Оглавление

1. Введение

В настоящей работе в рамках курса “конструирование компиляторов” реализуется фронтэнд компилятора упрощённой версии языка C--. В работе приводится использованная грамматика языка, реализуется лексический анализатор, синтаксический анализатор. Проводится поиск лексических, синтаксических и смысловых ошибок, таких как типовые и другие. В результате работы создаётся абстрактное синтаксическое дерево, соответствующее входной программе.

2. Выбор платформы

Для реализации фронтэнда компилятора решено было использовать язык Haskell. Был использован дистрибутив MinGHC 7.10.1[1], содержащий компилятор GHC 7.10.1[2], систему сборки и управления пакетами и библиотеками Haskell Cabal 1.22.4.0[3] и пакет утилит MSYS[4].

Также использовались генератор лексических анализаторов Alex 3.1.4[5] и генератор обобщённых LR-парсеров Happy 1.18.5[6]. Для сериализации получившегося дерева в формате JSON использовался пакет Aeson 0.6.1.0[7].

Выбор языка был обусловлен его строгой типизацией и удобством отладки благодаря жёстко отслеживаемым побочным эффектам функций, а также наличием большого количества библиотек.

3. Описание языка

За грамматику языка была принята упрощённая версия[8] грамматики языка С--, дополненная строковыми литералами. Грамматика имеет следующий вид:

1. *program* → *declaration-list*
2. *declaration-list* → *declaration* { *declaration* }
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID** [**[NUM]**]$_+$ **;**
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID** **(** *params*) *compound-stmt*
7. *params* → **void** | *param-list*
8. *param-list* → *param* { **,** *param* }
9. *param* → *type-specifier* **ID** [**[]**]$_+$
10. *compound-stmt* → **{** *local-declarations statement-list* **}**
11. *local-declarations* → *var-declarations*
12. *statement-list* → *statement*
13. *statement* → *expression-stmt*
    | *compound-stmt*
    | *selection-stmt*
    | *iteration-stmt*
    | *assignment-stmt*
    | *return-stmt*
    | *read-stmt*
    | *write-stmt*
14. *expression-stmt* → *expression* **;** | **;**
15. *selection-stmt* → **if** **(** *expression*) *statement* [**else** *statement*]$_+$
16. *iteration-stmt* → **while** **(** *expression*) *statement*
17. *return-stmt* → **return** [*expression*]$_+$ **;**
18. *read-stmt* → **read** *variable* **;**
19. *write-stmt* → **write** *expression* **;**
20. *expression* → { *var* **=** } *simple-expression*
21. *var* → **ID** [**[** *expression* **]**]$_+$

22. *simple-expression* → *additive-expression* [*relop additive-expression* $]_+$
23. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
24. *additive-expression* → *term* { *addop term* }
25. *addop* → **+** | **−**
26. *term* → *factor* { *multop factor* }
27. *multop* → $*$ | **/**
28. *factor* → **(** *expression* **)** | **NUM** | **ARR** | *var* | *call*
29. *call* → **ID** **(** *args* **)**
30. *args* → [*arg-list* $]_+$
31. *arg-list* → *expression* { **,** *expression* }

В грамматике также используются следующие регулярные выражения:

1. **ID** = $[a - z]+$
2. **NUM** = $[0 - 9]+$
3. **ARR** = "$PRINTABLE+$"
4. **PRINTABLE** -- соответствует любому печатаемому символу

Таблица 1: Смысловые значения выходных токенов лексера

Токен	Значение
Symbol	Управляющий символ либо ключевое слово
Num	Числовая константа
Array	Строка, преобразованная к массиву целых чисел
Name	имя переменной или функции

```
data Token =
    Symbol String |
    Array [Int]   |
    Num Int       |
    Name String
    deriving (Eq, Show)
data Posed a = Posed (Int,Int) a
```

Листинг 1: Выходные типы данных лексера

4. Лексический анализ

4.1. Входные и выходные структуры данных

На этапе лексического анализа происходит преобразование разбираемого кода в последовательность токенов, которые в дальнейшем будут обрабатываться в рамках синтаксического анализа и поиска ошибок.

На вход лексического анализатора подаётся строка символов, содержащая всё содержимое файла с исходным кодом. Выходом лексического анализатора является строка токенов типа, определённого в листинге на рис. 1. Смысловые значения токенов даны в таблице 1.

4.2. Обнаруживаемые ошибки

4.3. Конфигурационный файл Alex

Конфигурационный файл Alex содержит регулярные выражения, соответствующие различным токенам, набор правил для их преобразования и описание выходных структур данных. Также в нём содержатся вспомогательные функции, используемые в правилах преобразования токенов. Всё содержимое конфигурационного файла приведено в листинге в приложении 1. Haskel-модуль генерируется из конфигурационного файла с помощью утилиты командной строки *alex*.

4.4. Реализация лексического анализа

При анализе

5. Синтаксический анализ

6. Поиск ошибок

7. Список литературы

[1] FP Complete. *Minimum GHC Installer.* 14 мая 2015. URL: `https://github.com/fpco/minghc/releases` (дата обр. 18.05.2015).

[2] The Glasgow Haskell Team. *The Glasgow Haskell Compiler 7.10.1.* 27 марта 2015. URL: `https://www.haskell.org/ghc/download_ghc_7_10_1` (дата обр. 18.05.2015).

[3] Lemmih и др. *The cabal-install package 1.22.4.0.* 5 мая 2015. URL: `http://hackage.haskell.org/package/cabal-install` (дата обр. 18.05.2015).

[4] MinGW collective. *Msys -- GNU utilities collection.* 5 мая 2015. URL: `http://www.mingw.org/wiki/MSYS` (дата обр. 18.05.2015).

[5] Chris Dorian, Isaac Jones и Simon Marlow. *Alex Release 3.1.4.* 6 янв. 2015. URL: `https://github.com/simonmar/alex/releases/tag/3.1.4` (дата обр. 19.05.2015).

[6] Andy Gill, Simon Marlow и др. *Happy -- The Parser Generator for Haskell.* 17 июня 2010. URL: `https://www.haskell.org/happy` (дата обр. 19.05.2015).

[7] Bryan O'Sullivan. *Aeson Release 0.8.1.0.* 11 мая 2015. URL: `https://github.com/bos/aeson/releases/tag/0.8.1.0` (дата обр. 19.05.2015).

[8] Bob Broeg. *Extended BNF Grammar for C Minus.* 2014. URL: `http://www.wou.edu/~broegb/CS447/C_Minus_EBNF_Revised5.pdf` (дата обр. 19.05.2015).

Приложение 1

Конфигурационный файл Alex

```
{
{-# Language LambdaCase #-}
module Cmm_alex (Token(..), Posed(..), alexScanTokens) where
}

%wrapper "posn"

-- $white
 $letter  = [a-z A-Z]
 $digit  = 0-9

--{- name literal -}
@name = $letter+

--{- integer  literal  -}
@int  = \-? $digit+

--{- char  literal  -}
@escapeseq = 0 | a  | b  | f  | n  | r  | t  | \\ | \' | \" | \?
@escapechar = \\ @escapeseq
@char = $printable | @escapechar
@character = \' @char \'

--{- string  literal  -}
@string = \" $printable* \"

--{- symbol literal -}
@symbol = "[" | "]" | "{" | "}" | "(" | ")" | ";" | ","
          | "==" | "=" | "<=" | "<" | ">=" | ">" | "!="
          | "+" | "-" | "*" | "/"

--{- reserved word literal -}
@reserved = "int" | "void"   | "if"   | "else" | "while" | "return" | "read" | "write"

--{- comments -}
```

```
@comment = "//" .*
    | "/*" (. | \n)* "*/"

--------------------------------------------------------------
tokens :-

    $white+         ;
    @comment        ;
    @reserved       {pose Symbol}
    @name           {pose Name}
    @string         {pose (\s -> Array $ map fromEnum $ sanstr s)}
    @character      {pose (\s -> Num $ fromEnum $ (read s :: Char))}
    @int            {pose (\s -> Num $ read s)}
    @symbol         {pose Symbol}

{
data Token =
    Symbol String |
    Array [Int]   |
    Num Int       |
    Name String
    deriving (Eq, Show)

data Posed a = Posed (Int,Int) a
instance (Eq a => Eq (Posed a)) where
    (==) (Posed _ a) (Posed _ b) = a == b
instance (Show a => Show (Posed a)) where
    show (Posed p a) = concat [show p, "~", show a]

pose::(String->a)->AlexPosn->String->Posed a
pose constr (AlexPn abs line col) s = Posed (line, col) (constr s)

sanstr ('"':tail) = sanstrlast tail
sanstrlast = \case
    [] -> []
    a:[] | a == '"' ->'\0':[]
    a:b -> a:sanstrlast b

}
```

Приложение 2

Конфигурационный файл Happy

```
{
module Cmm_happy(
    happyParseToTree,
    Reference,
    TExpression (..) ,
    TDeclaration (..) ,
    TStatement(..),
    )where
import Cmm_alex
}

%name happyParseToTree DeclarationList
%tokentype { Posed Token}
%error { parseError }

%token
    array        {Posed _ (Array _)}
    num          {Posed _ (Num _)}
    name         {Posed _ (Name _)}
    "["          {Posed _ (Symbol "[")}
    "]"          {Posed _ (Symbol "]")}
    "{"          {Posed _ (Symbol "{")}
    "}"          {Posed _ (Symbol "}")}
    "("          {Posed _ (Symbol "(")}
    ")"          {Posed _ (Symbol ")")}
    ";"          {Posed _ (Symbol ";")}
    ","          {Posed _ (Symbol ",")}
    "=="         {Posed _ (Symbol "==")}
    "="          {Posed _ (Symbol "=")}
    "<="         {Posed _ (Symbol "<=")}
    "<"          {Posed _ (Symbol "<")}
    ">="         {Posed _ (Symbol ">=")}
    ">"          {Posed _ (Symbol ">")}
    "!="         {Posed _ (Symbol "!=")}
    "+"          {Posed _ (Symbol "+")}
    "-"          {Posed _ (Symbol "-")}
    "*"         {Posed _ (Symbol "*")}
    "/"         {Posed _ (Symbol "/")}
    "int"       {Posed _ (Symbol "int")}
    "void"      {Posed _ (Symbol "void")}
    "if"        {Posed _ (Symbol "if")}
    "else"      {Posed _ (Symbol "else")}
    "while"     {Posed _ (Symbol "while")}
    "return"    {Posed _ (Symbol "return")}
    "read"      {Posed _ (Symbol "read")}
    "write"     {Posed _ (Symbol "write")}

%nonassoc ")"
%nonassoc "else"

    %%

DeclarationList  ::{ [TDeclaration] }
DeclarationList  : Declaration                              { [$1] }
                 | DeclarationList  Declaration             { $1 ++ [$2] }

Declaration      ::{ TDeclaration }
Declaration      : VarDeclaration                           { $1 }
                 | FunDeclaration                           { $1 }

VarDeclaration   ::{ TDeclaration }
VarDeclaration   : "int" name ";"                           { let (Posed p (Name n)) =
    $2
                                                              in ( Intdecl (Posed p n)) }
                 | "int" name "[" num "]" ";"               { let (Posed p (Name n)) =
                     $2;
                                                                  (Posed p2 (Num i)) =
                                                                      $4
                                                              in (Arrdecl (Posed p n) (
                                                                  Posed p2 i)) }

FunDeclaration   ::{ TDeclaration }
FunDeclaration   : "int" name "(" Params ")" CompoundStmt { let (Posed p (Name n)
    ) = $2
                                                                  in (Fundecl (Posed p n
                                                                      ) $4 $6) }
                 | "void" name "(" Params ")" CompoundStmt { let (Posed p (Name n
                     )) = $2
```

```
                                                          in (Procdecl (Posed p
                                                              n) $4 $6) }

Params          ::{ [TDeclaration] }
Params          : "void"                          { [] }
                | ParamList                       { $1 }
ParamList       ::{ [TDeclaration] }
ParamList       : Param                           { [$1] }
                | ParamList "," Param             { $1 ++ [$3] }
Param           ::{ TDeclaration }
Param           : "int" name                      { let (Posed p (Name n)) =
    $2
                                                    in Intdecl (Posed p n) }
                | "int" name "[" "]"              { let (Posed p (Name n)) =
                    $2
                                                    in Arrdecl (Posed p n) (
                                                        Posed p 0) }
CompoundStmt    ::{ TStatement }
CompoundStmt    : "{" LocalDeclarations StatementList "}"   { CompSta $2 $3 }
---------
 LocalDeclarations   ::{ [TDeclaration] }
 LocalDeclarations   : {--}                       { [] }
                     | LocalDeclarations VarDeclaration  { $1 ++ [$2] }

StatementList   ::{ [TStatement] }
StatementList   : {--}                            { [] }
                | StatementList Statement         { $1 ++ [$2] }

Statement       ::{ TStatement }
Statement       : ExpressionStmt                  { $1 }
                | CompoundStmt                    { $1 }
                | SelectionStmt                   { $1 }
                | IterationStmt                   { $1 }
                | ReturnStmt                      { $1 }
                | ReadStmt                        { $1 }
                | WriteStmt                       { $1 }

ExpressionStmt  ::{ TStatement }
ExpressionStmt  : Expression ";"                  { ExpSta $1 }
                | ";"                             { EmpSta }

SelectionStmt   ::{ TStatement }
SelectionStmt    : "if" "(" Expression ")" Statement                    { SelSta $3
    $5 Nothing }
                 | "if" "(" Expression ")" Statement "else" Statement   { SelSta $3
                     $5 (Just $7 )}

IterationStmt    ::{ TStatement }
IterationStmt    : "while" "(" Expression ")" Statement { IterSta $3 $5 }

ReturnStmt       ::{ TStatement }
ReturnStmt       : "return" ";"               { let (Posed pos _) = $1
                                                 in RetSta pos Nothing}
                 | "return" Expression ";"    { let (Posed pos _) = $1
                                                 in RetSta pos (Just $2) }

ReadStmt         ::{ TStatement }
ReadStmt         : "read" Var ";"             { ReadSta $2 }

WriteStmt        ::{ TStatement }
WriteStmt        : "write" Expression ";"     { let (Posed p (Symbol "write")) = $1
                                                 in ExpSta $ CallEx (Posed p ">>") [$2
                                                     ] }
----------------------
Expression       ::{ TExpression }
Expression       : ExpressionHead SimpleExpression  { ComplEx $1 $2 }

ExpressionHead   ::{ [Reference] }
ExpressionHead   : {--}                       { [] }
                 | ExpressionHead Var "="     { $1 ++ [$2] }

Var              ::{ Reference }
Var              : name                       { let (Posed p (Name n)) = $1
                                                 in (Posed p n, Nothing) }
                 | name "[" Expression "]"    { let (Posed p (Name n)) = $1
                                                 in (Posed p n, Just $3) }

SimpleExpression ::{ TExpression }
SimpleExpression :  AdditiveExpression                            { $1 }
                 |  AdditiveExpression Relop AdditiveExpression   { let (Posed p (
                     Symbol n)) = $2
                                                                    in CallEx (Posed p
                                                                        n) [$1, $3] }

Relop            ::{ Posed Token }
Relop              : "<="             { $1 }
                   | "<"              { $1 }
                   | ">"              { $1 }
                   | ">="             { $1 }
                   | "=="             { $1 }
                   | "!="             { $1 }

 AdditiveExpression ::{  TExpression }
 AdditiveExpression   :  Term                               { $1 }
                      |  AdditiveExpression  Addop Term     { let  (Posed p (Symbol n))
                           = $2
                                                              in CallEx  (Posed p n)  [$1,
                                                                   $3] }

Addop              ::{  Posed Token }
Addop              : "+"              { $1 }
                   | "−"              { $1 }

Term               ::{  TExpression }
Term               :  Factor             { $1 }
                   |  Term Multop Factor { let (Posed p (Symbol n)) = $2
                                           in CallEx  (Posed p n)  [$1,  $3] }
Multop             ::{  Posed Token }
Multop             : "*"              { $1 }
                   | "/"              { $1 }

Factor             ::{  TExpression }
Factor             :  "("  Expression  ")" { $2 }
                   |  num                { let  (Posed p (Num n)) = $1
                                           in NumLiteral $ Posed p n }
                   |  array              { let  (Posed p (Array n)) = $1
                                           in  StringLiteral  $ Posed p n }
                   |  Var                {  Retrieval  $1 }
                   |   Call              { $1 }

 Call              ::{  TExpression }
 Call              :  name "(" Args ")"  { let  (Posed p (Name n)) = $1
                                           in CallEx  (Posed p n)  $3 }

Args               ::{  [TExpression] }
Args               :  {−−}                {  []  }
                   |  Expression          {  [$1]  }
                   |  Args "," Expression { $1 ++ [$3] }
```

```

{
parseError  :: Show b => [Posed b] -> a
parseError  ((Posed p t): rst )  = error $ "Parse error at " ++ (show p) ++ ":
    unexpected symbol " ++ (show t)

type Reference = (Posed String, Maybe (TExpression))
data TExpression =
    ComplEx [Reference] TExpression
    | CallEx  (Posed String) [TExpression]
    |  Retrieval  Reference
    |  StringLiteral   (Posed [Int])
    | NumLiteral (Posed Int)
    deriving (Show, Eq)
data TStatement =
    CompSta [TDeclaration] [TStatement]
    | SelSta  TExpression TStatement (Maybe TStatement)
    |  IterSta  TExpression TStatement
    | RetSta (Int,Int) (Maybe TExpression)
    | ReadSta Reference
    | ExpSta TExpression
    | EmpSta
    deriving (Show, Eq)
data TDeclaration =
     Intdecl  (Posed String)
    |  Arrdecl  (Posed String) (Posed Int)
    | Fundecl (Posed String) [TDeclaration]  TStatement
    | Procdecl (Posed String) [TDeclaration]  TStatement
    deriving (Show, Eq)
}
```

Приложение 3

Модуль проверки логических ошибок

```haskell
{-# LANGUAGE MultiWayIf #-}
{-# LANGUAGE LambdaCase #-}
module Checker(
    check
    )where
import Cmm_alex
import Cmm_happy
import Dictutils
import Data.Either

defaultFunctions ::[ Namedecl]
defaultFunctions  =
    [( "<=", (Function [Boolean, Number, Number],(0,0)))
    ,( "<", (Function [Boolean, Number, Number],(0,0)))
    ,( ">", (Function [Boolean, Number, Number],(0,0)))
    ,( ">=", (Function [Boolean, Number, Number],(0,0)))
    ,( "==", (Function [Boolean, Number, Number],(0,0)))
    ,( "!=", (Function [Boolean, Number, Number],(0,0)))

    ,( "+", (Function [Number, Number, Number],(0,0)))
    ,( "-", (Function [Number, Number, Number],(0,0)))
    ,( "*", (Function [Number, Number, Number],(0,0)))
    ,( "/", (Function [Number, Number, Number],(0,0)))

    ,( ">>", (Function [Void, Number],(0,0)))
    ,( "<<", (Function [Void, Reference],(0,0) ))]

data Typ = Boolean | Number | Reference | Void | Function [Typ] |  Any deriving (Show)
instance (Eq Typ) where
    a == b =
        case (a,b) of
            (Any,_) -> True
            (_, Any) -> True
            (Boolean,Boolean)->True
            (Number,Number)->True
            (Reference,Reference)->True
            (Void,Void)->True
            (Function a, Function b) -> a == b
            _ -> False

type Pos = (Int, Int)

type Namedecl = (String, (Typ, Pos))
type Error = (Pos, String)

check ::[TDeclaration]->[Error]
check decls = checkTopLevel decls defaultFunctions

checkTopLevel ::[TDeclaration]->[Namedecl]->[Error]
checkTopLevel [] _ = []
checkTopLevel (d:ecl) prevdecls =
    case d of
        Intdecl (Posed pos nam) ->
            if | haskey nam prevdecls  ->
                (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
               | otherwise ->
                checkTopLevel ecl ((nam, (Number, pos)):prevdecls)
        Arrdecl (Posed posn nam) (Posed poss siz) ->
            if | haskey nam prevdecls ->
                (posn, "Redefinition of variable"):checkTopLevel ecl prevdecls
               | siz <= 0 ->
                (poss, "Nonpositive array size"):checkTopLevel ecl prevdecls
               | otherwise ->
                checkTopLevel ecl ((nam, (Reference, posn)): prevdecls)
        Fundecl (Posed pos nam) paramsraw body ->
            if | haskey nam prevdecls ->
                (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
               | otherwise ->
                let (pardecl, parerrors) = morphdecl paramsraw prevdecls
                    bodyerrors = checkstat (pardecl++prevdecls) body Number
                    fntype = Number:(map (\(_, (t, _))-> t) pardecl)
                    fdcl = (nam, (Function fntype, pos))
                in bodyerrors ++ parerrors ++
                    checkTopLevel ecl (fdcl : prevdecls)
        Procdecl (Posed pos nam) paramsraw body ->
            if | haskey nam prevdecls ->
                (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
               | otherwise ->
```

```
                let (pardecl, parerrors) = morphdecl paramsraw prevdecls
                    bodyerrors = checkstat (pardecl++prevdecls) body Void
                    fntype = Void:(map (\(_, (t, _))-> t) pardecl)
                    fdcl = (nam, (Function fntype, pos))
                in bodyerrors ++ parerrors ++
                    checkTopLevel ecl (fdcl : prevdecls)

checkstat ::[Namedecl]->TStatement->Typ->[Error]
checkstat prevdecls statement rettyp =
    case statement of
        CompSta decls nested ->
            let (locdecl, declerr) = morphdecl decls prevdecls
                tail = concat $ map (\st -> checkstat (locdecl ++ prevdecls) st rettyp) nested
            in (declerr ++ tail)
        SelSta bexpr thn mels ->
            checkexpr Boolean prevdecls bexpr
            ++ checkstat prevdecls thn rettyp
            ++ case mels of Just els -> checkstat prevdecls els rettyp; Nothing -> []
        IterSta bexpr whl ->
            checkexpr Boolean prevdecls bexpr
            ++ checkstat prevdecls whl rettyp
        RetSta p Nothing ->
            if rettyp == Void then [] else [(p, "Expected empty return")]
        RetSta p (Just rexpr) ->
            if rettyp == Void then [(p, "Expected expression")] else checkexpr rettyp prevdecls rexpr
        ReadSta (Posed pos nam, Nothing) ->
            case lookup nam prevdecls of
                Just (Number, _) -> []
                Just _ -> [(pos, "Type mismatch: expected integer variable")]
                Nothing -> [(pos, "Unknown variable")]
        ReadSta (Posed pos nam, Just iexpr) ->
            case lookup nam prevdecls of
                Just (Number, _) -> [(pos, "Type mismatch: expected array variable")]
                Just (Reference, _) -> checkexpr Number prevdecls iexpr
                Nothing -> [(pos, "Unknown variable")]
        ExpSta sexpr -> checkexpr Any prevdecls sexpr
        EmpSta -> []

checkexpr :: Typ->[Namedecl]->TExpression->[Error]
checkexpr typ decls expr =
    case expr of
        ComplEx assigns cexpr ->
            case (typ, assigns) of
                (_, []) -> checkexpr typ decls cexpr
                (_, (Posed pos nam, mexpr):ssigns) ->
                    case (lookup nam decls, mexpr) of
                        (Nothing, _) -> (pos, "Unknown variable"):(checkexpr typ decls (ComplEx ssigns cexpr))
                        (Just (Reference, _), Just iexpr) | typ == Number ->
                            (checkexpr Number decls (ComplEx ssigns cexpr))
                            ++ (checkexpr Number decls iexpr)
                        (Just (_, _), Just iexpr) ->
                            (pos, "Cannot index non-array")
                            : if typ == Number then [] else [(pos, "Type mismatch: expected " ++ show typ)]
                            ++(checkexpr typ decls (ComplEx ssigns cexpr))
                            ++ (checkexpr Number decls iexpr)
                        (Just (chaintype, _), Nothing) | chaintype == typ ->
                            (checkexpr chaintype decls (ComplEx ssigns cexpr))
                        (Just (chaintype, _), Nothing) ->
                            (pos, "Type mismatch: expected " ++ show typ)
                            :(checkexpr typ decls (ComplEx ssigns cexpr))
        CallEx (Posed pos nam) argexprs ->
            case lookup nam decls of
                Nothing -> [(pos, "Unknown variable")]
                Just (Function (rett:argt), _) ->
                    let mterr = if (rett == typ) then [] else [(pos, "Type mismatch: expected " ++ (show typ) ++ " expression")]
                        argerr = checkcalltypes argt argexprs
                        checkcalltypes [] [] = []
                        checkcalltypes [] _ = [(pos, "Too many arguments in function call")]
                        checkcalltypes _ [] = [(pos, "Too few arguments in function call")]
                        checkcalltypes (ah:at) (bh:bt) =
                            (checkexpr ah decls bh)++(checkcalltypes at bt)
                    in mterr ++ argerr
                _ -> [(pos, "Expected function name")]
        Retrieval (Posed pos nam, adrexpr) ->
            let nameerr = case lookup nam decls of
                    Nothing -> [(pos, "Unknown variable")]
                    Just (rettyp, _) ->
```

```
                              if | ( rettyp == typ && (case adrexpr of Nothing -> True; _ -> False)) -> []
                                  | ( rettyp == Reference && (case adrexpr of Nothing -> False; _ -> True)) -> []
                                  | otherwise -> [(pos, "Type mismatch: expected " ++ ( show typ) ++ " expression")]
                    argerr = case adrexpr of
                                    Nothing -> []
                                    Just iexpr -> checkexpr Number decls iexpr
               in nameerr ++ argerr
           StringLiteral (Posed pos values) ->
               case typ of Reference -> []; _ -> [(pos, "Type mismatch: expected reference expression")]
          NumLiteral (Posed pos val) ->
               case typ of Number -> []; _ -> [(pos, "Type mismatch: expected integer expression")]

morphdecl::[TDeclaration]->[Namedecl]->([Namedecl], [Error])
morphdecl pars prevdecls =
     let ep1 = map (\case
                              Intdecl (Posed pos nam) -> Right (nam, (Number, pos))
                              Arrdecl (Posed pos nam) _ -> Right (nam, (Reference, pos))
                              Fundecl (Posed pos _) _ _ -> Left (pos, "Function declaration in nested scope")
                              Procdecl(Posed pos _) _ _ -> Left (pos, "Function declaration in nested scope"))
                        pars
          errs1 = lefts ep1
          pars1 = rights ep1
          (pars2, errs2) = checkdoubles prevdecls pars1
     in (pars2, errs2++errs1)

checkdoubles::[Namedecl]->[Namedecl]->([Namedecl], [Error])
checkdoubles prev [] = ([],[])
checkdoubles prev ((nam, (typ, pos)):ar) =
     if (haskey nam prev)
          then let tail = checkdoubles prev ar in (fst tail, (pos, "Redefinition of variable"):snd tail)
          else let p = (nam, (typ, pos)); tail = checkdoubles (p:prev) ar in (p:fst tail, snd tail)
```

Приложение 4

Модуль построения выходного синтаксического дерева

```haskell
{-# LANGUAGE MultiWayIf #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE DeriveGeneric,DeriveAnyClass #-}

module Astbuilder(
    mkAST
    )where
import Cmm_alex
import Cmm_happy
import Dictutils
import Data.Either
import Data.Aeson(ToJSON)
import GHC.Generics

data Type = Number | Reference Int deriving (Eq, Show, Generic, ToJSON)
data Vardecl = Vardecl String Type deriving (Eq, Show, Generic, ToJSON)
data Funcdecl = Funcdecl String [String] Statement deriving (Eq, Show, Generic, ToJSON)
type Declaration = Either Vardecl Funcdecl

data Statement =
    Complex [Vardecl] [Statement] |
     Ite  Expression  Statement (Maybe Statement) |
    While Expression  Statement |
    Expsta Expression  |
    Return (Maybe Expression)
    deriving (Eq, Show, Generic, ToJSON)

data Expression =
    ConstInt Int |              -- 7
    ConstArr [Int] |            -- [7,8,9]
    Takeval Expression |        -- (*7) :: Address->Value / first element of array
    Takeadr String |            -- (&x) :: Name->Address
     Call String [Expression] |
    Assign [Expression] Expression  -- adr1 = adr2 = adr3 = 7
    deriving (Eq, Show, Generic, ToJSON)
```

```

convexpr :: TExpression->Expression
convexpr = \case
    ComplEx [] right ->
        convexpr right
    ComplEx lefts right ->
        Assign (map convref lefts) (convexpr right)
    CallEx (Posed _ nam) parexprs ->
         Call nam (map convexpr parexprs)
     Retrieval ref ->
        Takeval $ convref ref
    NumLiteral (Posed _ num) ->
        ConstInt num
     StringLiteral (Posed _ arr) ->
        ConstArr arr

convref :: Reference->Expression
convref ((Posed _ nam), Nothing) = Takeadr nam
convref ((Posed _ nam), Just adrexpr) = Call "+" [Takeadr nam, convexpr adrexpr]

convvardecl :: TDeclaration->Vardecl
convvardecl = \case
     Intdecl (Posed _ nam) -> Vardecl nam Number
     Arrdecl (Posed _ nam) (Posed _ size) -> Vardecl nam (Reference size)
    _ -> error "Unexpected function declaration"

convstat :: TStatement->Statement
convstat = \case
    CompSta tdecls tstats ->
        Complex (map convvardecl tdecls) (map convstat tstats)
    SelSta ifexpr tstat mestat ->
         Ite (convexpr ifexpr) (convstat tstat) (fmap convstat mestat)
     IterSta whexpr wstat ->
        While (convexpr whexpr) (convstat wstat)
    RetSta _ mexpr ->
        Return $ fmap convexpr mexpr
    ReadSta ref ->
        Expsta (Call "<<" [convref ref])
    ExpSta texpr ->
        Expsta (convexpr texpr)
    EmpSta ->
        Complex [] []

```

```
mkAST::[TDeclaration]->[Declaration]
mkAST [] = []
mkAST (t:ree) = case t of
    Intdecl (Posed _ nam) ->
        (Left $ Vardecl nam Number):mkAST ree
    Arrdecl (Posed _ nam) (Posed _ size) ->
        (Left $ Vardecl nam (Reference size)):mkAST ree
    Fundecl (Posed _ nam) pardecls stat ->
        (Right $ Funcdecl nam (getnams pardecls) (convstat stat)):mkAST ree
    Procdecl (Posed _ nam) pardecls stat ->
        (Right $ Funcdecl nam (getnams pardecls) (convstat stat)):mkAST ree
    where
        getnams::[TDeclaration]->[String]
        getnams [] = []
        getnams (d:ecl) = case d of
            Intdecl (Posed _ nam) -> nam:getnams ecl
            Arrdecl (Posed _ nam) _ -> nam:getnams ecl
            _ -> error "Unexpected function declaration in function parameters"
```

Приложение 5

Главный модуль программы

```
{-# LANGUAGE BangPatterns #-}

import Prelude hiding (writeFile)
import System.IO hiding (writeFile)
import Control.Monad(when)

import Cmm_alex
import Cmm_happy
import Checker
import Astbuilder

{-import Data.Yaml(encode)
import Data.ByteString ( writeFile )-}
import Data.Aeson.Encode.Pretty(encodePretty)
import Data.ByteString.Lazy (writeFile)
encode = encodePretty


main :: IO ()
main = do
    program <- readFile "cmm_program.cmm"
    let tokens = alexScanTokens program
    --putStrLn $ concat $ map (\t -> show t ++ "\n") $ tokens
    let tree = happyParseToTree tokens
    print tree
    let errors = check tree
    print errors
    when (errors == []) $ do
        let ast = mkAST tree
        writeFile "cmm_program.ast" $ encode ast
    -- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    -- ⊠⊠⊠⊠⊠⊠⊠⊠⊠AST

{-

type Reference = (Posed String, Maybe (TExpression))
data TExpression =
    ComplEx [Reference] TExpression
    | CallEx (Posed String) [TExpression]
    | Retrieval Reference
    | StringLiteral (Posed [Int])
    | NumLiteral (Posed Int)
    deriving (Show, Eq)
data TStatement =
    CompSta [TDeclaration] [TStatement]
    | SelSta TExpression TStatement (Maybe TStatement)
    | IterSta TExpression TStatement
    | RetSta (Maybe TExpression)
    | ReadSta Reference
    | ExpSta TExpression
    | EmpSta
    deriving (Show, Eq)
data TDeclaration =
    Intdecl (Posed String)
    | Arrdecl (Posed String) (Posed Int)
    | Fundecl (Posed String) [TDeclaration] TStatement
    | Procdecl (Posed String) [TDeclaration] TStatement
    deriving (Show, Eq)
}
-}
```

Приложение 6

Модуль вспомогательных утилит

```
module Dictutils (
    module Dictutils,
    lookup)
where
import Data.List

haskey :: Eq a => a->[(a,b)]->Bool
haskey _ [] = False
haskey key (d:ict) = if key == fst d then True else haskey key ict

hasval :: Eq b => b->[(a,b)]->Bool
hasval _ [] = False
hasval val (d:ict) = if val == snd d then True else hasval val ict
```