Language Design Proposal: lispy

1 Compiler

1.1 Target Language

Python

1.2 Metalanguage

Python

1.2.1 Reasoning

I'm very experienced with the language; I've written a lexer and parser in Python before too. I want to remove as much overhead as possible so I can focus on implementing the compiler rather than learning a new language. Recent Python versions also support pattern matching. High performance is not a goal for this compiler.

2 Language

2.1 Name

lispy

2.2 Description

A compiled, statically typed, lexically scoped, and type-inferred LISP-like language. Rather than being based on the more modern and complex dialects of LISP, *lispy* is based on LISP 1.5, as described in *LISP 1.5 Programmer's Manual* (McCarthy et al., 1985) and *LISP 1.5 Primer* (Weismann, 1967). However, it has significant adjustments and deviations from LISP 1.5. This helps the language stay small and manageable, but a lot of the differences also stem from the need to make the language statically typed rather than dynamically typed.

2.3 Planned Restrictions

2.3.1 General

- No character objects / strings.
- No input support.
- No debugging, tracing, or error handling.
- No back-trace for runtime errors.
- No distinction between compiled and interpreted code; everything is compiled.
- Many built-in functions will be classified as special forms because the language's type system cannot describe these functions.
- Special forms are not first-class citizens.
- Special forms cannot be redefined.

2.3.2 Syntactic

- No comments.
- No octal numbers.
- No comma delimiter for list elements.
- No dot notation for S-expressions.

2.3.3 Typing

- All lists are homogenous.
- Types of lambda parameters must always be explicitly defined; they are never inferred.
- All branches of conditional expressions must evaluate to the same type.

2.3.4 Cut LISP 1.5 Features

- No macros.
- No arrays.
- No compiler/assembler functions.
- No `PROG`.
- No `QUOTE`.
- No `EVAL` or `EVALQUOTE`.
- No property lists (`GET`, `PUT`, `PROP`, `REMPROP`).
- No in-place list manipulation (`RPLACA`, `RPLACD`, `NCONC`).
- No user-defined functions using machine code.

2.4 Syntax

2.4.1 Abstract Syntax

Note: `*` denotes zero or more of the preceding term. `+` denotes one or more of the preceding term.

```
<name> ::= <string of letters and numbers>
<number> ::= <a decimal number> | "inf" | "nan"
<const> ::= <number> | "true" | "false"

<type> ::= "int" | "float" | "bool"
         | "(" "list" <type> ")"
         | "(" "func" "(" <type>* ")" <type> ")"

<nil> ::= "(" ")"

<func-param> ::= "(" <name> <type> ")"
<lambda> ::= "(" "lambda" "(" <func-param>* ")" <form> ")"

<list> ::= "(" "list" <form>* ")"
<cons> ::= "(" "cons" <form> <form> ")"
<car> ::= "(" "car" <form> ")"
<cdr> ::= "(" "cdr" <form> ")"

<progn> ::= "(" "progn" <form> <form>+ ")"
<set> ::= "(" "set" <name> <form> ")"

<let-binding> ::= "(" <name> <form> ")"
<let> ::= "(" "let" "(" <let-binding>+ ")" <form>+ ")"

<branch> ::= "(" <form> <form> ")"
<cond> ::= "(" "cond" <branch>+ <form> ")"
<select> ::= "(" "select" <form> <branch>+ <form> ")"

<elementary-form> ::= <const> | <name>
<composed-form> ::= "(" <form> <form>* ")"
<special-form> ::= <lambda> | <define> | <list> | <cons> | <car> | <cdr>
                 | <progn> | <set> | <let> | <cond> | <select> | <eq> | <equal>
                 | <evenp> | <lessp> | <null> | <member> | <and> | <or> | <sum>
                 | <prod> | <diff> | <neg> | <inc> | <dec> | <div> | <mod>
                 | <expt> | <sqrt> | <log> | <lb> | <lg> | <ln> | <recip>
                 | <abs> | <min> | <max> | <append> | <extend> | <copy>
                 | <reverse> | <length> | <efface> | <print> | <println>
                 | <map> | <mapcar>
<builtin-form> ::= <greaterp> | <not> | <float> | <floor> | <ceil> | <trunc>
                 | <round> | <logand> | <logior> | <logxor> | <lognot>
                 | <shift>
<form> ::= <elementary-form> | <composed-form> | <special-form> | <builtin-form>
```

```
<program> ::= <form>*

<eq> ::= "(" "eq" <form> <form> ")"
<equal> ::= "(" "equal" <form> <form> ")"

<greaterp> ::= "(" "greaterp" <form> <form> ")"
<evenp> ::= "(" "evenp" <form> <form> ")"
<lessp> ::= "(" "lessp" <form> <form> ")"

<null> ::= "(" "null" <form> ")"
<member> ::= "(" "member" <form> <form> ")"

<not> ::= "(" "not" <form> ")"
<and> ::= "(" "and" <form> <form>+ ")"
<or> ::= "(" "or" <form> <form>+ ")"

<sum> ::= "(" "sum" <form> <form>+ ")"
<prod> ::= "(" "prod" <form> <form>+ ")"
<diff> ::= "(" "diff" <form> <form> ")"
<neg> ::= "(" "neg" <form> ")"
<inc> ::= "(" "inc" <form> ")"
<dec> ::= "(" "dec" <form> ")"
<div> ::= "(" "div" <form> <form> ")"
<mod> ::= "(" "mod" <form> <form> ")"
<expt> ::= "(" "expt" <form> <form> ")"
<sqrt> ::= "(" "sqrt" <form> ")"
<log> ::= "(" "log" <form> <form> ")"
<lb> ::= "(" "lb" <form> ")"
<lg> ::= "(" "lg" <form> ")"
<ln> ::= "(" "ln" <form> ")"
<recip> ::= "(" "recip" <form> ")"
<abs> ::= "(" "abs" <form> ")"
<min> ::= "(" "min" <form> <form>+ ")"
<max> ::= "(" "max" <form> <form>+ ")"

<float> ::= "(" "float" <form> ")"
<floor> ::= "(" "floor" <form> ")"
<ceil> ::= "(" "ceil" <form> ")"
<trunc> ::= "(" "trunc" <form> ")"
<round> ::= "(" "round" <form> ")"

<logand> ::= "(" "logand" <form> <form> ")"
<logior> ::= "(" "logior" <form> <form> ")"
<logxor> ::= "(" "logxor" <form> <form> ")"
<lognot> ::= "(" "lognot" <form> <form> ")"
<shift> ::= "(" "shift" <form> <form> ")"

<append> ::= "(" "append" <form> <form> ")"
```

```
<extend> ::= "(" "extend" <form>  ")"
<copy> ::= "(" "copy" <form> ")"
<reverse> ::= "(" "reverse" <form> ")"
<length> ::= "(" "length" <form> ")"
<efface> ::= "(" "efface" <form> <form> ")"

<print> ::= "(" "print" <form>* ")"
<println> ::= "(" "println" <form>* ")"

<map> ::= "(" "map" <form> <form> ")"
<mapcar> ::= "(" "mapcar" <form> <form> ")"
```

2.4.2 Concrete Syntax

```
(* ------------------------ Character sets ------------------------- *)
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
       | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V"
       | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g"
       | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
       | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "_";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
ws = " " | "\r" | "\n" | "\t" | "\f" | "\v";

(* ---------------------------- Numbers ----------------------------- *)
sign = "+" | "-";
radix = ( digit, "." ) | ( ".", digit );
decimal = { digit }, radix, { digit }
        | digit, { digit };
exponent = ( "e" | "E" ), [ sign, digit ], { digit };
number = [ sign ], ( "inf" | "nan" )
       | [ sign ], decimal, [ exponent ];

(* ---------------------------- Atoms ------------------------------- *)
(* Literal atoms must start with a letter. *)
literal_atom = letter, { letter | digit };
bool = "true" | "false";
atom = number | bool | literal_atom;

(* ------------------------ S-expressions ------------------------- *)
(* Empty lists are valid too. *)
(* List elements must be delimited by >= 1 ws. *)
sexpr_elements = { s_expression, ws }, s_expression;
list_elements = sexpr_elements | s_expression | { ws };

sexp_non_atomic = "(", list_elements, ")";
s_expression = { ws }, ( atom | sexp_non_atomic ), { ws };

(* ---------------------------- Program ----------------------------- *)
(* Consecutive atoms must be separated by >= 1 ws. *)
atoms = { ws }, atom, { ws }, { ws, { ws }, atom, { ws } };
program = { [ { ws }, sexp_non_atomic, { ws } ], [ atoms ] };
```

2.5 Non-trivial Features

1. Higher-order functions (computation abstraction)
2. `cons`, `car`, and `cdr`
3. Type inference

Higher-order Functions Lambdas will not be implemented using Python's existing facilities for functions and lambda functions. Instead, this feature will be implemented as a map of function indices to custom objects representing closures.