Language Processing

2023

Compilation versus Interpretation

Computer languages are executed in one of two possible ways

- Compilation: the original program (aka the source code) is translated into machine code by a compiler. Once compiled, the hardware can directly execute the program.
- Interpretation: the operations and instructions of the source code are directly executed by an interpreter by performing the respective computations, without generating machine code.
- Hybrid scheme: the compiler, instead of machine code, generates a lower-level language, which in turn is interpreted. Examples: the Haskell interpreter GHCi, Java compilers.

Abstract Syntax Trees

- Trees can be used to achieve a more efficient representation for collections of items than ordinary lists.
- Example: to evaluate arithmetic expressions, we use trees to represent the expression structure. Similar, if more complex, tree representations are generally used to represent programs in interpreters and compilers - Abstract Syntax Trees (AST)
- 1) the input string is decomposed into individual symbols or tokens (by a process often called lexing)
- 2) the token stream is parsed and turned into an abstract syntax tree
- 3) this AST is processed (compiled or interpreted) by a recursive tree traversal.

Lexer

- Decomposes the string containing expressions into a sequence of symbols, which we call tokens. For the example of 2 + 7 * 13, we expect the following token sequence: 2, +, 7, *, 13.
- During lexing (or tokenisation), we discard all whitespace and similar characters that do not contribute to the meaning of the tokenised expression.
- This process is carried out by a lexer function:

$$lexer :: String -> [Token]$$

Tokens

The type of Tokens represents all symbols that may be produced when the lexer tokenises expressions:

```
data Token
  = PlusTok
  | TimesTok
  | OpenTok
  | CloseTok
  | IntTok Int
  deriving (Show)
```

In this example, the only type constructor that requires an argument is IntTok, as it is not sufficient to know that we read an integer literal, we need to know the value that it represents.

Lexer

Example: When we apply the lexer to the arithmetic expression string "$2 + 7 * 13$", we get the token sequence *[IntTok 2, PlusTok, IntTok 7, TimesTok, IntTok 13]*

```
lexer :: String -> [Token]
lexer []                = []
lexer ('+' : restStr) = PlusTok  : lexer restStr
lexer ('*' : restStr) = TimesTok : lexer restStr
lexer ('(' : restStr) = OpenP    : lexer restStr
lexer (')' : restStr) = CloseP   : lexer restStr
lexer (chr : restStr)
  | isSpace chr       = lexer restStr
```

Lexer

```
lexer str@(chr : _)
  | isDigit chr
  = IntTok (stringToInt digitStr) : lexer restStr
  where
     (digitStr, restStr) = break (not . isDigit) str
     -- convert a string to an integer
     stringToInt :: String -> Int
     stringToInt=foldl (\acc chr->10*acc+digitToInt chr) 0
  -- runtime error:
lexer (_ : restString)
  = error ("unexpected character: '" ++ show chr ++ "'")
```

Parsing

- The next step is to convert the sequence of tokens into a representation which exposes the fact that the expression "2 + 7 * 13"is the sum of 2 and the product of 7 and 13
- This process is carried out by a parser function:

$$parser :: [Token] -> Expr$$

Representing expressions (AST)

```
data Expr
  = IntLit Int           -- integer constants
  | Add    Expr Expr     -- addition node
  | Mult   Expr Expr     -- multiplication node
```

We represent the arithmetic expression "2 + 7 * 13"as

Add (IntLit 2) (Mult (IntLit 7) (IntLit 13))

Now, we need to convert the list of Tokens into a value of type Expr. Depending on the language, parsing can be a complex problem.

Parsing integer literals

```
parseInt :: [Token] -> Maybe (Expr, [Token])
parseInt (IntTok n : restTokens)
  = Just (IntLit n, restTokens)
parseInt tokens
  = Nothing
```

Parsing products

```
parseProdOrInt :: [Token] -> Maybe (Expr, [Token])
parseProdOrInt tokens
  = case parseInt tokens of
      Just (expr1, (TimesTok : restTokens1)) ->
          case parseProdOrInt restTokens1 of
            Just (expr2, restTokens2) ->
                  Just (Mult expr1 expr2, restTokens2)
            Nothing                        -> Nothing
      result -> result      -- can be 'Nothing' or valid
```

Parsing sums

```
parseSumOrProdOrInt :: [Token] -> Maybe (Expr, [Token])
parseSumOrProdOrInt tokens
  = case parseProdOrInt tokens of
      Just (expr1, (PlusTok : restTokens1)) ->
          case parseProdOrInt restTokens1 of
            Just (expr2, restTokens2) ->
                  Just (Add expr1 expr2, restTokens2)
            Nothing                       -> Nothing
      result -> result    -- could be 'Nothing' or valid
```

Parenthesised expressions

```
parseIntOrParenExpr :: [Token] -> Maybe (Expr, [Token])
parseIntOrParenExpr (IntTok n : restTokens)
        = Just (IntLit n,   restTokens)
parseIntOrParenExpr (OpenP : restTokens1)
        = case parseSumOrProdOrIntOrPar restTokens1 of
                Just (expr, (CloseP : restTokens2)) ->
                                Just (expr, restTokens2)
                Just _  -> Nothing -- no closing paren
                Nothing -> Nothing
parseIntOrParenExpr tokens = Nothing
```

Finally, adapt *parseSumOrProdOrInt* and *parseProdOrIntExpr* to call the new variant.

Parsing products or parenthesised expressions

```
parseProdOrIntOrPar :: [Token] -> Maybe (Expr, [Token])
parseProdOrIntOrPar tokens
  = case parseIntOrParenExpr tokens of
      Just (expr1, (TimesTok : restTokens1)) ->
          case parseProdOrIntOrPar restTokens1 of
            Just (expr2, restTokens2) ->
                     Just (Mult expr1 expr2, restTokens2)
            Nothing                        -> Nothing
      result -> result
```

Parsing sums or products or parenthesised expressions

```
parseSumOrProdOrIntOrPar::[Token] -> Maybe (Expr, [Token])
parseSumOrProdOrIntOrPar tokens
  = case parseProdOrIntOrPar tokens of
      Just (expr1, (PlusTok : restTokens1)) ->
          case parseSumOrProdOrIntOrPar restTokens1 of
            Just (expr2, restTokens2) ->
                        Just (Add expr1 expr2, restTokens2)
            Nothing                    -> Nothing
      result -> result
```

Top-level

The function *parse* calls *parseSumOrProdOrIntOrPar* and checks that it successfully consumed all input tokens, i.e., the result is of the form *Just (expr, [])* where the empty list indicates that no tokens where left over; otherwise, we have got a parse error.

```
parse :: [Token] -> Expr
parse tokens =
  case parseSumOrProdOrIntOrPar tokens of
       Just (expr, []) -> expr
       _               -> error "Parse error"
```

Parser libraries and parser generators

Compilers implement parsers using specific tools. In Haskell:

- Top-down parsing: *Parsec*
  `https://github.com/haskell/parsec`
- Bottom-up parsing: *Happy*
  `https://haskell-happy.readthedocs.io/en/latest/`