A DEFINITION OF AN ISWIM-LIKE LANGUAGE VIA SCHEME

Mirjana Ivanović, Zoran Budimac

University of Novi Sad
Faculty of Natural Sciences and Mathematics, Institute of Mathematics
Trg D. Obradovića 4
21000 Novi Sad, Yugoslavia

ABSTRACT: A full definition of a purely functional ISWIM-like language is given in the paper. A set of transformation rules for the translation of ISWIM expressions to expressions of a widely accepted Scheme programming language is given. The paper is supported with appropriate examples of equivalent expressions in both languages.

1 INTRODUCTION

The research in the field of functional (or applicative) programming has been of growing interest to computer scientists since its origin in the 1950's. Functional (or applicative) programming languages, especially pure ones, are of great importance for the future of computing, because they could solve the so-called "software crisis". Functional programs are short, concise, easy to maintain, and their (in)correctness is easily proved formally. They also offer a natural approach to parallelism and parallel programming languages.

Purely functional languages lack everything that is essential to procedural (or imperative) languages: statements, explicit sequencing and side effects. Purely functional languages are referentially transparent, insensitive to evaluation order, have strong mathematical basis and a small set of built-in features. Therefore, they are easy to learn and follow. Furthermore, all program identifiers are lexically scoped (i.e. bound at compile-time and not at run-time) and all functions are first-class objects (i.e. have the same rights like other data types). Finally, most purely functional languages have non-strict semantics (i.e. expressions are evaluated only when necessary), which gives those languages the potential of dealing with infinite data structures. For more details and an overview of functional programming languages and style, see for example [1,2,5].

The key issue in functional programming research is the definition of purely functional languages and implementation of their processors. Such languages must have friendly syntax resembling mathematical notation, and they have to be simple, concise, and, above all, strongly defined, so that correctness proofs could be completed. At the same time, language processors have to be fast, efficient, available on most hardware platforms, easy to implement and use etc. Currently, there is no functional language and its processor which fulfill above conditions.

The paper presents a full definition of an extension of purely functional language ISWIM (called UNS-ISWIM) and suggests a simple way of its implementation by translation to a functional subset of another, already implemented language - Scheme. With such approach, the processor for the new language can be easily implemented and made available in a very short time on a wide range of hardware platforms (at least as a prototype). Once translated into Scheme, UNS-ISWIM program can be immediately executed on widely available Scheme processors or further compiled into machine languages of various real or virtual machines.

On the other hand, friendly syntax, simplicity, compactness and purity are all features of UNS-ISWIM (and not of Scheme), which easily justifies the need for a new language beside the existing Scheme. Thus, the new language could be one of the rare strongly defined, purely functional languages with friendly syntax widely available even on low-end machines.

The paper is organized as follows: section 2 contains a short history of original ISWIM, in section 3 formal definitions of syntax and semantics of primitive operations are given, section 4 presents a set of transformation rules

for the translation of expressions of a defined language to expressions of Scheme and section 5 contains a short example of the translation. Section 6 concludes the paper.

2 HISTORY AND CHARACTERISTICS OF ISWIM

ISWIM (short for: If you See What I Mean) was intended to be a set of purely functional languages with common basis. It was introduced in [3] and represented an important step in the development of functional languages. Perhaps the two most important improvements were: the strong influence of lambda-calculus to the design and features of the language and the implementation technique via (virtual) SECD machine [4]. It can be said that ISWIM was the first declarative language and a predecessor of so-called modern functional languages whose best known representatives today are Miranda (trademark of Research Software Ltd) and Haskell [2].

From the point of view of LISP dialects (the only language with similar characteristics at the time), ISWIM introduced:

i) infix operators
ii) parentheses-free syntax
iii) local definitions in the form of let or where blocks

ISWIM is a functional language which satisfies all major criteria for a functional language [1,2]: it lacks statements, explicit sequencing and side effects, treats functions as first-class citizens and is lexically scoped. However, ISWIM lacks some of the characteristics of modern functional languages. From the point of view of today's most important representatives of modern functional languages, it lacks:

iv) strong (static) typing (which means that the set of possible types is built-in and that the types of all ISWIM objects are determined at run-time)
v) non-strict semantics (i.e. all operands in ISWIM expressions are evaluated prior to their applications)
vi) "curried" functions by default (i.e. ISWIM functions must be explicitly defined as "curried")
vii) syntactic sugaring like list comprehension (ZF expressions), pattern recognition and guards (which means that every ISWIM expression must be defined explicitly, in terms of built-in operators).

Most of the characteristics mentioned are of a syntactic nature and of no importance to the essence of the functional programming paradigm (except maybe non-strict semantics). If some of these characteristics (iv - vii) were introduced to the language, that will most certainly turn ISWIM into another language from a large family of functional programming languages. Since ISWIM was never standardized or even strongly defined, all functional languages satisfying the conditions i-iii and lacking the (syntactic) features iv - vii can be called ISWIM-like languages.

3 AN IMPLEMENTED ISWIM-LIKE LANGUAGE

An implemented ISWIM-like language has the following six data types: integer, real, logical, string, sequence and function. Objects belonging to all mentioned types are equal in their rights: they can all be an argument or a result of a function, an element of a sequence, a value of an expression etc. The data types integer, real, logical, string, sequence and function will be designated respectively as Z, R, B, C, S and F, while the set of all available types will be designated as τ. In the following sections, data values and a set of primitive operations for every type will be defined.

3.1 A Definition of Primitive Operations

Data values for every type will be simply enumerated or their form will be described. The syntax of every primitive operation will be given in the form of mapping from domain type(s) to the resulting type. Semantics will

be defined in the form of equations which hold for primitive operations. The sign ≡ will be used in following equations to avoid possible confusion with symbols of UNS-INSWIM language.

Every violation of syntax rules defined here as well as all undefined cases in semantic equations will produce run-time errors or unpredictable results.

Logical

B ∈ τ; **true, false** ∈ B

~ : B → B
&, |, ~=, = : B × B → B
... -> ... ; ... : B × T × T → T, T ∈ τ

For all $x,y \in$ B, the following equations hold:

~ :	~ true ≡ false	= :	true = true
	~ false ≡ true		false = false
& :	x & $y \equiv y$ & x	~= :	$x \sim= y \equiv \sim(x = y)$
	false & $x \equiv$ false		
	true & $x \equiv x$	...->...;... :	true -> x ; $y \equiv x$
			false -> x ; $y \equiv y$
| :	$x \mid y \equiv \sim(\sim x$ & $\sim y)$		

Integer and Real

Z, R ∈ τ; ..., -2, -1, 0, 1, 2, ... ∈ Z
all numbers of the form (given in EBNF notation) d{d}.d{d}['E'[-]d{d}] belong to R where d is a digit.

- : (Z ∪ R) → (Z ∪ R)
+, -, *, / : (Z ∪ R) × (Z ∪ R) → (Z ∪ R)
div, mod: (Z ∪ R) × (Z ∪ R) → Z
~=, =, >, <, >=, <= : (Z ∪ R) × (Z ∪ R) → B

The semantics of primitive operations for types Z and R will not be defined here. It is exactly the same as in functional programming language Scheme. The section 4 of the paper, where transformation rules for the translation of UNS-ISWIM into Scheme are given, can be considered as a definition of semantics for these operations.

String

C ∈ τ; "$x_1x_2...x_n$" ∈ C, where x_i, i=1,..,n are characters.

~=, =, >, <, >=, <= : C × C → B

For all $x,y \in$ C, the following equations hold:

= :	"$x_1x_2...x_n$" = "$y_1y_2...y_m$", if n=m and $x_i=y_i$, i=1,...,n
< :	"$x_1x_2...x_n$" < "$y_1y_2...y_m$", if ($\exists k\ 1 \le k \le \min(n,m)$, $x_i=y_i$, i=1,...,k-1 & $x_k<y_k$) or (n<m, $x_i=y_i$, i=1,...,n)

$\sim=$:	$x \sim= y \equiv \sim(x = y)$	$>=$:	$x >= y \equiv \sim(x < y)$
$>$:	$x > y \equiv y < x$	$<=$:	$x <= y \equiv \sim(x > y)$

Sequence

$S \in \tau$; **nil** $\in S$, $x{:}y \in S$, where $y \in S$, $x \in T \in \tau$

: : $T \times S \rightarrow S$, $T \in \tau$
hd : $S \rightarrow T$, $T \in \tau$
tl : $S \rightarrow S$
$\sim=$, $=$: $S \times S \rightarrow B$
: $S \rightarrow Z$
++ : $S \times S \rightarrow S$

! : $S \times Z \rightarrow T$, $T \in \tau$
@ : $S \times Z \rightarrow S$
in : $T \times S \rightarrow B$, $T \in \tau$
atom : $T \rightarrow B$, $T \in \tau$
[...] : $T_1 \times T_2 \times \ldots \times T_N \rightarrow S$, $T_1,T_2,\ldots,T_N \in \tau$

For all $x,x_1,x_2,\ldots,x_n,y \in T \in \tau$, $s,t \in S$, the following equations hold:

hd : **hd** $(x{:}s) \equiv x$

tl : **tl** $(x{:}s) \equiv s$

= : **nil** = **nil**
$(x{:}s) = (y{:}t) \equiv (x=y)$ & $(s=t)$

$\sim=$: $s \sim= t \equiv \sim(s = t)$

: **#** **nil** $\equiv 0$
$(x{:}s) = 1 +$ **#**s

++ : **nil** **++** $s \equiv s$
$(x{:}s)$**++**$t \equiv x{:}(s$**++**$t)$

! : $(x{:}s)!1 \equiv x$
$(x{:}s)!n \equiv (s!(n-1))$, if $n>1$

@ : $(x{:}s)@1 \equiv s$
$(x{:}s)@n \equiv (s@(n-1))$, if $n>1$

in : x **in nil** $\equiv$ **false**
x **in** $(x{:}s) \equiv$ **true**
x **in** $(y{:}s) \equiv (x$ **in** $s)$, if $x \sim= y$

atom : **atom** $(x{:}s) \equiv$ **false**
atom $x \equiv$ **true**, otherwise

[...] : $[x_1, x_2, \ldots, x_n] = x_1{:}(x_2{:}(\ldots{:}(x_n{:}\text{nil}))\ldots)$

Function

$F \in \tau$; all expressions of the form (**lambda** $(x_1, x_2, \ldots, x_n)$ *exp*) belongs to type F.

$f(\ldots) : T_1 \times T_2 \times \ldots \times T_N \rightarrow T$, $T,T_1,T_2,\ldots,T_N \in \tau$, $f \in F$

For all $x,x_1,x_2,\ldots,x_n \in T \in \tau$, $f \in F$, the following equations hold:

$f(\ldots) : f(x_1, x_2, \ldots, x_n) \equiv x$, where x is computed by the expression captured by *exp* and formal arguments $x_1,\ldots,x_n$. Note that f is implicitly or explicitly of the form (**lambda** $(x_1, x_2, \ldots, x_n)$ *exp*).

3.2 Operator Precedence

With respect to precedence, expressions of the language are grouped into following four groups, starting with those having the highest priority:

~, -, hd, tl, atom, #, f(x,y,...,z)
+, -, |, ++
*, /, div, mod, &, !, @, :
>, <, >=, <=, ~=, =, in, .->.;.

The order of the association for binary operators is from left to right, except in the case of : and ++, where the order of association is from right to left.

3.3 Grammar

The implemented UNS-ISWIM language has the following syntax (described by EBNF notation):

```
Block           = '{' ( WhereBlock | LetBlock ) '}'.
WhereBlock      = Expr WhereRec def { 'AND' def }
LetBlock        = LetRec def { 'AND' def } ';' Expr
WhereRec        = 'WHERE' ['REC']
LetRec          = 'LET' ['REC']
def             = id {idList} '=' Expr
id              = letter {letter | digit}
letter          = 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
digit           = '0' | '1' | ... | '9'
idList          = '(' [ id { ',' id } ] ')'
Expr            = Express | CondExpr
Express         = SimExpr [RelOp SimExpr]
CondExpr        = Expr '->' Expr ';' Expr
RelOp           = '=' | '<' | '<=' | '>' | '>=' | '~=' | 'IN'
SimExpr         = Term {AddOp Term}.
AddOp           = '+' | '-' | '|' | '++'
Term            = [ '-' ] Fact {MulOp Fact}
MulOp           = '*' | '/' | 'DIV' | 'MOD' | '&' | '!' | '@' | ':'
Fact            = FunApplication {ExprList} |
                  '[' Sequence ']' |
                   UnaryOp Fact |
                   Constant.
FunApplication  = '(' Expr ')' |
                  AnonFun |
                  Block |
                  id.
ExprList        = '(' [Expr { ',' Expr } ] ')'
UnaryOp         = '~' | 'HD' | 'TL' | '#' | 'ATOM'
Constant        = number | 'NIL' | '"' Id '"' | 'TRUE' | 'FALSE'
Sequence        = Expr [, Sequence ]
number          = digit {digit} ['.' digit {digit}] ['E'[-] digit {digit}]
AnonFun         = '(' 'LAMBDA' IdList Expr ')'
```

3.4 Differences

There are some differences between the language defined in this paper and the one defined in [3]. They all preserve the original concept of the language, no matter whether they are extensions or reductions of the original idea.

- In the original language let (or where) definitions were written without { and } delimiters, but with obligatory new lines and indentations. The introduction of delimiters in UNS-ISWIM allows for more relaxed program form.

- The data type Sequence was not explicitly mentioned in [3]. However, UNS-ISWIM includes it and supports it with primitive operations found in other functional languages (SASL and Miranda, for example).

- *Program points* as "ISWIM's nearest thing to jumping" [3] are not in UNS-ISWIM since the later development of functional languages proved that they are not necessary.

- Original ISWIM consisted of *demands* (i.e. commands to language interpreter) and *definitions*. UNS-ISWIM as defined here consists only of definitions, while the demands are left to be defined in the implementation phase.

- *Listings*, as a way of writing multiple definitions, are omitted from UNS-ISWIM. Instead of *listings*, an explicit multiple definition has to be written, that is $(x_1, x_2, ..., x_n) = (X_1, X_2, ..., X_n)$ must be written as: $x_1 = X_1$ and $x_2 = X_2$ and ... and $x_n = X_n$.

- A keyword **rec** standing before every recursive function definition in original ISWIM appears only once in UNS-ISWIM before all function definitions if at least one definition is recursive. This solution is adopted in many other functional languages as well.

- A notion and a definition of an anonymous function is introduced in UNS-ISWIM. Anonymous functions have several useful applications in a functional language, but all of them lead to more explicit functional programs and, in many cases, avoid the need for unnecessary function naming. For example, in addition to the following two ways of defining function returning function in ISWIM:

```
a)    f(n) = {g where g(m)=m+n}
b)    f(n)(m) = m+n
```

there is a new one in UNS-ISWIM:

```
c)    f(n) = (lambda(m) m+n)
```

which is more precise than the definition a) and avoids the need for explicit naming of a returned function (like in b).

3.5 Examples

In this section two expressions in UNS-ISWIM will be given as an illustration of the language features. The first expression will return a reversed sequence and all possible sub-sequences. The definition of function **revAll** uses a local definition to replace the head and tail of sequence **e** with the identifiers **head** and **tail** respectively.

```
{ revAll [1,2,3,[1,2,3,[1,2,3]]]
    where rec
       revAll(e) = e=nil -> nil;
                   { let head = hd e and
                         tail = tl e;
                     atom head  -> revAll(tail) ++ [head];
                     revAll(tail) ++ [revAll(head)]
                   }
}
```

The second expression returns the maximum of the sequence. Function **max** is defined via higher-order function **fold**, which, as its first argument, takes an anonymous function determining the maximum of two elements.

```
{ max [1,2,3,4,5,6,7,8,9,10]
    where rec
```

```
        max(e) = fold( (lambda(a,b)a<=b->b;a), hd e, tl e)
    and fold(f,init,e) = e=nil->init;
                         f(hd e, fold(f,init,tl e))
}
```

4 TRANSLATION TO OTHER FUNCTIONAL LANGUAGE

The proposed functional language can be implemented by the compilation to a machine language of some real or virtual machine or by the translation to some intermediate language. In the latter approach it is often useful to choose some existing functional language so that the translation is uncomplicated and fundamental features of (UNS-)ISWIM are preserved.

A subset of functional language Scheme [6] is among the most appropriate target languages for the translation. Scheme is a language with strict semantics, lexically scoped, not strongly typed, treats functions as first class objects, has anonymous functions, let and letrec blocks, and its function definitions are not curried by default. Although Scheme possesses some other features (including many imperative ones), they are not needed in the translation of ISWIM.

In this section a set of transformation rules for the translation of UNS-ISWIM language into Scheme is given. It can be regarded as a one more description of a new language, in terms of a known language. On the other hand, the set of transformation rules can be regarded as a simple recipe for implementation of a new language by translating it into an already implemented language.

The set of transformation rules will be organized hierarchically in accordance with grammar given in 3.3. The basic constructions of the UNS-ISWIM language are listed on the lefthand sides of every rule, while the corresponding Scheme constructions are listed on the righthand sides. In rules, constant constructions of both languages will be typed in **boldface**, while variable parts of constructions will be typed in *italics*.

The function T is defined by the set of transformation rules, thus mapping an UNS-ISWIM program into its Scheme equivalent. T is defined in the following way:

Program and Blocks

T[{ *e* where *def*$_1$ and *def*$_2$ ... and *def*$_n$ }] -> **(let (T[*def*$_1$] T[*def*$_2$] ... T[*def*$_n$]) T[*e*])**

T[{ *e* where rec *def*$_1$ and *def*$_2$ ... and *def*$_n$ }] -> **(letrec (T[*def*$_1$] T[*def*$_2$] ... T[*def*$_n$]) T[*e*])**

T[{ let *def*$_1$ and *def*$_2$ ... and *def*$_n$; *e* }] -> **(let (T[*def*$_1$] T[*def*$_2$] ... T[*def*$_n$]) T[*e*])**

T[{ let rec *def*$_1$ and *def*$_2$ ... and *def*$_n$; *e* }] -> **(letrec (T[*def*$_1$] T[*def*$_2$] ... T[*def*$_n$]) T[*e*])**

Definitions

T[*f* (x_1,...,x_n)(y_1,...,y_m)...(z_1,...,z_k) = *e*] -> **(f (lambda (T[x_1] ... T[x_n])**
(lambda (T[y_1] ... T[y_m]) ...
(lambda (T[z_1] ... T[z_k]) T[*e*])
...
)
)
)

T[*f* = *e*] -> **(T[*f*] T[*e*])**

Expression

T[*x* < *y*] -> **(if (string? *x*)**
(string<? T[*x*] T[*y*])

(< T[x] T[y])
)

T[x = nil] -> (null? T[x])
T[nil = x] -> (null? T[x])
T[x = y] -> (equal? T[x] T[y])
T[x ~= y] -> (not (equal? T[x] T[y]))

T[x > y] -> T[y < x]
T[x <= y] -> (not T[y < x])
T[x >= y] -> (not T[x < y])
T[x in y] -> (member T[x] T[y])

Conditional Expression

T[x -> y ; z] -> (if T[x] T[y] T[z])

Simple Expression

T[x + y] -> (+ T[x] T[y])
T[x - y] -> (- T[x] T[y])

T[x | y] -> (or T[x] T[y])
T[x ++ y] -> (append T[x] T[y])

Term

T[- x] -> - T[x]
T[x * y] -> (* T[x] T[y])
T[x / y] -> (/ T[x] T[y])
T[x div y] -> (quotient T[x] T[y])
T[x @ y] -> (list-tail T[x] T[y])

T[x mod y] -> (remainder T[x] T[y])
T[x & y] -> (and T[x] T[y])
T[x ! y] -> (list-ref T[x] (- T[y] 1))
T[x:y] -> (cons T[x] T[y])

Factor

T[(e)] -> (T[e])
T[f(x_1,...,x_n)(y_1,...,y_m)...(z_1,...,x_k)] -> (...(((f T[x_1] ... T[x_n]) T[y_1] ... T[y_m]) ...) T[z_1] ... T[z_k])
T[[x_1, x_2, ..., x_n]] -> (list T[x_1] T[x_2] ... T[x_n])

Unary operator + Factor

T[~ x] -> (not T[x])
T[hd x] -> (car T[x])
T[tl x] -> (cdr T[x])
T[#x] -> (length T[x])
T[atom x] -> (not (list? T[x]))

Constant

T[nil] -> '()
T[true] -> #t
T[false] -> #f

Anonymous Function

T[(lambda (x_1, x_2, ..., x_n) e)] -> (lambda (T[x_1] T[x_2] ... T[x_n]) T[e])

All other cases

T[x] -> x

The above definition of function T must be supported by several comments:

- In Scheme all expression different from **#f** are treated in logical context as logical truth value **true**. Since UNS-ISWIM is not strongly typed language (and type checking is not performed at compile-time), this feature of Scheme is not considered as a problematic one.

- Some Scheme operations used in transformation rules are not marked as "essential" to Scheme implementations in [6]. If they are not available in concrete Scheme implementation, they should be replaced with closest equivalents. For example, instead of **list?**, similar **pair?** should be used and instead of **list-tail**, a composition of **cdr**, **cddr**, **cdddr** and **cddddr**'s should be used.

- In most implementations of Scheme, expressions, in order to be evaluated, must be embedded into a top level definition. The top level definition consists of keyword **define** and is followed by the identifier to be bound to an expression. In such cases, an expression, which is the result of the translation function T, must be embedded into an appropriate form as well.

- If the transformation rules are applied in the order they are given in this section, a Scheme program conforming to UNS-ISWIM semantics will be produced. Exceptions to this are only the rules for **++** and **:**, whose associativity is from right to left. In these two cases, a special care must be taken.

5 AN EXAMPLE OF THE TRANSLATION

The UNS-ISWIM function **max** from section 3.6 would be translated into a corresponding Scheme program in the following way (only a few important steps are displayed):

T⟦*program*⟧ -> **(letrec (T⟦*def*$_1$⟧ T⟦*def*$_2$⟧) (max (list 1 2 3 4 5 6 7 8 9 10))**

T⟦*def*$_1$⟧ -> **(max (lambda (e)** T⟦e_1⟧**))**
T⟦e_1⟧ -> **(fold** T⟦*anon*⟧ **(car e) (cdr e))**
T⟦*anon*⟧ -> **(lambda (a b) (if (not (> b a)) b a))**

T⟦*def*$_2$⟧ -> **(fold (lambda (f init e)** T⟦e_2⟧ **))**
T⟦e_2⟧ -> **(if (null? e)**
init
(f (car e) (fold f init (cdr e)))
)

which finally produces the following Scheme definition of the function **max**:

```
( letrec
   ( (max
        (lambda (e)
            (fold (lambda (a b) (if (not (> b a)) b a))
                  (car e)
                  (cdr e)
      ) )   )
      (fold
         (lambda (f init e)
            (if (null? e)
                init
                (f (car e)
                   (fold f init (cdr e))
   ) ) ) )   )
   (max (list 1 2 3 4 5 6 7 8 9 10))
)
```

Note that in this example an UNS-ISWIM operation <= is translated into (not (> ...)) instead of a much longer expression from transformation rules. This is only to keep the example as concise as possible.

6 CONCLUSION

In the paper a full definition of a purely functional language UNS-ISWIM is given. The defined language is an extension of programming language ISWIM. It has syntax resembling mathematical notation and short and clear semantics. With the given set of transformation rules for the translation of UNS-ISWIM expressions into an equivalent Scheme expression, a precise recipe for the implementation of UNS-ISWIM is available. When translated into Scheme, an expression can be evaluated by some of Scheme processors or further compiled into machine language of a real or virtual machine.

With a full definition of UNS-ISWIM and a proposed precise way of implementation, a purely functional language becomes available on a large scale of hardware platforms, because of wide availability of Scheme language processors. Since the implementation of transformation rules is straightforward and automatically obtainable with the help of various software engineering tools, the defined language should be practically available in a very short time. This broad availability and conformity to the standards of purely functional languages are the main advantages of the proposed definition and its implementation.

References

1. Z. BUDIMAC, M. IVANOVIĆ, Z. PUTNIK and D. TOŠIĆ: *LISP by Examples*. Institute of Mathematics, Novi Sad, 1991. (in Serbian).
2. P. HUDAK: *Conception, Evolution, and Application of Functional Programming Languages*. ACM Comp. Surveys 3(1989), 359-411.
3. P. J. LANDIN: *The Next 700 Programming Languages*. Comm. of ACM 3(1966), 137 - 164.
4. P. J. LANDIN: *The Mechanical Evaluation of Expressions*. Comp. Journal 6(1964), 308 - 320.
5. B. J. MACLENNAN: *Functional Programming - Practice and Theory*. Addison Wesley, New York, 1989.
6. W. CLINGER and J. REES: *The Revised⁴ Report on the Algorithmic Language Scheme*. ACM LISP Pointers 3(1991), 1-55.