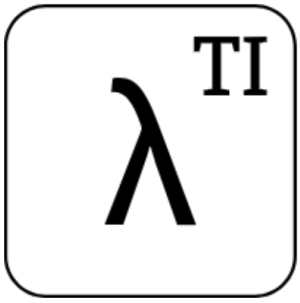


TI-Lisp

Type Inference Lisp

Final Report: COMS 4115 Programming Languages and Translators (Spring 2020)

Language Guru: Yifei Wang
C++ Master: Benson Li
Governor: Jianing Li
Detective: Jay Zern Ng

{yw3229, bbl2117, jl5543, jn2717}@columbia.edu

Contents

1 Introduction

1.1 Motivation

We have decided to design a LISP-like language with type inference, so the user will be able to utilize the power of a LISP language combined with a form of "type checking". We believe that type inference provides the best of both dynamically typed and statically typed languages: in that programs benefit from type checking, while at the same time not needing type annotations and being naturally generic, which can only be done in more conventional languages like Java and C++ through the use of templates. Modern C++ already has forms of type inference in its language, such as the 'auto' keyword. So it only makes sense that in whatever language we design, we should include type inference too.

Our language is appropriate for beginners to learn functional programming. For one, the language is simple. In fact, before they nerfed it, a dialect of LISP was used as the programming language of choice at Berkeley and MIT for their intro to programming class. Also, by introducing type inference to the beginner, we prepare them for the "harder" functional programming languages, such as OCaml and especially Haskell. Debugging in these languages is a fundamentally different beast, where one is, for the most part, thinking theoretically about inputting the correct kind (Haskell term).

1.2 Use cases

Classical hello-world example.

```
1 (define (hello-world) "Hello World!")
2 (hello-world) # => output: Hello World!
```

1.3 Getting started

1. Read "README.md" in the "docs" directory.
2. Run "make all" and "make test" to build and test the source codes.
3. Run "./tilisp.native -help" to see the usage.
4. After running "./tilisp.native -e source.tisp", an executable "a.out" would get generated under the current directory. Run "./a.out" to execute the compiled program.

2 Language Tutorial

2.1 Data Type Systems

Primitive type	Size	Examples
int	8 bytes	6
float	8 bytes	3.3, 0.2, 3.
char	1 bytes	'c'
string	2 bytes	"abc"
bool	1 bytes	true / false
func	24 bytes	(lambda a (+ a 1))

In the implementation of dynamic types, we used a type tag and a union of all different types to store typed values, so values of all types take the same space of 32 bytes.

2.2 Basic operators & built-in functions

Operator	Type	Function	Examples
+	Int → ... → Int	addition	(+ 1 2 3)
-	Int → ... → Int	subtraction	(- 3 2 1)
*	Int → ... → Int	multiplication	(* 1 2 3)
/	Int → ... → Int	division	(/ 3 2)
=	a' → a' → bool	equal	(= true false)
<	a' → a' → bool	less than	(< "a" "bb")
>	a' → a' → bool	greater than	(> 3. 2.)
<=	a' → a' → bool	less than or equal	(<= 2 3)
>=	a' → a' → bool	greater than or equal	(>= 6 5)
++	string → string → string	string concat	(++ "aa" "bb")
integer?	a'→bool	type predicate	(integer? 3)
display	a'→nil	display input to stdout	(display 1)
cons	a' → list → list	appends a' to front of list	(cons 1 '())
car	list → a'	returns first element of the given list	(car (list 1 2 3))
cdr	list → list	returns the same list without the first element	(cdr (list 1 2 3)
map	function → list → list	returns the same list but with the function applied to each member of the list.	(map f (list 1 3)
filter	pred_fun → list → list	returns the elements in the list that satisfy the pred	(filter even? (list 1 2 3))

Arithmetic operators accept 2 or more operands of the same value type (float / int). Logical operators accept exactly 2 operands of the same value type.

2.3 Variable assignment

Global variable binding

```
# Hello world as a variable
(define vhello "Hello world")      #1
```

2.4 Macro

Take the 'and' macro as an example. Macro expansion works similarly to pattern matching. A Lisp expression is matched against patterns in the clauses. Variables are bound in the matching process and the expression will be expanded as specified if it matches. Note that '_' matches against anything but does not bind the matched expression to an identifier. '...' indicates the variable preceding it will match all following expressions in the list.

```
(define-syntax and
  (syntax-rules
      () # literals, in this case no literals are specified
    # clause 1
    (# pattern
     (and)
     # what should the pattern be expanded to
     true)
```

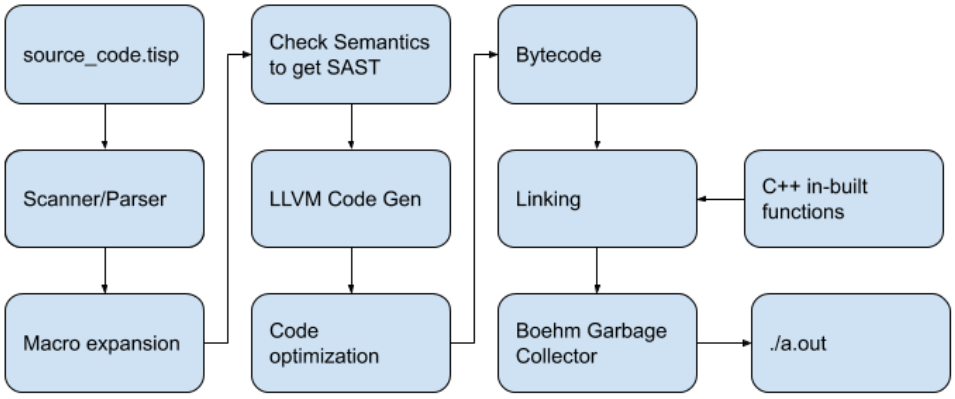
```
    # clause 2
    ((and test) test)
    # clause 3
    ((_ test1 test2 ...)
     (if test1 (and test2 ...) false))))

Example Output:
(and) => true # clause 1
(and 1) => 1 # clause 2
(and (= a 1)) => (= a 1) # clause 2
```

Macros were used to implement built-in variadic functions that can take arbitrary arguments. All the built-in macros included in TI-Lisp are:

1. cond: the cond structure, expanded into nested if
2. let*: the let* structure, expanded into nested let
3. and, or: implemented by macros instead of normal functions to support short-circuit evaluation.
4. define: extend the basic define ((define name value)) to support function declaration.
5. Arithmetic operators automatically expands, for example:
6. +: expands (+ 1 2 3) to (+ 1 (+ 2 3))
7. *: expands (* 1 2 3) to (* 1 (* 2 3))
8. Similarly, display: expands (display 1 2) to (begin (display 1) (display 2))

3 Architectural Design

3.1 Block diagram:



3.2 Major components & interfaces

3.2.1 Parsing

TI-Lisp uses a light weight parser. Besides basic types/literals, all tokens are wrapped into cons pairs. These tokens will later be parsed into keywords/variables/functions during the semantic phase.

program::=		program
	| EOF	end of file
	| expr program	
expr::=		expressions
	| (list_body)	list comprehension
	| INT	integer literal
	| FLT	float literal
	| STR	string literal
	| EXP	expansion
	| CHA	char literal
	| ‘ expr	quote
	| Id	identifier
list_body::=		
	| expr list_body	cons
	| expr . expr	cons
	|	nil

3.2.2 Macro

One of the most important features of TI-Lisp are macro expansions. Basically, there is a basic function that does macro expansion known as ‘expand1‘. It accepts a symbol table and an lisp expression, then:

1. If the expression is a define-syntax form (macro definition), it adds the macro definition to the symbol table, and returns the updated symbol and None. The None here indicates that the define-syntax form is stripped from AST.
2. If the expression that matches any macro in the symbol table (match_rule function), it will do expansion recursively until the expanded expression is no longer a macro form itself (replace_rule function) and return the expanded expression (with unchanged symbol table).
3. Otherwise, it returns the symbol table and expression unchanged.

Now that we have expand1, we can deal with macro definitions and expand macro forms like (and 1 2) recursively. However, normally macro forms are contained within other expressions like (set! a (and 1 2)) or (let ((a true) (b false)) (and 1 2)). As a result, recursive expansion is actually not complete now because we can't further expand macro hidden under nested structures in expanded code (example: (and 1 2) => (if 1 (and 2) false), we can't further expand the inner and now). We need a way to find macro forms in this nested structure. This is why we have the expand_over_list helper function.

expand_over_list first tries to expand the expression itself, in case it is given a macro form directly. Then it tries to expand each element in the expanded list (assume we are dealing with a list here, for other cases, expand_over_list can simply return the expression unchanged) (note that we are expanding elements in an expanded list, this is where full recursive expansion is enabled). If an element of the list is of the form “Cons [Cons something, ...]”, it means that we find a nested list (for example, the (and 1 2) in (set! a (and 1 2))), and we apply expand_over_list recursively to this inner list. This allows us to find macro forms in a nested list recursively.

Finally, since expand (a simple rename of expand_over_list) only works with one expr while our program is represented by an expr list, we create a simple wrapper expand_all, which applies expand sequentially to the list of expr and returns the expanded list of expressions. The initial symbol table in expand_all is made with the builtin macros.

3.2.3 Semant/SAST

The main goal of this stage is to semantically check AST after macro expansions. Using symbol tables, we can distinguish between language structures and function invocations. Our SAST as a program is essentially represented as a list of statements. For example, we have "set!" and "define" and something known as a fallback to "expr". Note that literals, Identifiers, symbols, lists, control structures all have their own clauses in "expr". Also, the "Cons" and "Nil" symbol no longer stores code and they only represent semantic lists.

A key design issue; that "set!" and "define" are statements while others are expressions because they will introduce side effects, i.e. this will introduce further issues with symbol tables. Basically, we need to update symbol tables for expressions while assuming that symbol tables are unchanged.

TI-Lisp is a dynamically typed language that includes type-checking features for functions, values and "any" types (i.e. when we cannot determine their types at compile time). Type checking in TI-Lisp basically checks for expressions for their value types in the return values. Types are not stored in SAST because we do not require them during the IR generation phase.

For example:

```
(define (fib n)
  (cond
   ((= n 0) 0)
   ((= n 1) 1)
   (else (+ (fib (- n 1)) (fib (- n 2))))))
```

Turns into:

```
(define fib (lambda (n)
  (if (funcall = n 0)
    (begin
      0)
    (if (funcall = n 1)
      (begin
        1)
      (begin
        (funcall + (funcall fib (funcall - n 1)) (funcall fib (
    funcall - n 2))))))))
```

3.2.4 Built-in functions

We decided to implement most of our built-in functions by linking with C++ code, rather than creating it in irgen. Because our language is dynamically typed, ir generation would involve a lot of basic blocks to account for the cases. Furthermore, any actual switch cases in the built in functions will mean that we have nested "if"s, which needs to be flattened for IR code. But by implementing the functions in C++, we only need to create the function declaration during irgen.

As for the built-in functions not written in C++, they were implemented by "bootstrapping" other built in functions. For example, "filter" was implemented by using "car", "cdr", "cons", which were all implemented in C++.

4 Test Plan

4.1 Automatic testing framework

We implemented an automated testing framework, so that each time we modified the codes or implemented new features, we can quickly check whether the compiler is broken or not.

In the "tilisp.native" executable, we support argument "-a", "-s", "-l" and "-e", corresponding to generation up to the AST, SAST, IR code, and executable phases. Then, in the "tests" folder, we put two types of test files. File "case.tisp" is source code of a test case and "case.log" is the expected output depending on the phase that we want to test. In the "testing.ml", a string matching is conducted to check whether a case is passed. A partial script of the testing is shown below.

```
~/ti-lisp> make test
./testing.native
Test case: [scanner] comments & identifier [passed].
Test case: [parser] unpaired bracket [passed].
Test case: [parser] incomplete string [passed].
Test case: [macro] parse expansion in macro [passed].
Test case: [macro] macro definition [passed].
Test case: [macro] multiple rules [passed].
Test case: [semant], built-in functions [passed].
...

Example:
Source codes:
(a b c)

Expected output with testing option "-a":
Cons [a, Cons [b, Cons [c, Nil]]]
```

5 Summary

We end this report by discussing various aspects of our contributions and what we learned throughout this awesome project.

5.1 Teamwork

How did we work together?

Yifei:

- Skeleton of the compiler. The implementation from parser through IR generation.
- Design and implementation of macro mechanisms.
- The framework and guideline for creating built-in libraries.

Jianing:

- Logical operators and display of boolean in the built-in functions implementation.
- Reconstruction of testing frameworks.
- Language support for "float" data type from parsing through IR codes.
- The skeleton of this report.

Jay:

- Design of general program argument structure and testing framework.
- Improvement of semantics/SAST and macros.
- Helped with implementing in-built functions.
- Design of report and logo.

Benson:

- Implemented list data structure + associated operations
- Implemented string concatenation + all the arithmetic operators.
- Minor fixes/changes throughout codebase

5.2 Takeaway

What did we learned throughout this project?

Benson:

- Learned more about specific LLVM IR instructions and the LLVM ocaml library bindings
- Learned how to debug IR code generation errors/how to read IR code.
- Personally felt that IR code generation was the hardest phase to implement

Jianing:

- Totally agreed that IR code generation is the most difficult part because it requires a clear thought about the memory allocation and instruction sequence for the compiled programs. Plus data types correspondence between IR codes and source codes is not quite easy to catch.
- A good framework makes a difference. The "macro" module of this compiler helps us reduce the amount of flow-control structures in IR generation.

Jay:

- IR generation is definitely the hardest part.
- Learned that Macros are super powerful; although our language is very lightweight during the parsing stage, the macro module and in-built functions help reduce implementation details in IR generation and adds functionality, especially using recursions.
- Learned about the LLVM lib and how to link things together with C++.

Yifei:

- Learned about how high level language features are mapped to primitive structures
- More clear understanding of the compiling and linking processes
- Begin to appreciate the wonderful (and efficient!) features provided by other popular programming languages