Writing an Interpreter

CS 331 Programming Languages
Lecture Slides
Friday, April 4, 2025

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

Unit Overview
Semantics & Interpretation

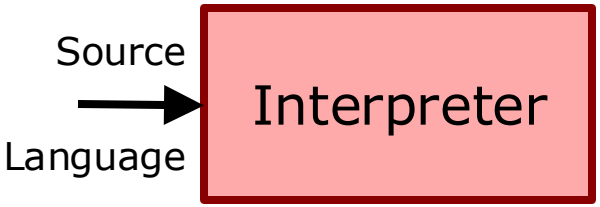
Topics

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

Review

Review
How Interpreters Work [1/2]

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



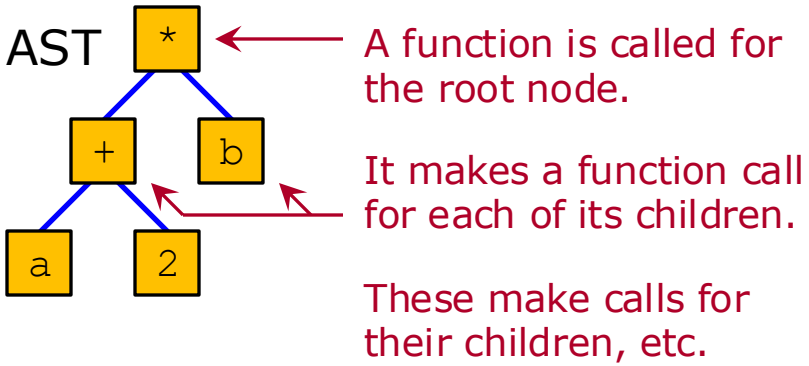
There are four main strategies for designing an interpreter. I list them from worst to best performance.

- 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.
- 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
How Interpreters Work [2/2]

Processing an AST is typically done via mutually recursive functions. A function is called for the root node. It makes a function call for each of its children, and so on. This is called *walking the tree*.

In a **tree-walk interpreter**, these functions do the execution without any further processing.



We know of faster methods; tree-walk interpreters are uncommon. However, they are easy to write. An early release of a PL might include a tree-walk interpreter, with faster interpreters written later. The Ruby PL was handled this way, for example.

Writing an Interpreter

Writing an Interpreter
Introduction

Now we look at how an interpreter is written.

First, we discuss issues that any interpreter will need to deal with, and how these might be handled.

Then we look at these issues for a simple tree-walk interpreter that evaluates arithmetic expressions. We conclude by writing such an interpreter.

Writing an Interpreter
General Issues — Semantics

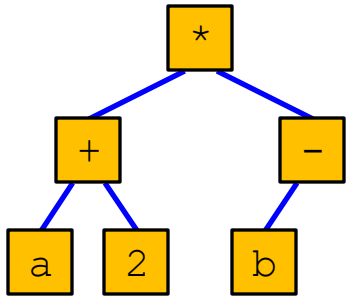
We can write a lexer and parser for a programming language without knowing anything about its semantics.

But to write an interpreter, we must know *everything* about the semantics. We cannot write an interpreter for a PL without a semantics specification.

If the source code is parsed, then there will be an AST. We need to know the format of this tree.

We will also need to process the AST, either by interpreting it directly (tree-walk interpreter), or by generating code or some other IR from it.

```
(a + 2) * -b
```



As we have said, rooted trees, including ASTs, are usually dealt with as follows.

- Call a function to handle the root node.
- Make a function call (recursive call?) on each child of the root, as appropriate.

While an interpreter is executing a program, it must store program **state**: variable values, the call stack, etc.

If a PL has static typing and scope (C++, Java, Haskell), then the compiler can determine the types and scopes of all variables. If it is being compiled to machine language, then these can be laid out in memory—for local values, in a stack frame. At runtime, a reference to a value is a reference to its memory location. The system stack can be used for the call stack.

In a dynamic PL (Lua, Python, etc.), it is common to place variables in an associative structure—typically a hash table—with the variable name as key. There will usually be a separate hash table for each namespace or scope.

There will usually need to be code to support the interpretation. For an executing program, this is a **runtime system** (often simply **runtime**): additional code that programs will use while running.

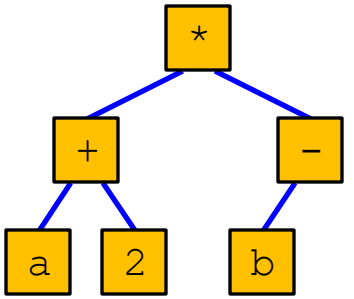
Some things that a runtime system might include:

- Program startup/initialization and shutdown.
- Memory management.
- Low-level I/O.
- Interfaces to other operating system functionality: threads, interprocess communication, etc.
- Support for error handling.
- Implementations of PL commands that perform complex operations: advanced floating-point computations, operations involving multiple data items like sorting or matrix operations, etc.

Writing an Interpreter
Arithmetic-Expression Evaluator — Overview

Let's write an evaluator for simple arithmetic expressions in Lua, as a tree-walk interpreter that takes an AST.

We use the arithmetic-expression syntax parsed by the last Recursive-Descent parser written in class: `lexer.lua` + `rdparser3.lua`.



We will need to deal with all the issues we have mentioned:

- **Semantics.** What is the semantics of our arithmetic expressions?
- **Processing an AST.** What format will our ASTs be stored in? How will our tree-walk interpreter evaluate an arithmetic expression, according to its semantics, based on an AST?
- **State.** Our expressions can involve named variables. How will we represent and access their values?
- **Support Code.** What support code (runtime system?) is needed?

Arithmetic-Expression Evaluator — Semantics [1/4]

We will specify the semantics of our expressions informally.

As we did some weeks ago, in *Regular Languages & Regular Expressions*, we can specify the semantics of arithmetic expressions by describing the semantics of the pieces (numeric literals, simple variables) and then the semantics of ways to build expressions out of smaller ones (operators).

However, we will need to specify details. For example, previously, we said, "The value of a numeric literal is its numeric value." But how will we represent this? What values are legal?

Since we are writing our expression evaluator in Lua, we can handle many of these issues easily, simply by saying that some particular Lua functionality will be used.

Expression Semantics

- The value of an expression is a Lua `number`. All `number` values are legal.
- The value of a numeric literal is the result when the string is passed to the Lua built-in function `tonumber`. (The format for numeric literals used by `lexer.lua` works with `tonumber`.)
- The value of a simple variable is found by looking it up. (More on this shortly.)
- There are four operators (`+`, `-`, `*`, `/`). All are binary. The result of each of these is the result when the corresponding Lua binary operator is applied to the values obtained by evaluating each of the two operands.

(Cont'd)

Expression Semantics (cont'd)

Lua follows the IEEE floating-point standard, which specifies **+infinity**, **–infinity**, and **NaN** (**Not a Number**) values for operations without numeric values. These do not crash or raise exceptions.

Do not forget edge cases!

- Dividing by zero gives whatever value the Lua operation returns.
- Other numeric-computation edge cases (overflow, underflow) are handled similarly: the result is the value of the Lua operation.
- When an undefined variable is used, it is treated as if it has value zero.

Arithmetic-Expression Evaluator — Semantics [4/4]

Our expression semantics has two important consequences.

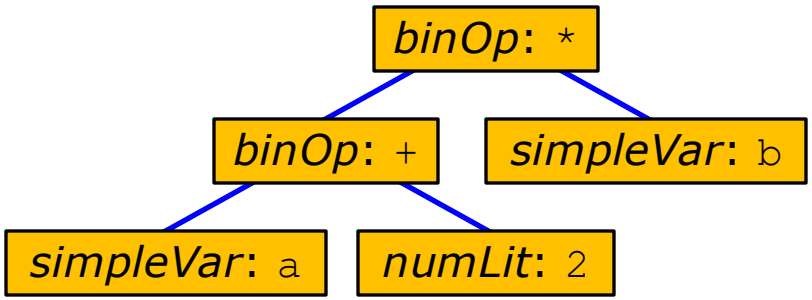
First, there are no fatal runtime errors. Every expression that parses correctly has a value that can be computed and displayed. The evaluation function requires no error-handling code. Similarly, its caller will not need to check for errors, once parsing is successful.

Second, the semantics of our arithmetic expressions is entirely about what their value is. We do not need to worry about declaring new variables or setting/changing their values.

Arithmetic-Expression Evaluator — Processing an AST [1/3]

It will be convenient to use the AST format from `rdparser3.lua`. We can then use `rdparser3.lua` as a source of ASTs.

```
(a + 2) * b
```

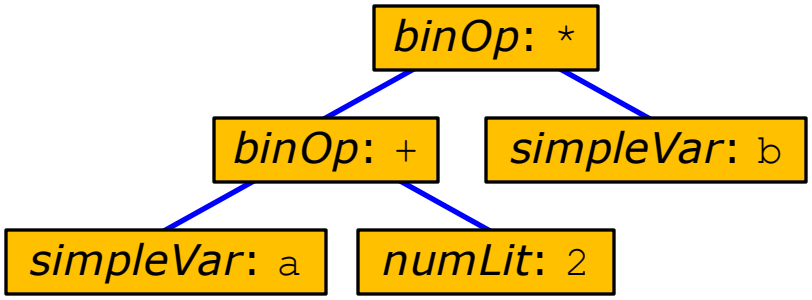


=

```
{ { BIN_OP, "*" },
  { { BIN_OP, "+" },
    { SIMPLE_VAR, "a" },
    { NUMLIT_VAL, "2" } },
  { SIMPLE_VAR, "b" } }
```

In these ASTs, each node represents one of three things: a numeric literal, a simple variable, or a binary operator.

Writing an Interpreter
Arithmetic-Expression Evaluator — Processing an AST [2/3]



=

```
{ { BIN_OP, "*" },
  { { BIN_OP, "+" },
    { SIMPLE_VAR, "a" },
    { NUMLIT_VAL, "2" } },
  { SIMPLE_VAR, "b" } }
```

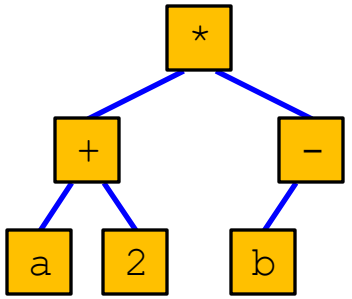
An AST is a Lua array.

- Numeric literal. 2-item array: `{ NUMLIT_VAL, `*STR*` }`
  STR is a string holding the literal.
- Simple variable. 2-item array: `{ SIMPLE_VAR, `*STR*` }`
  STR is a string holding the variable name.
- Other. 3-item array: `{ { BIN_OP, `*STR*` }, `*OPER1*`, `*OPER2*` }`
  STR is a string holding the operator.
  OPER1 and *OPER2* are the ASTs of the two operands.

`BIN_OP`, `NUMLIT_VAL`, `SIMPLE_VAR` are `1`, `2`, `3`, respectively.

Writing an Interpreter
Arithmetic-Expression Evaluator — Processing an AST [3/3]

To process an AST, we can write a function that takes the AST for an expression and returns its numeric value. To evaluate subexpressions, the function can call itself recursively.



Based on our arithmetic-expression semantics, the evaluation function can work like this.

- If the root node represents a numeric literal:
  - Convert the literal to a number and return it.
- Else if the root node represents a numeric variable:
  - Get the variable's value and return it.
- Else (the root node represents a binary operator):
  - Compute the value of the left subtree—recursive call.
  - Compute the value of the right subtree—recursive call.
  - Apply the appropriate operation and return the result.

More steps would be needed if our expressions involved function calls or operators of other arities.

Arithmetic-Expression Evaluator — State

None of our operations have side effects. So we do not need to maintain any **mutable** (changeable) state. However, values of variables do need to be stored.

Since we are writing Lua, we will not lay anything out in memory. We store values of variables as Lua `number` values in a Lua `table` with the variable name (Lua `string`) as the key.

There is only one namespace/scope. Just one Lua `table` is needed.

```
vars = {
         ["pi"] = 3.14159265358979323846,
         ["answer"] = 42,
         …
       }
```

Arithmetic-Expression Evaluator — Support Code

What code is needed to support our interpreter?

The interpreter will need to:

- Read an AST.
- Convert a numeric literal (string) to its numeric value.
- Look up a variable value in a table.
- Perform arithmetic operations (add, subtract, multiply, divide).

Facilities to do each of these are built into Lua. So we do not need to write code to implement details of our operations; we simply use the existing Lua functionality.

However, it would be nice to have a calculator program that inputs an expression from the user, parses it, calls our interpreter to compute its value, and outputs the result—or an error message.

This program can also set the values of a few variables.

Writing an Interpreter
Arithmetic-Expression Evaluator — CODE

Putting it all together, we can write an arithmetic-expression evaluator in the form of a tree-walk interpreter that handles the ASTs produced by `rdparser3.lua`.

We will write a function `eval` exported by a module `evaluator`, stored in the file `evaluator.lua`. This function will take an AST and a table holding values of variables.

TO DO

- Write an arithmetic-expression evaluator as described. Be sure to understand the AST format thoroughly before doing any coding.

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

Writing an Interpreter
Final Note

Looking at function `evaluator.eval`, if we count the lines that actually do something (so do not count blank lines, comments, `assert` calls, and "`end`"), we end up with a bit more than 20.

That is not very much!

Q. Why can it be so short?

A. Because a lot of the work is already done, in the lexing and parsing phases. (Counting lines in the same way, the parser is over 80 lines, while the lexer is well over 200 lines.)

Conclusion. Lexing and parsing are worthwhile things to do. An AST is a convenient, useful representation.