Translator Interpreter

```
module TranslatorInterpreter
( interpret_translator
) where

import Translator
import Utilities

interpret_translator :: TransCode -> IO Translator
interpret_translator transcode =
    let helper :: [String] -> Translator -> IO Translator
        helper [] trans = return trans
        helper ("filetype" : " " : filetype : ws) trans =
            helper ws $ set_filetype filetype trans
        helper (title : nests_str : args_str : ws) trans =
            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 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
        splitted_transcode = transcode `splitted_with` [" ", "<|", "|>"]
        empty_translator = Translator
            [] (Block "root" [] (\x -> "") True) (\fp -> fp)
    in do
        putStrLn $ 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)

-- gets the next <|...|> enclosed text
extract_next_text :: [String] -> Maybe (String, [String])
extract_next_text s =
    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 s

data ArgsType
    = ArgsNumber Int
    | ArgsStar

string_to_argstype :: String -> ArgsType
string_to_argstype s = case s of
    "*" -> ArgsStar
    int_str -> ArgsNumber (read int_str :: Int)

data ArgReference = ArgInt Int | ArgStar

break_text :: String -> [Either String ArgReference]
break_text s = case s of
    [] -> []
    ('\\': x : xs) -> break_text xs
    ('$' :'*': xs) -> (Right $ ArgStar) : break_text xs
    ('$' : x : xs) -> (Right $ ArgInt $ string_to_int [x]) : break_text xs
    _ -> error $ "couldn't break '" ++ s ++ "' properly"

interpret_star_items :: String -> String -> String
                        -> ([TargetCode] -> TargetCode)
interpret_star_items begin item end =
    let helper :: [Either String ArgReference] -> TargetCode -> TargetCode
        helper [] tgtcode = ""
        helper (Left s : xs) tgtcode = s ++ helper xs tgtcode
        helper (Right (ArgInt 1) : xs) tgtcode = tgtcode ++ (helper xs tgtcode)
    in \[tgtcode] -> begin ++ (helper (break_text item) tgtcode) ++ 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 (ArgInt i) : xs) ts = (ts `at` i) ++ helper xs ts
    in helper $ break_text item

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
```