8. Parsarea programelor folosind Parsec

Parsec este o bibliotecă de analiză lexicală și sintactică, una din multele disponibile pentru limbajul Haskell.

Ea este implementată într-un mod asemănător cu ceea ce am văzut la curs, dar cu mult mai multă atenție la eficiență și la mesajele de eroare.

Pornind de la mecanismele simple pe care le-am explorat la curs, Parsec implementează funcționalități complexe, dintre care vom explora astăzi pe cele legate de construirea unui parser pentru un limbaj.

Analiza lexicală

Pentru a face mai eficientă analiza sintactică, Parsec oferă posibilitatea definirii și folosirii unui analizor lexical (tokenizer) pentru limbajul nostru.

Inițializarea analizorului lexical se face prin specifiarea mai multor parametrii generici ai limbajului, ca membri ai unei înregistrări.

Parametrii sunt următorii:

commentStart începutul unui comentariu de tip bloc
commentEnd sfârșitul unui comentariu de tip bloc
commentLine începutul unui comentariu de tip linie
nestedComments dacă sunt permise comentarii în comentarii
identStart parser pentru primul caracter al unui identificator
identLetter parser pentru un caracter următor al unui identificator
opStart parser pentru primul caracter al unui operator
opLetter parser pentru un caracter următor al unui operator
reservedNames lista cuvintelor cheie
reservedOpNames lista operatorilor predefiniți
caseSensitive dacă limbajul face diferență între litere mari și mici

De exemplu, pentru limbajul imperativ SIMPLE, o astfel de inițializare ar putea arăta astfel:

```
import qualified Text.Parsec.Token as Token
import Text.Parsec.String ( Parser, parseFromFile )
import Text.Parsec.Expr
    ( buildExpressionParser,
      Assoc(..),
      Operator(..) )
import Text.ParserCombinators.Parsec.Language
    ( emptyDef,
      GenLanguageDef( .. ),
      LanguageDef)
import Text.Parsec ( alphaNum, letter, (<|>), eof )
```

```haskell
impLanguageDef :: LanguageDef ()
impLanguageDef =
    emptyDef
    { commentStart = "/*"
    , commentEnd = "*/"
    , commentLine  = "//"
    , nestedComments = False
    , caseSensitive = True
    , identStart = letter
    , identLetter = alphaNum
    , reservedNames =
        [ "while", "if", "else", "int", "bool"
        , "true", "false", "read", "print"
        ]
    , reservedOpNames =
        [ "+", "-", "*", "/", "%"
        , "==", "!=", "<", "<=", ">=", ">"
        , "&&", "||", "!", "="
        ]
    }
```

Pe baza unei astfel de "definiție de limbaj" putem crea un analizor lexical

```haskell
impLexer :: Token.TokenParser ()
impLexer = Token.makeTokenParser impLanguageDef
```

Acest TokenParser este la rândul său un tip înregistrare, care ne oferă mai multe analizoare sintactice (Parser) construite pe baza definiției de limbaj date de noi. Dintre acestea:

identifier Parser pentru identificatori
reserved parser pentru cuvinte cheie
reservedOp parser pentru operatorii predefiniți
integer parser pentru întregi
whiteSpace parser pentru spații neimportante
parens parser pentru ceva între paranteze
braces parser pentru ceva între acolade
semiSep parser pentru liste separate de `;`

Pentru a fi mai ușor de lucrat cu aceștia, fără a fi nevoie să specificați mereu înregistrarea de bază `impLexer`, se obișnuiește folosirea definițiilor locale:

```haskell
identifier :: Parser String
identifier = Token.identifier impLexer
reserved :: String -> Parser ()
reserved = Token.reserved impLexer
reservedOp :: String -> Parser ()
reservedOp = Token.reservedOp impLexer
parens :: Parser a -> Parser a
```

```
parens = Token.parens impLexer
braces :: Parser a -> Parser a
braces = Token.braces impLexer
semiSep :: Parser a -> Parser [a]
semiSep = Token.semiSep impLexer
integer :: Parser Integer
integer = Token.integer impLexer
whiteSpace :: Parser ()
whiteSpace = Token.whiteSpace impLexer
```

De exemplu, folosind aceste construcții, putem descrie un parser pentru instrucțiunea `if (exp) s1 else s2` astfel:

```
ifStmt :: Parser Stmt
ifStmt = do
    reserved "if"
    cond <- parens expression
    thenS <- statement
    reserved "else"
    elseS <- statement
    return (If cond thenS elseS)

statement :: Parser Stmt
statement = ifStmt <|> ....

expression :: Parser Exp
expression = ...
```

Parser-ul astfel obținut, va ști să sară peste spații și comentarii, să recunoască cuvintele cheie.

Recunoașterea expresiilor

Dificultatea în recunoașterea expresiilor constă în aceea că operatorii aritmetici și logici au anumite priorități și moduri proprii de grupare.

Deși nu ar fi imposibil să construim singuri un parser care să țină cont de toate aceste proprietăți, procesul ar fi anevoios și rezultatul destul de complex.

Din fericire, Parsec vine în ajutor, prin modulul Text.Parsec.Expr care oferă modalități de a specifica atributele operațiilor și precedența lor precum și un combinator pentru construirea unui parser pentru expresii dintr-o tabelă de operatori și un parser pentru expresiile atomice (identificatori, constante).

De exemplu, pentru limbajul imperativ SIMPLE, parser-ul pentru expresii ar putea arăta astfel:

```
expression :: Parser Exp
expression = buildExpressionParser operators term
```

```
  where
    operators =
        [ [ prefix "!" Not
          ]
        , [ binary "*" (BinA Mul) AssocLeft
          ]
        , [ binary "+" (BinA Add) AssocLeft
          ]
        , [ binary "==" (BinE Eq) AssocNone
          , binary "<=" (BinC Lte) AssocNone
          ]
        , [ binary "&&" (BinL And) AssocLeft
          , binary "||" (BinL Or) AssocLeft
          ]
        ]
    binary  name fun = Infix ( reservedOp name >> return fun)
    prefix  name fun = Prefix ( reservedOp name >> return fun)
```

unde parser-ul pentru expresii atomice poate fi definit ca:

```
term :: Parser Exp
term =
    parens expression
    <|> (I <$> integer)
    <|> (Id <$> identifier)
```

În exemplul de mai sus, `operators` este o listă de liste de operatori, specificând în ordine grupele de precedență ale operatorilor din limbaj.

O grupă aflată înaintea alteia indică o precedentă mai mare. Operatorii din același grup au aceeași precedență. `AssocLeft` indică că gruparea se face la stânga.

Constructorii, gen `Not` sau `BinA Mul` sunt funcțiile la care vrem să transmitem valorile parsate ale operanzilor corespunzători operatorului specificat.

Exercițiul I

Scrieți un parser pentru SIMPLE, care ar trebui să transforme un program de genul celui conținut în fișierul `1.simple` în sintaxa abstractă conținută în fișierul `SIMPLE.hs`.