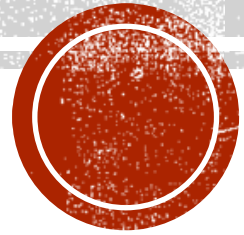
Department of
Computer Engineering
Faculty of Engineering
University of Sri Jayewardenepura

Interpretation



CO4205 - Compilers

Akarshani Amarasinghe

Use of an Interpretation

- After lexing and parsing, we have the abstract syntax tree of a program as a data structure in memory.
- But a program needs to be executed.
- The simplest way to execute a program is **interpretation**.

How Do We Evaluate a Mathematical Expression?

- We insert the values of variables in the expression and evaluate it bit by bit.
  - We start with the innermost parentheses and move out until we have the result of the expression.
- Then we repeat the process with other values for the variables.

Interpretation

- Interpretation is done by a program called an **interpreter**.
  - It takes the abstract syntax tree of a program and executes it by inspecting the syntax tree to see what needs to be done.
  - This is similar to how a human evaluates a **mathematical expression**.

Human vs. Interpreter

- A human being will copy the text of the formula with variables replaced by values and then write a sequence of more and more reduced copies of the formula until it is reduced to a single value.
- An interpreter will keep the formula (/the abstract syntax tree) unchanged and use a symbol table to keep track of the values of variables.
  - Instead of reducing a formula, the interpreter is a function that takes an abstract syntax tree and a symbol table as arguments and returns the value of the expression represented by the abstract syntax tree.
  - The function can call itself recursively on parts of the abstract syntax tree to find the values of subexpressions, and when it evaluates a variable, it can look its value up in the symbol table.

Role of an Interpreter

- A function takes the abstract syntax tree of the program and some extra information about the context (Eg: a symbol table or the input to the program).
- It returns the output of the program.
- Some input and output may be done as side effects by the interpreter.
- **Assumption:** The symbol tables are persistent.
  - No explicit action is required to restore the symbol table for the outer scope when exiting an inner scope.

The Structure of an Interpreter

- An interpreter will typically consist of one function per syntactic category.
- Each function will take as arguments an abstract syntax tree from the syntactic category and extra arguments such as symbol tables.
- Each function will return one or more results, which may be the value of an expression or an updated symbol table.
- The functions can be implemented in any language that we already have an implementation of.
- Eventually, we will need to either have an interpreter written in machine language or a compiler that compiles to machine language.

Department of
Computer Engineering
Faculty of Engineering
University of Sri Jayewardenepura

8

An Example for an Interpreter

Akarshani Amarasinghe

Example Language

Program → *Funs*

Funs → *Fun*
Funs → *Fun Funs*

Fun → *TypeId* (*TypeIds*) = *Exp*

TypeId → `int` **id**
TypeId → `bool` **id**

TypeIds → *TypeId*
TypeIds → *TypeId* , *TypeIds*

Exp → **num**
Exp → **id**
Exp → *Exp* + *Exp*
Exp → *Exp* = *Exp*
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → **id** (*Exps*)
Exp → `let` **id** = *Exp* `in` *Exp*

Exps → *Exp*
Exps → *Exp* , *Exps*

Evaluating Expressions

Program → *Funs*

Funs → *Fun*
Funs → *Fun Funs*

Fun → *TypeId* (*TypeIds*) = *Exp*

TypeId → `int` **id**
TypeId → `bool` **id**

TypeIds → *TypeId*
TypeIds → *TypeId* , *TypeIds*

Exp → **num**
Exp → **id**
Exp → *Exp* + *Exp*
Exp → *Exp* = *Exp*
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → **id** (*Exps*)
Exp → `let` **id** = *Exp* `in` *Exp*

Exps → *Exp*
Exps → *Exp* , *Exps*

$Eval_{Exp}(Exp, vtable, ftable)$ = `case` *Exp* `of`	
num	$getvalue(\textbf{num})$
id	$v = lookup(vtable, getname(\textbf{id}))$ *if* $v = unbound$ *then* **error**() *else* v
Exp_1 + Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ *if* v_1 *and* v_2 *are integers* *then* $v_1 + v_2$ *else* **error**()
Exp_1 = Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $v_2 = Eval_{Exp}(Exp_2, vtable, ftable)$ *if* v_1 *and* v_2 *are both integers or both booleans* *then if* $v_1 = v_2$ *then* **true** *else* **false** *else* **error**()
`if` Exp_1 `then` Exp_2 `else` Exp_3	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ *if* v_1 *is a boolean* *then if* $v_1 =$ **true** *then* $Eval_{Exp}(Exp_2, vtable, ftable)$ *else* $Eval_{Exp}(Exp_3, vtable, ftable)$ *else* **error**()
id (*Exps*)	$def = lookup(ftable, getname(\textbf{id}))$ *if* def = *unbound* *then* **error**() *else* $args = Eval_{Exps}(Exps, vtable, ftable)$ $Call_{Fun}(def, args, ftable)$
`let` **id** = Exp_1 `in` Exp_2	$v_1 = Eval_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), v_1)$ $Eval_{Exp}(Exp_2, vtable', ftable)$

$Eval_{Exps}(Exps, vtable, ftable)$ = `case` *Exps* `of`	
Exp	$[Eval_{Exp}(Exp, vtable, ftable)]$
Exp , *Exps*	$Eval_{Exp}(Exp, vtable, ftable)$ $:: Eval_{Exps}(Exps, vtable, ftable)$

Evaluating a Function Call

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & \mathit{Funs} \\
 & & \\
\mathit{Funs} & \rightarrow & \mathit{Fun} \\
\mathit{Funs} & \rightarrow & \mathit{Fun}\ \mathit{Funs} \\
 & & \\
\mathit{Fun} & \rightarrow & \mathit{TypeId}\ (\ \mathit{TypeIds}\ )\ =\ \mathit{Exp} \\
 & & \\
\mathit{TypeId} & \rightarrow & \texttt{int}\ \textbf{id} \\
\mathit{TypeId} & \rightarrow & \texttt{bool}\ \textbf{id} \\
 & & \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId} \\
\mathit{TypeIds} & \rightarrow & \mathit{TypeId}\ ,\ \mathit{TypeIds} \\
 & & \\
\mathit{Exp} & \rightarrow & \textbf{num} \\
\mathit{Exp} & \rightarrow & \textbf{id} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \mathit{Exp} = \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp} & \rightarrow & \textbf{id}\ (\ \mathit{Exps}\ ) \\
\mathit{Exp} & \rightarrow & \texttt{let}\ \textbf{id} = \mathit{Exp}\ \texttt{in}\ \mathit{Exp} \\
 & & \\
\mathit{Exps} & \rightarrow & \mathit{Exp} \\
\mathit{Exps} & \rightarrow & \mathit{Exp}\ ,\ \mathit{Exps}
\end{array}
$$

$Call_{Fun}(Fun, args, ftable) =$ case Fun of	
$TypeId\ (\ TypeIds\ )\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds, args)$ $v_1 = Eval_{Exp}(Exp, vtable, ftable)$ *if* v_1 is of type t_0 *then* v_1 *else* **error**()

$Get_{TypeId}(TypeId) =$ case $TypeId$ of	
`int` **id**	$(getname(\textbf{id}), \texttt{int})$
`bool` **id**	$(getname(\textbf{id}), \texttt{bool})$

$Bind_{TypeIds}(TypeIds, args) =$ case $(TypeIds, args)$ of	
$(TypeId,$ $[v])$	$(x, t) = Get_{TypeId}(TypeId)$ *if* v is of type t *then* $bind(emptytable, x, v)$ *else* **error**()
$(TypeId\ ,\ TypeIds,$ $(v :: vs))$	$(x, t) = Get_{TypeId}(TypeId)$ $vtable = Bind_{TypeIds}(TypeIds, vs)$ *if* $lookup(vtable, x) = unbound$ *and* v is of type t *then* $bind(vtable, x, v)$ *else* **error**()
_	**error**()

Interpreting a Program

$$
\begin{array}{lcl}
Program & \rightarrow & Funs \\
 & & \\
Funs & \rightarrow & Fun \\
Funs & \rightarrow & Fun\ Funs \\
 & & \\
Fun & \rightarrow & TypeId\ (\ TypeIds\ )\ =\ Exp \\
 & & \\
TypeId & \rightarrow & \texttt{int}\ \mathbf{id} \\
TypeId & \rightarrow & \texttt{bool}\ \mathbf{id} \\
 & & \\
TypeIds & \rightarrow & TypeId \\
TypeIds & \rightarrow & TypeId\ ,\ TypeIds \\
 & & \\
Exp & \rightarrow & \mathbf{num} \\
Exp & \rightarrow & \mathbf{id} \\
Exp & \rightarrow & Exp\ +\ Exp \\
Exp & \rightarrow & Exp\ =\ Exp \\
Exp & \rightarrow & \texttt{if}\ Exp\ \texttt{then}\ Exp\ \texttt{else}\ Exp \\
Exp & \rightarrow & \mathbf{id}\ (\ Exps\ ) \\
Exp & \rightarrow & \texttt{let}\ \mathbf{id}\ =\ Exp\ \texttt{in}\ Exp \\
 & & \\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

$Run_{Program}(Program, input) = \texttt{case}\ Program\ \texttt{of}$	
$Funs$	$ftable = Build_{ftable}(Funs)$ $def = lookup(ftable, \texttt{main})$ $if\ \text{def} = unbound$ $then\ \mathbf{error}()$ $else$ $Call_{Fun}(def, [input], ftable)$

$Build_{ftable}(Funs) = \texttt{case}\ Funs\ \texttt{of}$	
Fun	$f = Get_{fname}(Fun)$ $bind(emptytable, f, Fun)$
$Fun\ Funs$	$f = Get_{fname}(Fun)$ $ftable = Build_{ftable}(Funs)$ $if\ lookup(ftable, f) = unbound$ $then\ bind(ftable, f, Fun)$ $else\ \mathbf{error}()$

$Get_{fname}(Fun) = \texttt{case}\ Fun\ \texttt{of}$	
$TypeId\ (\ TypeIds\ )\ =\ Exp$	$(f, t_0) = Get_{TypeId}(TypeId)$ f

Department of
Computer Engineering
Faculty of Engineering
University of Sri Jayewardenepura

13

Advantages and Disadvantages of Interpretation

Akarshani Amarasinghe

Advantages

- The simplest way of executing a program once you have its abstract syntax tree.
- To get faster execution, we use the observation that a program that only executes each part of the program once will finish quite quickly.

Disadvantages

- Relatively slow.
- Spend much more time on figuring out what to do and if it is OK to do it than on actually doing it.