Thoughts on Assignment 6
PL Feature: Execution Model

CS 331 Programming Languages
Lecture Slides
Monday, April 7, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
Semantics & Interpretation

Topics

- ✓ Introduction to semantics
- ✓ Specifying semantics
- ✓ How interpreters work
- ✓ Writing an interpreter

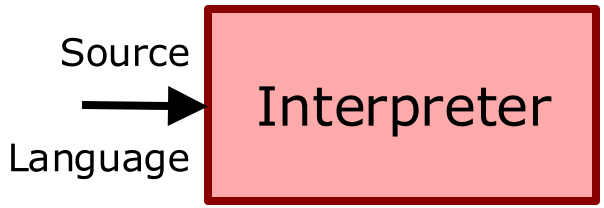
DONE

However, we will continue writing code for an interpreter in *Thoughts on Assignment 6.*

Review

Review
How Interpreters Work

An **interpreter** takes code in its source PL and executes it.



There are four main strategies for designing an interpreter. I list them from worst to best performance.

- Do little or no processing of the source code. Execute it line by line, using a **text-based interpreter**. Rare today, except for shells.
- Parse the source code to get an AST. Execute the AST directly, using a **tree-walk interpreter**. Rare today.

Ours

- Compile to a byte code. Execute the byte code directly, instruction by instruction, using a **virtual machine** (**VM**). Very common today.
- Compile to a byte code. Execute the byte code using a **JIT**, which compiles the byte code to machine language as it executes. Somewhat common today, and getting more common.

Review
Writing an Interpreter

We wrote an arithmetic-expression evaluator in the form of a tree-walk interpreter that handles the ASTs produced by `rdparser3.lua`.

To evaluate, check to see what kind of node the root is. Make function calls (recursive calls?) on each child, as appropriate.

See `evaluator.lua`*.*
See `calculator.lua` *for an appropriate main program.*

Thoughts on Assignment 6

Thoughts on Assignment 6
Introduction

Assignments 3 & 4 involved writing a lexer and parser for the Fulmar programming language. Assignment 6 completes the trilogy with a tree-walk interpreter that takes an AST in the format returned by the parser from Assignment 4.

As with the previous two parts, this will be written in Lua: a module `interpit`, which exports a function `interpit.interp`.

A specification of the semantics of Fulmar and requirements on your implementation are in the Assignment 6 description. These slides contain some relevant ideas & examples.

Thoughts on Assignment 6
The Goal

Here once again is a sample Fulmar program.

```
# fibo (param in variable n)
# Return Fibonacci number F(n).
func fibo()
    currfib = 0
    nextfib = 1
    i = 0  # Loop counter
    while i < n
        tmp = currfib + nextfib
        currfib = nextfib
        nextfib = tmp
        i = i+1
    end
    return currfib
end
```

```
# Main program
# Print some Fibonacci numbers
how_many_to_print = 20

println("Fibonacci Numbers")

j = 0  # Loop counter
while j < how_many_to_print
    n = j  # Set param for fibo
    ff = fibo()
    println("F(", j, ") = ", ff)
    j = j+1
end
```

Thoughts on Assignment 6
Function interpit.interp [1/2]

`interpit.interp` takes three parameters:

`ast`

The AST to interpret, in the format returned by `parseit.parse`.

`state`

Table holding the current state: values of functions, simple variables, and arrays. This is passed so that Fulmar code can be entered interactively, line by line, and handled as a series of separate programs, each getting its state from the earlier code.

`util`

Table with three function members, to be called when doing numeric input, string output, and random number generation.

`interpit.interp` will return the new state.

Thoughts on Assignment 6
Function interpit.interp [2/2]

You will need to write a number of helper functions. I suggest that, *at the very least*, you write the following four:

- A function that takes the AST for a program and executes it, updating the state appropriately.
- A function that takes the AST for a statement and executes it, updating the state appropriately.
- A function that takes the AST for an argument in a `print`/`println` statement, evaluates it, and returns its value (as a Lua `string`).
- A function that takes the AST for a numeric expression, evaluates it, and returns its value (as a Lua `number`).

These will mostly be recursive—perhaps indirectly. For example, the function that executes a program will be called to execute an entire program, or the body of a function, or the body of an if-statement or while-loop.

Thoughts on Assignment 6
State

State will be stored as a Lua table with three members: `f`, `v`, and `a`, holding functions, simple variables, and arrays, respectively:

- The AST for function `abc` will be in `state.f["abc"]`.
- The value of simple variable `abc` will be in `state.v["abc"]`.
- The value of array item `abc[2]` will be in `state.a["abc"][2]`.

All Fulmar identifiers are global and have dynamic scope. Once a variable/function is given a value, it has that value everywhere in the code. Therefore, only one state table is needed.

Fulmar has no fatal runtime errors. Undefined variables are treated as if they have a default value.

- The default AST for a function is `{ PROGRAM }`.
- The default value for a simple variable or array item is `0` (zero).

Thoughts on Assignment 6
Utilities [1/2]

I provide a runtime system for Fulmar, including the following.

`numToStr`

Number → string. Use in numeric output.

`strToNum`

String → number. Use in numeric input.

`numToInt`

Number → integer. Call this after *every* numeric computation.

`boolToInt`

Lua `boolean` → integer.

`astToStr`

Return a printable form of a given AST. *For debugging only*.

And all of Lua is available to be used.

Thoughts on Assignment 6
Utilities [2/2]

In addition, table `util`, which is passed to `interpit.interp`, has three members, all of which are functions.

`util.input`

Returns a string holding a line of input, without the ending newline. This must be used for all input (`readnum` calls).

`util.output`

Takes a string to output. This must be used for all output (`print`/`println` calls).

`util.random`

Takes an integer `n`. If `n >= 2`, returns a pseudorandom integer from `0` to `n-1`, inclusive. Otherwise, returns zero. This must be used to obtain the return value of each `rnd` call. The passed integer is the value of the argument to `rnd`.

Thoughts on Assignment 6
Numeric & Boolean Values

Fulmar has no separate Boolean type. When a Fulmar number is treated as a Boolean, it is *true* if it is nonzero (… `~= 0`) and *false* otherwise.

For most Fulmar operators, the computation is that done by the Lua operator with the same name, followed by a call to `numToInt` or `boolToInt`, as appropriate. Some exceptions:

- If the second operand of "`/`" or "`%`" is zero, then the result is zero.
- The Fulmar "`&&`", "`||`", and "`!=`" operators correspond to the Lua "`and`", "`or`", and "`~=`" operators, respectively.
- Unlike Fulmar, Lua has no unary "`+`" operator. The Fulmar unary "`+`" operator simply returns its operand unchanged.
- When the Fulmar bracket operator is used, the expression between brackets is evaluated; its value is used as a key for the appropriate member of `state.a`.

Thoughts on Assignment 6
Function Calls

A Fulmar function may be called as a function-call statement, or in an expression. When a function is called, its AST is executed in the same way the AST for a program is executed.

As a statement, a function call has no value.

As an expression, the value of a function call is the value of the simple variable `return` after the function body is executed. This variable is stored just like any other simple variable. But it is only used for return values of functions. Since “`return`” is a reserved word, we cannot say “`return = …`”.

Thoughts on Assignment 6
Handling Variables & Functions [1/3]

Saving a newly defined function is easy:

```
state.f[funcname] = ast
```

Similarly, setting the value of a simple variable is easy:

```
state.v[varname] = value
```

Thoughts on Assignment 6
Handling Variables & Functions [2/3]

Setting the value of an array item is a little trickier, since the array may not exist.

First, check if the array exists:

```
if state.a[arrayname] == nil then
    …
```

If the array does not exist, then create an empty array:

```
state.a[arrayname] = {}
```

In either case, the array item can now be set:

```
state.a[arrayname][index] = value
```

Thoughts on Assignment 6
Handling Variables & Functions [3/3]

When *getting* a simple variable, array item, or function, always check for nonexistence.

If a simple variable or array item does not exist, then its value is considered to be `0` (zero). If a function does not exist, then its AST is considered to be `{ PROGRAM }`

When getting an array item, first check whether the array exists. If not, then all array items have the value `0` (zero).

If the array does exist, then check whether the array item exists. Again, if not, then its value is considered to be `0` (zero).

Thoughts on Assignment 6
How I Did It [1/2]

Here is how I wrote the interpreter. As usual, you are not required to do things exactly the same way I did. However, my way does have the advantage that it is known to work.

My four main helper functions, mentioned a few slides back, are named as follows. Each takes an AST.

- `interp_program`
- `interp_stmt`
- `eval_print_arg` (returns `string`)
- `eval_expr` (returns `number`)

For the sake of modularity, it would a good idea to break some of these into multiple smaller functions.

Thoughts on Assignment 6
How I Did It [2/2]

Writing `eval_expr`

- This function takes an AST and returns the value of the expression.
- It is called for the right-hand side of an assignment statement, a `print`/`println` argument that is an expression, a condition after `if`, `elseif`, or `while`, an array index, and the argument to a `chr` or `rnd` call.
- It can call itself recursively, for arguments of operators.
- I wrote it in the form of a number of cases:
  - `ast[1] == NUMLIT_VAL`
  - `ast[1] == READNUM_CALL`
  - `ast[1] == RND_CALL`
  - `ast[1] == FUNC_CALL`
  - `ast[1] == SIMPLE_VAR`
  - `ast[1] == ARRAY_VAR`
  - `type(ast[1]) == "table"`, and:
    - `ast[1][1] == BIN_OP`
    - `ast[1][1] == UN_OP`

Thoughts on Assignment 6
General

Be DRY! If you have written a function, then you can use it.

You may assume the AST you are given is formatted correctly.

Write all functions local to `interpit.interp`.

- You do not need to pass around `state`, `util`. Do pass the AST.
- As in `parseit`, you will need to forward declare the local functions:

```
local ff

function ff(…)
    …
```

Thoughts on Assignment 6
CODE

TO DO

- Begin writing function `interpit.interp`.

Partially done. See `interpit.lua`*.*
I do not plan to make any further changes to this file.

Unit Overview
The Prolog Programming Language

Our seventh unit: The Prolog Programming Language.

Topics

- PL feature: execution model
- PL category: logic PLs
- Introduction to Prolog
- Prolog: simple programming
- Prolog: lists
- Prolog: flow of control
- Prolog: interaction

After this will be Student Presentations on Programming Languages.

PL Feature: Execution Model

PL Feature: Execution Model
Introduction, Commands, Evaluation

These slides are an incomplete summary of the reading "Programming Language Execution Models".

There is always something that drives the execution of a program.

- There is some **task** the computer is attempting to perform.
- There is some **strategy** for carrying out the execution.

In "C", the task is completing a call to function `main`. The strategy is to carry out the commands in function `main`. If other functions are called, these become **subtasks**.

Lua is similar, but the task is executing the code at global scope.

In Haskell, the task is evaluating some expression, perhaps `Main.main`. The strategy is to evaluate the primary function/operator in the expression, with subexpressions becoming subtasks.

PL Feature: Execution Model
Unification

To **unify** two constructions means to make them the same by **binding** variables (setting their values) as necessary.

Examples (upper-case letters are variables):

- We can unify X and 8 by setting X to 8.
- 5 and 8 cannot be unified.
- We can unify 5 and 5 by doing nothing.
- We can unify X and the list [1, 5] by setting X to [1, 5].
- We can unify [A, 6] and [4, B] by setting A to 4, and B to 6.

PL Feature: Execution Model
Unification in Prolog

Unification can be the basis of another execution strategy.

In the **Prolog** programming language (covered next), execution is driven by the task of answering some **query**—a question, roughly.

The strategy is to unify something we wish to prove true with something known to be true. The simplest example of the latter is a Prolog **fact**, which says, essentially, “Here is something true.”

But there is more complexity to it. Details to come.