PROJECT REPORT
Compiler Project 2010
Perttu Auramo

Table of Contents

PA-LISP language

This language is a minimal Lisp implementation. The language is influenced by Scheme, Clojure and Emacs Lisp.

Syntax

Being a Lisp, the language has a very minimal syntax.

PA-LISP consists of atoms:

- ID

  Example: my_var
- Number

  Example: 456
- String literal

  Example: ”Hello”

And lists at the syntactic level. Lists look like this:

```
(myfun 1 2)
```

They are built from parenthesis and atoms/other nested lists.

Syntax as EBNF

<program> ::= <members>

<sexpr> ::= <atom> | <list>

<atom> ::= <id> | <number> | <string>

<list> ::= "(" ")" | "(" <members> ")"

<members> ::= <sexpr> <sexpr>*

Regular expressions for atoms

Number: \d+

ID: [a-zA-Z*\/\+\-\<\>](\w)

String: ”.*”

Plain script code is ”main”

The PA-LISP equivalent of the ”main function” is any PA-LISP code outside of function definitions (defuns). This code gets executed when the program is run. For example:

```
(println (+ 1 1))
```

Is a valid, full program and can be executed (it prints 2 to the console).

Functions

Functions are defined with ID defun, followed by the desired ID of the function, parameter declarations (a list), and the contents of the function. Example:

```
(defun my_fun (par1 par2)
  (+ par1 par2))
```

The + is one of the pre-defined mathematical functions:+, -, * and /. All of these take two arguments of type integer.

Functions are called with the function ID as the first element of the list:

```
(my_fun 4 5)
```

When functions have no parameters, they are declared like this:

```
(defun noparams() ”retval”)
```

And called like this:

```
(noparams)
```

The return value of a function is always it's last form. For example, for noparams function the return value is the string constant ”retval”.

Built-in functions

The following functions are provided out-of-the-box by PA-LISP:

- Simple math functions: + , - , / , *

  All of them take exactly two arguments and work on integers. Example:

  ```
  (* (+ 1 1) (- 5 3))
  ```

  produces the number four
- println

  Takes one argument, either integer or string and prints it to the console
- Comparison functions: < , > , =

  Take exactly two arguments, compares the first argument to the second. Example:

  ```
  (< 1 2)
  ```

  Produces 1 (true)

Special forms

There are following "special forms" (these are not functions) in the language:

- if
- and
- or

if, and and or behave mostly like in other languages (e.g. Java).

Closely related to these forms is the concept of a ”boolean”, which is implemented so that the number **1** means **true**, and **0** means **false**.

The if special form

The form allows minimum two and maximum three nested PA-LISP structures (lists or atoms). If there is only two nested structures, it means that there is no ”else”. For example:

```
(if (< 1 2)
  (println "1 is smaller than 2"))
```

Is an if form without else. If we'd swap 1 and 2, it wouldn't print anything. Here is an if with an else:

```
(if (< 5 4)
 (my_fun1)
 (my_fun2))
```

The above if program will call function my_fun2.

The return value of an if form is it's true or false branch form's value. For example:

```
(if 1
  9
  88)
```

returns 9. *Remember, **1** meant true, **0** false.*

Or and And special forms

These forms always take two parameters. Examples:

```
(and (firstfun) (secondfun))
```

```
(or 0 1)
```

These two always return either 1 or 0 (true or false). Normal short circuiting is applied, so

for example here:

```
(defun call_func() (println "buhahaa") 1)
(or 1 (call_func))
```

The call_func is never called because the first parameter 1 (true) already short circuits the or and it returns 1. Similar rules apply to and.

Iteration

The only form of iteration supported is that via recursion, example:

```
(defun fib (n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Comments

Commented lines start with the semicolon character ”;”. Example:

```
; The ten here means tenth calculated
; The number is actually 11th in the series
(println (fib 10))
```

Environment prerequisites

To run run the compiler and the resulting programs, the following runtimes have to be installed:

- Java Virtual Machine (tested with Java 6)
- Ruby (tested with version 1.8)

These runtimes are installed in Computer Science department's Linux machines by default.

Using the compiler

Using the shell script commands

Once extracted, the PA-LISP compiler root directory contains the following shell (bash) scripts:

- plc
- plrun

plc is a thin wrapper on top of the Ruby script which compiles the source to jasmin assembly, and the Jasmin tool. *plrun* is a wrapper for the Java call to run the resulting class.

If you have, say, a PA-LISP file called *foo.pal* this is how you compile and run it:

```
<PA-LISP-DIR>/plc foo.pal
<PA-LISP-DIR>/plrun foo
```

If you add the <PA-LISP-DIR> to your PATH environment variable, you don't have to use the path before the commands:

```
plc foo.pal
plrun foo
```

The plc command produces these files to your working directory:

- *foo.j* Jasmin assembly file which can be inspected separately
- *foo.class* An ordinary Java class file which contains a main method and all other code the foo.pal source file had as Java bytecode.

The *plrun* command just runs the Java class' main method and adds the needed PaLispRuntime class to the classpath. *Note that plrun's argument doesn't contain the .class suffix!*

To compile existing example programs, see *<PA-LISP-DIR>/testdata* directory. For

example, you could compile and run the program:

testdata/fibotest1.pal

which prints the 10th calculated fibonacci number (11th in sequence) or :

testdata/sligthly_larger_test_program.pal

Which does a little bit more than most of the other test programs in *testdata.*

Using the raw commands

If you are using e.g. a Windows operating system which doesn't have bash shell by default, you can use the raw java- and ruby-commands for compilation directly. *Note that this compiler has not been tested on Windows, Linux or Mac OSX is strongly recommended!*

To compile foo.pal to jasmin assembly, run:

```
ruby -I <PA-LISP-DIR>/lib <PA-LISP-DIR>/lib/plc.rb foo.pal
```

And to compile the resulting foo.j to Java class:

```
java -jar <PA-LISP-DIR>/tools/jasmin.jar foo.j
```

And to run the resulting foo.class:

```
java -cp <PA-LISP-DIR>/palisp-runtime/bin:. foo
```

For windows, you should substitute colon with semicolon.

Source code, tests and resources

The compiler is written using Emacs with tab length 2 setting (the default Emacs settings for Ruby). If the code looks ugly, try setting the tab length to 2 and if you have Emacs available,

try using that.

Locations of resources

The source code of the compiler is under:

<PA-LISP-DIR>/lib

Test code is under:

<PA-LISP-DIR>/test

And testdata (example programs used in testing) is here:

<PA-LISP-DIR>/testdata

PA-LISP uses a runtime library which contains some basic methods like println, plus etc. These runtime methods are implemented in one Java class: *PaLispRuntime.* The location of the binary is:

<PA-LISP-DIR>/palisp-runtime/bin

And the Java source for the class is in here:

<PA-LISP-DIR>/palisp-runtime/src

There is also a small, one-line bash script to compile the runtime library:

<PA-LISP-DIR>/palisp-runtime/compile.sh

Running tests

Individual test modules containing Unit Tests can be run in the <PA-LISP-DIR> with the command:

```
ruby -I lib test/<test_module>.rb
```

For example, to run Unit Tests for the Scanner, run:

```
ruby -I lib test/scanner_test.rb
```

To run all Unit Tests, run command:

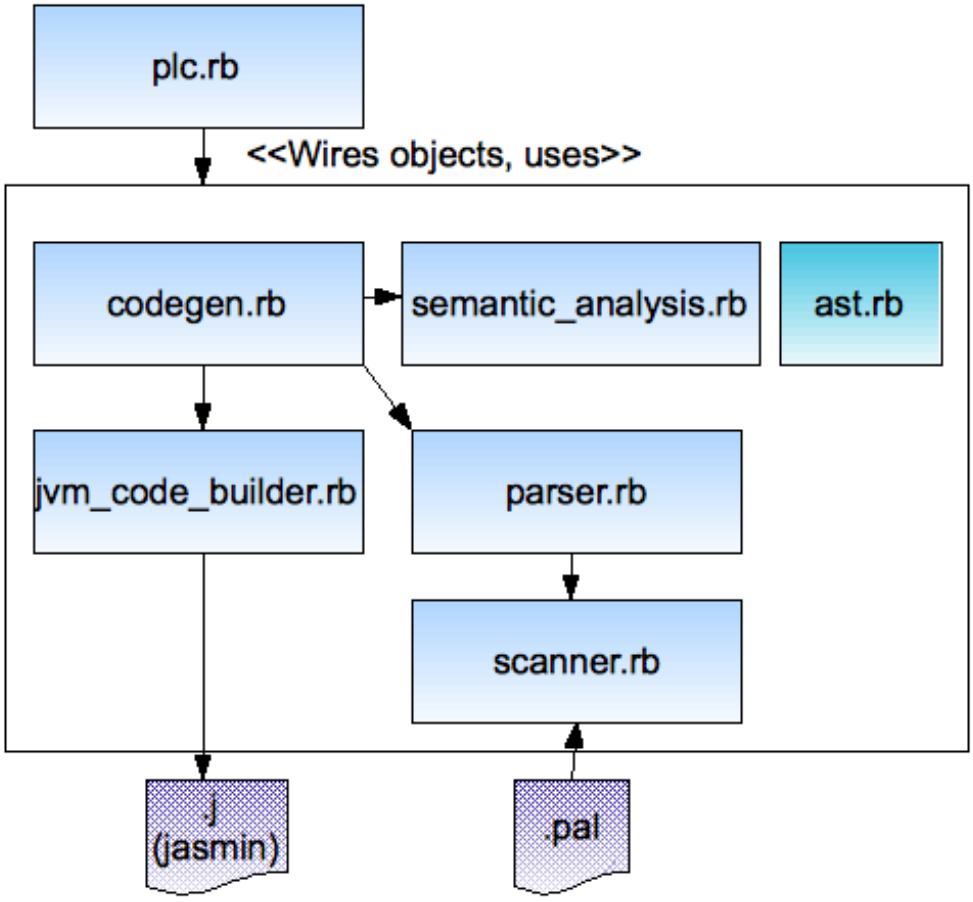
```
ruby -I lib:test test/plc_test_suite.rb
```

There is also a convenience script in <PA-LISP-DIR> doing exactly the above called *test-run.sh*

The module e2e_test.rb uses the compiler to create class and .j files under <PA-LISP-DIR>. So after running either all the tests or this module specifically, there is a huge amount of these files lying around. To get rid of those, use the script clean.sh.

Compiler Design

The compiler consists of seven Ruby modules. These modules contain one or many classes. The following image depicts the high level design of the compiler.



Module view

The modules are explained below.

plc.rb

This is the main module of the compiler. It instantiates objects from other modules and wires those objects together. Notably it instantiates objects from classes Parser, JvmBuilder and ClassGenerator. This is why plc.rb is drawn outside of the white box which represents the ”core system”, it just glues everything together but in itself it's not that interesting.

plc.rb also calls ClassCenerator's generate-method which launches the compilation process. It also handles the low-level details of file-naming.

codegen.rb

Orchestrates the whole compilation process. Uses key objects from other modules for the process: a *Parser* instance from parser.rb, a *JvmBuilder* instance from jvm_code_builder.rb, *VariableReferenceDecorator* instance from semantic_analysis.rb and a *Program* ast node instance from ast.rb. Note that these dependencies are not all reflected via the *require* statements at the top of the module. This is because of Ruby's dynamic nature (duck typing) and because the objects have already been instantiated elsewhere.

When codegen.rb's ClassGenerator instance's generate method is called, it:

- retrieves the abstract syntax tree from the *Parser* object
- decorates it with *VariableReferenceDecorator* (sets the references to correct declarations)
- Starts going through the abstract syntax tree and generating code via the *JvmBuilder* object which handles the really low-level details.

jvm_code_builder.rb

Contains the class *JvmBuilder* which handles:

- Writing the jasmin assembly to a file
- Generating labels
- Keeping track of limits (local variables and stack)
- Buffering of method contents to emit the correct limits at the end of method

parser.rb

The class *GenericParser* Uses Scanner from scanner.rb to pull tokens from input. Creates a generic abstract syntax tree from tokens. This tree consists of lists, ids numbers and string literals.

The class *Parser* uses *GenericParser*'s generic abstract syntax tree as an input and creates a more specific, further processed AST from it. For example, this AST contains no more lists, but instead they are replaced with Defuns, FuncCalls. This is done to make it easier for semantic analysis and code generation to focus on their core tasks.

semantic_analysis.rb

The class *VariableReferenceDecorator* is used to put the variable references in place: each *VarRef* node should have a *VarDecl* target after the tree has been processed. It also sets *ignore_retval* flags in place for the AST tree nodes. These are used in code generation phase to pop out elements from the stack which are not used (for example function calls which return values are not used).

scanner.rb

Reads PA-LISP input files, skips whitespaces and comments and converts the rest to valid PA-LISP tokens if the input file is a valid program.

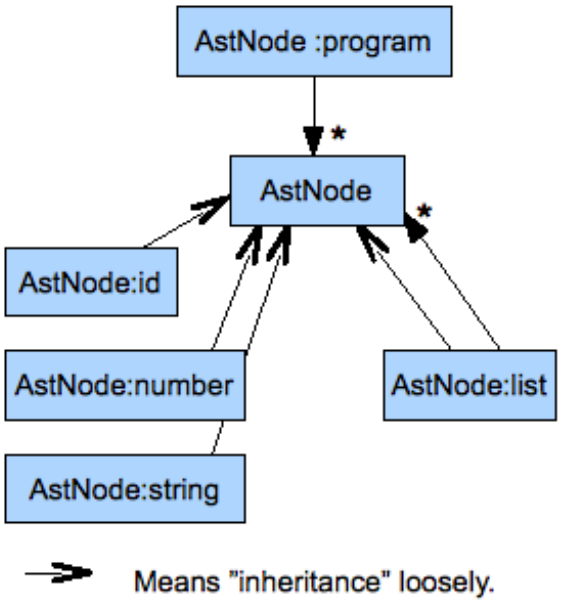
ast.rb

This module is so widely used, that it's color-coded differently and has no arrows pointing to it. The color tries to communicate the fact that this module is a basic building block for almost all others, and there is no sense to draw arrows to it from all other modules.

The module contains the classes for the abstract syntax tree nodes. Both the generic tree nodes and the more specific tree nodes (see parser.rb above for details) inherit from AstNode. Specific tree has more special classes for nodes, for example *MainFunc*, *VarDecl* etc.

Abstract Syntax Tree

Because PA-LISP syntax is so simple, the generic abstract syntax tree looks like this:



Generic AST

The picture represents the Abstract Syntax tree generated by *parser.rb*'s *GenericParser.* Only objects of class AstNode are created, and the type attribute, token's value and contained children vary.

The more specific AST generated by *parser.rb*'s *Parser* class which uses the Generic AST as an input contains certain specific subclasses of *AstNode*, like *Defun, VarDecl* etc. For details see *ast.rb*.

Ruby as the implementation language

This chapter tries to make it easier to read the source code of the compiler for a reader who is not familiar with Ruby. I try to list those features which might be confusing or less common among other dynamic languages like Python, Groovy etc.

Blocks

Ruby programs contain a lot of blocks. They look like this:

```
5.times do |i|
  puts i
end
```

Or like this:

```
my_list.each { |item| puts item }
```

The first example prints numbers 0-4 to output, the second prints the items of a list.

The method *times* exists on a number and it accepts a Ruby *block* which gets called back as many times as the number tells us. The callback value to our caller code is passed between pipe characers ”|”.

The *do..end* form is equivalent to the curly bracket form we see on the list iteration example. In that case we pass our block to list's each method which then gets called back as many times as there are items in the list with the list item as the parameter between pipe characters.

Blocks have access to the surrounding scope, so they are very similar to Closures or Lambdas in other languages.

Coding convention regarding function's parenthesis

In Ruby, parenthesis for method declarations and method calls are optional. It is considered

a ”best practice” in the Ruby community to leave the parenthesis out from both method declaration and method call if the method has no arguments. Otherwise the parenthesis should be used. This style is followed in PA-LISP compiler's source code.

Constructors and member variables

In Ruby, constructors are defined using the keyword *initialize.* The member variables are set using the prefix ”@” and they are always referred to with ”@” as well. Example:

```
class MyClass
  def initialize(param1, param2)
    @member1 = param1
    @member2 = param2
  end
  def my_fun
     @member1  # returns the member1 instance variable
  end
end
```

Ruby also supports ”properties”, meaning that getters and setters don't have to be manually written like in e.g. Java. In Ruby, the methods attr_reader and attr_accessor are used to expose member variables:

```
class MyClass
  attr_reader :member1
  attr_accessor :member2
  def initialize(param1, param2)
    @member1 = param1
    @member2 = param2
  end
end
```

In the above example, the *attr_reader call* with the member1 name parameter (as a Ruby symbol, they start with colon) causes the @member1 to be readable from other objects. *attr_accessor* is called for member2 which means that other objects can also set this member.

It is also a convention (don't ask why :)) to leave parenthesis out of attr_reader and attr_accessor calls.

Character literals

Ruby's character literals are prefixed with a question mark and followed by the character without any quotes. Examples:

```
?a
?\n
```

Many times in the PA-LISP compiler code, the character is first converted to a string, with the *chr* method, and after that it looks ”normal”, like this: ”a”.