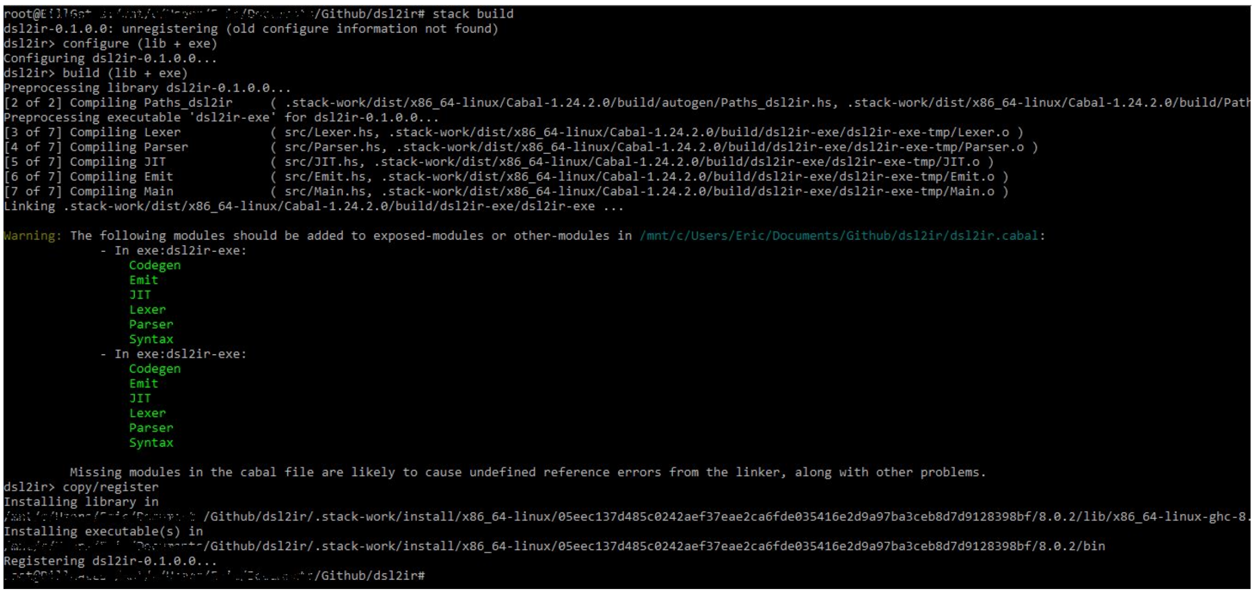
Technical Document - Threesum Language Compiled with Haskell and LLVM

Axel Zuchovicki	A01022875
Kyun-tak Woo	A01372055
Eric Parton	A01023503

Installation and usage

1. Before starting, you should install LLVM 4.0, GHC 7.8, and Stack
2. Download or clone the project from Github
3. Navigate to the project directory in your terminal
4. Run `stack build` to compile the files



5. You can now run files using our compiler with the following command:
   `stack exec dsl2ir-exe <file path>`
   Here's an example using the esotericTest file we included in the src folder, at the end of the code you can see the file is evaluated to 3 as the return value comes only from the final function. (image on the next page)
   Alternatively, you can simply run `stack exec dsl2ir-exe` to permit individual command line instructions (thanks JIT compilation, image on the next, next page)

```
[redacted]/Github/dsl2ir# stack exec dsl2ir-exe src/esotericTest.3sum
; ModuleID = 'dsl2ir jit'
source_filename = "<string>"

; Function Attrs: norecurse nounwind readnone
define double @"binary:"(double %x, double returned %y) local_unnamed_addr #0 {
entry:
  ret double %y
}

; Function Attrs: norecurse nounwind readnone
define double @newtest(double %x) local_unnamed_addr #0 {
entry:
  %0 = fdiv double %x, 3.000000e+00
  %1 = tail call double @"binary:"(double undef, double %0)
  ret double %1
}

; Function Attrs: norecurse nounwind readnone
define double @newesttest(double %x) local_unnamed_addr #0 {
entry:
  %0 = fcmp ule double %x, 2.000000e+00
  %x. = select i1 %0, double %x, double 1.000000e+00
  ret double %x.
}

; Function Attrs: norecurse nounwind readnone
define double @testWhileMultUntil(double %x) local_unnamed_addr #0 {
entry:
  %0 = fcmp ult double %x, 2.900000e+01
  br i1 %0, label %while.loop.preheader, label %while.exit

while.loop.preheader:                             ; preds = %entry
  br label %while.loop

while.loop:                                       ; preds = %while.loop.preheader, %while.loop
  %.01 = phi double [ %1, %while.loop ], [ %x, %while.loop.preheader ]
  %1 = fmul double %.01, 2.000000e+00
  %2 = fcmp ult double %1, 2.900000e+01
  br i1 %2, label %while.loop, label %while.exit.loopexit

while.loop.preheader:                             ; preds = %entry
  br label %while.loop

while.loop:                                       ; preds = %while.loop.preheader, %while.loop
  %.01 = phi double [ %1, %while.loop ], [ %x, %while.loop.preheader ]
  %1 = fmul double %.01, 2.000000e+00
  %2 = fcmp ult double %1, 2.900000e+01
  br i1 %2, label %while.loop, label %while.exit.loopexit

while.exit.loopexit:                              ; preds = %while.loop
  br label %while.exit

while.exit:                                       ; preds = %while.exit.loopexit, %entry
  %.0.lcssa = phi double [ %x, %entry ], [ %1, %while.exit.loopexit ]
  %3 = tail call double @"binary:"(double undef, double %.0.lcssa)
  ret double %3
}

; Function Attrs: norecurse nounwind readnone
define double @main() local_unnamed_addr #0 {
entry:
  %0 = tail call double @newtest(double 9.000000e+00)
  ret double %0
}

attributes #0 = { norecurse nounwind readnone }

Evaluated to: 3.0
[redacted]/Github/dsl2ir#
```

```
[redacted]/github/dsl2ir# stack exec dsl2ir-exe
dsl2ir> _____ unary!(x) ______ x _ 0 _ 1;
; ModuleID = 'dsl2ir jit'
source_filename = "<string>"

; Function Attrs: norecurse nounwind readnone
define double @"unary!"(double %x) local_unnamed_addr #0 {
entry:
  %0 = fcmp ueq double %x, 0.000000e+00
  %. = select i1 %0, double 1.000000e+00, double 0.000000e+00
  ret double %.
}

attributes #0 = { norecurse nounwind readnone }

dsl2ir> !1;
; ModuleID = 'dsl2ir jit'
source_filename = "<string>"

; Function Attrs: norecurse nounwind readnone
define double @"unary!"(double %x) local_unnamed_addr #0 {
entry:
  %0 = fcmp ueq double %x, 0.000000e+00
  %. = select i1 %0, double 1.000000e+00, double 0.000000e+00
  ret double %.
}

; Function Attrs: norecurse nounwind readnone
define double @main() local_unnamed_addr #0 {
entry:
  %0 = tail call double @"unary!"(double 1.000000e+00)
  ret double %0
```

Compiler

Description

The challenge was to use a purely functional programming language and its benefits to implement a compiler from the lexer and parser all the way to the intermediate representation (IR) utilized by LLVM. There are multiple reasons to why we want to produce LLVM IR but some of them could be the optimization passes that LLVM offers, the wide range of Backends that LLVM supports and can generate code for, along with a complete toolchain to analyze, experiment and optimize the back end for. It is important to remember that LLVM IR uses "unlimited single-assignment register machine instruction set". This means that despite CPUs having a fixed number of registers, LLVM IR has an infinite number and new registers are created to hold the result of every instruction. This also leads us to use Static Single Assignment (SSA) as registers may be assigned to only once which may cause a lot of redundant memory operations but this is solved by the use of Scalar Replacement of Aggregates (SROA) to clean it up[1].

JIT Compilation

This is a JIT (Just-In-Time) compiler meaning that the program is compiled at run-time. The benefit of this is that code can be compiled more efficiently since our compiler can have more pertinent and recent data than a non-JIT compiler that compiles all the code before start time. This also means that our JIT compilers tend to be more resource efficient than non-JIT Compilers

Language Esotericism

Taking inspiration from esoteric languages such as Brainfuck and LOLCODE, we have also added some esoteric touches of our own to Threesum. These deviations are described in the Esoteric Components section of this document.

---

[1] https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf

Grammar Used

Before we move into explaining how our compiler works, it would be useful to understand our grammar. In Threesum, being a functional/procedural language, everything is evaluated to a floating point. Having that in mind we should have only an initial (EXP) that would always evaluate to a floating point. We ended up with a really simple grammar reflecting this.

Without keyword estorism

Since we built an esoteric language, it might be difficult to understand the grammar. Therefore we want to first present its grammar without the esoteric keywords, and later on will correct the definitions with the esoteric keywords.

EXP ⇒ EXTERN; | FUNCTION; | UNARYDEF; | BINARYDEF; | EXPR;
EXPR ⇒ FLOATING | INT | ESOTERICINT | CALL | VARIABLE | IFCLAUSE | LETINS | FOR | WHILE | (EXPR) | UNOP EXPR | EXPR BINOP EXPR

EXTERN ⇒ extern STRING (IDS)
IDS ⇒ epsilon | STRING IDS
STRING ⇒ "any valid string"

FUNCTION ⇒ def STRING (IDS) EXPR

UNARYDEF ⇒ def unary OP (IDS) EXPR
OP ⇒ "single char op"

BINARYDEF ⇒ def binary OP (IDS) EXPR

FLOATING ⇒ "floating number"
INT ⇒ "integer"
ESOTERICINT ⇒ "base 3 integer"

CALL ⇒ STRING (EXPRS)
EXPRS ⇒ epsilon | EXPR COMMAEXPR
COMMAEXPR ⇒ epsilon | , EXPR

VARIABLE ⇒ STRING

IFCLAUSE ⇒ if EXPR then EXPR else EXPR

LETINS ⇒ var LETBODY
LETBODY ⇒ STRING = EXPR NEXTLET
NEXTLET ⇒ epsilon | , LETBODY

FOR ⇒ for STRING = EXPR , EXPR , EXPR in EXPR
WHILE ⇒ while EXPR in EXPR

*BINOP ⇒ = | * | / | + | - | < | > | == | != | >= | <= | && | “||”*
note: the amounts of operations may change since the user might declare their own.

UNOP ⇒ defined unary op
note: We don’t define any unary operation, but the user is able to do it in a program, and use it in the language.

With keyword estorism

The grammar from above is the same, but we are substituting the keywords with underscores as shown below. The idea is that the number of underscores will determine the path, and even though some keywords share the same amount of underscores, we made sure that it wouldn’t make the grammar ambiguous.

FUNCTION ⇒ _____ STRING (IDS) EXPR
UNARYDEF ⇒ _____ unary OP (IDS) EXPR
BINARYDEF ⇒ _____ binary OP (IDS) EXPR

LETINS ⇒ ____ LETBODY
IFCLAUSE ⇒ ______ EXPR _ EXPR _ EXPR
FOR ⇒ __ STRING = EXPR , EXPR , EXPR __ EXPR
WHILE ⇒ ___ EXPR __ EXPR

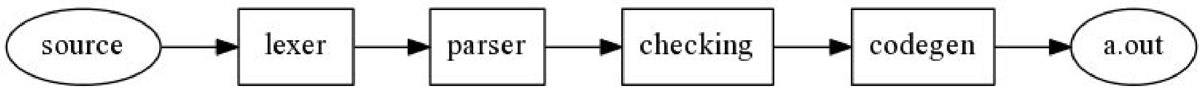
We explore the keywords more in-depth later on.

Now that we understand our grammar used, we can move on to explain how our compiler was made.

Compilation Phases

Overall View

Overall, the project will consist in the use of Haskell to produce all of the following steps leading to IR which is in the middle of the following chain.



The output of our project will create either a “.bc” (bitcode format) file or a “.ll” (assembly) flavors which can be converted between the two forms using llvm-dis.

Lexical and Syntactic Analysis

For the Lexer and Parser and AST, we used a family of Haskell libraries called *Parser Combinators*[2] which generate grammars in a very similar manner as a BNF (Backus-Naur Form). We started off with a very simple lexical syntax supporting floats and identifiers which we will later extend. It is a **procedural language.**

Here’s an example for both a custom lexer and its parser:

```
esotericInteger_ l = Tok.lexeme l int3 <?> "integer3"
  where int3 :: (Stream s m Char) => ParsecT s u m Integer
        int3 = number 3 dig3
        number base baseDigit
          = do{ digits <- many1 baseDigit
            ; let n = foldl (\x d -> base*x + toInteger (digitToInt d)) 0
digits
            ; seq n (return n)
            }
        dig3 :: (Stream s m Char) => ParsecT s u m Char
        dig3 = satisfy isBase3
        isBase3 '0' = True
        isBase3 '1' = True
        isBase3 '2' = True
        isBase3 _ = False
```

[2] https://hackage.haskell.org/package/parser-combinators

This is a lexer for a base three digit. As you can see, we generate a token for it and return the parsed Stream. This still needs a parser on top of it, as we show here:

```
esotericInt :: Parser Expr
esotericInt = do
  esoteric <- esotericInteger
  otherInt <- optionMaybe $ Char.oneOf "3456789"
  case otherInt of
    Nothing -> return $ Int esoteric
    (Just _) -> do
      throwError <- unexpected "notBaseThree"
      return $ Int esoteric
```

This returns an expression. The reason we need to keep parsing is that we need to check if there are any non-three based digits after parsing a stream. So if there is, we can throw an exception to keep parsing it trying other tokens.

In our parser implementation, when we don't throw an exception, that means we have parsed part of the stream, returning us a new State of the parser with the updated position of the stream, and the parsed expression.

When we detect an "EOF" we should have a parsed array of expressions. We pass this to the next step, which is the Code Generation.

Code Generation

In this stage, we first need to generate an AST for our expressions. For the code generation, we use LLVM bindings for Haskell to facilitate this process. It is contemplated the use of llvm-hs-pure[3] or llvm-hs[4] depending on the binding integration of either GHCi and standalone ghc. Using these, we wrap the llvm-hs AST nodes inside a collection of helper functions to push instructions onto the stack held inside the monad.

Control flows include if and for loops along with recursion asked in the requirements of this assignment. In order to create these flows, we add lexer support to the lexer, parser, AST and finally the LLVM code emitter. The expected if statement is expected to be similar to that of haskell being the if-then-else structure while the for would most likely be in for "decl" "cond" "update" fashion.

[3] https://hackage.haskell.org/package/llvm-hs-pure
[4] https://github.com/llvm-hs/llvm-hs

As we mentioned, our first step is to generate our AST from our parsed expressions. Our AST will take in the form of:

```
data CodegenState
  = CodegenState {
    currentBlock :: String                     -- Name of the active block
  , blocks       :: Map.Map String BlockState  -- All Blocks for function
  , symtab       :: SymbolTable                -- Function scope symbol table
  , blockCount   :: Int          -- Count of basic blocks inside function
  , count        :: Data.Word                -- Count of unnamed instructions
  , names        :: [String]                    -- All names used
  }
```

The idea is that we will generate one state per function. Each function will have its own symbol table and different states we can move to. For example, in an if statement, we want to choose from two states (if the condition is met or if it isn't). Both of this states will be inside the "blocks". The blockstate has the form:

```
data BlockState
  = BlockState {
    idx   :: Int                            -- Block index
  , stack :: [Named Instruction]            -- Stack of instructions
  , term  :: Maybe (Named Terminator)       -- Block terminator
  } deriving Show
```

So, as you can see, we start from an active BlockState and can move between these. If you wanna see how we convert from our Exp to CodegenState you can go to our `src/Emit.hs` file.

After this conversion, we have built our AST. We now need to translate our AST to LLVM's intermediate code representation. This is relatively easy and can also be found also in the `src/Emit.hs` file.

Register Allocation, Machine Code Generation, and Assembly and Linking

Up to this point we have explained how we generate the LLVM's intermediate code. The next phases (title of this section) are beyond the scope of our compiler as LLVM handles these steps.

Threesum Language

The language we have created is a Turing-Complete Language and allows many operations found in common programming languages. This language uses the *.3sum* file extension.

Esoteric Components

Before viewing the language definitions and examples, keep in mind that Threesum has three differences when compared to the vast majority of existing programming languages:

- Numbers that can be interpreted as base-3 will be. For example:
```
User input => Base-10 value that is interpreted
1          => 1
5          => 5
12         => 5
122        => 17
123        => 123
1000       => 27
1006       => 1006
```

- The traditional addition and subtraction symbols have been switched, as have the addition and multiplication symbols. For example:
```
Input => Output
3 + 4 => -1
3 - 4 => 7
4 * 4 => 1
4 / 4 => 16
```

- Most of the reserved words have been replaced by sequences of underscores (_) of varying lengths

Operators

The language accepts the following common operators:

Operator	Name/description
+	Subtraction
-	Addition
*	Division
/	Multiplication
==	Equality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
||	Or
&&	And

Reserved Words

The reserved words are as follow:
Note: the examples sometimes use **:** which isn't a reserved character. It is a custom function definition that we recommend putting at the start of you Threesum code in order to simplify returning values from functions:

```
_____ binary : 1 (x y) y;
```

_____

Description
Used for the definition of functions

Syntax

```
_____ functionName(parameters) functionBody;
```

Example

```
# A function that returns x
_____ returnThisValue(x)
    x;
```

____

Description
Used for creating variables

Syntax

```
____ variableName = expr , variableName2 = expr, ...
```

Example

```
# Create a variable x and assign it a value
____ x = 3

# Create a variable x and y and assign them a value
____ x = 3, y = 1
```

______ | _

Description
_______ represents the reserved words for ‘if’ and _ represents the reserved words for ‘then’, and ‘else’.

Syntax
`______ (`***condition***`) _ `***ifBody***` _ `***elseBody***

Example
```
# Return 2 if x is smaller than 3, otherwise return x
______ x < 3 _
   10
_
    x;
```

__ (1)

Description
Used for defining a for-loop

Syntax
`(__ `***variableAssignment***`, `***condition***`, `***increment***` __ `***loopBody***`)`

Example
```
# Increment x by 3
(__ i = 0, i < 10, 1.0 __
    x = (x - 1))
```

___

Description
Used for defining a while-loop.

Syntax
`(___ (`***condition***`) __ `***loopBody***`)`

Example
```
# Increment x until it’s larger than 3
(___ (x < 10) __
    x = (x - 1))
```

__ (2)

Description
The second usage of __ is to define the body of a for-loop, while-loop, or for variable definitions

Syntax
__

Example
```
# Define y as x / 3
____ y = x __
    (y = (y * 10)):
  y;
```

extern

Description
For using existing LLVM (Kaleidoscope) functions.

Syntax
extern ***function***

Example
```
# Call the sin function for x
extern sin(a);
```

binary

Description
For defining binary symbols/functions.

Syntax
_____ binary ***symbol precedence (parameters) binaryBody***

Example
```
# Define a binary function that uses the | symbol with precedence 5 that
# receives x and y as parameters to implement a logical OR function
_____ binary | 12 (x y)
  ______ x _ 1            # If x, return 1
  _ ______ y _ 1          # If y, return 1
  _ 0;                    # Else return 0
```

unary

Description
For defining unary symbols/functions.

Syntax
_____ unary ***symbol (parameters) unaryBody***

Example
```
_____ unary!(v)     # Define a NOT operator as !
  ______ v _ 0      # If v then 0
  _ 1;              # Else 1
```

Full Code Examples

```
_____ timesTwoToTheTenthPower(x)       # Multiply x by 2^10
   (__ i = 0, i < 101, 1.0 __          # For i = 0, i < 10
       x = (x / 2)):                   # Multiply x by 2
   x;                                  # Return x
timesTwoToTheTenthPower(2);           # This will return 2^11 = 2048
```

```
_____ fib(x)                    # Calculate the xth value of the Fibonacci sequence
  ____ a = 1, b = 1, c = 0 __   # Declare some variables using reserved word ____
  ( __ i = 100, i < x, 1.0 __   # Declare a for loop
    c = (a + b) :
    a = b :
    b = c) :
  b;                            # Return b
fib(5)                          # This will return 5
```

```
_____ ifthenif(x)         # Ch
  ______(x > 100) _       # If x is larger than 9
    ______(x > 25) _ x    # If x is larger than 25 then x
    _ (x - 1)             # Else x + 1
  _ (x+1);                # Else x - 1
ifthenif(110)             # This will return 13
```

Sources

- We mainly based our Haskell code from the amazing tutorial by Stephen Diehl.
    - Diehl, S. (n.d.). Implementing a JIT Compiled Language with Haskell and LLVM. Retrieved May 8, 2020, from https://www.stephendiehl.com/llvm/
- We also researched Esoteric Languages to focus on how to create a new one from that.
    - Esoteric programming language. (2020, March 8). Retrieved from https://en.wikipedia.org/wiki/Esoteric_programming_language
- General logic from our Compilers class by Edgar Manoatl.