

ATL Compile

1 Introduction

The function `compile` takes a Translator file name, `T.atf_trans`, and a list of source file names, `x1.atf_src`, ..., `xN.atf_src`, and does the following:

1. Parse `T.atf_trans`.
2. Compile the parse `T.atf_trans` into an ATF translator specification. This specification, T , specifies the source syntax to be parsed and the target text format to write the compiled source as. The same T 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 T .

```
module Compile
( compile
) where

import Debug
import Utilities
import Translator
import TranslatorInterpreter
import TranslatorLibrary
```

2 The Compile Function

```
compile :: FilePath -> [FilePath] -> IO ()
compile fp_trans fp_srcs = do
  putStrLn $ "compiling translator: " ++ fp_trans
  transcode <- readFile fp_trans
  trans <- interpret_translator transcode
  foldl (>>) (putStr "") $
    map (\fp_src -> do
      putStrLn $ "compiling source: " ++ fp_src
      srccode <- readFile fp_src
      tgtcode <- compile_sourcecode trans srccode
      writeFile (trans_convert_filepath trans fp_src) tgtcode)
    fp_srcs
```

3 Example Translator

4 Compiling SourceCode

```
compile_sourcecode :: Translator -> SourceCode -> IO TargetCode
compile_sourcecode translator sourcecode = do
  blocktree <- sourcecode_to_blocktree translator sourcecode
  debug $ "blocktree: " ++ (show blocktree)
  targetcode <- blocktree_to_targetcode translator blocktree
  debug $ "targetcode: " ++ targetcode
  return targetcode
```

4.1 SourceCode to Block-tree

```
set_children (Block title _ format does_nest) children =
  Block title children format does_nest

prepend_blockchild (Block title children format does_nest) child =
  Block title (child : children) format does_nest

add_child :: Block -> BlockChild -> Block
add_child block child = case child of
  -- new child is code
  (ChildCode x) -> case (block_children block) of
    -- if most recent child is code, append x to that
    (ChildCode y : cs_) -> set_children block $ ChildCode (y ++ x) : cs_
    -- otherwise, prepend to children
    _ -> prepend_blockchild block $ ChildCode x
  -- new child is a block
  (ChildBlock x) -> prepend_blockchild block $ ChildBlock x

add_partition :: Block -> Block
add_partition block = case (block_children block) of
  (ChildCode "":_) -> block
  _ -> prepend_blockchild block $ ChildCode ""

sourcecode_to_blocktree :: Translator -> SourceCode -> IO Block
sourcecode_to_blocktree trans sourcecode =
  let root_block = trans_root_block trans
      all_blocks = trans_blocks trans

  extract_empty_block :: SourceCode -> (Block, SourceCode)
  extract_empty_block srccode =
    let helper :: [Block] -> (Block, SourceCode)
        helper blocks = case blocks of
          -- none of blocks matched
          [] -> error $ "no block title found at: "
              ++ (show $ (take 20 srccode) ++ " ...")
          -- check if srccode starts with the scope's title
          (b:bs) -> case srccode 'beheaded_by' (block_title b) of
            -- this scope matches
            Just srccode_rest -> (b, srccode_rest)
            -- this scope does not match
            Nothing -> helper bs
    in helper all_blocks
```

```

helper :: Block -> SourceCode -> IO (Block, SourceCode)
helper block srccode = if (block_does_nest block)
  -- parse nesting
  then case srccode of
    -- begin block
    ('<':'|':xs) ->
      let (empty_block, xs_) = extract_empty_block xs
      in do
        (new_block, xs_rest) <- helper empty_block xs_
        debug $ "begin block: " ++ (block_title new_block)
        helper (add_child block $ ChildBlock new_block) xs_rest
    -- end current block
    ('|':':>':xs) -> do
      debug $ "end block: " ++ (block_title block)
      return (block, xs)
    -- add partition to current block
    ('|':':xs) ->
      let new_block = add_partition block
      in do
        debug $ "part block: " ++ (block_title new_block)
        helper new_block xs
    -- escape character
    ('\\':':x:xs) ->
      let new_block = add_child block (ChildCode [x])
      in do
        -- debug $ "add escaped: " ++ [x]
        helper new_block xs
    -- add normal character
    (x:xs) ->
      let new_block = add_child block (ChildCode [x])
      in do
        -- debug $ "add char: " ++ [x]
        helper new_block xs
    -- end of sourcecode
    "" -> do
      debug "end of sourcecode"
      return (block, "")
  -- parse raw; no nesting
  else case srccode of
    -- end current block
    ('|':':>':xs) -> do
      debug $ "end block: " ++ (block_title block)
      return (block, xs)
    -- escape character
    ('\\':':x:xs) ->
      let new_block = if x 'elem' ">"
      then add_child block (ChildCode [x])
      else add_child block (ChildCode $ '\\':':x:[])
      in helper new_block xs
    -- add normal character
    (x:xs) ->
      let new_block = add_child block (ChildCode [x])
      in helper new_block xs

```

```

        -- end of sourcecode
        "" -> do
            debug "end of sourcecode"
            return (block, "")
    in do
        (block, _) <- helper root_block sourcecode
        return block

```

4.2 Block-tree to TargetCode

```

blocktree_to_targetcode :: Translator -> Block -> IO TargetCode
blocktree_to_targetcode trans block =
    let helper :: [BlockChild] -> IO [TargetCode]
        helper children = case children of
            [] -> return []
            (child:cs) -> case child of
                ChildCode x -> do
                    transed_cs <- helper cs
                    return $ transed_cs
                        ++ (if is_empty_string x then [] else [x])
                ChildBlock x -> do
                    transed_child <- blocktree_to_targetcode trans x
                    transed_cs <- helper cs
                    return $ transed_cs ++ [transed_child]
    in do
        transed_children <- helper (block_children block)
        return $ block_format block $ transed_children

```