Symbol Table - Documentation

Hufflepuff

June 24 - 2024

With this module `SymbolTable` we will manage the symbol table, which is an important part for the next step, semantic analysis, where we will store the necessary information about the different types of information that can be stored in the symbol table.

To achieve this approach, we will use the Haskell module `Data.Map`, which will help us implement our symbol table as a kind of map or dictionary, thanks to the key-value functionality it provides.

Now, we will starting to explain in more detail how our `SymbolTable` works.

1 Definition of `SymbolInfo`

```haskell
data SymbolInfo
    = VariableInfo Identifier Type (Maybe Value)
    | FunctionInfo Identifier Type [FuncParam] FuncBody
    | TaskInfo Identifier Task
    | MemberInfo Identifier Member
    | ListInfo Identifier List
    | BoolExpressionInfo BoolExpression
    | LiteralInfo Literal
    | DoAssignmentInfo Identifier Type Statement
    deriving (Show)
```

This `dataType` will represent the different types of information that can be stored in our `SymbolTable`. Explaining a bit more in detail, we have the following:

- `VariableInfo` will store the necessary information about a variable, where it will have its identifier, type, and an optional value.
- `FunctionInfo` will store information about a function, including its identifier, return type, parameters, and body.
- `TaskInfo` will store information about a task along with its identifier.
- `MemberInfo` will store information about a member along with its identifier.

- `ListInfo` will store information about a list along with its identifier.
- `BoolExpressionInfo` will store information about a boolean expression.
- `LiteralInfo` will store information about a literal.
- `DoAssignmentInfo` will store information about a `do` expression along with its identifier, type, and statement.

2 Definition of `SymbolTable`

```
newtype SymbolTable = SymbolTable (M.Map Identifier
    ↪ SymbolInfo)
    deriving (Show)
```

Defining the `dataType` of the `SymbolTable`, we have wrapped the 'Map' from the module `Data.Map` so that our keys are 'Identifier' and our values are `SymbolInfo`. Thanks to this, we have defined our dictionary, so to speak, and it is ready to work with this symbol table.

3 Manipulation of the `SymbolTable`

3.1 Creating an empty table

```
emptyTable :: SymbolTable
emptyTable = SymbolTable M.empty
```

With this function, we will create an empty symbol table, which will serve as our starting point to begin working.

3.2 Inserting symbols

```
insertVariable :: Identifier -> Type -> Maybe Value ->
    ↪ SymbolTable -> SymbolTable
insertVariable name typ val (SymbolTable table) =
    SymbolTable (M.insert name (VariableInfo name typ val)
        ↪ table)

insertFunction :: Identifier -> Type -> [FuncParam] ->
    ↪ FuncBody -> SymbolTable -> SymbolTable
insertFunction name retType params body (SymbolTable table)
    ↪ =
    SymbolTable (M.insert name (FunctionInfo name retType
        ↪ params body) table)
```

```
insertTask :: Identifier -> Task -> SymbolTable ->
    ↪ SymbolTable
insertTask name task (SymbolTable table) =
    SymbolTable (M.insert name (TaskInfo name task) table)

insertMember :: Identifier -> Member -> SymbolTable ->
    ↪ SymbolTable
insertMember name member (SymbolTable table) =
    SymbolTable (M.insert name (MemberInfo name member)
        ↪ table)

insertList :: Identifier -> List -> SymbolTable ->
    ↪ SymbolTable
insertList name list (SymbolTable table) =
    SymbolTable (M.insert name (ListInfo name list) table)

insertBoolExpression :: Identifier -> BoolExpression ->
    ↪ SymbolTable -> SymbolTable
insertBoolExpression name boolExpr (SymbolTable table) =
    SymbolTable (M.insert name (BoolExpressionInfo boolExpr)
        ↪  table)

insertLiteral :: Identifier -> Literal -> SymbolTable ->
    ↪ SymbolTable
insertLiteral name lit (SymbolTable table) =
    SymbolTable (M.insert name (LiteralInfo lit) table)

insertDoAssignment :: Identifier -> Type -> Statement ->
    ↪ SymbolTable -> SymbolTable
insertDoAssignment name typ stmt (SymbolTable table) =
    SymbolTable (M.insert name (DoAssignmentInfo name typ
        ↪ stmt) table)
```

All these defined functions will help us add information to our symbol table. Basically, I have created a function for each specific data type, all of this using `M.insert` from the module `Data.Map`.

3.3 Searching for symbols

```
lookupSymbol :: Identifier -> SymbolTable -> Maybe
    ↪ SymbolInfo
lookupSymbol name (SymbolTable table) = M.lookup name table
```

This function will be mostly used in the part that will be implemented in the semantic analysis, because with this function, the symbols or information stored during the parser process can be searched.

3.4 Deleting symbols

```
removeSymbol :: Identifier -> SymbolTable -> SymbolTable
removeSymbol name (SymbolTable table) = SymbolTable (M.
    ↪ delete name table)
```

And well, a function that will help us delete a symbol from our table.

4 How the `SymbolTable` was implemented or adapted within the `Parser`

Explaining in more detail how the `SymbolTable` was integrated into our `Parser`. Obviously, I will not put all the code because it is almost 1000 lines of code, so I will show the most important and relevant parts to better understand the process.

Originally, the parser was implemented in this way.

```
code :: Parser Code
code = do
  whiteSpace
  f <- funcs
  whiteSpace
  d <- doStatement
  whiteSpace
  return $ Code f d

doStatement :: Parser DoStatement
doStatement = do
  reserved "do"
  whiteSpace
  reservedOp "{"
  whiteSpace
  c <- funcCall
  whiteSpace
  reservedOp "}"
  whiteSpace
  return $ DoStatement c
```

These are just two functions of the many we have. So, these two functions, if we notice, only parse to the data type `Code` and `DoStatement`. Everything is normal so far, but the question here is how we should implement the `SymbolTable` here. After thinking for a long time, the way that occurred to me was to add a new parameter to all functions. This parameter was to receive a `SymbolTable` to pass this parameter to all the necessary functions that need to interact with our symbol table. To better understand this, here is an example of how it was adapted.

`Note:` You may have noticed that the previous version was implemented with `do return`, and thus with all the functions. So, once the `SymbolTable` was adapted, it was changed to use `applicatives` or in this case `Functors`.

4.1 Adapted Parser

```
parseCode :: Parser (Code, SymbolTable)
parseCode =
  let initialSymbolTable = emptyTable
   in code initialSymbolTable

code :: SymbolTable -> Parser (Code, SymbolTable)
code symTable =
  whiteSpace *> funcs symTable >>= \(f, symTable1) ->
    whiteSpace *> doNotation symTable1 >>= \(d, symTable2)
      ↪ ->
      whiteSpace *> return (Code f d, symTable2)

doStatement :: SymbolTable -> Parser (DoStatement,
  ↪ SymbolTable)
doStatement symTable =
  try (doAssignment symTable)
    <|> try (doPrint symTable)
```

So, as I explained, my idea of how to implement it was this. First, a function that initializes the empty `SymbolTable`, and then adding as a parameter to the functions that need to interact with this table. And instead of just returning the parsed types, now we return a `tuple` that will contain the `(parsed code, updated symbol table)` and so on with the rest.

4.2 Example of `SymbolTable` manipulation

Here we have an example where we are parsing a `func` that is a function defined in our grammar. We receive the `SymbolTable` as a parameter, and once the parsing is done, we add it to our symbol table.

```
func :: SymbolTable -> Parser (Func, SymbolTable)
func symTable = do
  whiteSpace
  reserved "func"
  whiteSpace
  funId <- identifier
  whiteSpace
  reservedOp "->"
  whiteSpace
  funType <- dataType
  whiteSpace
  _ <- string "{"
  whiteSpace
  reserved "params"
  whiteSpace
  _ <- string "{"
  whiteSpace
```

```
  (params, symTable') <- funcParams symTable
  whiteSpace
  _ <- string "}"
  whiteSpace
  (body, symTable'') <- funcBody symTable'
  whiteSpace
  _ <- string "}"
  whiteSpace
  let func = Func funId (str2type funType) params body
  let symTable''' = insertFunction funId (str2type funType)
      ↪ params body symTable''
  return (func, symTable''')
```

We remember that for a function we have a specific function for them which was `insertFunction`. Using this, we insert into our symbol table the data we defined `Identifier Type [FuncParam] FuncBody`. Finally, we return the parsed code and the updated symbol table in a tuple to be used by other functions.

And in this way with the other data types that are required in our symbol table, this is how the SymbolTable works, and it is ready to be used for the work of semantic analysis.