A J0 to LLVM Compiler in Haskell

dreamycactus

April 5, 2014

Abstract

For the undergraduate senior compilers course final project, J0 to LLVM compiler was implemented in Haskell. The J0 toy language consists of a subset of the Java language stripped down to the most essential features. The compiler code was documented using literate programming techniques and open sourced on Github as an additional reference to novice compilers construction in Haskell.

1 Introduction

The primary goal in this project was to learn about and implement the main components of compilers while demonstrating good software practices including proper documentation, modularization, and readability to name a few. The compiler's input language, J0 is a tiny toy language(ENBF here) with Java-like syntax which is then compiled into the LLVM Intermediate Representation (LLVM IR). From here users can convert IR into bitcode with the LLVM toolchain. (LLVM-as in this case).

2 LLVM

The LLVM Project consists of resuable compiler and toolchain technologies which all started from a research project at the University of Illinois.LLVM Intermediate Representation(LLVM IR) is a relatively high level intermediate representation supporting an static single assignment form (SSA)-based compilation strategy capable of static and dynamic compilation. The essential idea behind SSA is that each variable is assigned to only once, allowing certain compiler optimizations to be done cheaply.

LLVM's bitstream file format is identified and verified by "magic number" bits. The file itself consists of blocks and records. There are many types of blocks, with basic blocks which roughly correspond to labels in assembly, and blocks that contain function bodies. Data records which contain a record code and a number keep track of data objects and describe entities in the file.

3 Approach

The main components of the J0 compiler are shown in the below diagram. Lexing and Parsing will be implemented using Haskell's Parsec library, and the Analyzer and Code Generator will use llvm-general Haskell bindings.

3.1 Parsing

The J0 compiler is implemented in Haskell, which provides a few benefits over C. Haskell allows for elegant parsing, particularly with the aid of the Parsec parsing library. Parsec contains various features that support top-down recursive descent parsing in addition to other methods and handles details such as lookahead characters and provides many useful parsing aids. The ¡—¿ operator implements choice in parsing.

3.2 Documentation

Haskell has native support for literate programming. Haskell code conforming to the principles of literate programming use the file extension .lhs rather than .hs. Most of J0 compiler's code is written in this format. From .lhs files, there are latex utilities, namely lhs2tex, that then format and present the literate program.

4 Resources

The canonical Dragon book was be used as the primary academic resource for the construction of the J0 compiler. Let's Build A Compiler, written by Jack Crenshaw was used as a resource for more practical aspects of compiler construction. Source code repositories `https://github.com/sdiehl/kaleidoscope` and `https://github.com/alephnullplex/cradle` were excellent resources which had well documented code for a JIT compiler for Kaleidoscope and Pascal0 respectively.

5 Acknowledgements

Much of the code was based on Stephen Diel's Haskell LLVM tutorials, which without, it would have been much more difficult getting started with this project. The attribute grammar for the language was based on the minijava compiler repository on Github.

6 Implementation Overview

6.1 Features

The single pass j0 compiler takes an input .j0 file, and outputs the corresponding LLVM IR. It can also work interactively taking input via command line, as long as the input code is given appropriate chunks (at a class definition granularity). Were this a fully featured and more practical compiler, the compiler should output LLVM bitcode rather than IR. As it stands, an extra step is required to generate the bitcode from the IR.

6.2 Limitations and Problems

There are quite a number of bugs at this moment with the tool. In particular, bugs relating to symbol conflicts are not caught by the tool. For example one can define a class member "a" then a class function "a" in the same class, and the compiler will not complain. However during conversion of IR to bitcode, the LLVM llc tool detects the problem.

Another problem is that each class function is available at global scope. This is due to proper class method calling not being fully implemented.

One other big problem is the lack of type checking. llvm-general constructs do some type checking via its operands internally, but is not a complete type checking solution. For example, we can use an object as a condition, which is simply nonsense. The compiler will not detect this, but llc will.

Due to time constraints, I did not implement code generation for boolean logic and unary and binary comparison operators. They are, however, being parsed correctly. In addition, the only control statement that emits code is the if statement. There is no code generation for loops. All of these features should be trivial to implement as will be seen further below in the source code.

6.3 High Level Design

The structure of the j0 compiler is quite simple. The program primarily consists of the following modules: Lexer, Syntax Definitions, Parser, Codegeneration backend, and Code emitter. Their functions should be self explanatory. The compiler flow is as follows:

The Parsec library is used in lexing and parsing, and the llvm-general library is used for to emit IR from an LLVM Module data structure which contains all the translated definitions from the input source.

7 Lexer

This module simply contains definitions that Parsec uses to configure its lexer. There is not much of note other than some convenience functions defined near the end of the file which are used in the parser.

```
module Lexert where

import Text.Parsec.String (Parser)
import Text.Parsec.Language (emptyDef)

import qualified Text.Parsec.Token as Tok

lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser style
  where
    ops = ["+","*","-","/",";",",","<","=",">","."]
    names = ["class","static","if","then","else","in"
           , "while","return","print","null","new","int"]
    style = emptyDef {
                Tok.commentLine = "//"
              , Tok.reservedOpNames = ops
              , Tok.reservedNames = names
              }

integer    = Tok.integer lexer
float      = Tok.float lexer
parens     = Tok.parens lexer
braces     = Tok.braces lexer
commaSep   = Tok.commaSep lexer
semiSep    = Tok.semiSep lexer
identifier = Tok.identifier lexer
whitespace = Tok.whiteSpace lexer
reserved   = Tok.reserved lexer
reservedOp = Tok.reservedOp lexer
```

8 Syntax

The Syntax Module contains data structures which implement attribute grammar for the language. The structure of the grammar encourages top-down parsing.

```haskell
module SyntaxMini where
```

A Program consists of 1+ class declarations, with the first declaration being considered the main class.

```haskell
data Program
  = Program ClassDecl [ClassDecl]
      deriving(Eq, Show)

progMainClass (Program mc cd) = mc
progClassDecls (Program mc cd) = cd
```

The class declaration data structure is designed in such a way to support inheritance, but in this revision is not used. The extends field will always be Nothing.

```haskell
data ClassDecl
    = ClassDecl { class_name :: Id
                , extends    :: Maybe Id
                , vars       :: [VarDecl]
                , methods    :: [MethodDecl] }
    deriving (Eq, Show)
```

Variable and Method declaration data structures are what one might expect of an imperative language. VarDecl is used both for local declarations in a function body, arguments to functions, and class member declarations. This makes sense as the syntax and data are similar for all of these purposes.

The Method declaration is a bit stricter than usual imperative languages. It mandates that each method must have exactly one return expression.

```haskell
data VarDecl
    = VarDecl { var_type :: Type, var_id :: Id }
    deriving (Eq, Show)

varName (VarDecl ty id) = id

data MethodDecl
    = MethodDecl { m_type   :: Type
                 , m_name   :: Id
                 , m_args   :: [VarDecl]
                 , decls    :: [VarDecl]
                 , body     :: [Statement]
                 , m_return :: Exp }
    deriving (Eq, Show)
```

There is only one “first class” type in my implementation, the integer. All other types are declared as classes.

```haskell
type Id = String

data Type
    = T_Int
    | T_Id Id
    deriving (Eq, Show)
```

Some of the statement constructors are not used. They are simply there to demonstrate the possibilities of this schema ofsyntax attributes. All Statements do not have any values associated with them, in contrast to expressions, which must represent a value.

```
data Statement
    = S_Block        [Statement]
    | S_If           { cond :: Exp, then_arm :: Statement
                     , else_arm :: Statement }
    | S_While        { cond :: Exp, while_body :: Statement }
    | S_Print        Exp
    | S_Return       Exp
    | S_Void         Exp
    | S_Assign       { var :: Id, classId :: Id, value :: Exp }
    | S_ArrayAssign { var :: Id, arr_index :: Exp
                     , value :: Exp}
    deriving (Eq, Show)
```

A value with a type must be associated with every expression. Note that in this scheme, a function call is considered an expression, and hence the only way to call a function (a naked expression) is through a S_Void statement, which is designed exactly for this purpose.

```
data Exp
    = B_Op Op Exp Exp
    | E_Index { array_exp, index_exp :: Exp }
    | Length Exp
    | Call { class_ :: Id, callee :: Exp, method :: Id, args :: [Exp] }
    | Function Id [Id] Exp
    | E_Int Int
    | E_false
    | E_true
    | E_Id Id Id
    | E_this
    | E_Not Exp
    deriving (Eq, Ord, Show)
```

The Op data type covers binary operators. The parser is capable of parsing these, but in this revision, LessThan and GreaterThan may not be implemented fully.

```
data Op
    = Add | Subtract | Multiply | Divide | LessThan
          | GreaterThan
    deriving (Eq, Ord, Show)
```

9 Parser

This module contains the functions for using the combinatoric parsers that the Parsec library offers. The parser is designed to go top-down.

```
module ParserMini where

import Text.Parsec
import Text.Parsec.String (Parser)
import Control.Applicative ((<$>))

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok
import Text.Parsec.Perm

import Lexert
import SyntaxMini
```

9.1 Types

Using the "do" notation, the parsing functions are quite readable. typeInt and typeId are trivial parsers that parse types, which are part of all expressions and variables.

```
typeInt :: Parser Type
typeInt = do
    reserved "int"
    return T_Int

typeId :: Parser Type
typeId = do
    name <- identifier
    return $ T_Id name
```

The type_ function wraps all types and attempts to match the input to a previously defined type. Using the "try" notation, in the event the parser fails to match, the parser acts as though no input was consumed, effectively rolling back infinitely.

```
type_ :: Parser Type
type_ = do
        try typeInt
    <|> try typeId
```

9.2 Expressions

Binary operators are implemented via a table, which is then passed to Parsec. The reason that this table is built is due to Parsec being unable to handle left recursive grammars. By creating this table, Parsec can rewrite the grammar internally to a non-left recursive one.

```
binary s f assoc
    = Ex.Infix (reservedOp s >> return (B_Op f)) assoc

binops = [[binary "*" Multiply Ex.AssocLeft
          ,binary "/" Divide Ex.AssocLeft]
         ,[binary "+" Add Ex.AssocLeft
          ,binary "-" Subtract Ex.AssocLeft]
         ,[binary "<" LessThan Ex.AssocLeft
          ,binary ">" GreaterThan Ex.AssocLeft]]
```

The <?>operator is used to provide more meaningful error messages in case invald syntax is detected during parsing. In this case, if an expression is expected during parsing, but is not matched, the parser will state that an expression is expected.

```
expr :: Parser Exp
expr =  Ex.buildExpressionParser binops factor
    <?> ("Expression")
```

For each type constructor defined for the Exp data type, a corresponding expression parser must exist. Each parser must return the data type specified in the Parser type constructor in the function signature definition. The "return" at the end of each parser returns a complete data type that is not modified any more in the parsing phase.

```
exprInt :: Parser Exp
exprInt = do
    n <- integer
    return $ E_Int (fromInteger n)
```

This parser attempts to match an optional class identifier followed be a "." before a variable identifier. This is needed for class member referencing.

```
exprVar :: Parser Exp
exprVar = do
    classId <- option "" (try (do
        classId <- identifier
        reservedOp "."
        return classId))
    n <- identifier
    return $ E_Id classId n

call :: Parser Exp
call = do
    classId <- option "" (try (do
        classId <- identifier
        reservedOp "."
        return classId))
    name <- identifier
    args <- parens $ commaSep expr
    return $ Call classId (E_Id classId "") name args --todo fix


returnStatement :: Parser Exp
returnStatement = do
    reserved "return"
    value <- expr
    reservedOp ";"
    return value
```

Only the types of expressions referenced by the factor function are accepted currently. returnStatement is not included here as it is explicitly used in the methodDeclaration.

```
factor :: Parser Exp
factor = try exprInt
      <|> try call
      <|> try exprVar
      <|> (parens expr)
```

9.3 Statements

Statements are handled by these functions. All statements are followed by the semicolon reserved operator.

```
assign :: Parser Statement
assign = do
    classId <- option "" (try (do
        classId <- identifier
        reservedOp "."
        return classId))
    name <- identifier
    reservedOp "="
    val <- expr
    reservedOp ";"
    return $ S_Assign name classId val

printst :: Parser Statement
printst = do
    reserved "print"
    exp <- parens expr
    reservedOp ";"
    return $ S_Print exp
```

The void statement is a wrapper for an expression. This allows functions calls to be made without being part of another statement for example.

```
voidst :: Parser Statement
```

```
voidst = do
    exp <- expr
    reservedOp ";"
    return $ S_Void exp

ifStatement :: Parser Statement
ifStatement = do
    reserved "if"
    cond <- parens expr
    reserved "then"
    tr <- statement
    reserved "else"
    fl <- statement
    return $ S_If cond tr fl

whileStatement :: Parser Statement
whileStatement = do
    reserved "while"
    cond <- parens expr
    body <- statement
    return $ S_While cond body

block :: Parser Statement
block = do
    s <- braces $ many statement
    return $ S_Block s

statement :: Parser Statement
statement = try block
    <|> try assign
    <|> try printst
    <|> try voidst
    <|> try ifStatement
    <|> try whileStatement
```

9.4 Member Declarations

The variable parser simply matches a type followed by an identifier. It is used by many other parsers. The fieldDeclaration parser is responsible for matching class member declarations. Although "static" is matched against, it holds no special meaning as there is no corresponding code generation feature for it.

```
variable :: Parser VarDecl
variable = do
    t <- type_
    name <- identifier
    return $ VarDecl t name

fieldDeclaration :: Parser VarDecl
fieldDeclaration = do
    optional (reserved "static")
    var <- variable
    reservedOp ";"
    return var <?> ("field declaration")
```

Method generation parser is slightly more complex than the other parsers due to a method having more parts to it than regular statements.

```
methodDeclaration :: Parser MethodDecl
methodDeclaration = do
    optional (reserved "static")
```

First we attempt to match a type and a name,

```
    t       <- type_
    name    <- identifier
```

then a comma delimited array of variables surrounded by parenthesis,

```
    args    <- parens $ commaSep variable
```

and finally the method body definition, which starts and ends with curly braces.

```
    (vars, stats, ret) <- braces (do {
```

I experimented with a permutation parser to try and allow arbitrary interspersion of local variable declarations to little success. In the end I decided to enforce all local declarations to be at the top of the method body.

```
          vars    <- many $ try ( do { v <- variable;
                                       reservedOp ";";
                                       return v
                                  })
        ; stats   <- many statement
```

Every method body must contain exactly one return statement. This makes parsing much easier, but may make some function design awkward. Perhaps a "goto" construct in the language could help with this.

```
        ; ret     <- returnStatement
        ; return (vars, stats, ret)
    })
    return $ MethodDecl t name args vars stats ret
    <?> ("method declaration")
```

9.5 Class Definitions

The classBody parser attempts to match all the field and method member declarations in a class definition. It is another instance of failed experimentation with permutation parsers. Hence, all field declarations must preceed method declarations.

```
classBody :: Parser ([VarDecl], [MethodDecl])
classBody = do
    fd <- option [] (many (try fieldDeclaration))
    md <- option [] (many (try methodDeclaration) )
    return (fd, md)
```

Every program consists of one or more class declarations.

```
classDeclaration :: Parser ClassDecl
classDeclaration = do
    reserved "class"
    name <- identifier
    (fd, md) <- braces classBody
    return $ ClassDecl name Nothing fd md

program :: Parser Program
program = do
    mc <- classDeclaration
    cs <- many classDeclaration
    return $ Program mc cs
```

9.6 Top Level

```
defn :: Parser Program
defn = program
```

This helper function removes starting whitespaces.

```
contents :: Parser a -> Parser a
contents p = do
  whitespace
  r <- p
  eof
  return r
```

These functions are the ones that start the top down parsing execution.

```
toplevel :: Parser Program
toplevel = do
    def <- defn
    return def

--This function is not used.
--parseExpr :: String -> Either ParseError Exp
--parseExpr s = parse (contents expr) "<stdin>" s

parseToplevel :: String -> Either ParseError Program
parseToplevel s = parse (contents toplevel) "<stdin>" s
```

10 Code Generation Backend

This module contains the backend for generating LLVM IR from the syntax tree. It wraps many llvm-general functions for use by the IR emitter.

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Codegen where
import Data.Word
import Data.String
import Data.List
import Data.Function
import qualified Data.Map as Map
import Control.Monad.State
import Control.Applicative
import LLVM.General.AST
import qualified LLVM.General.AST.Global as A.G
import qualified LLVM.General.AST as AST
import LLVM.General.AST.AddrSpace
import qualified LLVM.General.AST.Constant as C
import qualified LLVM.General.AST.Attribute as A
import qualified LLVM.General.AST.CallingConvention as CC
import qualified LLVM.General.AST.IntegerPredicate as IP
import qualified LLVM.General.AST.FloatingPointPredicate as FP
import qualified LLVM.General.AST.Linkage as L
import Debug.Trace
```

10.1 Module Level

Since LLVM has no support for keeping track of named class members, I had to use this hack to add this functionality onto the llvm-general module. It makes code rather ugly, but it works. The Module is what is translated by llvm-general and used to emit IR.

```haskell
data Module2 =
    Module2 { modul   :: AST.Module,
              classFT :: [ClassFieldTable] }

module2modul (Module2 m _) = m
module2Definitions (Module2 m _) = moduleDefinitions m
module2FTs (Module2 _ ft) = ft
```

The Module is wrapped in a state monad which changes as the syntax tree unravels.

```haskell
newtype LLVM a = LLVM { unLLVM :: State Module2 a }
  deriving (Functor, Applicative, Monad, MonadState Module2)
runLLVM :: Module2 -> LLVM a -> Module2
runLLVM = do
    flip (execState . unLLVM)
```

An empty module is used as an intial state.

```haskell
emptyModule :: String -> Module2
emptyModule label
  = Module2 { modul = defaultModule
                          { moduleName = label
                          , moduleDefinitions = []}
            , classFT = [] }
```

These external definitions are useful for debugging and testing the compiler.

```haskell
putcharDef = external (IntegerType 32) "putchar"
                      [(IntegerType 32, "i")]
printDef = external (IntegerType 32) "printf"
                    [(IntegerType 8, "i"), (IntegerType 8, "s")]
```

Any function definition, class definition is added to the module using either the define function or defineClassStruct function. Both of these functions then call the addDefn function, which modifies the excuting Module state.

```haskell
addDefn :: Definition -> LLVM ()
addDefn newdef = do
  defs <- gets module2Definitions
  (Module a b c d) <- gets module2modul
  modify $ \s -> s { modul = (Module a b c (defs ++ [newdef])) }
define ::  Type -> String -> [(Type, Name)] -> [BasicBlock]
        -> LLVM ()
define retty label argtys body = addDefn $
  GlobalDefinition $ functionDefaults {
    A.G.name        = Name label
  , A.G.parameters  = ( [Parameter ty nm [] | (ty, nm) <- argtys], False)
  , A.G.returnType  = retty
  , A.G.basicBlocks = body
  }
```

The external function handles external function prototypes like putchar.

```haskell
external ::  Type -> String -> [(Type, Name)] -> LLVM ()
external retty label argtys = addDefn $
  GlobalDefinition $ functionDefaults {
    A.G.name        = Name label
  , A.G.parameters  = ( [Parameter ty nm [] | (ty, nm) <- argtys], False)
  , A.G.returnType  = retty
  , A.G.basicBlocks = []
  }
```

Whenever a class is defined, a llvm struct type is created with the appropriate field members. Functions are assigned to a class via special compiler prefixes, or would be in a more complete version.

```
defineClassStruct :: Name -> [(Type, Name)] -> LLVM ()
defineClassStruct nm@(Name nn) vars = do
    fts <- gets module2FTs
    modify $ \s -> s { classFT = fts ++ [ft] }
```

In this version a global variable of the class type is declared automatically for each class definition. I have found this useful for debugging.

```
    addDefn $ ty
    addDefn $ GlobalDefinition $ globalVariableDefaults
        { A.G.name = AST.Name $ nn ++ "0"
        , A.G.type' = NamedTypeReference nm
        }
    where
        ty = TypeDefinition nm $ Just st
        st = (StructureType False $ map fst vars )
        vars2 = map (\(x, AST.Name n) -> (x, n)) vars
        ft = ClassFieldTable ty vars2
```

10.2 Names

Names are used to keep track of the symbols defined. This code is untouched from Stephen Diel's tutorials.

```
type Names = Map.Map String Int
uniqueName :: String -> Names -> (String, Names)
uniqueName nm ns =
  case Map.lookup nm ns of
    Nothing -> (nm,  Map.insert nm 1 ns)
    Just ix -> (nm ++ show ix, Map.insert nm (ix+1) ns)
instance IsString Name where
  fromString = Name . fromString
```

10.3 Codegen

A symbol table contains a list of id, type and operand pairs. Operand roughly maps to a reference to an instance of a symbol here.

```
type SymbolTable = [(String, (Type, Operand))]
```

ClassFieldTable is something I made up on the spot to keep track of which named member fields corresponded to which type in each class. It is necessary as LLVM does not keep track of named fields.

```
data ClassFieldTable
    = ClassFieldTable { ty :: AST.Definition
                      , fields :: [(AST.Type, String)] }
    deriving (Eq, Show)
getFTFields (ClassFieldTable _ fds) = fds
```

I did not look into BlockState, which worked fine straight out of Stephen Diel's tutorials. It is another data type used in managing the blocks that LLVM uses.

```
data BlockState
  = BlockState {
    idx   :: Int
  , stack :: [Named Instruction]
  , term  :: Maybe (Named Terminator)
} deriving Show
-- %%\ignore{cc = 5}
```

The data type CodegenState keeps track of many important code generation states(obviously). During the generation of code, each class has its own code generation state which keeps track of the LLVM blocks assigned, the symbol table, the class field table, and the current class and class instance. Like the LLVM Module state, the CodegenState mutates frequently as the program traverses the syntax tree.

```
data CodegenState
  = CodegenState {
    -- Name of the active block to append to
    currentBlock :: Name
    -- Blocks for function
  , blocks       :: Map.Map Name BlockState
  , symtab       :: SymbolTable                -- Function scope symbol table
  , blockCount   :: Int                        -- Count of basic blocks
  , count        :: Word                       -- Count of unnamed instructions
  , names        :: Names                      -- Name Supply
  , ft           :: [ClassFieldTable]
  , currentClass :: (String, Maybe Operand)
  } deriving Show

codeFieldTable :: CodegenState -> [ClassFieldTable]
codeFieldTable (CodegenState _ _ _ _ _ _ ft _) = ft
codeCurrentClass (CodegenState _ _ _ _ _ _ _ cl) = cl
```

Since each class declaration is passed its own CodegenState, a global table of all classes definitions must be passed to each CodegenState. This is a hack around global mutatable states in Haskell.

```
emptyCodegen :: [ClassFieldTable] -> String -> CodegenState
emptyCodegen cft clazz = CodegenState (Name entryBlockName)
                                      Map.empty [] 1 0
                                      Map.empty cft
                                      (clazz, Nothing)

execCodegen :: [ClassFieldTable]-> String -> Codegen a -> CodegenState
execCodegen cft clazz m = execState ( runCodegen m) $ emptyCodegen cft clazz

newtype Codegen a = Codegen{ runCodegen::State CodegenState a }
  deriving (Functor,Applicative,Monad,MonadState CodegenState )
```

10.4 Block Management

As stated before, this part of the code I did not touch much. These functions are used to manage blocks and reference them.

```
sortBlocks :: [(Name, BlockState)] -> [(Name, BlockState)]
sortBlocks = sortBy (compare `on` (idx . snd))

createBlocks :: CodegenState -> [BasicBlock]
createBlocks m = map makeBlock $ sortBlocks $ Map.toList (blocks m)

makeBlock :: (Name, BlockState) -> BasicBlock
makeBlock (l, (BlockState _ s t)) = BasicBlock l s (maketerm t)
  where
    maketerm (Just x) = x
    maketerm Nothing = error $ "Block has no terminator: "++ (show l)

entryBlockName :: String
entryBlockName = "entry"

emptyBlock :: Int -> BlockState
emptyBlock i = BlockState i [] Nothing

entry :: Codegen Name
```

```haskell
entry = gets currentBlock

addBlock :: String -> Codegen Name
addBlock bname = do
  bls <- gets blocks
  ix <- gets blockCount
  nms <- gets names
  let new = emptyBlock ix
      (qname, supply) = uniqueName bname nms
  modify $ \s -> s { blocks = Map.insert (Name qname) new bls
                   , blockCount = ix + 1
                   , names = supply
                   }
  return (Name qname)

setBlock :: Name -> Codegen Name
setBlock bname = do
  modify $ \s -> s { currentBlock = bname }
  return bname

getBlock :: Codegen Name
getBlock = gets currentBlock

modifyBlock :: BlockState -> Codegen ()
modifyBlock new = do
  active <- gets currentBlock
  modify $ \s -> s { blocks = Map.insert active new (blocks s) }

current :: Codegen BlockState
current = do
  c <- gets currentBlock
  blks <- gets blocks
  case Map.lookup c blks of
    Just x -> return x
    Nothing -> error $ "No such block: " ++ show c
```

10.5 Instructions

All LLVM instructions results are named and stored in virtual registers, which LLVM keeps track of. Fresh keeps track of the current unnamed number in a block.

```haskell
fresh :: Codegen Word
fresh = do
  i <- gets count
  modify $ \s -> s { count = 1 + i }
  return $ i + 1
```

This function wraps any LLVM instruction and does the necessary block manipulation .

```haskell
instr :: Instruction -> Codegen (Operand)
instr ins = do
  n <- fresh
  let ref = (UnName n)
  blk <- current
  let i = stack blk
  modifyBlock (blk { stack = i ++ [ref := ins] } )
  return $ local ref
```

The terminator instruction indicates the end of a block, which corresponds to a method.

```haskell
terminator :: Named Terminator -> Codegen (Named Terminator)
terminator trm = do
  blk <- current
  modifyBlock (blk { term = Just trm })
  return trm
```

10.6 Symbol Table

These functions are associated with retrieving, or modifying the symbol table elements. Their purposes should be self evident.

```haskell
assign :: String -> Type -> Operand -> Codegen ()
assign var ty x = do
  lcls <- gets symtab
  modify $ \s -> s { symtab = [(var, (ty, x))] ++ lcls }
getvar :: String -> Codegen (Type, Operand)
getvar var = do
  syms <- gets symtab
  case lookup var syms of
    Just x  -> return x
    Nothing -> error $ "Local variable not in scope: "++show var
```

10.7 References

These are just wrappers to LLVM references.

```haskell
local ::  Name -> Operand
local = LocalReference
global ::  Name -> C.Constant
global = C.GlobalReference
externf :: Name -> Operand
externf = ConstantOperand . C.GlobalReference
```

10.8 Arithmetic and Contants

```haskell
icmp :: IP.IntegerPredicate -> Operand -> Operand -> Codegen Operand
icmp cond a b = instr $ ICmp cond a b []
iadd :: Operand -> Operand -> Codegen Operand
iadd a b = instr $ Add False False a b []
isub :: Operand -> Operand -> Codegen Operand
isub a b = instr $ Sub False False a b []
imul :: Operand -> Operand -> Codegen Operand
imul a b = instr $ Mul False False a b []
idiv :: Operand -> Operand -> Codegen Operand
idiv a b = instr $ SDiv False a b []
fadd :: Operand -> Operand -> Codegen Operand
fadd a b = instr $ FAdd a b []
fsub :: Operand -> Operand -> Codegen Operand
fsub a b = instr $ FSub a b []
fmul :: Operand -> Operand -> Codegen Operand
fmul a b = instr $ FMul a b []
fdiv :: Operand -> Operand -> Codegen Operand
fdiv a b = instr $ FDiv a b []
fcmp :: FP.FloatingPointPredicate -> Operand -> Operand -> Codegen Operand
fcmp cond a b = instr $ FCmp cond a b []
cons :: C.Constant -> Operand
cons = ConstantOperand
uitofp :: Type -> Operand -> Codegen Operand
uitofp ty a = instr $ UIToFP a ty []
```

10.9 Side effects

These functions include calling functions, allocation of stack memory, and loading and storing values to and from registers.

```
toArgs :: [Operand] -> [(Operand, [A.ParameterAttribute])]
toArgs = map (\x -> (x, []))
call :: Operand -> [Operand] -> Codegen Operand
call fn args = instr $ Call False CC.C [] (Right fn) (toArgs args) [] []
alloca :: Type -> Codegen Operand
alloca ty = instr $ Alloca ty Nothing 0 []
store :: Operand -> Operand -> Codegen Operand
store ptr val = instr $ Store False ptr val Nothing 0 []
load :: Operand -> Codegen Operand
load ptr = instr $ Load False ptr Nothing 0 []
```

10.10 Control Flow

These functions include breaks, conditional breaks, and keeping track of control statement labels.

```
br :: Name -> Codegen (Named Terminator)
br val = terminator $ Do $ Br val []
cbr :: Operand -> Name -> Name -> Codegen (Named Terminator)
cbr cond tr fl = terminator $ Do $ CondBr cond tr fl []
phi :: Type -> [(Operand, Name)] -> Codegen Operand
phi ty incoming = instr $ Phi ty incoming []
ret :: Operand -> Codegen (Named Terminator)
ret val = terminator $ Do $ Ret (Just val) []
```

11 Code Emitter

This module contains the front end code to travel the syntax tree and emit the corresponding IR.

```
{-# LANGUAGE OverloadedStrings #-}
module Emit where

import LLVM.General.Module
import LLVM.General.Context

import qualified LLVM.General.AST as AST
import qualified LLVM.General.AST.Constant as C
import qualified LLVM.General.AST.Float as F
import qualified LLVM.General.AST.FloatingPointPredicate as FP
import qualified LLVM.General.AST.IntegerPredicate as IP

import Data.Word
import Data.Int
import Data.List
import Control.Monad.State
import Control.Monad.Error
import Control.Applicative
import qualified Data.Map as Map

import Debug.Trace

import Codegen
import qualified SyntaxMini as S
```

11.1 Compilation

These functions start the process in which the syntax tree data is transformed into LLVM module data.

```
liftError :: ErrorT String IO a -> IO a
liftError = runErrorT >=> either fail return
```

The codegen function takes a module and a syntax tree and wraps it in an error handling context and executes the llvm-general Module to IR generator.

```
codegen :: Module2 -> S.Program -> IO (Module2, String)
codegen mod fns = withContext $ \context ->
  liftError $ withModuleFromAST context newast $ \m -> do
    llstr <- moduleLLVMAssembly m
    putStrLn llstr
    return (mimi, llstr)
  where
    modn    = codegenTop fns
    mimi@(Module2 newast _ ) = runLLVM mod modn
```

11.2 Top Down Generation

Top down traversal starts here. The codegenClass function runs on each class definition in the program.

```
codegenTop :: S.Program -> LLVM ()
codegenTop prog = trace "entering codegen" $ do
    putcharDef
    mc <- codegenClass True (S.progMainClass prog)
    cs <- mapM (codegenClass False) (S.progClassDecls prog)
    return ()
```

Here we call the function that declares an LLVM struct with the appropriate fields, and run codegenMethod on the method declarations in the class.

```
codegenClass :: Bool -> S.ClassDecl -> LLVM ()
codegenClass isMain (S.ClassDecl name _ fd md) = do
    ft <- gets module2FTs
    (res1) <- defineClassStruct (AST.Name name) (map (varDeclTuple ft) fd)
    mapM (codegenMethod name) md
    return ()
```

Method codegen is more involved. On the highest level, we define a function after retrieving the type data, the function id, and the transformed syntax data types to types LLVM recognizes. A code block must also be specified.

```
codegenMethod :: String -> S.MethodDecl -> LLVM ()
codegenMethod clazz (S.MethodDecl ty name args decl body retty) = do
    ft <- gets module2FTs
    define (typeToAST ty ft)
           (classFunc clazz name)
           (map (varDeclTuple ft) ([args2]++args))
           (blks ft)
    return ()
        where
        args2 = (S.VarDecl (S.T_Id clazz) "this")
```

This is the subfunction that creates the block containing the function logic. First we initialize the block with entry label.

```
        blks dd = createBlocks $ execCodegen dd clazz $ do
          entry <- addBlock entryBlockName
          setBlock entry
```

Each class function takes a reference to itself as the first argument. This would not be the case for static functions, but they are not implemented right now. A copy of each argument is made in order to implement pass-by-value behavior.

```
          thisop <- alloca $ fst $ param dd args2
          store thisop (local $ AST.Name $ vn args2)
          assign (vn args2) (fst (param dd args2)) thisop
```

For each argument, we do the same as above.

```
          forM args $ \a -> do
            var <- alloca $ fst $ param dd a
            store var (local $ AST.Name $ vn a)
            assign (vn a) (fst (param dd a)) var
```

Now we call the functions that generate code for each local declaration, and make the adjustments in the symbol table.

```
          rr <- mapM (cgenVarDecl) decl
          sss <- gets symtab
```

We also need to pass information about the current class to the code responsible for generating child nodes of the syntax tree.

```
          (curTy, _ ) <- gets codeCurrentClass
          modify $ \s -> s{ currentClass= (curTy, Just thisop) }
```

Then we generate the method body which consists of an array of statements, and the return expression.

```
          res <- mapM (cgenStatement) body
          resret <- cgenExp retty
          ret $ resret
        vn a = S.varName a
        param dd a = varDeclTuple dd a
```

11.3 Statement Codegen

Statement code generation is the next step in the top down process. A block statement simply does a recursive call on each of its statement elements.

```
cgenStatement :: S.Statement -> Codegen AST.Operand
cgenStatement (S.S_Block ss) = do
    res <- mapM cgenStatement ss
    return $ last res
```

In the case the lval is not prefixed with an identifier, the assignment statement either finds a local reference matching the lval, or a class field matching the lval,or returns an error. If there is a class identifier, then there is no ambiguity. The function classFieldPtr is defined later, which we will see calculates the offset of the struct field to assign to.

```
cgenStatement (S.S_Assign id classId val) = do
    valop <- cgenExp val
    syms <- gets symtab
    case classId of
        ""   -> case (lookup id syms) of
                    Just (symty, symop)  -> do
                        store symop valop
                        return symop
                    Nothing -> do
                        ptrop <- classFieldPtr classId id
                        store ptrop valop
        cid  -> do
                    ptrop <- classFieldPtr classId id
                    store ptrop valop
cgenStatement (S.S_Print e) = do
```

```
    res <- cgenExp e
    call (externf (AST.Name "putchar")) [res]
    return res
```

To implement an if statement, 3 blocks are needed.

```
cgenStatement (S.S_If cond t e) = do
    ifthen <- addBlock "if.then"
    ifelse <- addBlock "if.else"
    ifexit <- addBlock "if.exit"
```

First we calculate which branch to jump to by calculating the value of the condition expression. Note there are no type checks so we do not know if the condition is even valid or not. An integer value 0 corresponds to FALSE, and all other values are TRUE.

```
    --Entry
    cond <- cgenExp cond
    test <- icmp IP.NE (AST.ConstantOperand (C.Int 32 0)) cond
    cbr test ifthen ifelse
```

For each branch, we create a label and run codegen on the conditional statements in the branch.

```
    --if.then
    setBlock ifthen
    trval <- cgenStatement t
    br ifexit
    ifthen <- getBlock

    --if.else
    setBlock ifelse
    flval <- cgenStatement e
    br ifexit
    ifelse <- getBlock

    --ifexit
    setBlock ifexit
```

phi is used to keep track of which block we came from and the values stored in registers.

```
    phi (AST.IntegerType 32) [(trval, ifthen), (flval, ifelse)]
cgenStatement (S.S_Return exp) = do
    e <- cgenExp exp
    t <- ret e
    return e
cgenStatement (S.S_Void exp) = do
    e <- cgenExp exp
    return e
```

11.4 Expression Codegen

An expression which is just a variable is very much like the assign lval resolution code. We need to figure out which reference an id refers to.

```
cgenExp :: S.Exp -> Codegen AST.Operand
cgenExp (S.E_Int n)
    = return $ cons $ (C.Int 32 (fromIntegral n))
cgenExp (S.E_Id classId id) = do
    syms <- gets    symtab
    case classId of
        ""   -> case (lookup id syms) of
                    Just (symty, symop)  -> do
                        load symop
                    Nothing -> do
                        ptrop <- classFieldPtr classId id
```

```
                        load ptrop
            cid   -> do
                    ptrop <- classFieldPtr classId id
                    load ptrop
```

A function does not know by itself to which class it belongs. This is why we must extract the class id from the call syntax node.

```
cgenExp (S.Call cid callee fn args) = do
    syms <- gets symtab
    largs <- mapM cgenExp args
    objs <- case (lookup cid syms) of
            Just (cty, cop) -> do
                ll <- load cop
                return [ll]
            Nothing -> return []

    call (externf (AST.Name (classFunc cid fn))) $ objs++largs
```

Binary expression generation calls code generation on each of the operands to get references to their values, and then calls the operator function passing the two operand values.

```
cgenExp (S.B_Op op a b) = do
  case Map.lookup op binops of
    Just f   -> do
      ca <- cgenExp a
      cb <- cgenExp b
      f ca cb
    Nothing -> error "No such operator"
```

cgenVarDecl handles code generation for local declarations.

```
cgenVarDecl :: S.VarDecl -> Codegen ()
cgenVarDecl vd = do
    ft <- gets codeFieldTable
    let (typ, AST.Name nm) = varDeclTuple ft vd
    newvar <- alloca typ
    lcls <- gets symtab
    modify $ \s -> s { symtab = [(nm, (typ, newvar))] ++ lcls }
```

This function returns a reference to a structure field given an object id and an offset.

```
structFieldFromOff :: AST.Operand -> Int -> Codegen AST.Operand
structFieldFromOff ty off
    = do
          res <- instr $ AST.GetElementPtr
                True
                ty
                [ AST.ConstantOperand $ C.Int 32 0
                , AST.ConstantOperand $ C.Int 32 (fromIntegral off)]
                []
          return res
```

classFieldPtr returns the reference to a structure field given an object id and field id by doing a lookup on the class field table. I certainly will not winning any functional elegance awards for this one.

```
classFieldPtr :: String -> String -> Codegen AST.Operand
classFieldPtr classId fieldId = do
    ft <- gets codeFieldTable
```

First we lookup the object id to see whether or not the instance registered in the class field table. We attempt to discern the class type from this lookup.

```
    (ctynm, currClazzOp) <- case classId of
        ""      -> gets codeCurrentClass
        cid     -> do
            syms <- gets symtab
            case lookup classId syms of
                Just (tty, top)
                    -> return $ (findTypeName ft tty, Just top)
                Nothing
                    -> error $ "No local object named "
                        ++ classId
```

Then we lookup to see if the named field exists inside the class definition.

```
    case currClazzOp of
        Nothing -> return $ error $ "No class field named"++fieldId++" in object "++classId
        Just cop -> do
            let clazzFieldTable = findCurrentClassTable ft ctynm
                in case (clazzFieldTable) of
```

Finally, we find the offset of the field in the class struct type and return the reference to the field.

```
                    Just cc@(ClassFieldTable (AST.TypeDefinition nm (Just ty)) fields)
                        -> case (findIndexOfField cc fieldId) of
                              (Just fieldty, Just n)
                                           -> structFieldFromOff cop $ findOffestOfField ty n
                              (_, Nothing) -> do error ("In class, symbol with id not defined:"
                                                       ++ctynm ++ "."
                                                       ++ fieldId)
                    Nothing ->  do error $ "Symbol with id not defined: " ++ ctynm
                                        ++ "." ++ fieldId
```

This methods returns the name of a class type.

```
findTypeName :: [ClassFieldTable] -> AST.Type -> String
findTypeName ft ty = do
    case (find (\(ClassFieldTable (AST.TypeDefinition nm (Just td)) _) -> td == ty) ft) of
        Just (ClassFieldTable (AST.TypeDefinition (AST.Name nm) (Just td) ) _ ) -> nm
        Nothing -> ""
```

This methods looks up the class field table given a class name from a list of class field tables. It is basically a convenience function to search the dictionary.

```
findCurrentClassTable :: [ClassFieldTable] -> String-> Maybe ClassFieldTable
findCurrentClassTable fts cur
    = find (\(ClassFieldTable (AST.TypeDefinition nm _) _) -> nm == (AST.Name cur)) fts
```

The rest of these functions are convenience functions to calculate field offsets, and find type from names.

```
findIndexOfField :: ClassFieldTable -> String -> (Maybe AST.Type, Maybe Int)
findIndexOfField (ClassFieldTable _ fields) fd
    = ( liftM fst $ find matchName fields
      , findIndex matchName fields)
    where matchName = (\(ty, nm) -> nm == fd)

findOffestOfField :: AST.Type -> Int -> Int
findOffestOfField (AST.StructureType _ tys) idx
    = foldr (\x acc -> acc + sizeofType x) 0 $ take idx tys

sizeofType :: AST.Type -> Int
sizeofType (AST.IntegerType 32) = 1
sizeofType (AST.StructureType _ tys) = sum $ map sizeofType tys
--sizeofType x = error $ show x

typeToAST :: S.Type -> [ClassFieldTable] -> AST.Type
typeToAST (S.T_Int) _ = AST.IntegerType 32
```

```
typeToAST (S.T_Id id) ft = case (findTypeFromModule id ft) of
    Nothing -> error $ "No type exists " ++ id ++ (show ft)
    Just t  -> t

findTypeFromModule :: String -> [ClassFieldTable] -> Maybe AST.Type
findTypeFromModule nm ft =
    case typedef of
        Nothing -> Nothing
        Just (ClassFieldTable (AST.TypeDefinition _ mty) _) -> mty
    where
        typedef = find (\def -> case def of
            ClassFieldTable (AST.TypeDefinition (AST.Name id) mt) _-> id == nm -> False) ft
```

varDeclTuple is a useful function to convert a variable declaration syntax node to a tuple llvm-general can use.

```
varDeclTuple :: [ClassFieldTable] -> S.VarDecl -> (AST.Type, AST.Name)
varDeclTuple ft (S.VarDecl ty nm)= (typeToAST ty ft, AST.Name nm)

-- This function does basically nothing.
classFunc :: String -> String -> String
classFunc nm func = func
```

11.5 Binary Operator Codegen

```
lt :: AST.Operand -> AST.Operand -> Codegen AST.Operand
lt a b = do
  test <- icmp IP.SLT a b
  uitofp (AST.IntegerType 32) test

gt :: AST.Operand -> AST.Operand -> Codegen AST.Operand
gt a b = do
  test <- icmp IP.SGT a b
  uitofp (AST.IntegerType 32) test

binops = Map.fromList [
      (S.Add, iadd)
    , (S.Subtract, isub)
    , (S.Multiply, imul)
    , (S.Divide, idiv)
    , (S.LessThan, lt)
    , (S.GreaterThan, gt)
  ]
```

12 Main Program

This module is rather unremarkable. It simply runs the parser and starts codegeneration. It operates in two modes, line by line or file input.

```
module Main where

import ParserMini
import Codegen
import Emit

import Control.Monad.Trans

import System.IO
import System.Environment
import System.Console.Haskeline

import qualified LLVM.General.AST as AST
```

```haskell

initModule :: Module2
initModule = emptyModule "my cool compiler"

process :: Module2 -> String -> IO (Maybe Module2, String)
process modo source = do
  let res = parseToplevel source
  case res of
    Left err -> print err >> return (Nothing, "")
    Right ex -> do
      (ast, out) <- codegen modo ex
--        tt <- mapM (print . show) $ module2Definitions modo
      return (Just ast,out)

repl :: IO ()
repl = runInputT defaultSettings (loop initModule)
  where
  loop mod = do
    minput <- getInputLine "ready> "
    case minput of
      Nothing -> outputStrLn "Goodbye."
      Just input -> do
        (modn, _) <- liftIO $ process mod input
        case modn of
          Just modn -> loop modn
          Nothing -> loop mod

processFile :: String -> IO (Maybe Module2)
processFile fname = do
    inp <- readFile fname
    (m, out) <- process initModule inp
    writeFile (fname ++ ".ll") out
    return m

main :: IO ()
main = do
  args <- getArgs
  case args of
    []      -> repl
    [fname] -> processFile fname >> return ()
```

13 Concluding Thoughts

I think I have learned quite a bit writing this compiler. Everything was a bit new to me, even Haskell itself to some extent. There were many potential features that could have been implemented, but all in all, I think this project will serve as a reasonble reference to other undergraduate students taking compilers.

References

[dragon(2007)] *Compilers: principles, techniques, and tools 2nd ed 2007*, Addison-Wesley Longman Publishing Co., Alfred V. Aho, Ravi Sethi, & Jeffrey D. Ullman.

[lbac(1995)] *Let's Build A Compiler 1995*, I.E.C.C., Jack Crenshaw.

[ssa(2014)] *Static single assignment form 2014*, Wikipedia, `http://en.wikipedia.org/wiki/Static_single_assignment_form`

[llvm(2014)] *LLVM Bitcode File Format 2014*, LLVM Reference, `http://llvm.org/docs/BitCodeFormat.html`

[llvm(2014)] *The LLVM Compiler Infrastructure 2014*, LLVM Site, `http://llvm.org/`