YAX: Yet Another Cross Referencer

Name: Xuheng Li(UNI: xl2784)

December 22, 2021

1 Introduction

A cross referencing tool, or commonly known as cross referencer, is a software that indexes source code and provides information for symbols and definitions on a given code base such that the user can find where a symbol is defined or used in that code base. The cross referencer, such as Cscope [1], is widely used in software development and integrated into IDEs and editors like vscode or vim. Since parsing the symbols on a single source code file is usually independent from the rest of the files for the given code base, the procedure of building the database can be paralleled. Therefore, I implemented **YAX: Yet Another Cross Referencer**, a parallelized version of Cscope written in Haskell. Given the time constraint, YAX will only work on preprocessed C99 [3] source code. Other languages, including various C extensions, such as GNU C extension or LLVM C extension, are not supported.

Alex [5] and Happy [6] are the Haskell counterpart for Lex and YACC [4] for C, respectively. They can be used together to parse source code into the Abstract Syntax Tree(AST) and in turn used by YAX to build the cross reference database. However, writing Alex and Happy compatible parsing rules is time consuming and off the topic of this lecture. Therefore, I use an existing Haskell module, language-c [2] that leverages Alex and Happy, to translate the C source code into ASTs. YAX can then analyze the ASTs and extract symbols together with necessary information, including the location of the symbol and how the symbol used, to the database.

2 Design and Implementation

YAX takes a source code or a directory of source code tree as the input, parses the source code by language-c into the AST, traverses the AST to extract symbols, together with how the symbol is used, the file, column and row where the symbol is located, and finally adds them to the database. Not all of the symbols will be added to the database. For example, local variables are always considered as temporary variables only visible to a certain scope and thus is less meaningful to be indexed.

2.1 Parsing

language-c parses each source code file into an AST. The full definition of the C AST is pages long and thus is not included in this report but can be found in [3]. I present an example of a simple C source code shown in Figure 1 and its AST shown in Figure 2. Each box in Figure 2 is a node of the AST and each node is tagged with its location information in the source code. For example, the `Decl: g0` box which is the left child of `Root` has the location information `("example.c", 1, 5)`, which means the symbol `g0` is defined in the first row, fifth column of file `example.c`. Underscored symbols in Figure 1 and shadowed boxes in Figure 2 represent the symbol added to the database, while others are omitted.

More specifically, only the following symbols will be added to the database and indexed:

- declaration of global variables,

```
//example.c
int g0 = 0;
struct st1 { int f1; long f2; };
void func3(int arg) {
    int j, k, i;
    j = g2;
lbl:
    i = st2->f1;
    if (cond1)
        j = g0;
    func2(foo, 2, arg);
}
```

Figure 1: Example of C source code.

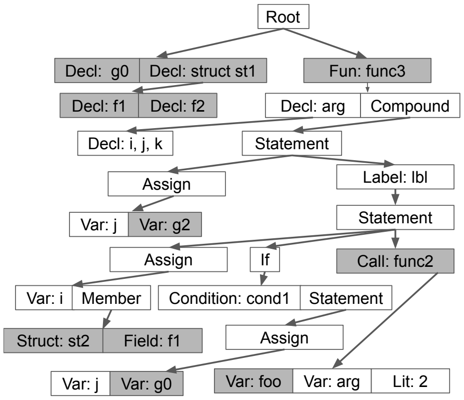

Figure 2: AST of the example of C source code.

- reference of global variables, either in the global scope or the local scope,
- definition of global composite data types such as `struct`,
- declaration of and reference to members of global composite data types,
- declaration of functions,
- labels.

Key words, local variables as well as components of other C extensions are omitted. Together with the location, a tag of how the symbol is used is also saved. YAX defines four types of symbol usage: variable or function declaration, function call, label and regular reference. Regular reference means the symbol is used in the way other than the first three. Since a function in C can also be used as a pointer variable, not all references to a function name is considered as a function call but only when the function is explicitly called by the C function call syntax. Calling a function pointer is usually determined at runtime and therefore is not considered as a function call, even if the name of the function pointer is the same as the function.

The information of a symbol is saved in a tuple of `(file::String, column::String, row:: String, entryType::EntryType)` where `EntryType` is a defined Haskell data type of how the symbol is used as introduced above. Then the tuple is saved to a map of which the key is the name of the symbol. Since a symbol usually appears more than once in a code base, the value is a list of tuples. The map uses the strict map module as opposed to the lazy map because for a cross referencer, the database should only be queried after it is fully build-up and the strict map has better performance than the lazy map.

2.2 Local Variables

A cross referencer generally does not index a local variable to avoid excess temporary variables flushing the database. To address this problem, YAX traverses the AST with two databases - a global database stores information that will be merged to the final result and a local database stores local variables visible to the current scope. More specifically, for a C program, the scope for a local variable is a compound and if a local variable has the same name of a global variable, the local variable shadows the global one. Therefore when parsing a compound, YAX takes the local database from its parent as an argument. When a local variable declaration is found, the variable is added to the local database and if a symbol is used in the following code and that symbol is in the local database, it will not be added to the global database. After a compound is parsed and returns to its parent compound, which means the life cycle for local variables in the compound is terminated, the local database is discarded and the parent can still keep its own local database unchanged.

```
1  void func(void){
2      int i;
3      {int k; func2(g,i,j,k);}
4      func2(g,i,j,k);
5  }
```

Figure 3: Example of local variables indexing.

Figure 3 shows an example of how symbols are indexed when local variables are involved. Underlined symbols are added to the database. In line 3, the function call to `func2` is indexed, together with variable `g` and `j`. `i` is declared as a local variable in the scope of the function and `k` is in the scope of the compound in line 3, so they will not be indexed. Similarly in line 4, `func2`, `g` and `j` is indexed but since `k` is no longer a local variable here, it will also be indexed.

2.3 Parallelism

Potentially, YAX can be paralleled in two manners: (1) parse an AST in parallel or (2) parse an AST sequentially and process multiple files in parallel to scale to a large code base. The first one is less practical because regardless the size of a target project, a single source code file should always have a reasonable size. The overhead introduced by parsing an AST in parallel can swamp the performance gained from parallelism.

Therefore YAX chooses to use a single thread to parse an AST and launches multiple threads when working on a large code base. Currently, YAX uses one spark for each AST. YAX takes the root directory of the source code as the input, recursively reads source code into a list of ByteString, one file per element and map the parsing function to each element in the list in parallel. The reading and mapping procedure are connected via a `pseq` function so all data are enforced to be read into the memory before the paralleled part running. The parsing function returns the reference database as a map and thus the main thread gets a list of maps when all source code are parsed. Then YAX unions the maps in the list to build the final result. When there is a key conflict when union-ing the map, i.e. a symbol appears in different files, the values, which is a list of symbol information, are concatenated to each other. Since the location of a symbol in the database, as well as the order of the information of the symbol do not affect the result of querying the database, the returned map is an monoid and therefore can also be unioned in parallel. However, based on my experiment, parallel fold and union the list of maps has minimal impact on the performance.

Various parallelism schema is tested to reach the best performance of YAX, including dynamically chunking, statically chunking and lazy stream with `parBuffer`. But the experiment shows different parallelism schema has barely no impact on the performance. Therefore a simple but more scalable `parList rpar` is used.

3 Performance Evaluation

Since YAX can only work on preprocessed C source code, to evaluate the performance of YAX and its parallelism implementation, I ran YAX on a synthesized code based. The code are randomly generated through a Python script outputting various C component, including global variable declaration, composite data type definition, function definition and different C statements such as assign, condition, function call, etc. The size of each file is also randomized so different sparks may have different workload. The synthesized code based has 16K C files and a total of 13M LOC.

To better demonstrate the performance for YAX on the real world project, the distribution of the size of files in the synthesized code based mimics the Linux kernel source code tree.

I ran YAX on a HP ML350 workstation, with a 10-core Intel Xeon 2640v4 CPU at 2.40GHz, hyperthreading off, 64GB of RAM and 1TB SSD. The performance is measured by the time from

YAX reading the source code into the memory until the database being build, not including the time for querying the database.

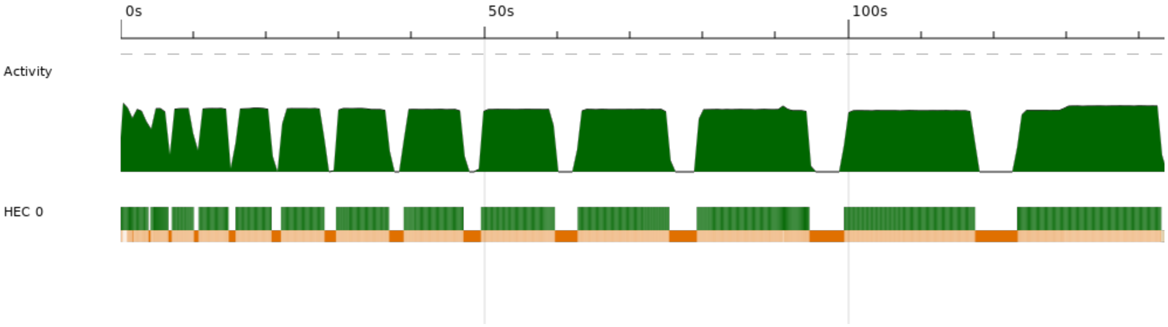

Figure 4: Threadscope of single threaded YAX.

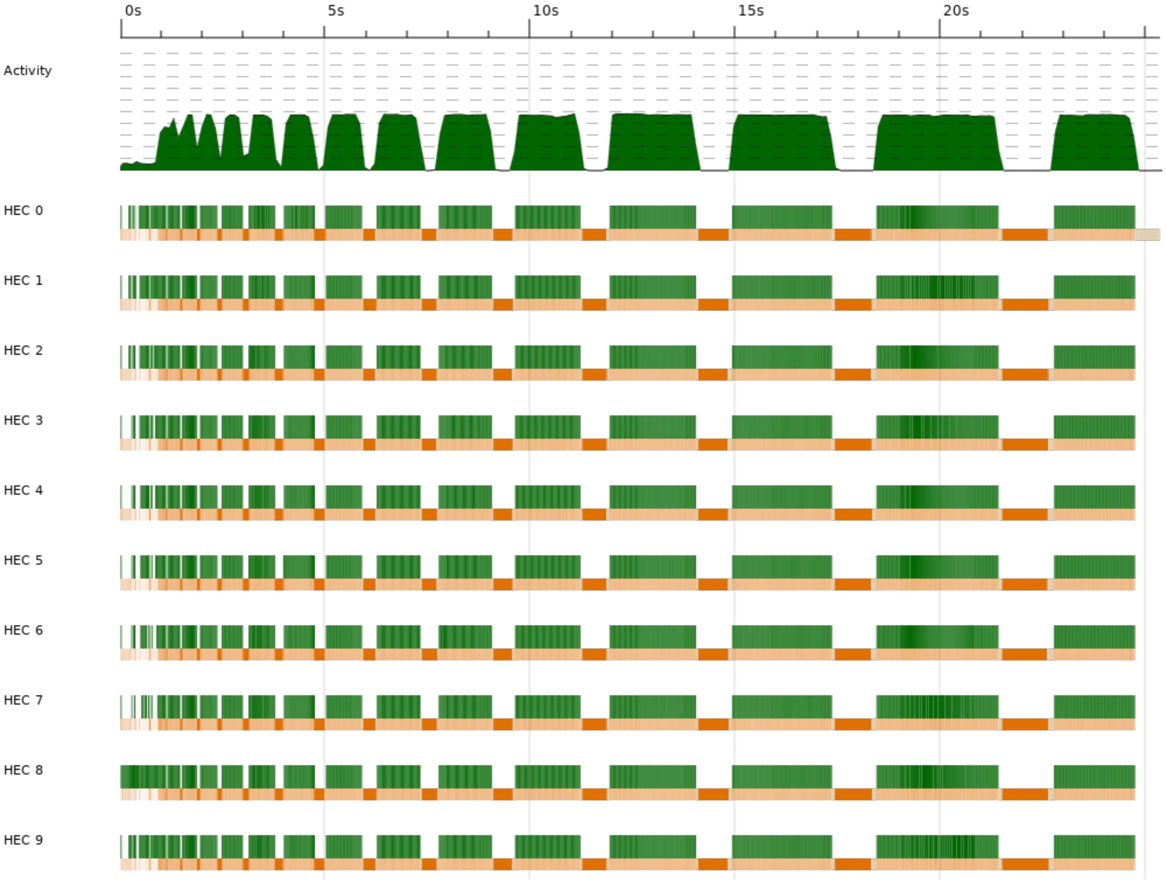

Figure 5: Threadscope of 10 threads YAX.

Figure 4 and Figure 5 show the Threadscope information of single threaded YAX and YAX with 10 paralleled threads, respectively. Figure 5 indicates the workload is evenly distributed into all 10 threads Evenly.

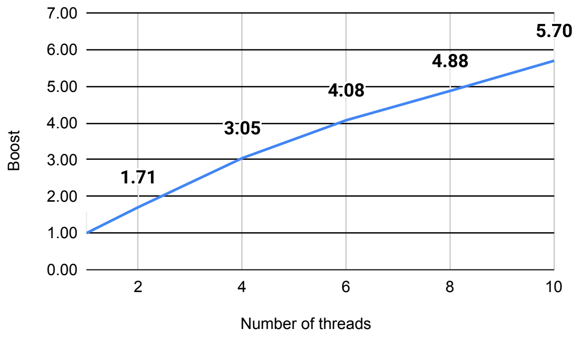

Figure 6: Performance boost for parallel YAX.

In an earlier version of YAX, the I/O action of reading source code into the memory is not enforced by the `pseq` and the paralleled performance is hit by the I/O when evaluating the input list. When `pseq` is used and all I/O are enforced to be done before the parallel evaluation, YAX gains a slightly performance improvement especially when more threads are used. Figure 6 shows the performance boost for parallel YAX from 2 threads to 10 threads with the baseline of sequential YAX. 2-threads is 1.71 times faster and 10-threads has a multiplier of 5.70. One of the major overhead for YAX is the Garbage Collection. Because YAX has to read all source code into a list and chunk that list for parallel evaluation, lots of memory will used to hold the entire code base and makes GC expensive. The Threadscope figure shows almost 50% of time is used for doing GC.

References

[1] *Cscope Home Page*. 2012. URL: `http://cscope.sourceforge.net/`.

[2] Joe Hermaszewski. *language-c: Analysis and generation of C code*. 2020. URL: `https://hackage.haskell.org/package/language-c-0.9.0.1`.

[3] ISO. *ISO C Standard 1999*. Tech. rep. 1999. URL: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[4] John R Levine et al. *Lex & yacc.* " O'Reilly Media, Inc.", 1992.

[5] Simon Marlow. *Alex: A lexical analyser generator for Haskell.* URL: `https://www.haskell.org/alex/`.

[6] Simon Marlow. *Happy: The Parser Generator for Haskell.* URL: `https://www.haskell.org/happy/`.

Appendix: List of Haskell Source Code of YAX

app/Main.hs

```haskell
module Main where
import ParseAST

import Language.C
import Language.C.System.GCC

import System.Environment
import System.Directory
import System.Exit
import qualified Data.Map.Strict as Map

import Control.Monad
import System.FilePath
import System.Posix.Files

import Control.Parallel
import Control.Parallel.Strategies

usage :: IO ()
usage = do
    prog <- getProgName
    die $ "Usage: " ++ prog ++ " <filename>|<directory> -p|-s"

-- Borrowed from https://stackoverflow.com/a/23822913
traverseDir :: FilePath -> (FilePath -> Bool) -> IO [FilePath]
traverseDir top exclude = do
  ds <- getDirectoryContents top
  paths <- forM (filter (not.exclude) ds) $ \d -> do
    let path = top </> d
    s <- getFileStatus path
    if isDirectory s
      then traverseDir path exclude
      else return [path]
  return (concat paths)

filesToStreamList :: [FilePath] -> IO [(InputStream, FilePath)]
filesToStreamList fs = sequence $ map (\f -> do
                                        s <- readInputStream f
                                        return (s, f))
                                        fs

-- Credit: https://stackoverflow.com/questions/19117922/parallel-folding-in-haskell/19119503
pfold :: (a -> a -> a) -> [a] -> a
pfold _ [x] = x
pfold mappend' xs  = (ys `par` zs) `pseq` (ys `mappend'` zs) where
    len = length xs
    (ys', zs') = splitAt (len `div` 2) xs
    ys = pfold mappend' ys'
```

```haskell
    zs = pfold mappend' zs'

doHandleStream :: (InputStream, FilePath) -> IdDB
doHandleStream (s, f) = case parseC s $ initPos f of
    Right tu -> case tu of
        CTranslUnit l _ -> parseTranslUnit Map.empty l
    Left _ -> Map.singleton "" [dummyEntry]
handleStreams :: [(InputStream, FilePath)] -> IdDB
handleStreams ss = foldl (Map.unionWith unionResult) Map.empty $
                      map doHandleStream ss
parHandleStreams :: [(InputStream, FilePath)] -> IdDB
parHandleStreams ss =
    pfold (Map.unionWith unionResult) $
        withStrategy (parList rpar) . map doHandleStream $ ss
unionResult :: [IdEntry] -> [IdEntry] -> [IdEntry]
unionResult new old = new ++ old

-- Simple query interface for the database
loopQuery :: IdDB -> IO ()
loopQuery db = do
    putStrLn "Search symbol:"
    sym <- getLine
    print $ Map.lookup sym db
    loopQuery db

main :: IO ()
main = do
    args <- getArgs
    case args of
        [f, c] -> handleFileDir f c
        _    -> usage
    where
    handleFileDir f c = do
        isF <- doesFileExist f
        if isF then readWithPrep f
        else  handleDir f c
    handleDir f c = do
        isD <- doesDirectoryExist f
        if isD then do
            files <- traverseDir f excludeDot
            contents <- pseq () (filesToStreamList files)
            case c of
                "-s" ->
                    loopQuery $ handleStreams contents
                "-p" ->
                    loopQuery $ parHandleStreams contents
                _ -> usage
        else die $ ("File does not exists: " ++) $ show f
    excludeDot "." = True
    excludeDot ".." = True
    excludeDot _ = False
```

src/ParseAST.hs

```haskell
module ParseAST where

import qualified Data.Map.Strict as Map
import Language.C
import Language.C.System.GCC


data EntryType = IdDecl | IdRef | IdCall | IdLabel | IdLocal deriving (Eq, Show)
-- Not meaningful, just in case of sorting for searching
instance Ord EntryType where
    IdLocal `compare` _         = EQ
    _       `compare` IdLocal   = EQ
    IdDecl  `compare` _         = LT
    IdRef   `compare` IdDecl    = GT
    IdRef   `compare` _         = LT
    IdCall  `compare` IdLabel   = LT
    IdCall  `compare` _         = GT
    IdLabel `compare` _         = GT

-- | IdEntryVal stores the information about a symbol:
--  (file, row, column, type)
type IdEntryVal = (String, Int, Int, EntryType)
-- | (ident, key)
-- type IdEntry = (String, IdEntryVal)
-- type IdDB = [IdEntry]

type IdEntry = IdEntryVal
type IdDB = Map.Map String [IdEntry]

-- dummy entry for local symbols to avoid unnecessary GC
dummyEntry :: IdEntry
dummyEntry = ("", 0, 0, IdLocal)

identToEntry :: Ident -> EntryType -> IdEntry
identToEntry ident entry_type =
    let id_file = case fileOfNode ident of
                    Nothing -> ""
                    Just p -> p in
    let id_pos = posOfNode $ nodeInfo ident in
    let id_row = posRow $ id_pos in
    let id_col = posColumn $ id_pos in
    (id_file, id_row, id_col, entry_type)

-- Just use linear search as the size of the local list should be handy
inLocalList :: IdDB -> String -> Bool
-- "true" and "false" are excluded since they are widely used as keywords
inLocalList _ "true" = True
inLocalList _ "false" = True
inLocalList db id_name =  case Map.lookup id_name db of
    Just _ -> True
```

```haskell
    _ -> False

addEntry :: Ident -> EntryType -> IdDB -> IdDB
addEntry ident IdLocal gl =
    let id_name = (identToString ident) in
    Map.insert id_name [dummyEntry] gl
addEntry ident t gl =
    let id_name = (identToString ident) in
    let id_entry = identToEntry ident t in
    Map.insertWith mergeEntry id_name [id_entry] gl
    where
    mergeEntry :: [IdEntry] -> [IdEntry] -> [IdEntry]
    mergeEntry [n] o = n : o
    mergeEntry _ o = o -- we know new_value must be a singleton list

parseDeclList :: IdDB -> IdDB -> [(Maybe (CDeclarator a0), b0, c0)] ->
    (IdDB, IdDB)
parseDeclList gl ll [] = (gl, ll)
parseDeclList gl ll ((cDeclr, _, _):xs) = case cDeclr of
    Nothing -> (gl, ll)
    Just (CDeclr (Just ident) _ _ _ _) ->
        case null ll of
            True -> let gl' = addEntry ident IdDecl gl in parseDeclList gl' ll xs
            _ -> let ll' = addEntry ident IdLocal ll in parseDeclList gl ll' xs
    _ -> (gl, ll)

parseCSU :: IdDB -> IdDB -> CStructureUnion a -> (IdDB, IdDB)
parseCSU gl ll (CStruct _ mident mdecl _ _) = case mident of
    Just ident ->
        -- struct variable declarations are always indexed
        let gl' = addEntry ident IdDecl gl in
        case mdecl of
            Just declL -> (parseStructDeclList gl' declL, ll)
            _ -> (gl', ll)
    _ -> (gl, ll)
    where
        -- struct fields are always indexed
        parseStructDeclList gl' [] = gl'
        parseStructDeclList gl' (x:xs) =
            let (dl, _) = (parseDecl gl' Map.empty x) in parseStructDeclList dl xs

parseCType :: IdDB -> IdDB -> [CDeclarationSpecifier a] ->
    [(Maybe (CDeclarator a0), b0, c0)] -> (IdDB, IdDB)
parseCType gl ll [] _ = (gl, ll)
parseCType gl ll (cType:_) declList = case cType of
    -- struct or union
    CTypeSpec (CSUType (csu) _) ->
        let (gl', ll') = parseCSU gl ll csu in
        case declList of
            [] -> (gl', ll')
            _ -> parseDeclList gl' ll' declList
```

```haskell
    -- other types
    _ -> parseDeclList gl ll declList

--parseDecl :: CDeclaration a -> IdEntry
parseDecl :: IdDB -> IdDB -> (CDeclaration a) ->
    (IdDB, IdDB)
parseDecl gl ll (CDecl cTypeList declrList _) =
    parseCType gl ll cTypeList declrList
parseDecl gl ll _ = (gl, ll)

-- expr and stmt won't introduce new symbols so local DB is always discarded
parseExprList :: IdDB -> IdDB -> [CExpression a] -> EntryType -> IdDB
parseExprList gl _ [] _ = gl
parseExprList gl ll (expr:xs) id_type =
    let gl' = parseExpr gl ll expr id_type in
    parseExprList gl' ll xs id_type

parseExpr :: IdDB -> IdDB -> (CExpression a) -> EntryType -> IdDB
parseExpr gl ll cexpr id_type = case cexpr of
    CComma exprList _ -> parseExprList gl ll exprList IdRef
    CAssign _ expr1 expr2 _ ->
        parseExpr2 gl ll (expr1, IdRef) (expr2, IdRef)
    CCond expr1 Nothing expr2 _ ->
        parseExpr2 gl ll (expr1, IdRef) (expr2, IdRef)
    CCond expr1 (Just expr2) expr3 _ ->
        parseExpr3 gl ll (expr1, IdRef) (expr2, IdRef) (expr3, IdRef)
    CBinary _ expr1 expr2 _ ->
        parseExpr2 gl ll (expr1, IdRef) (expr2, IdRef)
    CCast _ expr _ -> parseExpr gl ll expr IdRef
    CUnary _ expr _ -> parseExpr gl ll expr IdRef
    CSizeofExpr expr _ -> parseExpr gl ll expr IdRef
    CIndex expr1 expr2 _ ->
        parseExpr2 gl ll (expr1, IdRef) (expr2, IdRef)
    CCall expr exprList _ ->
        -- callee must be defined so ll can't be changed
        let gl' = parseExpr gl ll expr IdCall in
        parseExprList gl' ll exprList IdRef
    CMember struct field _ _ -> -- field :: Ident is always indexed
        let gl' = parseExpr gl ll struct IdRef in
        addEntry field IdRef gl'
    CVar ident _ ->
        -- if the ident is a local variable, just discard it
        if inLocalList ll (identToString ident)
            then gl
            else addEntry ident id_type gl
    _ -> gl

parseExpr2 :: IdDB -> IdDB -> (CExpression a, EntryType) ->
    (CExpression a, EntryType) -> IdDB
parseExpr2 gl ll (expr1, t1) (expr2, t2) =
    let gl' = parseExpr gl ll expr1 t1 in
```

```haskell
    parseExpr gl' ll expr2 t2

parseExpr3 :: IdDB -> IdDB -> (CExpression a, EntryType) ->
    (CExpression a, EntryType) -> (CExpression a, EntryType) -> IdDB
parseExpr3 gl ll exprt1 exprt2 (expr3, t3)  =
    let gl' = parseExpr2 gl ll exprt1 exprt2 in
    parseExpr gl' ll expr3 t3

parseStmt :: IdDB -> IdDB -> (CStatement a) -> IdDB
parseStmt gl ll cstmt = case cstmt of
    CLabel label stmt _ _ ->
        let gl' = addEntry label IdLabel gl in
        parseStmt gl' ll stmt
    CCase expr stmt _ ->
        let gl' = parseExpr gl ll expr IdRef in
        parseStmt gl' ll stmt
    CCases expr1 expr2 stmt _ ->
        let gl' = parseExpr2 gl ll (expr1, IdRef) (expr2, IdRef) in
        parseStmt gl' ll stmt
    CDefault stmt _ ->
        parseStmt gl ll stmt
    CExpr (Just expr) _ ->
        parseExpr gl ll expr IdRef
    CCompound label compoundItems _ ->
        parseCompound gl ll label compoundItems
    CIf expr stmt Nothing _ ->
        let gl' = parseExpr gl ll expr IdRef in
        parseStmt gl' ll stmt
    CIf expr stmt1 (Just stmt2) _ ->
        let gl' = parseExpr gl ll expr IdRef in
        let gl'' = parseStmt gl' ll stmt1 in
        parseStmt gl'' ll stmt2
    CSwitch expr stmt _ ->
        let gl' = parseExpr gl ll expr IdRef in
        parseStmt gl' ll stmt
    CWhile expr stmt _ _ ->
        let gl' = parseExpr gl ll expr IdRef in
        parseStmt gl' ll stmt
    CFor _ _ _ _ _ -> parseCFor cstmt
    CGoto label _ ->
        addEntry label IdLabel gl
    CReturn (Just expr) _ ->
        parseExpr gl ll expr IdRef
    _ -> gl
    where
        mParseExpr gl' ll' mexpr = case mexpr of
            Nothing -> Just gl'
            Just expr -> Just (parseExpr gl' ll' expr IdRef)
        parseCFor (CFor (Left mexpr1) (mexpr2) (mexpr3) stmt _) =
             case mParseExpr gl ll mexpr1 >>= \gl1 ->
                  (mParseExpr gl1 ll) mexpr2 >>= \gl2 ->
```

```
                  (mParseExpr gl2 ll) mexpr3 of
                Nothing -> gl
                Just gl3 -> parseStmt gl3 ll stmt
        parseCFor (CFor _ _ _ _ _) = gl
        parseCFor _ = gl

-- C code compound, gl is global symbol DB, ll is local symbol DB
-- Updates to a local symbol in a compound is discarded when the compound
-- is parsed
parseCompound :: IdDB -> IdDB -> [Ident] -> [CCompoundBlockItem a]
    -> IdDB
parseCompound gl _ _ [] = gl -- end of parsing, ll is discarded
parseCompound gl ll labels (blockItem:xs) = case blockItem of
    CBlockStmt stmt -> -- Stmt won't introduce new symbols
        let gl' = parseStmt gl ll stmt in
        parseCompound gl' ll labels xs
    CBlockDecl decl ->
        let (gl', ll') = parseDecl gl ll decl in
        parseCompound gl' ll' labels xs
    CNestedFunDef (_) -> gl -- GNU C nested function is not supported

parseFunDeclr :: IdDB -> (CDerivedDeclarator a) -> IdDB
parseFunDeclr ll (CFunDeclr (Left _) _ _ ) = ll -- old-style function declaration is not supported
parseFunDeclr ll (CFunDeclr (Right (cDecls, _)) _ _) =
    forEachCDecl ll cDecls
    where
    forEachCDecl :: IdDB -> [CDeclaration a] -> IdDB
    forEachCDecl rl [] = rl
    forEachCDecl rl (cDecl:xs) =
        let (new_rl, _) = (parseDecl rl Map.empty cDecl) in forEachCDecl new_rl xs
parseFunDeclr ll _ = ll

-- Function definitions
parseDef :: IdDB -> (CFunctionDef a) -> IdDB
parseDef gl (CFunDef _ cDeclr _ cCompound _) = case cDeclr of
    (CDeclr (Just ident) [cFunDeclr] _ _ _) ->
        let gl' = addEntry ident IdDecl gl in -- add function name to global list
        let ll = parseFunDeclr Map.empty cFunDeclr in -- add function arguments to local list
        case cCompound of
            (CCompound labels items _) ->
                parseCompound gl' ll labels items
            _ -> gl'
    _ -> gl

parseTranslUnit :: IdDB -> [CExternalDeclaration a] -> IdDB
parseTranslUnit gl [] = gl
parseTranslUnit gl (x:xs) = case x of
    CDeclExt decl -> let (dl, _) = (parseDecl gl Map.empty decl) in parseTranslUnit dl xs
    CFDefExt def -> let dl = parseDef gl def in parseTranslUnit dl xs
    _ -> gl
```

```haskell
parseAST :: CTranslationUnit a -> IdDB
parseAST (CTranslUnit l _) = parseTranslUnit Map.empty l

readWithPrep :: String -> IO ()
readWithPrep input_file = do
    ast <- errorOnLeftM "Parse Error" $
        parseCFile (newGCC "gcc") Nothing [""] input_file
    mapM_ print $ parseAST ast

errorOnLeft :: (Show a) => String -> (Either a b) -> IO b
errorOnLeft msg = either (error . ((msg ++ ": ")++).show) return
errorOnLeftM :: (Show a) => String -> IO (Either a b) -> IO b
errorOnLeftM msg action = action >>= errorOnLeft msg
```