Translator Interpreter

```
module TranslatorInterpreter
( interpret_translator
) where

import Translator
import Utilities
import Debug

interpret_translator :: TransCode -> IO Translator
interpret_translator transcode =
    let helper :: [String] -> Translator -> IO Translator
        helper words trans = case words of
            [] -> return trans
            ("(#)":ws) -> helper (extract_comment ws) trans
            ("" :ws)  -> helper ws trans
            (" " :ws)  -> helper ws trans
            ("\n":ws)  -> helper ws trans
            ("filetype":" ":filetype : ws) ->
                helper ws $ set_filetype filetype trans
            (title:" ":nests_str:" ":args_str : ws) ->
                let does_nest = string_to_bool nests_str
                    args_type = string_to_argstype args_str
                in case args_type of
                    ArgsNumber args_count ->
                        let (block_format, rest) =
                                make_count_formatter args_count ws
                            trans_new =
                                add_block title block_format does_nest trans
                        in do
                            -- putStrLn $ "title  : " ++ title
                            -- putStrLn $ "before : " ++ (show ws)
                            -- putStrLn $ "rest   : " ++ (show rest)
                            helper rest trans_new
                    ArgsStar ->
                        let formatter = make_star_formatter
                            (block_format, rest) = formatter ws
                            trans_new =
                                add_block title block_format does_nest trans
                        in helper rest trans_new
            _ -> error
                $ "couldn't interpret as translator code: " ++
                    (if length words < 10
                        then show words
                        else show $ take 10 words) ++ "..."
        splitted_transcode = transcode `splitted_with`
            [" ", "\n", "<|", "|>","(#)"]
        empty_translator = Translator
            [] (Block "root" [] (\xs -> join xs) True) (\fp -> fp)
    in do
        foldl (>>) (putStr "") (map (debug . show) splitted_transcode)
```

```
        helper splitted_transcode empty_translator

set_filetype :: String -> Translator -> Translator
set_filetype s (Translator blocks root _) =
    Translator blocks root (\fp -> fp ++ "." ++ s)

extract_comment :: [String] -> [String]
extract_comment ws = case ws of
    ("\n":rest) -> rest
    (_:rest) -> extract_comment rest

-- gets the next <|...|> enclosed text
extract_next_text :: [String] -> Maybe (String, [String])
extract_next_text strings =
    let extract_start :: [String] -> Maybe (String, [String])
        extract_start ss = case ss of
            [] -> Nothing
            ("<|":rest) -> extract_end rest
            (_   :rest) -> extract_start rest
        extract_end :: [String] -> Maybe (String, [String])
        extract_end ss = case ss of
            [] -> Nothing
            ("|>":rest) -> Just ("", rest)
            (s:rest) -> case extract_end rest of
                Nothing -> Nothing
                Just (ss, rest) -> Just (s ++ ss, rest)
    in extract_start strings

data ArgsType
    = ArgsNumber Int
    | ArgsStar

string_to_argstype :: String -> ArgsType
string_to_argstype s = case s of
    "*" -> ArgsStar
    int_str -> ArgsNumber $ string_to_int int_str

data ArgReference = ArgRefIndex Int

instance Show ArgReference where
    show (ArgRefIndex i) = show i

break_text :: String -> [Either String ArgReference]
break_text string =
    let helper :: String -> String -> [Either String ArgReference]
        helper str work = case str of
            "" -> case work of
                "" -> []
                _  -> [Left work]
            -- ('\\': x : xs) -> error $ "got escape for: " ++ [x]
            ('$' : x : xs) -> case (readMaybe [x] :: Maybe Int) of
                Nothing -> helper xs (work ++ ['$'] ++ [x])
                Just i  -> case work of
                    "" -> (Right $ ArgRefIndex i)
```

```
                        : helper xs ""
                _  -> (Left work)
                        : (Right $ ArgRefIndex i)
                        : helper xs ""
        (x : xs) -> helper xs (work ++ [x])
    in helper string ""

interpret_star_items :: String -> String -> String -> ([TargetCode]->TargetCode)
interpret_star_items begin item end =
    let helper :: [Either String ArgReference] -> (TargetCode -> TargetCode)
        helper [] _ = ""
        helper (Left  s          : xs) ts = s  ++ helper xs ts
        helper (Right (ArgRefIndex 1) : xs) ts = ts ++ helper xs ts
        splitted_item = break_text item
    in \ts -> begin ++ (join $ map (helper splitted_item) ts) ++ end

interpret_count_item :: String -> ([TargetCode] -> TargetCode)
interpret_count_item item =
    let helper :: [Either String ArgReference] -> [TargetCode] -> TargetCode
        helper [] _ = ""
        helper (Left s           : xs) ts = s           ++ helper xs ts
        helper (Right (ArgRefIndex i) : xs) ts = if i <= length ts
            then (ts `at` (i-1)) ++ helper xs ts
            else helper xs ts
        splitted_item = break_text item
    in \ts -> helper splitted_item ts

just :: Maybe a -> a
just mb_x = case mb_x of
    Just x -> x
    _ -> error "tried to get something from nothing"

make_count_formatter :: Int -> [String] -> ([TargetCode]->TargetCode, [String])
make_count_formatter count ss =
    let (item, rest) = just $ extract_next_text ss
    in (interpret_count_item item, rest)

make_star_formatter :: [String] -> ([TargetCode]->TargetCode, [String])
make_star_formatter ss =
    let (item1, rest1) = just $ extract_next_text ss
        (item2, rest2) = just $ extract_next_text rest1
        (item3, rest3) = just $ extract_next_text rest2
    in (interpret_star_items item1 item2 item3, rest3)

make_block :: String -> ([TargetCode] -> TargetCode) -> Bool -> Block
make_block title block_format does_nest = Block title [] block_format does_nest

add_block :: String->([TargetCode]->TargetCode)->Bool->Translator->Translator
add_block title block_format does_nest (Translator blocks root convert_fp) =
    let new_block = make_block title block_format does_nest
    in case title of
        "root" -> Translator blocks new_block convert_fp
        _      -> Translator (blocks ++ [new_block]) root convert_fp
```