ATL Compile

1 Introduction

The function `compile` takes a language file name, `L.atf_lang`, and a list of source file names, `x1.atf_src, ..., xN.atf_src`, and does the following:

1. Parse `L.atf_lang`.
2. Compile the parse `L.atf_lang` into an ATF language specification. This specification, L, specifies the source syntax to be parsed and the target text format to write the compiled source as. The same L will be used for transpiling each of the `xi.atf_src`.
3. For each `xi.atf_src`, do the following.
   (a) Parse `xi.atf_src`.
   (b) Compile `xi.atf_src` into x_i, which is interpreted abstractly as in the framework of the target format specified by L.
   (c) Translate x_i into text format, written into `xi.atf_tgt`, where "`atf_tgt`" is the target file format specified by L.

```
module Compile
( compile
) where

import Debug
```

2 The Compile Function

```
compile :: FilePath -> [FilePath] -> IO ()
compile fp_lang fp_srcs = do
    putStrLn $ "compiling language: " ++ fp_lang
    langcode <- readFile fp_lang
    lang <- compile_language fp_lang
    foldl (>>) (putStr "") $
        map (\fp_src -> do
            putStrLn $ "compiling source: " ++ fp_src
            srccode <- readFile fp_src
            tgtcode <- compile_source lang fp_src
            writeFile (lang_convert_filepath lang fp_src) tgtcode)
        fp_srcs
```

3 Tokens

```
type Token = String
```

4 Compiling Language

```
type SourceCode = String
type LangCode   = String

data Language = Language
    { lang_wrapper_block    :: Block
    , lang_static_blocks    :: [Block]
    , lang_blocks           :: [Block]
    , lang_sections         :: [String]
    , lang_convert_filepath :: FilePath -> FilePath }

type TargetCode = String

data Block = Block
    { block_section        :: String
    , block_token_bounds   :: (Token, Token)
    , block_format_content :: [Either Token Block] -> TargetCode
    , block_content        :: [Either Token Block] }

add_content :: Block -> Either Token Block -> Block
add_content (Block sect tkbs form cont) x = Block sect tkbs form (cont++[x])

compile_language :: LangCode -> IO Language
compile_language langcode = -- TODO: implementation
    return example_language
```

5 Example Language

```
make_block sect tkbs form = Block sect tkbs form []

make_static_block :: String -> String -> Block
make_static_block sect cont = make_block sect ("","") (\_ -> cont)

join_content :: String -> String -> [Either Token Block] -> TargetCode
join_content header footer xs =
    let f :: Either Token Block -> TargetCode -> TargetCode
        f x tgtcode = case x of
            Left  t -> tgtcode ++ t
            Right b -> tgtcode ++ (block_format_content b $ block_content b)
    in (foldr f header xs) ++ footer

example_language = Language
    ( make_block "wrapper" ("","") (join_content "" "") )
    [ make_static_block "header" "" , make_static_block "footer" "" ]
    [ make_block "body" ("(",")") (join_content "<p>" "</p>") ]
    [ "header", "body", "footer" ]
    (\fp -> fp ++ ".exp_tgt")
```

6 Compiling Source

```
compile_source :: Language -> SourceCode -> IO TargetCode
compile_source lang srccode = return
    $ blocktree_to_targetcode lang
    $ tokens_to_blocktree lang
    $ sourcecode_to_tokens lang
    $ srccode


-- separates SourceCode into Tokens,
-- splitting with the tokens reserved by the Language.
sourcecode_to_tokens :: Language -> SourceCode -> [Token]
sourcecode_to_tokens lang srccode =
    let helper :: SourceCode -> [Token] -> Token -> [Token]
        helper srccode lang_tkns work_token = case (srccode, lang_tkns) of
            -- finished all of the srccode.
            -- append work_token if its non-empty.
            ("", _) -> if work_token == ""
                then []
                else [work_token]
            -- have gone through all tokens, so add the front char
            -- to the work_token, and recurse through all of
            -- the lang tokens.
            (c:s, []) -> helper s all_lang_tkns (work_token++[c])
            -- check to see if t extracts from s. if so, then
            -- prepend t and then restart recurse on rest of srccode.
            -- prepend work_token before the newfound token, if work_token
            -- is non-empty
            (s, t:ts) -> case t `extracted_from` s of
                Nothing -> helper s ts work_token
                Just s_rest -> if work_token == ""
                    then t : helper s_rest all_lang_tkns ""
                    else work_token : t : helper s_rest all_lang_tkns ""
        all_lang_tkns = language_special_tokens lang
    in helper srccode all_lang_tkns ""

-- if target is a substring of string and starts at the beginning of string,
-- then is Just the rest of string after target ends.
-- otherwise, is Nothing
extracted_from :: String -> String -> Maybe String
target `extracted_from` string =
    case (target, string) of
        ("", s ) -> Just s
        (_ , "") -> Nothing
        (x:xs, s:ss) -> if s == x
            then xs `extracted_from` ss
            else Nothing

language_special_tokens :: Language -> [Token]
language_special_tokens lang =
    let helper :: [Block] -> [Token] -> [Token]
        helper []      ts = ts
        helper (b:bs)  ts =
            let (t1, t2) = block_token_bounds b
```

```
                add1 = if t1 `elem` ts
                    then [] else [t1]
                add2 = if t2 `elem` ts || t2 == t1
                    then [] else [t2]
            in helper bs (add1 ++ add2 ++ ts)
    in helper (lang_blocks lang) []


-- breaks Token list into a Block tree
tokens_to_blocktree :: Language -> [Token] -> Block
tokens_to_blocktree lang ts =
    let helper :: [Token] -> Block -> (Block, [Token])
        helper ts work_block = case ts of
            [] -> (work_block, [])
            -- does token begin new block?
            (t:ts) -> case block_that_begins_with t of
                -- token begins new block
                Just block ->
                    let new_work_block = add_content
                            work_block (Right new_block)
                        (new_block, ts_rest) = helper ts block
                    in helper ts_rest new_work_block
                -- does token end current block?
                Nothing ->
                    if (t == (snd $ block_token_bounds work_block)
                    && ("wrapper" /= block_section work_block))
                        -- token ends current work_block
                        then (work_block, ts)
                        -- token is normal; append to current block
                        else helper ts $ add_content work_block (Left t)
        block_that_begins_with :: Token -> Maybe Block
        block_that_begins_with t =
            let helper :: [Block] -> Maybe Block
                helper [] = Nothing
                helper (b:bs) =
                    let (t1, t2) = block_token_bounds b
                    in if t == t1
                        then Just b
                        else helper bs
            in helper $ lang_blocks lang
    in fst $ helper ts (lang_wrapper_block lang)


-- arranges the Block tree into the finalized TargetCode
blocktree_to_targetcode :: Language -> Block -> SourceCode
blocktree_to_targetcode _ _ = unimplemented
```