XV Conference on Applied Mathematics
N. Krejić, Z. Lužanin, eds.
Department of Mathematics and Informatics
Novi Sad, 2002, pp. xx-xx

AN IMPLEMENTATION OF LISPKIT LISP IN JAVA[1]

Miloš Radovanović[2], Mirjana Ivanović[2]

Abstract. Functional programming languages (FPL) and their implementations are still very interesting topics in the computer community. Declarative languages are also extremely interesing in the field of inter-agent communication. Thus it seems that a FPL could be a good starting point for the implementation of particular Agent Communication Languages. In this paper a description of a concrete LispKit LISP system is given. The system is implemented in Java, and consists of a translator from LispKit LISP to SECD machine code and an interpreter for that code.

The emphasis of the paper is on simplicity and clarity of underlying concepts, rather than actual practical use of the system. But, it can be used as an interesting tool for educational purposes, in the field of functional programming and in the field of compiler construction, as well. Particular focus is on features of the Java programming language which are extensively used in the implementation, such as packages, class inheritance, interfaces, garbage collection and the standard library. The implemented LispKit LISP system will be analyzed as a possible core language for the realization of a concrete ACL.

1. Introduction

The development of programming languages over the years has been marked by attempts to construct languages that are more similar to human concepts of abstraction and communication, while preserving efficiency of translation and execution of programs written in those languages.

An important paradigm of programming is the *functional programming style*. In this style, program execution is based on the notion of a *function*. Basically, there are only three activities which form the foundation of the whole programming process:

[1]This work is part of the project *Development of (intelligent) techniques based on software agents for application in information retrieval and workflow*, 2002-2004, supported by Ministry of Science, Technology and Development (Republic of Serbia), project no. 1844.

[2]Department of Mathematics and Informatics, University of Novi Sad, Trg D. Obradovića 4, 21000 Novi Sad, Serbia and Montenegro, e-mail: radacha@Eunet.yu (M. Radovanović), mira@im.ns.ac.yu (M. Ivanović)

- Function definition, that is, assignment of the expression which calculates the value of the function to the function name.
- Application of a function to a list of arguments.
- Function composition.

One programming language which supports the functional programming style, and which will be discussed in this paper, is a variant of LISP called LispKit LISP [6]. It is a small *purely functional* programming language.

A common trait of all dialects of LISP is the *S-expression*. It is the basic data structure and the main building block of all LISP programs. An **S-expression** is either:

1. an *atom* (symbolic or numeric), or
2. a *constructed expression* `(`s_1`.`s_2`)`, where s_1 and s_2 denote S-expressions.

The constructed expression is also referred to as the *cons* expression. The s_1 in the former definition is often called the *head*, or *car* part of the expression, and s_2 the *tail*, or *cdr* part.

More formally, using the Extended Backus-Naur Form, the S-expression can be defined as:

```
<atom>           ::=  <symbolic> | <numeric>
<symbolic>       ::=  <letter> { <letter> | <digit> }
<numeric>        ::=  <integer>
<integer>        ::=  ['-'] digit { <digit> }
<SExpression>    ::=  <atom> | '(' <SExpressionList> ')'
<SExpressionList> ::=  <SExpression> | <SExpression> '.' <SExpression> |
                       <SExpression> <SExpressionList>
```

The implementation of a parser for S-expressions in Java will closely follow this definition. Note that room was left for possible extension of the numeric atom with other types of numbers.

The rest of the paper is organized as follows. In Section 2 the essence of the LispKit LISP language is presented. Section 3 outlines the basics of the SECD machine, while Section 4 gives a description of a LispKit system implemented in Java. In the last section, a conclusion and some comments on possible further work are given.

2. The LispKit LISP Language

Programs written in the programming language LispKit LISP are in form of S-expressions, which have to abide to certain syntactic rules that will briefly be described in this section.

A LispKit LISP *expression*, in general, is a list with the first element denoting the type of the expression, and the following elements being the arguments. Every expression returns a value.

The simplest expression is the QUOTE expression, which simply returns its single argument. Then, there are arithmetic expressions ADD, SUB, MUL, DIV and REM, that return a numeric atom representing the result of applying the respective arithmetic operation to their two arguments. Logical expressions return either `T` or `F`, which depict logical values of *true* and *false*. The EQ expression tests whether its two arguments are equal atoms, while the LEQ expression checks if its first numeric argument is less than or equal to the second. The ATOM expression returns `T` if its single argument is an atom. LispKit LISP expressions that manipulate with the structure of S-expressions include CAR and CDR, which return the *car* and *cdr* parts of their single arguments, together with CONS, which returns the *cons* expression of its two arguments. The IF expression, based on the logical value of its first argument, returns the value of its second or third argument. The value of the LAMBDA expression is a *function*. Formal arguments of this function are given as a list of symbolic atoms that is the first argument of the LAMBDA expression, and the body of the function is given in the second argument. A *function call* expression enables such a function to be evaluated. LET and LETREC expressions permit the assignment of symbols to various LispKit LISP expressions. Their first argument is an expression, and the second is a list of *cons* pairs whose heads are symbolic atoms and tails are expressions. The values of LET and LETREC expressions are values of their first argument, calculated after substitutions of symbols with approrpiate expressions are made. The main difference between LET and LETREC is that, with LETREC, the defined symbols are also visible in the expressions from the list of *cons* pairs, not just from the main expression. This permits the definition of recursive functions. A LispKit LISP program is either a LAMBDA, LET or LETREC expression.

A formal definition of LispKit LISP syntax, using an EBNF-like grammar, follows.

```
<symbolicList> ::= NIL | (<symbolic> {<symbolic>})
<exp>          ::= <symbolic>|<quoteExp>|<eqExp>|<leqExp>|<carExp>|
                   <cdrExp>|<atomExp>|<consExp>|<ifExp>|<lambdaExp>|
                   <letExp>|<letrecExp>|<callExp>
<quoteExp>     ::= (QUOTE <SExpression>)
<eqExp>        ::= (EQ <exp>)
<aritExp>      ::= (<aritOp> <exp> <exp>)
<aritOp>       ::= ADD | SUB | MUL | DIV | REM
<leqExp>       ::= (LEQ <exp> <exp>)
<carExp>       ::= (CAR <exp>)
<cdrExp>       ::= (CDR <exp>)
<atomExp>      ::= (ATOM <exp>)
<consExp>      ::= (CONS <exp> <exp>)
<ifExp>        ::= (IF <exp> <exp> <exp>)
<lambdaExp>    ::= (LAMBDA <symbolicList> <exp>)
<letExp>       ::= (LET <exp> { (<symbolic> . <exp>) })
<letrecExp>    ::= (LETREC <exp> { (<symbolic> . <lambdaExp>) })
<callExp>      ::= (<function> { <exp> })
<function>     ::= <symbolic>|<lambdaExp>|<letExp>|<letrecExp>
<program>      ::= <letrecExp>|<letExp>|<lambdaExp>
```

3. The SECD Machine

The SECD machine, originally intorduced by Landin, is still a useful structure for implementation of functional programming languages. It consists of four registers, which hold S-expressions [6]:

s	the stack	used to hold intermediate and final results
e	the environment	used to hold the values of variables during evaluation
c	the control list	used to hold the machine-language program being executed
d	the dump	used as a stack to save values of other registers.

The basic SECD machine language consists of 21 instructions. Execution of an instruction results in changes to the contents of one or more registers, that is, in a change of the machine state. This may be denoted as $s\ e\ c\ d \rightarrow s'\ e'\ c'\ d'$, where the part left of the arrow represents the machine state before instruction execution, and the part on the right the state after. Changes to the SECD machine state, initiated by appropriate instructions, are presented below.

1	LD	$s\ e$ (LD $locate(i,e).c$) d	$\rightarrow$	$(a.s)\ e\ c\ d$
2	LDC	$s\ e$ (LDC $a.c$) d	$\rightarrow$	$(a.s)\ e\ c\ d$
3	LDF	$s\ e$ (LDF $c'.c$) d	$\rightarrow$	$((c'.e).s)\ e\ c\ d$
4	AP	$((c'.e')\ v.s)\ e$ (AP.c) d	$\rightarrow$	NIL $(v.e)\ c'\ (s\ e\ c.d)$
5	RTN	$(a)\ e'$ (RTN) $(s\ e\ c.d)$	$\rightarrow$	$(a.s)\ e\ c\ d$
6	DUM	$s\ e$ (DUM.c) d	$\rightarrow$	$s\ (\Omega.e)\ c\ d$
7	RAP	$((c'.e')\ v.s)\ (\Omega.e)$ (RAP.c) d	$\rightarrow$	NIL $rplaca(e',v)\ c'\ (s\ e\ c.d)$
8	SEL	$(a.s)\ e$ (SEL $c_T\ c_F.c$) d	$\rightarrow$	$s\ e\ c_a\ (c.d)$
9	JOIN	$s\ e$ (JOIN) $(c.d)$	$\rightarrow$	$s\ e\ c\ d$
10	CAR	$((a.b).s)\ e$ (CAR.c) d	$\rightarrow$	$(a.s)\ e\ c\ d$
11	CDR	$((a.b).s)\ e$ (CDR.c) d	$\rightarrow$	$(b.s)\ e\ c\ d$
12	ATOM	$(a.s)\ e$ (ATOM.c) d	$\rightarrow$	$(l.s)\ e\ c\ d$
13	CONS	$(a\ b.s)\ e$ (CONS.c) d	$\rightarrow$	$((a.b).s)\ e\ c\ d$
14	EQ	$(a\ b.s)\ e$ (EQ.c) d	$\rightarrow$	$(b = a.s)\ e\ c\ d$
15	ADD	$(a\ b.s)\ e$ (ADD.c) d	$\rightarrow$	$(b + a.s)\ e\ c\ d$
16	SUB	$(a\ b.s)\ e$ (SUB.c) d	$\rightarrow$	$(b - a.s)\ e\ c\ d$
17	MUL	$(a\ b.s)\ e$ (MUL.c) d	$\rightarrow$	$(b \cdot a.s)\ e\ c\ d$
18	DIV	$(a\ b.s)\ e$ (DIV.c) d	$\rightarrow$	$(b/a.s)\ e\ c\ d$
19	REM	$(a\ b.s)\ e$ (REM.c) d	$\rightarrow$	$(b\%a.s)\ e\ c\ d$
20	LEQ	$(a\ b.s)\ e$ (LEQ.c) d	$\rightarrow$	$(b \leq a.s)\ e\ c\ d$
21	STOP	$s\ e$ (STOP) d	$\rightarrow$	$s\ e$ (STOP) d

Function *locate* returns the element in the environment at the specified 2-tuple position, while *rplaca* replaces the head of the given *cons* expression and returns the resulting expression. For every LispKit LISP expression there exists a rule for translating it to a sequence of SECD machine instructions.

4. An Implementation

This section presents a possible implementation of the described LispKit LISP system in Java.

4.1. Basic Structure

The system is organized into package `lispkit`. Subpackage `sexp` contains the implementation of the S-expression data type, while subpackage `secd` implements the registers and instructions of the SECD machine. The input of S-expressions, LispKit LISP syntax checking, the compiler which translates LispKit LISP programs to SECD machine code, the interpreter for SECD machine code and the facilities for error handling are implemented in the root of package `lispkit`.

4.2. S-expressions

The S-expression data type can be naturally represented in an object-oriented manner with the class hierarchy shown in Figure 1. Class `SExp` is the root of the hierarchy, and, theoretically, it is an abstract class. The "leaf" classes are the ones whose instances may actually appear in a program, while anything else represents an error. But, class `SExp` is not implemented as a Java *abstract* class, nor as an *interface*, for that matter. Every method applicable to the leaf classes is defined in class `SExp` to simply *throw* an exception of type `SExpException`.

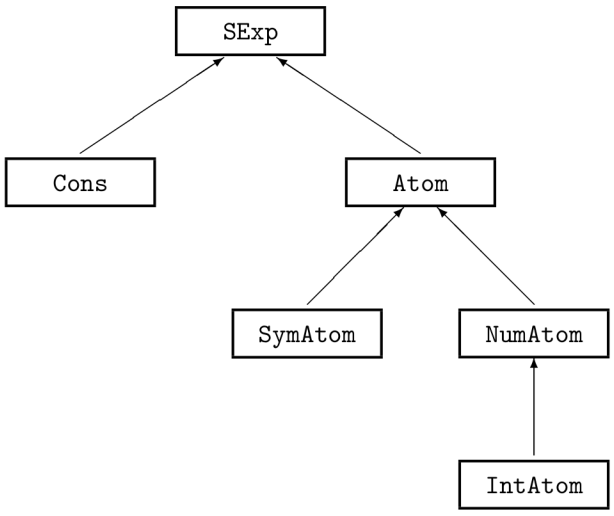

Figure 1: S-expression class hierarchy

For instance, class `IntAtom` has method `intValue()` which returns the *int* value of the integer atom. If this method gets called from, say, an instance of class `Cons`, the `intValue()` from `SExp` will be executed, indicating an error. The exception may later be caught and the error treated according to the context in which it appeared.

The input of S-expressions has been implemented in a straightforward manner using the compiler generator *Coco/R for Java* [9], with the S-expression grammar given in Section 1. While performing recursive descent parsing, two `SExp`s are constructed—one representing the actual S-expression from the input text, and the other having the same structure as the former, but with all atoms replaced with 2-tuple `Cons`es of `IntAtom`s, that is, by pairs of integers representing the line and column numbers of the concerning atom in the input text. This simple scheme will be an important contribution to error reporting discussed later.

The output of S-expressions can be done by defining appropriate `toString()` methods.

4.3. Syntax Checking

Checking the syntax of a LispKit LISP program, once it has been loaded into an `SExp` is relatively simple. A recursive descent parser can be made in a rather straightforward fashion, using the grammar from Section 2. The one thing to be kept in mind is that parsing is done through an instance of class `SExp`, and not through an input character stream.

For every nonterminal in the grammar, a procedure with the same name is defined, all being placed in class `LispSyntaxChecker`. Every such procedure needs to have three parameters of type `SExp`: the portion of the input program to be checked, the current *environment*, and the "parallel" structure of line and column numbers. The environment is nothing more than a list of symbols, organized into sublists, each sublist representing a *block*. A new block is defined and a new list of symbols is added to the environment every time a LAMBDA, LET or LETREC expression is being processed. On completion of the analysis of the block, the added sublist is removed.

For example, consider the procedure corresponding to the nonterminal <addExp>:

```
private static void addExp(SExp e, SExp n, SExp num) {
  try {
    if (e.car().eq("ADD")) {
      exp(e.cadr(), n, num.cadr());
      exp(e.caddr(), n, num.caddr());
      if (!e.cdddr().eq(SymAtom.NIL))
        LispSynError(6, e, num.car()); // invalid ADD expression
    }
    else
      LispSynError(6, e, num.car());
```

```
    }
    catch (SExpException ex) {
      LispSynError(6, e, num);
    }
  }
```

In case the S-expression `e` does not have the expected overall structure, an `SExpException` will be caught, and the error will be reported. An error will also be signaled if `e` is a longer list than expected, or if `e` does not begin with the symbolic atom `ADD`. In any case, parsing of the remaining LispKit LISP code will resume normally, making detection of additional errors possible.

4.4. The Compiler

Earlier it was said that there are precise rules for translating valid LispKit LISP expressions into SECD machine code. If $e{*}n$ denotes SECD machine code that is the result of translating the LispKit LISP expression e with respect to the environment n, and | denotes a list concatenation operator, then the translation rules for all LispKit LISP expressions can be written in a simple way. Some of those rules are given below.

`(QUOTE` s`)`$*n$	=	`(LDC` s`)`
`(ADD` e_1 e_2`)`$*n$	=	$e_1{*}n \mid e_2{*}n \mid$`(ADD)`
`(LAMBDA (`$x_1 \ldots x_k$`)` e`)`$*n$	=	`(LDF` $e{*}$`((`$x_1 \ldots x_k$`).`n`)|(RTN))`

Implementation of such translation is only a matter of carefully expressing the translation rules in Java. It is achieved by class `Compiler` with method

```
static SExp compile(SExp e);
```

which, for a given LispKit LISP program `e` returns the appropriate SECD code. Method `compile`, in turn, simply calls

```
private static SExp comp(SExp e, SExp n, SExp c);
```

which returns SECD code of the form `e*n|c`. Here, `c` acts as an accumulating parameter, holding the already translated parts of the expression. As an example, a portion of method `comp` implementing the rules mentioned would be

```
if (e.car().eq("QUOTE"))
  return new Cons(2, new Cons(e.cadr(), c));
else if (e.car().eq("ADD"))
  return comp(e.cadr(), n, comp(e.caddr(), n, new Cons(15, c)));
else if (e.car().eq("LAMBDA")) {
  SExp body = comp(e.caddr(), new Cons(e.cadr(), n),
                   new Cons(5, SymAtom.NIL));
  return new Cons(3, new Cons(body, c));
}
  ...
```

where 2, 15, 5 and 3 are numeric codes for approrpiate SECD instructions (see Section 3).

It is convenient to wrap the body of method `comp` in a `try...catch` block, so that any `SExpException` may be caught, and the error reported, similarly to the way it was done in class `LispSyntaxChecker`. But, such errors should never occur in practice, so a manifestation of an error at this stage would indicate a failure on the part of the syntax checker, and compilation would have to be aborted immediately. After extensive testing, the current implementation showed no signs of failing in such fashion.

4.5. The Interpreter

The implementation of the "physical" parts of the SECD machine—the "circuits" which compose the registers and the microcode of the instructions, can be conveniently grouped into subpackage `secd` of package `lispkit`. Within this subpackage, class `Reg` contains the registers: `public static SExp s, e, c, d, w;` where `w` is an internal "working register" used for storing temporary values during execution of some instructions. The instructions themselves are implemented by an *interface* `Instruction`, containing a sole method

```
public void execute() throws SExpException, STOPException;
```

For each SECD instruction, a separate class which *implements* interface `Instruction` is defined. For example, the ADD instruction is implemented by the following class.

```
public class ADDInstruction implements Instruction {
  public void execute() throws SExpException {
    Reg.s = new Cons(Reg.s.cadr().intValue()+Reg.s.car().intValue(),
                     Reg.s.cddr());
    Reg.c = Reg.c.cdr();
  }
}
```

Java's garbage collection prooves handy with such register manipulation, since there are no worries about the exessive amounts of `SExp`s which may become unreachable somewhere in memory. A `STOPException` is explicitly `thrown` by method `execute` of class `STOPInstruction`:

```
public class STOPInstruction implements Instruction {
  public void execute() throws STOPException {
    throw new STOPException();
  }
}
```

to announce the end of program execution. A `thrown` `SExpException`, on the other hand, signals a runtime error, which may be a result of a logical error in programming—a type mismatch, or some other kind of programming error which was not anticipated by `LispSyntaxChecker`.

The interpreter itself is implemented in a separate class `Interpreter`, outside package `secd`. Its only method `static SExp exec(SExp code, SExp argList);` takes as arguments the SECD machine program to be executed, and the list of parameters to be passed to it, and returns the result of the program's computations. An array of `Instruction`s named `instr` contains instances of all instructions from package `secd`, which are indexed by their respective numeric codes. Therefore, after the initialization of registers:

```
Reg.s = new Cons(argList, SymAtom.NIL);
Reg.e = SymAtom.NIL;
Reg.c = code;
Reg.d = SymAtom.NIL;
```

the main execution loop will simply be

```
for (;;)
  instr[Reg.c.car().intValue()].execute();
```

This statement needs to be nested inside a `try...catch` block, handling an `SExpException` as a runtime error and a `STOPException` by returning the top element of the stack:

```
return Reg.s.car();
```

A runtime error may also occur during the execution of this statement, if `Reg.s` is not of the expected structure, so a nested `try...catch` is necessary to handle a possible `SExpException`.

5. Conclusion

The purpose of this work was not to create a powerful LISP system applicable in everyday programming, but rather to concentrate on those aspects of the system which illustrate various implementation techniques, and features of the programming language Java. An attempt was made to accompany the simplicity and clarity of LispKit LISP and SECD machine design by an adequate implementation of a system. Java mechanisms, such as garbage collection, object oriented facilities (class inheritance, interfaces, method overloading, polymorphism), exceptions and the standard library were extensively used where they seemed a natural choice.

Further work on this implementation may be split in two directions. The first direction will be concerned with improvements to the language and the implementation as a programming tool, like adding new LISP expressions, with corresponding SECD machine instructions if needed, new types of numeric atoms, closer linking with Java (perhaps through introduction of object oriented features in LispKit LISP), or, in fact, any kinds of extensions which would make the system more suitable to a specific application in programming.

The second direction of system improvements will be concerned with the area of Multi-Agent Systems. Agent communication plays an extremely important

role in MASs, and therefore the existence of a suitable Agent Communication Language is essential. An architecture suitable for building MASs, called AJA [2, 3], was developed at the Institute of Mathematics in Novi Sad. In [3] several ACLs for the communication of AJA agents are proposed. It seems that this LISP system could be a good starting point for the implementation of a useful KQML-like ACL [8, 1, 7, 5]. It already posesses several advantages:

- Declarative style, with relatively simple, clear and concise language constructs.
- Java as the implementation language (of both the LISP system and AJA) offers a lot of possibilities for further improvements.

The *performatives* of the ACL could be realized as LISP function calls, where the functions themselves may be implemented in LISP or some other language. If necessary, the S-expression parser can be extended to allow the naming of performative (function call) parameters.

References

[1] ARPA Knowledge Sharing Initiative. Specification of the KQML Agent-Communication Language. ARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993.

[2] Badjonski, M., Implementation of Multi-Agent Systems using Java. MSc thesis, Institute of Mathematics, Faculty of Science, University of Novi Sad, Yugoslavia, 1998.

[3] Badjonski, M., Adaptable Java Agents (AJA)—a Tool for Programming of Multi-Agent Systems. Draft version of PhD thesis, Institute of Mathematics, Faculty of Science, University of Novi Sad, 2002.

[4] Budimac, Z., Ivanović, M., An Implementation of Functional Language Using S-expressions. In Proceedings of 14th Information Technologies Conference "Sarajevo 1990", pp. 111.1–111.8, Sarajevo, 1990.

[5] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification. Available at http://www.fipa.org/specs/fipa00061/

[6] Henderson, P., Functional Programming—Application and Implementation. London: Prentice Hall, 1980.

[7] Labrou, Y., Finin, T., A Proposal for a new KQML Specification. Technical Report TR-CS-97-03, University of Maryland Baltimore County, 1997.

[8] Labrou, Y., Finin, T., Peng, Y., Agent Communication Languages: The Current Landscape. IEEE Intelligent Systems, 14(2) (1999), 45-52.

[9] Mössenböck, H., Coco/R for Java, amended by P. D. Terry. Electronic document, available as http://cs.ru.ac.za/homes/cspt/javacoco.htm

[10] Radovanović, M., Implementacija programskog jezika LispKit LISP u Javi. BSc thesis, Institute of Mathematics, Faculty of Science, University of Novi Sad, 2001.