Fall 2007:
Building an Interpreter for an Imperative Language

Professor Maida
CMPS 450.2
First Draft

September 25, 2007

1 Introduction

This is a first draft of the project specifications. Updates will be posted on the class web page accessible from:

http://www.cacs.louisiana.edu/~maida

This project uses the JAVA programming language to build a symbol-table-based interpreter for a small imperative programming language. Features of the language include global scalar variables and functions. The functions allow local scalar variable declarations. In this language, primary statements are arithmetic assignment statements. The language also has if-else statements.

The interpreter does lexical and syntactic analysis and maintains a run-time environment that consists of a stack-oriented symbol table. The implemented language supports recursion and obeys rules of dynamic scoping. Functionally the interpreter provides a read-eval-print loop for the user. The interpreter reads a statement, evaluates it and updates the runtime environment, and prints the value of the evaluated expression. After that, the interpreter reads the next statement and executes another cycle of the loop. Statements can appear in any order with the only constraint being that all references must be defined before use, or evaluation.

2 Sample Program

A program consists of a sequence of top-level statements that are evaluated in the order that they are given. Allowable top-level statement types are: global declarations, function definitions, assignment statements, and expressions (including function calls). Global declarations and function definitions must be top-level statements.
In the program below, a result is printed after each statement is evaluated, although this is not shown.

```plaintext
global i := 5; // declares and initializes global var "i" to 5
global ret;   // declares and initializes global var "ret" to 0

func fact(n) { // recursive factorial
    if (n == 0) 1;
    else n * fact(n - 1);
}

func ifact(n) { // iterative factorial
    local ans := 1;
    while (n != 0) {
        ans := ans * n;
        n := n - 1;
    }
    ret := ans;
    ans;
}

i;       // print the value of i
fact(i); // invoke recursive factorial
ifact(i); // invoke iterative factorial
ret;     // print new value of "ret"
```

3 The Language

3.1 Tokens

The tokens in the language must be identified by a scanner. The scanner obeys the normal rules: 1) obeys principle of longest substring; 2) treats white space as a delimiter; and, 3) skips over extra whitespace.

Tokens fall into different classes.

3.1.1 Reserved Words

The words below are reserved words built into the programming language. They cannot be used as variable or function names.

```
global, local, func, if, else, while
```

3.1.2 Operators

The operators of the language are listed below from highest to lowest precedence. All operators are left associative, except for assignment (:=) which is right associative. In regard to precedence, the lower the number, the higher the precedence.
### Operator Precedence Table

<table>
<thead>
<tr>
<th>Description</th>
<th>Operator(s)</th>
<th>Precedence</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary minus</td>
<td>&quot;-&quot;</td>
<td>1</td>
<td>not applicable</td>
</tr>
<tr>
<td>Multiplication and division</td>
<td>&quot;*&quot;, &quot;/&quot;</td>
<td>2</td>
<td>left</td>
</tr>
<tr>
<td>Addition and subtraction</td>
<td>&quot;+&quot;, &quot;-&quot;</td>
<td>3</td>
<td>left</td>
</tr>
<tr>
<td>Integer equality and inequality</td>
<td>&quot;==&quot;, &quot;!=&quot;</td>
<td>4</td>
<td>left</td>
</tr>
<tr>
<td>Assignment</td>
<td>&quot;:=&quot;</td>
<td>5</td>
<td>right</td>
</tr>
</tbody>
</table>

#### 3.1.3 Other Tokens

- Left and right curly braces: "{", "}"
- Left and right parentheses: "("", ")"
- Semicolon: ";"
- Comma: ","
- Identifier: IDENT
- Unsigned integer: SCALAR

An IDENT begins with a letter and can be followed by zero or more letters, digits, or underscores. A SCALAR is one or more digits, that may optionally be preceded by a minus sign.

#### 3.2 The Grammar

The syntax of the language is specified using the EBNF given below.

```
1: <top_stmt> ::= <global_decl> | <func_def> | <assign_stmt> | <exp> ";
2: <global_decl> ::= "global" IDENT { ":=" <exp> } ";"
3: <local_decl> ::= "local" IDENT { ":=" <exp> } ";"
4: <func_def> ::= "func" IDENT "(" { formal_params } ")" <block>
5: <formal_params> ::= IDENT {"," IDENT}
6: <block> ::= <stmt> | "{" { <stmts> } "}" |
7: <stmts> ::= <stmt> { <stmt> }
8: <stmt> ::= <assign_stmt> | <f_call> | <if_else_stmt> | <local_decl>
9: <if_else_stmt> ::= "if" "(" <test> ")" <block> { "else" <block> }
10: <while_stmt> ::= "while" "(" <test> ")" <block>
11: <assign_stmt> ::= IDENT "=" ( <exp> | <assign_stmt> ) ";"
12: <f_call> ::= IDENT "(" <actual_params> ")"|
13: <actual_params> ::= <exp> {, <exp> }
14: <test> ::= <exp> ( "==" | ";!=" ) <exp>
15: <exp> ::= <term1> { ( "+" | "-" ) <term1> }
16: <term1> ::= <term2> { ( "+" | ";-" ) <term2> }
17: <term2> ::= { "-" } <factor>
18: <factor> ::= "(" <exp> ")" | SCALAR | IDENT | <f_call>
```

Line 18 requires that the scanner use two-step look ahead because both IDENT and <f_call> begin with an identifier. There are two ways to resolve this ambiguity. The first is to look at the next symbol in the grammar. If it is a left parentheses, then the identifier was a function name. The second way to resolve this
ambiguity is to use the symbol table. Since an identifier must be defined before it is used, then the name will exist in the symbol table and the symbol table knows whether the name represents a function or a variable.

3.2.1 Abstract syntax trees

Figures 1 and 2 show a partial list of abstract syntax trees (ASTs) for various non-terminals in the grammar. The trees represent the subset of syntax that is semantically meaningful.

Note that function calls or invocations are divided into two forms. First, there are built-in primitive functions, such as “+” and “*” that appear in the grammar. These correspond to rules 14, 15, 16, and 17 of the grammar. The AST for a primitive function has its name in the node. Second, there are the user-defined functions as in “square(x)”. These correspond to rule 12 of the grammar. For these functions, the AST has f-call in the node.

3.3 Semantics

The runtime state of the program is stored in a stack-based symbol table. This table stores:

1. local and global variable names and values;
2. function definitions;
3. runtime stack updates implemented via the symbol table (dynamic scoping).

Names (identifiers) can refer either to a variable or a function. However, one cannot use a name both for a variable and a function. All variables are integer valued.

A list of the different types of statements in the language is given below. The semantic effect of each statement type is explained. This involves informally saying what the expression does and how the execution of the statement changes the state of the program in regard to its representation on the symbol table. The boldface names correspond to nonterminals in the grammar.

global_decl A global declaration declares a global variable and gives it a value. If the value is not explicit, then the default is the integer 0. The semantic effect is to associate the variable name with the value and add the association (binding) into the symbol table. If a global declaration for a variable is entered more than once, the later declarations act as if they were assignment statements, updating the value of the initial declaration. The returned value of this statement is the new value of the variable.

func_def A function definition defines a function that has a return value. The effect of evaluating a function definition is to place an association between a name and the definition into the symbol table. This involves entering the name, the list of formal parameters, the local variables, and the body of the function. The body of the function is represented using an abstract syntax tree. If the function is defined successfully, a value of one is returned. If the name is already associated with a function
6. stmt or stmts

7. stmt or sequence
   stmts stmt

9. if
   exp block block

11. 
   IDENT exp IDENT :=
   or

12. f-call
   IDENT args

Figure 1: Abstract syntax trees (ASTs) corresponding to the numbered rules in the grammar.

definition then the new definition replaces the old definition. If the name is already declared as a variable, then no action is taken and the value of zero is returned.

local_decl A local declaration can only appear within a function definition. It cannot be entered in the read-eval-print loop as a top-level statement. When the function definition is evaluated, its declaration will be entered into the local variables for the definition that it appears in. At runtime, when the function is executed, these variable declarations will be pushed onto the symbol table. Local variable declarations can shadow global variable declarations according to the rules of dynamic scoping.

exp An expression is evaluated to obtain a value. It is represented as an abstract syntax tree that encodes operator precedence and associativity. If identifiers are encountered in the evaluation of an expression their values are obtained by accessing the symbol table. If a value is not found, then a runtime error is issued. If there is no error, the returned value is the value of the expression.

assign_stmt The right hand side of an assignment statement is evaluated in the current environment to obtain a value. The value of the variable on the left hand side of the statement is then set to this value. The returned value after executing this statement is the new value of the variable.

f_call When a function is called, or invoked, the following steps take place. The function arguments are evaluated (Parameter passing is call by value). The formal parameters must be internally renamed so that their declarations do not conflict with other definitions in the source code. These evaluated arguments are then paired-up with the formal parameters that were declared in the function definition and pushed onto the symbol table. After that the body of the function is evaluated in the context of the current environment. When a function finishes execution, it pops the symbol table. A function returns the last value of the last expression evaluated.

if_else_stmt To execute an if-else statement, the test must be evaluated in the current environment. If the test evaluates to any value that is not zero, then the first block of the statement is executed. Otherwise, the second block is executed. The returned value is the value returned by the block that was executed.
Figure 2: ASTs corresponding to numbered rules in the grammar (part 2).

4 Examples

4.1 Evaluating an Assignment Statement

Consider the global assignment statement below.

global x := 5;

Evaluating this statement must create a symbol table entry consisting of three pieces of information:

1. Type: global_variable
2. Name: x
4.2 Evaluating a Function Definition

An example of a function definition appears below.

```plaintext
func square (n) { n * n;}
```

Evaluation of the definition must create a binding, or association, from the name “square” to a symbol table entry containing four pieces of information.

1. Type: function_definition
2. Name: square
3. Formal parameters: ( n )
4. Body: an abstract syntax tree representation of n * n

4.3 Evaluating a Function Invocation

Consider the requirements to process the simplest possible function invocation. An example appears below.

```plaintext
1: global m := 5;
2: global x;
3: func square( n ) { n * n;}
   ...
4: x := square( m );
```

In the above, Line 3 defines a function that accepts one argument and returns the value of an expression. We have already explained the effect of evaluating Line 3. Now we want to explain the effect of evaluating a function that has been previously defined. Line 4 invokes, or calls, the function “square” on the particular value of “m” (which is 5) and assigns the result to the variable “x”.

The following are the steps in applying a function call to a user-defined function. The function must have been previously defined and stored in the symbol table.

Let’s start with the part “square( m )”. According to the grammar, this expression is a function call and the parser will supply an abstract syntax tree representing this expression. The AST for this function call is given in Box 13 of Figure 1. In this case, the value of IDENT is “square” and the value of args is “m”.

3. Value: 5
1: int eval_f_call(AST ast) {
2:   f_name = get_f_name(ast);
3:   f_args = get_args(ast);
4:   s_tab_entry = get_entry(f_name);
5:   body = get_body(s_tab_entry);
6:   f_params = get_params(s_tab_entry);
7:   evaled_args = evaluate_list(f_args);
8:   local_env = match_up(f_params, evaled_args);
9:   push_symtab(local_env);
10:  value = evaluate(body);
11:  pop_symtab();
12:  return value;
}

Table 1: Pseudo code to evaluate a user-defined function call.

1. Retrieve the definition for “square” from the symbol table. This includes the formal parameter list and the body of the function which is represented by an abstract syntax tree (AST). In this example, the formal parameter list is simply the list containing the single variable name “n”. The body is an AST that represents the expression “n*n”.

2. Create a new stack frame and push it onto the symbol table.

3. Evaluate the actual parameters in the function invocation and save them in a list. In this example, the actual parameter “m” evaluates to 5. So the list has one element, which is 5.

4. Match the values of the actual parameters to the formal parameters, rename the formal parameters and the variables in the body, and add the resulting bindings to the current stack frame. In this example, there is only one binding. Specifically, n equals 5.

5. Evaluate the body in the resulting environment. To evaluate the body for this example, compute the value of n*n and remember it when the stack frame is popped.

6. Return the result. In this case, the stack frame is popped and the global variable “x” is set to 25.

In the following, evaluation always takes place in the context of the environment defined by the global symbol table.

Pseudo-code to implement this algorithm given in the function eval_f_call shown Table 1. The function accepts an abstract syntax tree as input. In Lines 2 and 3, it extracts the name and arguments of the function. In Line 4, it uses the name to retrieve the symbol table entry. In Lines 5 and 6, it extracts the body and formal parameters from the symbol table entry. In Line 7, it evaluates the arguments to the function, using a call to a function evaluate_list. This function evaluates each element of a list and returns the list of the resulting values. Line 8 matches the formal parameters to the evaluated arguments. Line 9 creates a new abstract syntax tree that replaces the formal parameters in the body of the function with the values in the local environment. Line 10 evaluates the new abstract syntax tree and returns the result.

The evaluate function in Line 10 is capable of evaluating every type of node that can occur in an abstract syntax tree. This function is a dispatch function that does not do any of its own work. It looks at the node type of the abstract syntax tree, and then calls a function that is designed to evaluate that node type. Part of the pseudo code for this function is shown in Table 2. In the present example, evaluate will call the built-in multiply function (Line 5). The returned result will be 25.
1: int evaluate(AST ast) {
2:   switch (type_of(ast)) {
3:     case f_call : return eval_f_call(ast);
4:     case assign_stmt : return eval_assign_stmt(ast);
5:     case if_else_stmt: return eval_if_else(ast);
6:     case while_stmt : return eval_while(ast);
7:     case built_in_* : return eval_multiply(ast);
8:     ...  
9:   }

Table 2: Pseudo code for an evaluation dispatcher.

1: int eval_assign_stmt(AST ast) {
2:   v_name = get_v_name(ast);
3:   exp = get_rhs_exp(ast);
4:   value = evaluate(exp);
5:   put_var_into_symtab(v_name, value);
6:   return value;
}

Table 3: Pseudo code to evaluate an assignment statement.

We can now return to the issue of evaluating the assignment statement in its entirety. Let us look use the pseudo code given in Table 3 to do this. The AST for this function call is given in Box 11 of Figure 1. Line 2 obtains the variable name from the abstract syntax tree and Line 3 obtains the expression to be evaluated. The evaluation process takes place in Line 4 and have just finished explaining this is done (the returned value was 25). Line 5 associates the value with the variable name and puts it into the symbol table. Line 6 returns the value of the assignment statement.

4.4 A Recursive Function

Consider the recursive “fact” function shown below. It requires that we define the evaluation process for an if-else statement. After this is done, the recursive factorial function should work automatically.

```c
func fact(n) {
   if (n == 0) 1;
   else n * fact( n - 1 );
}
```

The pseudo code for evaluating the if-else statement is given in Table 4. This function accepts an abstract syntax tree as input. The AST for this function call is given in Box 10 of Figure 1. In Line 3, the function evaluates the test expression. If the value is zero and there is an else branch, it evaluates the else branch. Otherwise, it evaluates the then branch.

4.5 An Iterative Function

Consider the iterative “ifact” function shown below. It requires that we define the evaluation process for an while statement and also for the local variable declaration. After this is done, the iterative factorial
Table 4: Pseudo code to implement an if-else statement.

```c
int eval_if_else (AST ast) {
    test = get_if_test(ast);
    choice = evaluate(test);
    if (choice == 0)
        if (has_else_branch(ast))
            return evaluate(get_else_branch(ast));
        else return 0;
    else
        return evaluate(get_then_branch(ast));
}
```

Table 5: Pseudo code to implement an if-else statement.

```c
int eval_while (AST ast) {
    test = get_while_test(ast);
    block = get_while_block(ast);
    while (evaluate(test) != 0) {
        evaluate(block);
    }
    return 1;
}
```

The pseudo code for evaluating the if-else statement is given in Table 5.