

Оглавление

1	Введение	3
2	Выбор платформы	4
3	Описание языка	5
4	Лексический анализ	7
4.1	Входные и выходные структуры данных	7
4.2	Обнаруживаемые ошибки	7
4.3	Реализация лексера	8
5	Синтаксический анализ	9
5.1	Входные и выходные структуры данных	9
5.2	Обнаруживаемые ошибки	10
5.3	Реализация парсера	10
6	Поиск ошибок	11
7	Список литературы	11
	Приложение 1	12
	Приложение 2	14
	Приложение 3	20
	Приложение 4	25
	Приложение 5	28
	Приложение 6	29

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 полученного 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 C--, дополненная строковыми литералами. Грамматика имеет следующий вид:

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

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

В грамматике также используются следующие регулярные выражения:

1. $\mathbf{ID} = [a - z]^+$
2. $\mathbf{NUM} = [0 - 9]^+ \mid 'PRINTABLE'$
3. $\mathbf{ARR} = "PRINTABLE" + "$
4. $\mathbf{PRINTABLE}$ -- соответствует любому печатаемому символу

Таблица 1: Смысловые значения выходных токенов лексера

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

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 1. Выходные токены аннотированы их позицией. Смысловые значения токенов даны в таблице 1.

```

1 data Token =
2   Symbol String
3   | Array [Int]
4   | Num Int
5   | Name String
6   deriving (Eq, Show)
7 data Posed a = Posed (Int,Int) a

```

Листинг 1: Выходные типы данных лексера

4.2. Обнаруживаемые ошибки

На данном этапе единственными ошибками, обнаруживаемыми фронтэндом, вызываются наличием в исходном коде программы символов, неприводимых к описанным выше типам токенов. К примеру, лексическую ошибку вызовет встреченный лексером в любом месте программы кроме как внутри одинарных либо двойных кавычек символ %.

Ошибки на данном этапе являются критическими: при возникновении ошибки работа фронтэнда немедленно завершается с выводением сообщения, содержащего номер строки и столбца, где был встречен ошибочный символ.

`1 lexical error at line 39, column 15`

Листинг 2: Сообщение о лексической ошибке

Пример сообщения об ошибке, вызываемой не входящим в грамматику языка символом, приведён в листинге 2.

4.3. Реализация лексера

Лексер был реализован с помощью средства генерации лексеров 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. Haskell-модуль генерируется из конфигурационного файла с помощью утилиты командной строки *alex*.

5. Синтаксический анализ

5.1. Входные и выходные структуры данных

На этапе синтаксического анализа происходит преобразование последовательности токенов, полученных лексером, в дерево выражения, которое будет в дальнейшем проверяться на типовую согласованность и отсутствие неопределённых имён.

На вход синтаксического анализатора подаётся список аннотированных позиций токенов, полученных лексером из исходного кода. Выходом анализатора является преобразованное дерево разбора исходной грамматики. От собственно дерева разбора оно отличается отсутствием элементов, не являющихся необходимыми для дальнейших расчётов -- так, в выходном дереве нет отдельных узлов, соответствующих нетерминалам *factor* и *term*, и они оба соответствуют узлу *Expression* в результирующем дереве.

Определение выходных типов парсера приведено в листинге 3.

```
1 type Reference = (Posed String, Maybe (TExpression))
2 data TExpression =
3   ComplEx [Reference] TExpression
4   | CallEx (Posed String) [TExpression]
5   | Retrieval Reference
6   | StringLiteral (Posed [Int])
7   | NumLiteral (Posed Int)
8   deriving (Show, Eq)
9 data TStatement =
10  CompSta [TDeclaration] [TStatement]
11  | SelSta TExpression TStatement (Maybe TStatement)
12  | IterSta TExpression TStatement
13  | RetSta (Int,Int) (Maybe TExpression)
14  | ReadSta Reference
15  | ExpSta TExpression
16  | EmpSta
17  deriving (Show, Eq)
18 data TDeclaration =
19  Intdecl (Posed String)
20  | Arrdecl (Posed String) (Posed Int)
21  | Fundecl (Posed String) [TDeclaration] TStatement
22  | Procdecl (Posed String) [TDeclaration] TStatement
23  deriving (Show, Eq)
```

Листинг 3: Выходные типы данных парсера

5.2. Обнаруживаемые ошибки

На данном этапе обнаруживаются ошибки несоответствия исходного кода грамматике языка. Ошибки данного типа также критические: хотя более сложные парсеры и могут восстанавливаться после ошибок, к примеру, с помощью поиска синхронизирующих элементов [9], но данный механизм является весьма громоздким в реализации.

При возникновении ошибки работа фронтэнда немедленно завершается с выводением сообщения, содержащего номер строки и столбца, где был встречен ошибочный токен.

Пример сообщения об ошибке, вызываемой ошибочным фрагментом кода `"int void arrln;"`, приведён в листинге 4.

```
1 Syntax error at (39,9): unexpected symbol "Symbol "void""
```

Листинг 4: Сообщение о синтаксической ошибке

5.3. Реализация парсера

Парсер был реализован с помощью средства генерации GLR-парсеров *Happy*.

Конфигурационный файл *Happy* содержит:

1. описание выходных типов данных в языке *Haskell*
2. функцию-обработчик ошибок разбора
3. определение терминалов и соответствующих им токенов
4. информацию о ассоциативности различных терминалов
5. правила вывода, содержащие цепочки из элементов объединённого алфавита
6. правила преобразования правых частей правил к соответствующим левым частям структурам данных

Haskell-модуль генерируется из конфигурационного файла с помощью утилиты командной строки *happy*. Содержимое конфигурационного файла *Happy* приведено в приложении 2.

6. Поиск ошибок

7. Список литературы

- [1] FP Complete. *Minimum GHC Installer*. 14 мая 2015. URL: <https://github.com/fpc/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).
- [9] Википедия. *Синтаксический анализ --- Википедия, свободная энциклопедия*. [Online; accessed 19-05-2015]. 2015. URL: <http://ru.wikipedia.org/?oldid=69538992>.

Приложение 1

Конфигурационный файл Alex

```
1 {
2 {-# Language LambdaCase #-}
3 module Cmm_alex (Token(..), Posed(..), alexScanTokens) where
4 }
5
6 %wrapper "posn"
7
8 -- $white
9 $letter = [a-z A-Z]
10 $digit = 0-9
11
12 --{- name literal -}
13 @name = $letter+
14
15 --{- integer literal -}
16 @int = \-? $digit+
17
18 --{- char literal -}
19 @escapeseq = 0 | a | b | f | n | r | t | \\ | \' | \" | \?
20 @escapechar = \\ @escapeseq
21 @char = $printable | @escapechar
22 @character = \' @char \'
23
24 --{- string literal -}
25 @string = \" $printable* \"
26
27 --{- symbol literal -}
28 @symbol = "[" | "]" | "{" | "}" | "(" | ")" | ";" | ","
29           | "==" | "=" | "<=" | "<" | ">=" | ">" | "!="
30           | "+" | "-" | "*" | "/"
31
32 --{- reserved word literal -}
33 @reserved = "int" | "void" | "if" | "else" | "while" | "return" | "read" | "write"
34           "
35
36 --{- comments -}
```

```

36 @comment = "//" .*
37   | "/" ( . | \n ) * "*" /
38
39 -----
40 tokens :-
41
42   $white+      ;
43   @comment     ;
44   @reserved    {pose Symbol}
45   @name        {pose Name}
46   @string      {pose (\s -> Array $ map fromEnum $ sanstr s)}
47   @character   {pose (\s -> Num $ fromEnum $ (read s :: Char))}
48   @int         {pose (\s -> Num $ read s)}
49   @symbol      {pose Symbol}
50
51 {
52 data Token =
53   Symbol String |
54   Array [Int]   |
55   Num Int       |
56   Name String
57   deriving (Eq, Show)
58
59 data Posed a = Posed (Int,Int) a
60 instance (Eq a => Eq (Posed a)) where
61   (==) (Posed _ a) (Posed _ b) = a == b
62 instance (Show a => Show (Posed a)) where
63   show (Posed p a) = concat [show p, "~", show a]
64
65 pose :: (String -> a) -> AlexPosn -> String -> Posed a
66 pose constr (AlexPn abs line col) s = Posed (line, col) (constr s)
67
68 sanstr (' ':tail) = sanstrlast tail
69 sanstrlast = \case
70   [] -> []
71   a:[] | a == "' ' -> '\0':[]
72   a:b -> a:sanstrlast b
73
74 }

```

Приложение 2

Конфигурационный файл Happy

```
1 {
2 module Cmm_happy(
3     happyParseToTree,
4     Reference,
5     TExpression(..),
6     TDeclaration(..),
7     TStatement(..),
8     )where
9 import Cmm_alex
10 }
11
12 %name happyParseToTree DeclarationList
13 %tokentype { Posed Token}
14 %error { parseError }
15
16 %token
17 array      {Posed _ (Array _)}
18 num        {Posed _ (Num _)}
19 name       {Posed _ (Name _)}
20 "["        {Posed _ (Symbol "[")}
21 "]"        {Posed _ (Symbol "]")}
22 "{"        {Posed _ (Symbol "{")}
23 "}"        {Posed _ (Symbol "}")}
24 "("        {Posed _ (Symbol "(")}
25 ")"        {Posed _ (Symbol ")")}
26 ";"        {Posed _ (Symbol ";")}
27 ","        {Posed _ (Symbol ",")}
28 "=="       {Posed _ (Symbol "==")}
29 "="        {Posed _ (Symbol "=")}
30 "<="        {Posed _ (Symbol "<=")}
31 "<"         {Posed _ (Symbol "<")}
32 ">="        {Posed _ (Symbol ">=")}
33 ">"         {Posed _ (Symbol ">")}
34 "!="       {Posed _ (Symbol "!=")}
35 "+"        {Posed _ (Symbol "+")}
36 "-"        {Posed _ (Symbol "-)}
```

```

37      "*"          {Posed _ (Symbol "*")}
38      "/"          {Posed _ (Symbol "/")}
39      "int"         {Posed _ (Symbol "int")}
40      "void"        {Posed _ (Symbol "void")}
41      "if"          {Posed _ (Symbol "if")}
42      "else"        {Posed _ (Symbol "else")}
43      "while"       {Posed _ (Symbol "while")}
44      "return"      {Posed _ (Symbol "return")}
45      "read"        {Posed _ (Symbol "read")}
46      "write"       {Posed _ (Symbol "write")}
47
48      %nonassoc ")"
49      %nonassoc "else"
50
51      %%
52
53      DeclarationList ::{ [TDeclaration] }
54      DeclarationList : Declaration          { [$1] }
55                      | DeclarationList Declaration { $1 ++ [$2] }
56
57      Declaration     ::{ TDeclaration }
58      Declaration     : VarDeclaration       { $1 }
59                      | FunDeclaration      { $1 }
60
61      VarDeclaration  ::{ TDeclaration }
62      VarDeclaration  : "int" name ";",
63                      $2
64                      | "int" name "[" num "]" ";",
65                      $2;
66
67                      (Posed p2 (Num i)) =
68                      $4
69                      in (Arrdecl (Posed p n) (
70                      Posed p2 i)) }
71
72      FunDeclaration  ::{ TDeclaration }
73      FunDeclaration  : "int" name "(" Params ")" CompoundStmt { let (Posed p (Name n)
74                      ) = $2
75
76                      in (Fundekl (Posed p n
77                      ) $4 $6) }
78
79                      | "void" name "(" Params ")" CompoundStmt { let (Posed p (Name n
80                      )) = $2
81
82                      in (Procdecl (Posed p

```

```

n) $4 $6) }

73
74 Params      ::{ [TDeclaration] }
75 Params      : "void"                { [] }
76             | ParamList             { $1 }
77 ParamList    ::{ [TDeclaration] }
78 ParamList    : Param                 { [$1] }
79             | ParamList "," Param    { $1 ++ [$3] }
80 Param        ::{ TDeclaration }
81 Param        : "int" name           { let (Posed p (Name n)) =
    $2                                     in Intdecl (Posed p n) }
82                                     { let (Posed p (Name n)) =
83           | "int" name "[" "]"       { let (Posed p (Name n)) =
    $2                                     in Arrdecl (Posed p n) (
84                                     Posed p 0) }

85 CompoundStmt ::{ TStatement }
86 CompoundStmt : "{" LocalDeclarations StatementList "}" { CompSta $2 $3 }
87 -----
88 LocalDeclarations ::{ [TDeclaration] }
89 LocalDeclarations : {--}                { [] }
90                 | LocalDeclarations VarDeclaration { $1 ++ [$2] }
91
92 StatementList  ::{ [TStatement] }
93 StatementList  : {--}                { [] }
94                 | StatementList Statement { $1 ++ [$2] }
95
96 Statement      ::{ TStatement }
97 Statement      : ExpressionStmt       { $1 }
98                 | CompoundStmt        { $1 }
99                 | SelectionStmt       { $1 }
100                | IterationStmt        { $1 }
101                | ReturnStmt           { $1 }
102                | ReadStmt             { $1 }
103                | WriteStmt            { $1 }
104
105 ExpressionStmt ::{ TStatement }
106 ExpressionStmt : Expression ";"       { ExpSta $1 }
107                 | ";"                { EmpSta }
108
109 SelectionStmt  ::{ TStatement }
110 SelectionStmt  : "if" "(" Expression ")" Statement { SelSta $3
    $5 Nothing }

```

```

111         | "if" "(" Expression ")" Statement "else" Statement { SelSta $3
                    $5 (Just $7) }
112
113 IterationStmt ::{ TStatement }
114 IterationStmt : "while" "(" Expression ")" Statement { IterSta $3 $5 }
115
116 ReturnStmt    ::{ TStatement }
117 ReturnStmt    : "return" ";" { let (Posed pos _) = $1
                                in RetSta pos Nothing }
118
119         | "return" Expression ";" { let (Posed pos _) = $1
                                in RetSta pos (Just $2) }
120
121
122 ReadStmt      ::{ TStatement }
123 ReadStmt      : "read" Var ";" { ReadSta $2 }
124
125 WriteStmt     ::{ TStatement }
126 WriteStmt     : "write" Expression ";" { let (Posed p (Symbol "write")) = $1
                                in ExpSta $ CallEx (Posed p ">>") [$2
                                ] }
127
128 -----
129 Expression    ::{ TExpression }
130 Expression    : ExpressionHead SimpleExpression { ComplEx $1 $2 }
131
132 ExpressionHead ::{ [Reference] }
133 ExpressionHead : {--} { [] }
134         | ExpressionHead Var "=" { $1 ++ [$2] }
135
136 Var           ::{ Reference }
137 Var           : name { let (Posed p (Name n)) = $1
                                in (Posed p n, Nothing) }
138
139         | name "[" Expression "]" { let (Posed p (Name n)) = $1
                                in (Posed p n, Just $3) }
140
141
142 SimpleExpression ::{ TExpression }
143 SimpleExpression: AdditiveExpression { $1 }
144         | AdditiveExpression Relop AdditiveExpression { let (Posed p (
                                Symbol n)) = $2
                                in CallEx (Posed p
145                                         n) [$1, $3] }
146
147 Relop         ::{ Posed Token }
148 Relop         : "<=" { $1 }
149         | "<" { $1 }

```



```

150         | ">"           { $1 }
151         | ">="         { $1 }
152         | "=="          { $1 }
153         | "!="          { $1 }
154
155 AdditiveExpression ::{ TExpression }
156 AdditiveExpression : Term { $1 }
157                   | AdditiveExpression Addop Term { let (Posed p (Symbol n))
                                     = $2
                                     in CallEx (Posed p n) [$1,
                                     $3] }
158
159
160 Addop              ::{ Posed Token }
161 Addop              : "+" { $1 }
162                   | "-" { $1 }
163
164 Term               ::{ TExpression }
165 Term               : Factor { $1 }
166                   | Term Multop Factor { let (Posed p (Symbol n)) = $2
                                     in CallEx (Posed p n) [$1, $3] }
167
168 Multop             ::{ Posed Token }
169 Multop             : "*" { $1 }
170                   | "/" { $1 }
171
172 Factor             ::{ TExpression }
173 Factor             : "(" Expression ")" { $2 }
174                   | num { let (Posed p (Num n)) = $1
                                     in NumLiteral $ Posed p n }
175                   | array { let (Posed p (Array n)) = $1
                                     in StringLiteral $ Posed p n }
176                   | Var { Retrieval $1 }
177                   | Call { $1 }
178
179
180 Call               ::{ TExpression }
181 Call               : name "(" Args ")" { let (Posed p (Name n)) = $1
                                     in CallEx (Posed p n) $3 }
182
183
184
185 Args              ::{ [TExpression] }
186 Args              : {--} { [] }
187                   | Expression { [$1] }
188                   | Args "," Expression { $1 ++ [$3] }
189
190 {

```

```

191 parseError :: Show b => [Posed b] -> a
192 parseError ((Posed p t):rst) = error $ "Syntax error at " ++ (show p) ++ ":
    unexpected symbol \"\" ++ (show t) ++ "\"\"
193
194 type Reference = (Posed String, Maybe (TExpression))
195 data TExpression =
196     ComplEx [Reference] TExpression
197     | CallEx (Posed String) [TExpression]
198     | Retrieval Reference
199     | StringLiteral (Posed [Int])
200     | NumLiteral (Posed Int)
201     deriving (Show, Eq)
202 data TStatement =
203     CompSta [TDeclaration] [TStatement]
204     | SelSta TExpression TStatement (Maybe TStatement)
205     | IterSta TExpression TStatement
206     | RetSta (Int,Int) (Maybe TExpression)
207     | ReadSta Reference
208     | ExpSta TExpression
209     | EmpSta
210     deriving (Show, Eq)
211 data TDeclaration =
212     Intdecl (Posed String)
213     | Arrdecl (Posed String) (Posed Int)
214     | Fundecl (Posed String) [TDeclaration] TStatement
215     | Procdecl (Posed String) [TDeclaration] TStatement
216     deriving (Show, Eq)
217 }

```

Приложение 3

Модуль проверки логических ошибок

```
1 {-# LANGUAGE MultiWayIf #-}
2 {-# LANGUAGE LambdaCase #-}
3 module Checker(
4     check
5 )where
6 import Cmm_alex
7 import Cmm_happy
8 import Dictutils
9 import Data.Either
10
11 defaultFunctions :: [ NamedDecl]
12 defaultFunctions =
13     [( "<=", (Function [Boolean, Number, Number],(0,0)))
14     ,(" < ", (Function [Boolean, Number, Number],(0,0)))
15     ,(" > ", (Function [Boolean, Number, Number],(0,0)))
16     ,(" >=", (Function [Boolean, Number, Number],(0,0)))
17     ,(" == ", (Function [Boolean, Number, Number],(0,0)))
18     ,(" != ", (Function [Boolean, Number, Number],(0,0)))
19
20     ,(" + ", (Function [Number, Number, Number],(0,0)))
21     ,(" - ", (Function [Number, Number, Number],(0,0)))
22     ,(" * ", (Function [Number, Number, Number],(0,0)))
23     ,(" / ", (Function [Number, Number, Number],(0,0)))
24
25     ,(" >> ", (Function [Void, Number],(0,0)))
26     ,(" << ", (Function [Void, Reference],(0,0)))
27
28 data Typ = Boolean | Number | Reference | Void | Function [Typ] | Any deriving (
29     Show)
30 instance (Eq Typ) where
31     a == b =
32         case (a,b) of
33             (Any, _) -> True
34             (_, Any) -> True
35             (Boolean, Boolean) -> True
36             (Number, Number) -> True
```

```

36         (Reference, Reference) -> True
37         (Void, Void) -> True
38         (Function a, Function b) -> a == b
39         _ -> False
40
41     type Pos = (Int, Int)
42
43     type Namedecl = (String, (Typ, Pos))
44     type Error = (Pos, String)
45
46     check :: [TDeclaration] -> [Error]
47     check decls = checkTopLevel decls defaultFunctions
48
49     checkTopLevel :: [TDeclaration] -> [Namedecl] -> [Error]
50     checkTopLevel [] _ = []
51     checkTopLevel (d:ecl) prevdecls =
52         case d of
53             Intdecl (Posed pos nam) ->
54                 if | haskey nam prevdecls ->
55                     (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
56                 | otherwise ->
57                     checkTopLevel ecl ((nam, (Number, pos)):prevdecls)
58             Arrdecl (Posed posn nam) (Posed poss siz) ->
59                 if | haskey nam prevdecls ->
60                     (posn, "Redefinition of variable"):checkTopLevel ecl prevdecls
61                 | siz <= 0 ->
62                     (poss, "Nonpositive array size"):checkTopLevel ecl prevdecls
63                 | otherwise ->
64                     checkTopLevel ecl ((nam, (Reference, posn)):prevdecls)
65             Fundecl (Posed pos nam) paramsraw body ->
66                 if | haskey nam prevdecls ->
67                     (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
68                 | otherwise ->
69                     let (pardecl, parerrors) = morphdecl paramsraw prevdecls
70                         bodyerrors = checkstat (pardecl++prevdecls) body Number
71                         fntype = Number:(map (\(_, (t, _)) -> t) pardecl)
72                         fdcl = (nam, (Function fntype, pos))
73                     in bodyerrors ++ parerrors ++
74                         checkTopLevel ecl (fdcl:prevdecls)
75             Procdecl (Posed pos nam) paramsraw body ->
76                 if | haskey nam prevdecls ->
77                     (pos, "Redefinition of variable"):checkTopLevel ecl prevdecls
78                 | otherwise ->

```

```

79         let (pardecl, parerrors) = morphdecl paramsraw prevdecls
80         bodyerrors = checkstat (pardecl ++ prevdecls) body Void
81         fntype = Void:(map (\(_, (t, _)) -> t) pardecl)
82         fdcl = (nam, (Function fntype, pos))
83         in bodyerrors ++ parerrors ++
84         checkTopLevel ecl (fdcl : prevdecls)
85
86 checkstat :: [Nameddecl] -> TStatement -> Typ -> [Error]
87 checkstat prevdecls statement rettyp =
88     case statement of
89     CompSta decls nested ->
90         let (locdecl, declerr) = morphdecl decls prevdecls
91         tail = concat $ map (\st -> checkstat (locdecl ++ prevdecls) st
92             rettyp) nested
93         in (declerr ++ tail)
94     SelSta bexpr thn mels ->
95         checkexpr Boolean prevdecls bexpr
96         ++ checkstat prevdecls thn rettyp
97         ++ case mels of Just els -> checkstat prevdecls els rettyp; Nothing
98             -> []
99     IterSta bexpr whl ->
100         checkexpr Boolean prevdecls bexpr
101         ++ checkstat prevdecls whl rettyp
102     RetSta p Nothing ->
103         if rettyp == Void then [] else [(p, "Expected empty return")]
104     RetSta p (Just rexpr) ->
105         if rettyp == Void then [(p, "Expected expression")] else checkexpr
106             rettyp prevdecls rexpr
107     ReadSta (Posed pos nam, Nothing) ->
108         case lookup nam prevdecls of
109         Just (Number, _) -> []
110         Just _ -> [(pos, "Type mismatch: expected integer variable")]
111         Nothing -> [(pos, "Unknown variable")]
112     ReadSta (Posed pos nam, Just iexpr) ->
113         case lookup nam prevdecls of
114         Just (Number, _) -> [(pos, "Type mismatch: expected array
115             variable")]
116         Just (Reference, _) -> checkexpr Number prevdecls iexpr
117         Nothing -> [(pos, "Unknown variable")]
118     ExpSta sexpr -> checkexpr Any prevdecls sexpr
119     EmpSta -> []
120
121 checkexpr :: Typ -> [Nameddecl] -> TExpression -> [Error]

```

```

118 checkexpr typ decls expr =
119   case expr of
120     ComplEx assigns cexpr ->
121       case (typ, assigns) of
122         (_, []) -> checkexpr typ decls cexpr
123         (_, (Posed pos nam, mexpr):ssigns) ->
124           case (lookup nam decls, mexpr) of
125             (Nothing, _) -> (pos, "Unknown variable"):(checkexpr typ
126               decls (ComplEx ssigns cexpr))
127             (Just (Reference, _), Just iexpr) | typ == Number ->
128               (checkexpr Number decls (ComplEx ssigns cexpr))
129               ++ (checkexpr Number decls iexpr)
130             (Just (_, _), Just iexpr) ->
131               (pos, "Cannot index non-array")
132               : if typ == Number then [] else [(pos, "Type mismatch:
133                 expected " ++ show typ)]
134               ++ (checkexpr typ decls (ComplEx ssigns cexpr))
135               ++ (checkexpr Number decls iexpr)
136             (Just (chaintype, _), Nothing) | chaintype == typ ->
137               (checkexpr chaintype decls (ComplEx ssigns cexpr))
138             (Just (chaintype, _), Nothing) ->
139               (pos, "Type mismatch: expected " ++ show typ)
140               :(checkexpr typ decls (ComplEx ssigns cexpr))
141     CallEx (Posed pos nam) argexprs ->
142       case lookup nam decls of
143         Nothing -> [(pos, "Unknown variable")]
144         Just (Function (rett : argt), _) ->
145           let mterr = if (rett == typ) then [] else [(pos, "Type
146             mismatch: expected " ++ (show typ) ++ " expression")]
147           argerr = checkcalltypes argt argexprs
148           checkcalltypes [] [] = []
149           checkcalltypes [] _ = [(pos, "Too many arguments in
150             function call")]
151           checkcalltypes _ [] = [(pos, "Too few arguments in function
152             call")]
153           checkcalltypes (ah:at) (bh:bt) =
154             (checkexpr ah decls bh)++(checkcalltypes at bt)
155           in mterr ++ argerr
156       _ -> [(pos, "Expected function name")]
157     Retrieval (Posed pos nam, adrexpr) ->
158       let nameerr = case lookup nam decls of
159         Nothing -> [(pos, "Unknown variable")]
160         Just (rettyp, _) ->

```

```

156         if | ( rettyp == typ && (case adrexp of Nothing ->
              True; _ -> False)) -> []
157         | ( rettyp == Reference && (case adrexp of Nothing ->
              False; _ -> True)) -> []
158         | otherwise -> [(pos, "Type mismatch: expected " ++ (
              show typ) ++ " expression")]
159     argerr = case adrexp of
160               Nothing -> []
161               Just iexpr -> checkexpr Number decls iexpr
162     in nameerr ++ argerr
163   StringLiteral (Posed pos values) ->
164     case typ of Reference -> []; _ -> [(pos, "Type mismatch: expected
              reference expression")]
165   NumLiteral (Posed pos val) ->
166     case typ of Number -> []; _ -> [(pos, "Type mismatch: expected
              integer expression")]
167
168 morphdecl :: [TDeclaration] -> [Nameddecl] -> ([Nameddecl], [Error])
169 morphdecl pars prevdecls =
170   let ep1 = map (\case
171                 Intdecl (Posed pos nam) -> Right (nam, (Number, pos))
172                 Arrdecl (Posed pos nam) _ -> Right (nam, (Reference, pos))
173                 Fundecl (Posed pos _) _ _ -> Left (pos, "Function
              declaration in nested scope")
174                 Procdecl (Posed pos _) _ _ -> Left (pos, "Function
              declaration in nested scope"))
175       pars
176   errs1 = lefts ep1
177   pars1 = rights ep1
178   (pars2, errs2) = checkdoubles prevdecls pars1
179   in (pars2, errs2 ++ errs1)
180
181 checkdoubles :: [Nameddecl] -> [Nameddecl] -> ([Nameddecl], [Error])
182 checkdoubles prev [] = ([], [])
183 checkdoubles prev ((nam, (typ, pos)):ar) =
184   if (haskey nam prev)
185     then let tail = checkdoubles prev ar in (fst tail, (pos, "Redefinition of
              variable"):snd tail)
186   else let p = (nam, (typ, pos)); tail = checkdoubles (p:prev) ar in (p:fst
              tail, snd tail)

```

Приложение 4

Модуль построения выходного синтаксического дерева

```
1 {-# LANGUAGE MultiWayIf #-}
2 {-# LANGUAGE LambdaCase #-}
3 {-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
4
5 module Astbuilder(
6     mkAST
7 )where
8 import Cmm_alex
9 import Cmm_happy
10 import Dictutils
11 import Data.Either
12 import Data.Aeson(ToJSON)
13 import GHC.Generics
14
15 data Type = Number | Reference Int deriving (Eq, Show, Generic, ToJSON)
16 data Vardecl = Vardecl String Type deriving (Eq, Show, Generic, ToJSON)
17 data Funcdecl = Funcdecl String [String] Statement deriving (Eq, Show, Generic,
18     ToJSON)
19
20 type Declaration = Either Vardecl Funcdecl
21
22 data Statement =
23     Complex [Vardecl] [Statement] |
24     Ite Expression Statement (Maybe Statement) |
25     While Expression Statement |
26     Expsta Expression |
27     Return (Maybe Expression)
28     deriving (Eq, Show, Generic, ToJSON)
29
30 data Expression =
31     ConstInt Int | -- 7
32     ConstArr [Int] | -- [7,8,9]
33     Takeval Expression | -- (*7) :: Address->Value / first element of array
34     Takeadr String | -- (&x) :: Name->Address
35     Call String [Expression] |
36     Assign [Expression] Expression -- adr1 = adr2 = adr3 = 7
37     deriving (Eq, Show, Generic, ToJSON)
```



```

36
37 convexpr :: TExpression -> Expression
38 convexpr = \case
39   ComplEx [] right ->
40     convexpr right
41   ComplEx lefts right ->
42     Assign (map convref lefts) (convexpr right)
43   CallEx (Posed _ nam) parexprs ->
44     Call nam (map convexpr parexprs)
45   Retrieval ref ->
46     Takeval $ convref ref
47   NumLiteral (Posed _ num) ->
48     ConstInt num
49   StringLiteral (Posed _ arr) ->
50     ConstArr arr
51
52 convref :: Reference -> Expression
53 convref ((Posed _ nam), Nothing) = Takeadr nam
54 convref ((Posed _ nam), Just adrexpr) = Call "+" [Takeadr nam, convexpr adrexpr]
55
56 convvardecl :: TDeclaration -> Vardecl
57 convvardecl = \case
58   Intdecl (Posed _ nam) -> Vardecl nam Number
59   Arrdecl (Posed _ nam) (Posed _ size) -> Vardecl nam (Reference size)
60   _ -> error "Unexpected function declaration"
61
62 convstat :: TStatement -> Statement
63 convstat = \case
64   CompSta tdecls tstats ->
65     Complex (map convvardecl tdecls) (map convstat tstats)
66   SelSta ifexpr tstat mestat ->
67     ITE (convexpr ifexpr) (convstat tstat) (fmap convstat mestat)
68   IterSta whexpr wstat ->
69     While (convexpr whexpr) (convstat wstat)
70   RetSta _ mexpr ->
71     Return $ fmap convexpr mexpr
72   ReadSta ref ->
73     Expsta (Call "<<" [convref ref])
74   ExpSta texpr ->
75     Expsta (convexpr texpr)
76   EmpSta ->
77     Complex [] []
78

```

```

79 mkAST::[TDeclaration]->[Declaration]
80 mkAST [] = []
81 mkAST (t:ree) = case t of
82   Intdecl (Posed _ nam) ->
83     (Left $ Vardecl nam Number):mkAST ree
84   Arrdecl (Posed _ nam) (Posed _ size) ->
85     (Left $ Vardecl nam (Reference size)):mkAST ree
86   Fundecl (Posed _ nam) pardecls stat ->
87     (Right $ Funcdecl nam (getnams pardecls) (convstat stat)):mkAST ree
88   Procdecl (Posed _ nam) pardecls stat ->
89     (Right $ Funcdecl nam (getnams pardecls) (convstat stat)):mkAST ree
90 where
91   getnams::[TDeclaration]->[String]
92   getnams [] = []
93   getnams (d:ecl) = case d of
94     Intdecl (Posed _ nam) -> nam:getnams ecl
95     Arrdecl (Posed _ nam) _ -> nam:getnams ecl
96     _ -> error "Unexpected function declaration in function parameters"

```

Приложение 5

Главный модуль программы

```
1 {-# LANGUAGE BangPatterns #-}
2
3 import Prelude hiding (writeFile)
4 import System.IO hiding (writeFile)
5 import Control.Monad(when)
6
7 import Cmm_alex
8 import Cmm_happy
9 import Checker
10 import Astbuilder
11
12 {-import Data.Yaml(encode)
13 import Data.ByteString ( writeFile )-}
14 import Data.Aeson.Encode.Pretty(encodePretty)
15 import Data.ByteString.Lazy (writeFile)
16 encode = encodePretty
17
18
19 main :: IO ()
20 main = do
21     program <- readFile "cmm_program.cmm"
22     let tokens = alexScanTokens program
23     --putStrLn $ concat $ map (\t -> show t ++ "\n") $ tokens
24     let tree = happyParseToTree tokens
25     print tree
26     let errors = check tree
27     print errors
28     when (errors == []) $ do
29         let ast = mkAST tree
30         writeFile "cmm_program.ast" $ encode ast
```

Приложение 6

Модуль вспомогательных утилит

```
1 module Dictutils (  
2     module Dictutils,  
3     lookup)  
4 where  
5 import Data.List  
6  
7 haskey :: Eq a => a -> [(a,b)] -> Bool  
8 haskey _ [] = False  
9 haskey key (d:ict) = if key == fst d then True else haskey key ict  
10  
11 hasval :: Eq b => b -> [(a,b)] -> Bool  
12 hasval _ [] = False  
13 hasval val (d:ict) = if val == snd d then True else hasval val ict
```