -> Parser ()
reserved s = lexeme (string s) >> return ()

expr :: Parser Expr
expr = buildExpressionParser table term <?> "expression"

-- Binary operator definition
binary :: String -> (Expr -> Expr -> Expr) -> Assoc -> Operator String
    () Identity Expr
binary name f assoc = Infix (try (symbol name) >> return f) assoc

-- Updated operator precedence table
table :: OperatorTable String () Identity Expr
table = [ [binary "*" Mul AssocLeft, binary "/" Div AssocLeft]
        , [binary "++" Concat AssocLeft]
```

```
        , [binary "+" Add AssocLeft]
        , [binary "&&" And AssocLeft]
        , [binary "||" Or AssocLeft]
        ]

-- Term parser refinement
term :: Parser Expr
term = choice [ parens expr
              , Num . read <$> lexeme (many1 digit)
              , Bool True <$ reserved "True"
              , Bool False <$ reserved "False"
              , Str <$> stringLiteral
              , reserved "Not" >> Not <$> term
              , ifExpr
              ] <?> "term"

ifExpr :: Parser Expr
ifExpr = do
    reserved "If"
    cond <- expr
    reserved "Then"
    tr <- expr
    reserved "Else"
    fl <- expr
    return $ If cond tr fl


stringLiteral :: Parser String
stringLiteral = lexeme $ char '"  *> many (noneOf "\"") <* char '"

parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")

-- REPL and Runner
runEval :: Expr -> Env -> IO (Either ErrorMsg Value)
runEval expr env = evalStateT (runExceptT (runReaderT (eval expr) env))
   env

repl :: IO ()
repl = forever $ do
    putStr "> "
    line <- getLine
    case parse (spaces *> expr <* eof) "" line of
        Left err -> print err
        Right e -> runEval e [] >>= either putStrLn print

main :: IO ()
main = repl
```

Irodalomjegyzék

Bird, R. *Thinking Functionally with Haskell*. Cambridge University Press, 2014. URL https://www.cambridge.org/core/books/thinking-functionally-with-haskell/.

Bishop, C. M. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2nd edition, 2016. ISBN 978-1493938438.

cdsmith, . Why do monads matter?, 2012.

Crichton, W. Typed design patterns for the functional era. *arXiv preprint arXiv:2307.07069*, 2023. URL https://arxiv.org/abs/2307.07069.

Dijkstra, E. W. *An ALGOL 60 Translator for the X1*. 1961. URL https://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF.

Egi, S. és Nishiwaki, Y. Functional programming in pattern-match-oriented programming style. *arXiv preprint arXiv:2002.06176*, 2020. URL https://arxiv.org/abs/2002.06176.

Hutton, G. és Meijer, E. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437--444, 1999. doi: 10.1017/S0956796898003082.

Hutton, G. és Meijer, E. Monadic parsing in haskell. *Journal of functional programming*, 8(4):437--444, 1998a.

Hutton, G. és Meijer, E. Monadic parsing in haskell. *Journal of Functional Programming*, 8(4):437--444, 1998b. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/abs/monadic-parsing-in-haskell/.

Kuncak, V. Modular interpreters in haskell. 2000.

Leijen, D. és Meijer, E. Parsec: Direct style monadic parser combinators for the real world. 2001.

Leroy, X. Formal verification of a realistic compiler. In *Communications of the ACM*, volume 52, pages 107--115. ACM, 2009. doi: 10.1145/1538788.1538814.

Malakhovski, J. és Soloviev, S. Programming with applicative-like expressions. *arXiv preprint arXiv:1905.10728*, 2019. URL https://arxiv.org/abs/1905.10728.

McCool, M., Reinders, J., és Robison, A. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2017. ISBN 978-0124159938.

Meijer, E., Fokkinga, M., és Paterson, R. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on*

functional programming languages and computer architecture, pages 124--144. Springer, 1991.

Moggi, E. Notions of computation and monads. *Information and Computation*, 93(1):55--92, 1991. URL https://www.sciencedirect.com/science/article/pii/0890540191900524.

Moll, R. N., Arbib, M. A., és Kfoury, A. J. *An introduction to formal language theory*. Springer Science & Business Media, 2012.

O'Sullivan, B., Goerzen, J., és Stewart, D. B. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008a.

O'Sullivan, B., Stewart, D., és Goerzen, J. *Real World Haskell*. O'Reilly Media, 2008b. URL http://book.realworldhaskell.org/.

Popa, D. How to build a monadic interpreter in one day. *Stud. Cercet. Stiint., Ser. Mat., Supplement Proceedings of CNMI*, 17:173--192, 2007.

Rivas, E. és Jaskelioff, M. Notions of computation as monoids. *arXiv preprint arXiv:1406.4823*, 2014. URL https://arxiv.org/abs/1406.4823.

Steele, G. L. Building interpreters by composing monads. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 1994. URL https://api.semanticscholar.org/CorpusID:207178347.

Swierstra, S. D. Combinator parsing: A short tutorial. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 252--300. Springer, 2008.

Wadler, P. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 113--128. Springer, 1985.

Wadler, P. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61--78, 1990.

Wadler, P. The essence of functional programming. *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1--14, 1992. URL https://dl.acm.org/doi/10.1145/143165.143169.

Wadler, P. Monads for functional programming. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24--30, 1995 Tutorial Text 1*, pages 24--52. Springer, 1995.