Homework 4

Due: Thu, May 4, 2017

In this assignment, you will implement a compiler for a limited C-like programming language (functions, mutable variables, and state). Your implementation must parse an input file to produce an internal representation of the program. The compiler can either a) interpret that program by evaluating it, or b) generate code that can be compiled and executed as a native binary.

1 Abstract language

This section defines the abstract syntax and semantics of the programming language. It describes the internal representation of types, expressions, declarations, and statements along with rules for constructing well-formed and well-typed programs.

1.1 Object model

This section describes the fundamental notion of storage. Note that this language does not provide direct access to raw storage; it only provides facilities that operate on objects and their values. It is not possible, for example, to reinterpret an integer object as a sequence of bytes.

A program can create, destroy, reference, access, and modify objects. An *object* is region of storage created by a variable declaration or as the result of a value computation. Objects created by a declaration are *stored objects*. An object created by a computation is a *temporary object*. Temporary objects are not stored.

Every stored object has a *memory location*, which can be used as to refer to the object (e.g. as the value of a reference).

Every object has a *type*. The type of an object determines how its values are represented.

All objects created by a declaration have *automatic storage duration*. These objects are destroyed when the block (statement) in which they were created exits. Note that storage for a variable can be reused. For example, a loop containing a variable declaration creates multiple objects for that variable, all of which may occupy the same storage.

A function is not an object.

1.2 Types and values

The calculator defines two types, which are defined as follows:

t	::=	**bool**	
		int	
		$(t_1, t_2, ..., t_n) \to t$	function types
		ref t	reference types

The type **bool** describes the values true ($\top$) and false ($\bot$). The type **int** describes integer values in the left-open range $[-2^{31-1}, 2^{31-1})$.

Collectively, **bool** and **int** are called the *object types*. Values of object types can be stored in memory.

The set of types $(t_1, t_2, ..., t_n) \to t$ are the *function types*; they are the types of functions (see declarations). For example, the type $(\textbf{int}, \textbf{int}) \to \textbf{bool}$ is the type of all functions taking two **int** arguments and returning a **bool** value. The types $t_1, t_2, ..., t_n$ are the *parameter types* of a function, and t is *return type* of the function. The values of a function type are the functions themselves. Note that functions are stored as part of a program and are typically accessed by a pointer.

The set of types **ref** t are the *reference types*. The values of a reference type are the memory locations of stored objects of type t. For example, **ref int** is a reference to an integer. Its values are the locations of **int** objects in memory.

1.3 Expressions

The language defines a number of expressions. These are:

e	::=	**true**			
		false		$e_1 < e_2$	less than
		$\mathbb{Z}$	integer values	$e_1 > e_2$	greater than
		ref d	references	$e_1 \leq e_2$	less than or equal to
		val e	valuation	$e_1 \geq e_2$	greater than or equal to
		$e_1 \leftarrow e_2$	assignment	$e_1 + e_2$	addition
		e_1 **and** e_2	and	$e_1 - e_2$	subtraction
		e_1 **or** e_2	inclusive or	$e_1 * e_2$	multiplication
		not e_1	logical negation	e_1 **div** e_2	integer division
		if e_1 **then** e_2 **else** e_3	conditional	e_1 **rem** e_2	remainder of division
		$e_1 = e_2$	equal to	$-e_1$	arithmetic negation
		$e_1 \neq e_2$	not equal to	$e(e_1, e_2, ..., e_n)$	function call

An expression is a sequence of operands and operators that specifies a computation. The evaluation of an expression results in a value and side effects. A *side effect* of an expression can create or destroy objects (e.g., function calls) or modify the values of existing objects (e.g., assignment). Unless otherwise specified, expressions have no effect.

The type of an expression determines how expressions can be combined to produce complex computations and the kind of value it produces. The following paragraphs define the requirements on operands and the result types of each expression as well the values they produce.

Many expressions require their operands to be values of some type t. When an operand e for such an expression has type **ref** t, it can be implicitly converted by replacing e with a new expression

val e. The effect of this conversion is to access the value of a reference for use in the expression. This process is called *value conversion*.

To convert an expression e to an object type t, value conversion is applied to e, possibly replacing it. If the converted expression does not have type t, the program is ill-formed.

The order in which an expression's operands are evaluated is unspecified unless otherwise noted.

1. The expressions **true** and **false** have type **bool** and the values $\top$ and $\bot$, respectively.

2. Integer values have type **int**. The value of an integer literal is the one indicated by the expression.

3. Expressions of the form **ref** d denote a reference variable or function (i.e., declarations) within an expression. The type and value of a depends on the kind of d.

   - If the d is a variable whose declared type is t, then the type is **ref** t and the value is the location of the object created by d.
   - If d is a function, the type is the declared type of the function, and the value is d.

4. The expression **val** e denotes the (implicit) conversion of a reference to the value of the referenced object. The operand e shall have type **ref** t and the result type of the expression t. The value of the expression is the value of the object referred to by e.

5. The expression $e_1 \leftarrow e_2$ denotes the assignment of the value of e_2 to the object referred to by e_1. The operand e_1 shall have reference type **ref** t. The operand e_2 is converted to t. The effect of the expression is to store the value of e_2 in the object referred to by e_1. The type of the expressions is **ref** t and the value is that of e_1.

6. The operands of e_1 **and** e_2 and e_1 **or** e_2 are converted to **bool**. The result type of each is **bool**. The result of e_1 **and** e_2 is $\top$ if both operands are $\top$ and $\bot$ otherwise. If e_1 is $\bot$, then e_2 is not evaluated. Note that this is equivalent to **if** e_1 **then** e_2 **else false**. The result of e_1 **or** e_2 is $\bot$ if both operands are $\bot$ and $\top$ otherwise. If e_1 is $\top$, then e_2 is not evaluated. Note that this is equivalent to **if** e_1 **then true else** e_2.

7. The operand of **not** e_1 is converted to **bool**. The type of the expression is **bool**. The result of the expression is $\top$ when the e_1 is $\bot$ and $\bot$ otherwise.

8. In the conditional expression **if** e_1 **then** e_2 **else** e_3, e_1 is first converted to **bool**. If e_2 and e_3 have the same type, no other conversions are performed. If either e_2 or e_3 has type **ref** t and the other operand has type t, value conversion is applied to the reference. Otherwise, the program is ill-formed. The type of the expression is the type of e_2 and e_3. The result of expression is determined by first evaluating e_1. If that value is $\top$ then the result of the expression is the value of e_2, otherwise, it is the value of e_3. Only one of e_2 or e_3 is evaluated.

9. Value conversion is appled to the operands of $e_1 = e_2$ and $e_1 \neq e_2$. After conversion, the operands shall have the same type. The result type is **bool**. The result of $e_1 = e_2$ is $\top$ if e_1 and e_2 are equal and $\bot$ otherwise. The result of $e_1 \neq e_2$ is $\top$ if e_1 and e_2 are different and $\bot$ otherwise.

10. The operands of $e_1 < e_2$, $e_1 > e_2$, $e_1 \leq e_2$, and $e_1 \geq e_2$ are conveted to **int**. The type of these expressions is **bool**. The result of $e_1 < e_2$ is $\top$ if e_1 is less than e_2 and $\bot$ otherwise. The

result of $e_1 > e_2$ is $\top$ if e_1 is greater than e_2 and $\bot$ otherwise. The result of $e_1 \leq e_2$ is $\top$ if e_1 is less than or equal to e_2 and $\bot$ otherwise. The result of $e_1 \geq e_2$ is $\top$ if e_1 is greater than or equal to e_2 and $\bot$ otherwise.

11. The operands of all arithmetic expressions are converted to **int**. The type of these expressions is **int**. The result of $e_1 + e_2$ is the sum of the operands. If the sum is greater than the maximum value of **int**, the result is undefined. The result of $e_1 + e_2$ is the difference resulting from the subtraction of the e_2 from e_1. If the difference is less than the minimum value of **int**, the result is undefined. The result of $e_1 * e_2$ is the product of the operands. If the product is greater than the maximum value of **int**, the result is undefined. The results of e_1 **div** e_2 and e_1 **rem** e_2 are the quotient and remainder of dividing e_1 by e_2, respectively. In either case, if e_2 is 0, the result is undefined. If e_2 is the minimum value of **int**, the result is undefined. For division, the fractional part of the value is discarded (the value is truncated towards zero). If the expression a **div** b is defined, $(a$ **div** $b) * b + a$ **rem** b is equal to a.

12. The operand of $-e_1$ is converted to **int**. The type of the expression is **int**. The result of the expression is the additive inverse of the value of e_1. Note that $-e_1$ is equivalent to $0 - e_1$.

13. In a call expression $e(e_1, e_2, ..., e_n)$, the expression e is the *callee* and $e_1, e_2, ..., e_n$ are the *arguments*. The callee shall have function type $(t_1, t_2, ..., t_m) \to t$. The number of arguments shall be the same as the number of parameters of the callee (i.e., $n = m$ must be true). Each parameter is initialized by its corresponding argument (see variables). Note that initialization may result in conversions. The type of the call expression is the function's return type t.

    The value of the function call is the value returned by the called function. Note that control is passed to the function's definition in order to compute the return value.

1.4 Declarations

A declaration associates meaning with an identifier. The language defines three kinds of declarations:

d	::=	**def** $n(p_1, p_2, ..., p_n) \to t\ s$	functions
		$t\ n$	parameters
		$t\ n = e$	variables

A *function* **def** $n(p_1, p_2, ..., p_n) \to t\ s$ defines an identifier n to be a parameterized statement s that computes or returns a value of type t. The *parameter*s of a function $p_1, p_2, ..., p_n$ are declarations that determine the number and type of arguments accepted by the function. The type t is called the function's *return type*.

The *signature* of a function is its name, parameters, and return type. The type of a function whose signature is **def** $n(p_1, p_2, ..., p_n) \to t$ has the function type $(t_1, t_2, ..., t_n) \to t$.

The *definition* of the function is the statement s.

Within the definition of the function, parameters are variables, which contain the values of arguments supplied at the function calls (as described above). These objects created for parameters are destroyed when the function returns. The *return value* of a function is variable that is initialized by a return statement (see below). Note that this language does not require return values to be allocated by the caller; return values can be returned directly (e.g., via a register).

Note that parameters are never declared outside of a function. Also note that this language does not permit a function to return without initializing the return value.

A *variable* defines an identifier n to be a stored object of type t whose value that is initialized by the expression e. The object created by a variable declaration is destroyed when the block in which it is declared exits.

Initialization provides an initial value for a variable or parameter. The process of initialization depends on the type of the declaration.

- If the variable has object type (i.e., **bool** or **int**) or function type, variable is *copy initialized*. Value conversion is applied to the initializer e. After initialization, e shall have the same type as the variable.
- If the variable has reference type, it is *reference initialized*. The initializer shall have the same type as the variable. The reference is bound to the value of the initializer.

1.5 Statements

A statement is a command that is executed as part of a function. The statements are:

s	::=	$\{s_1\ s_2\ ...\ s_n\}$	block statement
		if(e) s_t **else** s_f	
		while(e) s	
		break	
		continue	
		return e	
		assert e	
		e	expression statement
		d	declaration statement

Statements have neither type nor value. Statements are executed to produce side effects. The meaning of these statements are:

- A *block statement* is a sequence of statements that are executed in sequence. A block statement exits after its last statement has executed, a nested break or continue statement causes control to pass outside to a non-nested statement, or a nested return statement causes is executed.
- In an *if statement* **if**(e) s_t **else** s_f the condition e is converted to **bool**. The statement's condition is evaluated. If the result is **true**, the *true branch* s_t is executed. Otherwise, the *false branch* s_f is executed.
- In a *while loop* **while**(e) s the condition e is converted to **bool**. The loop repeatedly executes its body s until the condition e becomes **false**. The *top* of a while loop evaluates the condition and, if **true** executes the statement s. The *bottom* of the loop passes control to the statement following the loop.
- A *break statement* **break** shall appear only inside a while loop. Its execution causes the control to pass to bottom of the innermost loop in which the statement occurs.

- A *continue statement* **continue** shall appear only inside a while loop. Its execution causes the control to return return to the top of the innermost while loop in which the statement occurs.
- In a *return statement* **return** e, e is converted to the return type of the function. The execution of a return statement initializes the return value with the value of e, and control is returned to the caller of the function.
- In an *assert statement* **assert** e, e is converted to **bool**. If the e is false, the program is terminated in an implementation defined way. Otherwise, the statement has no effect.
- An *expression statement* e evaluates the expression e and discards the value.
- A *declaration statement* d declares a variable within its block. This creates an object that is destroyed when the nearest enclosing block exits.

2 Program Translation

There are two required phases of translation:

1. The source text is decomposed into tokens according to the lexical conventions below.
2. The resulting tokens are analyzed syntactically (i.e., parsed) and semantically to produce an abstract representation of the program (i.e., an AST).

Note that the abstract program may be subsequently translated into one or more lower-level languages suitable for virtualized or native execution.

3 Program execution

A program shall contain a function named `main` and that function shall have the type $() \to int$. The function `main` shall not be called by any other functions in the program.

The program is started by a call to main from the execution environment. Note that the call to main could be initiated by an interpreter of the abstract program, or it could be invoked by the operating system after the program has been translated to machine code.

The program is terminated when `main` returns. The return value of `main` is returned to the execution environment.

4 Lexical conventions

This section describes the lexical conventions of the language. That is, it determines the set of symbols used in the concrete syntax of the language. The information in this section can be used to derive a specification of a lexical analyzer and the tokens that it produces.

The lexical conventions describe the symbols (tokens) of the grammar's language. The tokens are punctuators, operators, or literals. The language defines the following tokens representing punctuators and operators:

```
+   -   *  /  %
&&  ||  !
==  !=  <  >  <=  >=
?   :
(   )   {  }
```

The following identifiers are reserved as keywords.

```
break  continue  def   else  false
if     return    true  var   while
```

The boolean literals are:

boolean-literal → `true` | `false`

The integer literals are:

integer-literal → *digit digit*∗
digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Identifiers have the following form:

identifier → *letter*|(*letter*|*digit*)∗
letter → `a` | `b` | ... | `z` | `A` | `B` | ... | `Z` | `_`

Note that this language effectively has only 4 kinds of tokens:

- punctuators and operators are tokens that have no information beyond their name,
- boolean literals have an associated boolean value as an attribute,
- integer literals have an associated integer value as an attribute,
- identifiers are associated with their spelling.

5 Concrete syntax

This section gives the concrete syntax of the language as a context-free grammar in Backus-Naur form (i.e., a set of productions that define the valid strings of the language. This specification can be used to derive a parser that matches a sequence of tokens to produce a corresponding abstract syntax tree.

A program is well-formed only if its tokens form a string in the language. Said otherwise, if the parser fails to match a sequence of tokens to a particular production, the input contains a syntax error. In such cases, the program is not evaluated, and calculator should diagnose the specific source of the error.

Most productions have associated semantic actions which include additional rules that determine whether or not a program is well-formed. These are defined below. Note that an "$_{opt}$" following a production name indicates that the production may be omitted.

5.1 Scope, declarations, and lookup

The *scope of a name* is the region of the program where that name is valid. A *scope* is the set of all names declared within some part of a program. This language defines three kinds of scope, corresponding to different parts of the program text.

- *Global scope* covers the entirety of a *program*. Only function names have global scope
- *Parameter scope* covers the region of text corresponding to the start of a *parameter-list* of a *function-declaration* and ending at the closing brace of its *block-statement*. Only parameter names have parameter scope.
- *Block scope* covers the region of text starting with the opening brace of a *block-statement* and ending at the closing brace of that statement. Only variable name have block scope.

The *point of declaration* of a name is the location in the text where a name becomes visible (for the purpose of lookup). The point of declaration for names is described with the semantic actions below.

As a general rule, a scope shall not contain two declarations with the same name. For example:

```
def f(int n) -> int { return 0; }
def f() -> bool { return false; } // error: redefinition of f
```

Name lookup is the process of identifying the declaration of an *identifier* within a program. Name lookup searches searches each scope for a declaration with the given *identifier*, starting with the innermost scope and proceeding outwards towards the global scope. Name lookup ends as soon as a declaration is found. If no declaration is found, the program is ill-formed. In all cases, a name shall be declared before it is used. Note that this implies that the order of declarations matters.

5.2 Program structure

program	→	*declaration-seq*
declaration-seq	→	*declaration*
		declaration-seq declaration
declaration	→	*function-declaration*
function-declaration	→	`def` *identifier* `(` *parameter-list*$_{opt}$ `)` `->` *type block-statement*
parameter-list	→	*parameter*
		parameter-list `,` *parameter*
parameter	→	*type identifier*

The point of declaration for a function is immediately after the return type and before the opening brace of its definition. Function names have global scope.

The point of declaration for a parameter is immediately after its *identifier*.

5.3 Types

The declarations of functions and variables are annotated by their types.

type	→	*simple-type*
		simple-type`&`
simple-type	→	`bool`
		`int`

The *simple-type*s `bool` and `int` correspond to the types **bool** and **int** respectively.

A type of the form `t&` corresponds to the type **ref** t where `t` corresponds to the abstract type t.

5.4 Statements

statement	→	*block-statement*
		if-statement
		while-statement
		break-statement
		continue-statement
		return-statement
		assert-statement
		expression-statement
		declaration-statement
block-statement	→	{ *statement-seq* }
statement-seq	→	*statement*
		statement-seq statement
if-statement	→	`if (` *expression* `)` *statement* `else` *statement*
while-statement	→	`while (` *expression* `)` *statement*
break-statement	→	`break ;`
continue-statement	→	`continue ;`
return-statement	→	`return` *expression* `;`
assert-statement	→	`assert` *expression* `;`
expression-statement	→	*expression* `;`
declaration-statement	→	*variable-declaration*
variable-declaration	→	`var` *type identifier* `=` *expression* `;`

The semantics of statements are as follows:

1. A block statement `s1 s2 ... sn` corresponds to the abstract syntax $\{s_1\ s_2\ ...\ s_n\}$ where each sub-statement `si` corresponding to the abstract statement s_i. A block defines a block scope that begins at its opening { and ends and its closing }.

2. An if statement `if(e) s1 else s2` corresponds to an abstract **if**(e) s_1 **else** s_2 where e corresponds to the abstract expression e, and `s1` and `s2` correspond to the abstract expressions s_1 and s_2, respectively.

3. A while statement `while(e) s` corresponds to an abstract **while**(e) s where `e` and `s` correspond to their abstract expressions and statements e and s.

4. The statement `break;` corresponds to the abstract statement **break**.

5. The statement `continue;` corresponds to the abstract statement **continue**.

6. An expression statement `e;` corresponds to the abstract statement e where `e` corresponds to the abstract expression e.

7. A declaration statement `d` corresponds to the abstract statement d where `d` corresponds to the abstract declaration d. A declaration statement declares a variable.

8. A variable declaration `var t n = e;` corresponds to an abstract declaration $n\ t = e$ where `t` corresponds to the type t, `n` is the declared name n, and `e` corresponds to the abstract expression e. The point of declaration for a variable is immediately after its *identifier* `n` and before its `=` symbol. Variable names have block scope.

5.5 Expressions

expression	→	*assignment-expression*
assignment-expression	→	*conditional-expression*
		logical-or-expression `=` *assignment-expression*
conditional-expression	→	*logical-or-expression*
		logical-or-expression `?` *expression* `:` *assignment-expression*
logical-or-expression	→	*logical-and-expression*
		logical-or-expression `||` *logical-and-expression*
logical-and-expression	→	*equality-expression*
		logical-and-expression `&&` *equality-expression*
equality-expression	→	*ordering-expression*
		equality-expression `==` *ordering-expression*
		equality-expression `!=` *ordering-expression*
ordering-expression	→	*additive-expression*
		ordering-expression `<` *additive-expression*
		ordering-expression `>` *additive-expression*
		ordering-expression `<=` *additive-expression*
		ordering-expression `>=` *additive-expression*
additive-expression	→	*multiplicative-expression*
		additive-expression `+` *multiplicative-expression*
		additive-expression `-` *multiplicative-expression*
multiplicative-expression	→	*unary-expression*
		multiplicative-expression `*` *unary-expression*
		multiplicative-expression `/` *unary-expression*
		multiplicative-expression `%` *unary-expression*
unary-expression	→	*primary-expression*
		`!` *unary-expression*
		`-` *unary-expression*
primary-expression	→	*boolean-literal*
		integer-literal
		id-expression
		`(` *expression* `)`
id-expression	→	*identifier*

Note that this grammar can be easily implemented as a top-down, recursive descent parser. However, you must remove all left-recursive productions from the grammar by refactoring them as right-recursive productions.

Semantically, when a production of the grammar matches a sequence of tokens, it produces a corresponding abstract syntax tree. The semantic rules in this section define that correspondence. If the types of operands do not conform to the requirements specified in the abstract syntax, the program is ill-formed, and the calculator must diagnose the error.

1. An *assignment-expression* `e1 = e2` corresponds to the abstract syntax $e_1 \leftarrow e_2$ where each subexpression is represented by its corresponding abstract syntax.

2. A *conditional-expression* `e1 ? e2 : e3` corresponds to the abstract syntax **if** e_1 **then** e_2 **else** e_3 where each subexpression is represented by its corresponding abstract syntax.

3. A *logical-or-expression* `e1 || e2` corresponds to the abstract syntax e_1 **or** e_2 where each subexpression is represented by its corresponding abstract syntax.

4. A *logical-and-expression* `e1 && e2` corresponds to the abstract syntax e_1 **and** e_2 where each subexpression is represented by its corresponding abstract syntax.

5. An *equality-expression* `e1 == e2` corresponds to the abstract syntax $e_1 = e_2$ where each subexpression is represented by its corresponding abstract syntax. An *equality-expression* `e1 != e2` corresponds to the abstract syntax $e_1 \neq e_2$ where each subexpression is represented by its corresponding abstract syntax.

6. A *relational-expression* `e1 < e2` corresponds to the abstract syntax $e_1 < e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 > e2` corresponds to the abstract syntax $e_1 > e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 <= e2` corresponds to the abstract syntax $e_1 \leq e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 >= e2` corresponds to the abstract syntax $e_1 \geq e_2$ where each subexpression is represented by its corresponding abstract syntax.

7. An *additive-expression* `e1 + e2` corresponds to the abstract syntax $e_1 + e_2$ where each subexpression is represented by its corresponding abstract syntax. An *additive-expression* `e1 - e2` corresponds to the abstract syntax $e_1 - e_2$ where each subexpression is represented by its corresponding abstract syntax.

8. An *multiplicative-expression* `e1 * e2` corresponds to the abstract syntax $e_1 * e_2$ where each subexpression is represented by its corresponding abstract syntax. An *multiplicative-expression* `e1 / e2` corresponds to the abstract syntax e_1 **div** e_2 where each subexpression is represented by its corresponding abstract syntax. An *multiplicative-expression* `e1 % e2` corresponds to the abstract syntax e_1 **rem** e_2 where each subexpression is represented by its corresponding abstract syntax.

9. A *unary-expression* `!e1` corresponds to the abstract syntax **not** e_1 where the subexpression is represented by its corresponding abstract syntax. A *unary-expression* `-e1` corresponds to the abstract syntax $-e_1$ where the subexpression is represented by its corresponding abstract syntax.

10. A *boolean-literal* `true` represent the expressions **true** and **false** respectively.

11. An *integer-literal* represents the integer value determined by its spelling. For example, the literal `42` represents the integer value forty-two.

12. For an *id-expression*, lookup is performed on its *identifier*. If lookup succeeds, the expression corresponds to the abstract syntax **ref** d where d is the declaration found by lookup.

13. The expression `(e)` corresponds to the abstract syntax e. That is, applying parentheses to a well-formed expression affects neither the type nor its value.

In your parser, define a function for each non-terminal. Each production should return its corresponding abstract syntax tree. If the production does not match a sequence of tokens, or if a corresponding abstract syntax tree cannot be formed, the input is ill-formed, and you should diagnose the error. Note that throwing an uncaught exception is perfectly reasonable for now.

6 Requirements

Implement this language specification. Your compiler must (minimally) translate input source into an abstract representation (AST). You should provide an interpreter for that representation. You should provide a code generator that emits either JVM or LLVM compatible definitions.

Write a report describing the capabilities of your compiler.

Submission: Submit a printed copy of your report on the due date. I should have a link to your compiler.

Above and beyond: There are lots of things that can be done with this language:

- Extend the language and grammar to include the bitwise operators and, inclusive or, exclusive or, left shift, and right shift, and complement for both boolean and integer values.
- Add language support for arrays and pointers.
- Implement function currying.
- Add support for lambda expressions.
- Add a foreign function interface that allows you to call C functions.