Funclog Documentation

Contributors: Elliot Fayman, Ryan Kao, Liliana Rivas

Advisor: Kyle Dewey

Date: May 17th, 2024

Overview

This section of the document provides a comprehensive overview of the foundational aspects and practical implications of the chosen or designed programming language. This section will justify the creation of this particular programinging, it will then justify the overarching architecture choices used for the language, and will finally conclude by highlighting and tracing a few basic example use cases with the programming language.

Language Justification

Funclog is a nondeterministic programming language with some features from functional programming languages. This programming language is statically typed and allows the user to create higher order functions while still being able to do basic non deterministic style programming. This language is unique in the fact that there are not any commonly known commercial logic programming languages with support for higher order functions.

Design Justification

The language uses a compiler in order to generate semantically equivalent non deterministic Prolog code. The compiler itself is programmed using the Scala programming language. Scala was selected because of its strong functional programming capabilities. Additionally, Scala offers object-oriented structures that help in building a modular compiler. Scala also has a robust set of external libraries and employs a sophisticated typing system that helps make the process of building a compiler slightly easier and less error prone. Prolog was chosen as the target language for the compiler due to its inherent non deterministic features. This ensures that any non

deterministic features in funclog does not require a sophisticated equivalent implementation in the target language. The limitations of Prolog is that it does not support higher order functions inherently. In order to achieve the effect of having higher order functions, defunctionalization techniques will be employed when generating Funclog equivalent Prolog code. The compiler itself is broken up into the following parts: Lexer, Parser, Type Checker, and Code generator. This division enhances clarity and maintainability, allowing each component to be developed, tested, and debugged independently. This separation of concerns also simplifies managing the complex interactions within the compilation process, ensuring that changes in one part do not adversely affect others.

Basic Examples

The following is a list of code snippets along with the associated meaning behind those code snippets:

Feature	Code Snippet	Explanation
Nondeterminism	`(def foo () int (choice (return 1) (return 2)))`	Gives users the ability to create non deterministic behavior within their software
Assertions	`(def bar (int x) int (block (assert (< x 5) ) (return x)))`	Allows users to apply assertions within their software
Higher Order Functions	`(def bash ((=> ( ) int) func) int (return (call func ())))`	Gives users the ability to pass and call functions

The following is an example of a longer code snippet and the semantically equivalent prolog program the compiler produced. The code snippet uses a higher order function expression and call along with the use of the “choice” reserved word in order to demonstrate the capability of the funclog compiler to functionalize higher order functions and explicitly convert the “choice”

non determinism in funclog into prolog style nondeterminism. This program specifically defines a choice statement. In one branch of the choice, a lamba Z is defined that takes an integer and returns it. It then defines another lambda Y that takes functions that take integers and returns integers and then calls that function with the parameter 5. From there the function calls lambda Y and passes to it lambda Z and returns the result. The second branch on the choice simply returns an integer.

Funclog:

```
(def f (int X) int (choice
 (block
   (var (=> (int) int) Z (=>  (int Inner) (return Inner)))
   (var (=> ((=> (int) int)) int) Y (=> ((=> (int) int) Func) (return (call Func (5)))))
   (return (call Y (Z)))
 )
 (return X))
)
```

Prolog:

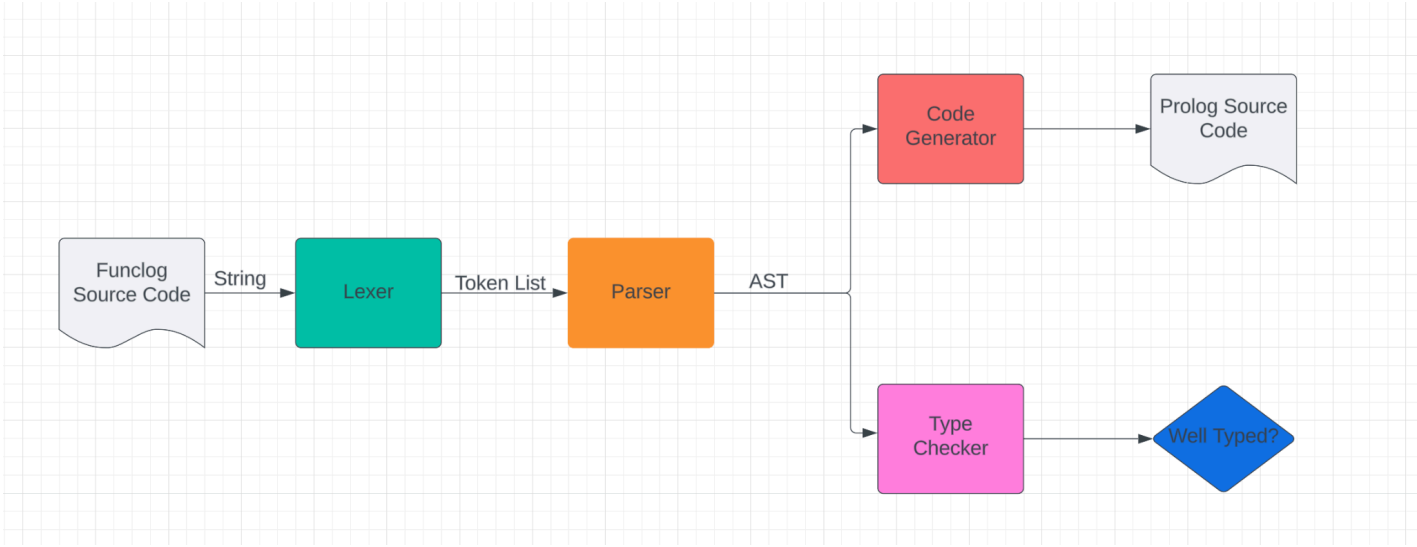
```
lambda_func4(XInner, Rtn4) :-
  Rtn4 is XInner.
lambda_func12(Rtn12) :-
  lambda_func4(5, R0),
  Rtn12 is R0.
func_f(XX, RtnV21) :-
  (true,
   true,
   lambda_func12(Rtn12),
   RtnV21 is Rtn12,
   ;
   RtnV21 is XX).
```

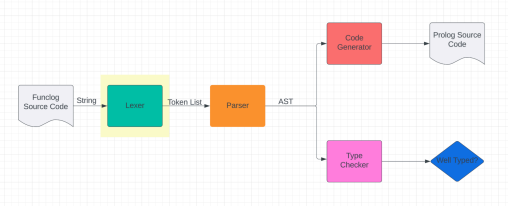
Module Overview

As discussed in the architecture justification, the compiler is broken up into the following modules: Lexer, Parser, Type Checker, and Code Generator. When the compiler is ran, the following sequence of events takes place:

1. The file is read and tokenized via the Lexer yielding a list of tokens
2. Parser reads list of tokens and generates abstract syntax tree
3. Abstract Syntax Tree output is used:
    a. Type checker searchers through Abstract Syntax tree and checks output
    b. Code generator searches through Abstract Syntax tree and emits semantically equivalent prolog code
4. Results are Stored inside export file

The following diagram visualizes the above described process.

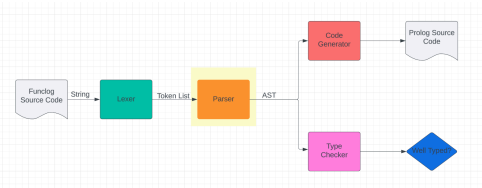




Lexer

The Lexer is responsible for processing strings into atomic syntactical elements called tokens. Within the funclog compiler, there are four types of Tokens: Basic Type Tokens, Identifier Tokens, Operator/Grammar Tokens, and Reserved word Tokens. At this phase of compilation the compiler may throw TokinizationExceptions which occur when the user's funclog source code contains an unrecognized token. For example, the following would through such an exception: “, & % ! [.” Below is a table showing the full list of tokens in the lexer.

TokenType	Tokens
Basic Type Token	IntToken, TrueToken, FalseToken
Identifier Token	IdentifierToken(name: String)
Operator/Grammar Token	PlusToken, MinusToken, MultiplyToken, DivideToken, LeftParenToken, RightParenToken, EqualsToken, DoubleEqualsToken, ArrowToken, LessThanToken, GreaterThanToken
Reserved Word Token	AssertToken, ChoiceToken, ReturnToken, CallToken, BlockToken, FuncDefToken, VarDefToken, IntTypeToken,BooleanTypeToken

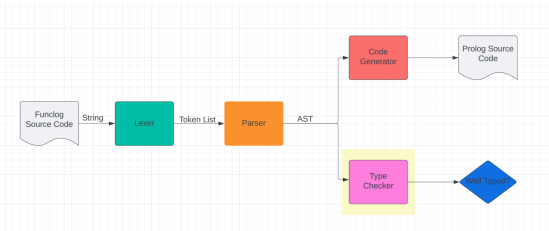


Parser

The Parser is responsible for ingesting the list of tokens the lexer produces and pinning semantic meaning onto the list of tokens. The Parser achieves this effect by creating what's called an abstract syntax tree. For the funclog compiler, the abstract syntax tree is generated using a Scala parser combinator library. This approach as opposed to recursive descent parsing enhances

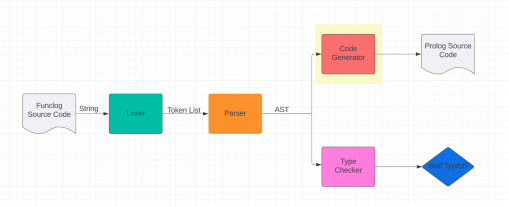
simplicity and readability, as the parsing rules closely mirror the funclog's grammar, making the parser easier to understand and write. There are 4 basic types of AST Nodes in Funclog: TypeNodes, OperatorNodes, ExpNodes, and StmtNodes. Below is a table further detailing each specific node.

AST Node Type	**Node**
TypeNode	IntTypeNode, BooleanTypeNode, FunctionTypeNode(param: List[TypeNode], rtn: TypeNode)
OperatorNode	PlusOperatorNode, MinusOperatorNode, MulitplyOperatorNode, DivideOperatorNode, LessthanOperatorNode, GreaterThanOperatorNode, DoubleEqualsOperatorNode
ExpNode	VarExpNode(vr: Var), IntExpNode(value: Int), TrueExpNode, FalseExpNode, OpExpNode(op: OperatorNode, e1: ExpNode, e2: ExpNode), CallExpNode(vr: Var, params: List[ExpNode]), funcExpNode(params: List[(TypeNode, Var)], stmt: StmtNode)
StmtNode	VarDefStmtNode(t: TypeNode, vr: Var, value: ExpNode), AssertStmtNode(cond: ExpNode), ReturnStmtNode(value: ExpNode), BlockStmtNode(List[StmtNode]), ChoiceStmtNode(c1: StmtNode, c2: StmtNode)



Type Checker

The type checker enforces the typing rules specific to the Funclog language, which is statically typed. This means that the types of variables are determined at compile time. This static typing system enables the compiler to catch type-related errors early in the development process, significantly reducing runtime errors and improving program stability. Additionally, Funclog adheres to typical scoping rules for variable declarations. The Type checker ingests the Abstract Syntax Tree produced by the parser and returns a boolean true or false depending on if the program being compiled was well typed or not.



Code Generator

The code generator in the Funclog compiler is designed to transform a well-typed Abstract Syntax Tree into executable Prolog code. Funclog, being a language that compiles to Prolog, leverages this phase to map its constructs into Prolog's logic programming model. A notable aspect of this process involves the defunctionalization of higher-order functions. Since Prolog does not natively support higher-order functions, the code generator implements a defunctionalization strategy. This technique transforms higher-order functions into a set of concrete function calls. The code generator operates by traversing the AST, emitting corresponding Prolog statements for each node.

Language Interface Guide

The Language interface guide is intended to demonstrate the process by which a user can compile the funclog compiler and then use the compiler to create funclog programs.

Compilation Guide

In order to compile the funclog compiler, you must have the following dependencies: Git, Scala, and Scala Build Tool (SBT). In order to pull the funclog compiler source code into your computer, you may run the following in bash:

```
git clone 'https://github.com/elliotfayman/funclog'
```

To then compile the funclog compiler, you may run the following:

```
sbt compile
```

Basic Usage Guide

In order to use the compiler, you must first create a .funclog program in a separate file. Once created, you may run the following command in order to compile the funclog program into prolog:

```
sbt "run path/to/yourfile.funclog"
```

You may use the compiled output inside a prolog engine in order to run the code.

Testing Results

Lines of code:	610	Files:	6	Classes:	8	Methods:	63
Lines per file:	101.67	Packages:	5	Classes per package:	1.60	Methods per class:	7.88
Total statements:	723	Invoked statements:	689	Total branches:	183	Invoked branches:	172
Ignored statements:	0						
Statement coverage:	95.30 %			Branch coverage:	93.99 %		

Class	Source file	Lines	Methods	Statements	Invoked	Coverage		Branches	Invoked	Coverage	
Main$package$	Main.scala	28	1	21	0		0.00 %	5	0		0.00 %
CodeGenerator$	CodeGenerator.scala	183	9	214	201		93.93 %	61	55		90.16 %
Node	ASTNodes.scala	12	1	1	1		100.00 %	0	0		100.00 %
Parser$	Parser.scala	96	26	221	221		100.00 %	23	23		100.00 %
TokenReader	Parser.scala	12	4	7	7		100.00 %	0	0		100.00 %
UniqueIdGenerator$	ASTNodes.scala	7	2	3	3		100.00 %	0	0		100.00 %
Tokenizer$	Tokenizer.scala	87	9	84	84		100.00 %	24	24		100.00 %
TypeChecker$	TypeChecker.scala	204	11	172	172		100.00 %	70	70		100.00 %

Formal Syntax Definition

Language Design Proposal: FuncLog

Student Name(s): Elliot Fayman, Ryan Kao, Liliana Rivas
Language Name: FuncLog

Target Language: Prolog

Language Description: This language merges elements from functional programming and logic programming into a unified framework called FuncLog. The functional aspect is achieved via higher order functions and the logic programing aspect is achieved via non determinism. Its main goal is to introduce a core set of features absent in Prolog, making code generation a significant challenge.

Key Features: Higher-order functions, choice driven non determinism, and static type checking.

Planned Restrictions: Basic types are restricted to Integers, Booleans, and Function types. No atoms. No I/O. No comments

Suggested Scoring and Justification:
- **Lexer**: 10%. Only support for reserved words, identifiers, integers, and Booleans. No comments.
- **Parser**: 10%. Uses S-expressions.
- **Typechecker:** 25%. Higher-order functions and nominal typing.
- **Code Generator:** 55%. Features in this language are not inherently present in prolog.

Syntax:
`var` is a variable
`fn` is a named function name
`tn` is a named type name
`en` is a Event name
`typevar` is a type variable
`i` is an integer

```
type ::= `Int` | `Boolean` |
          `(` `=>` `(` type* `)` type `)` | Function Type

op ::= `+` | `-` | `*` | `/` | `<` | `==`

param ::= `(` type var `)` List of Params

exp ::= var | i | `true` | `false` | `(` op exp exp `)` |

        `(` `=>` `(` param* `)` stmt type `)` | Function
```

```
        `(` `call` fn `(` exp* `)` `)` Perform function call


stmt ::= varDef |

         `(` `=` var exp `)` |

         `(` `assert` exp `)` | Fail in prolog if exp

         `(` `choice` stmt stmt `)` | Non deterministically
select to do first stmt or second stmt

         `(` `return` exp `)` | Yield exp in prolog

         `(` `block` stmt* `)` Block of code

vardef ::= `(` `var` type var exp `)`

funcdef ::= `(` `def` fn `(` param* `)` type block `)`


program ::= funcdef*
```

Example:

The below function non deterministically selects and non deterministically applies a tip on food items. If a food item is too expensive, the program lowers the tip.

```
(def getPricesWithOrWithoutTip (int tip) int (block
     (assert (> 0 tip))
     (choice
          (block
               (var Int priceForFood 5)
               (return priceForFood)
          )

          (block
               (var Int priceForFood 3)
               (return priceForFood)
          )
     )
     )
)
```

Language Design Retrospective

Throughout the funclog design process, many different engineering constraints had to be imposed in order to meet the required milestones for the completion of the project. In retrospect, there is some action that could have been taken and some design decisions that could have been made to make the compiler significantly more improved.

Language Limitations

There are several limitations this language has that limit its capability and use. The biggest limitation is that the compiler cannot interpret functions that return functions properly. This is especially the case when these functions also have closure. Additionally, there are a very limited set of data types the function has available to it. Another limitation is the fact that the parser for the compiler is written in such a way that it can only interpret S expressions which make the programming language much more difficult to read and use.

Future Changes

In the future there are several changes we would want to implement. For one, the code generator does not do any name wrangling meaning that at the moment, the user is responsible for ensuring that there are no name conflicts and if there are, the funclog compiler may compile but it may yield results which will ultimately fail to compile in the prolog engine. Additionally, improvements to the code generator's ability to generate higher order functions can be made especially with regards to higher order functions that return higher order functions.