

CSE 131A
Lecture 13

Building an Onyx Interpreter

Scott B. Baden

Department of Computer Science & Engineering

University of California, San Diego

Winter 2007

Announcements

- Midterm return on Friday

From Parser to Interpreter

- Assignment 3 is to construct an Onyx interpreter
- The parser plays an important role
 - Analyzes the syntactic structure of Onyx input
 - Determines if an input string abides by the Onyx syntax rules
 - Reports syntax errors
- **But most important of all:** *translates* the input program into a form that a machine can “run,” preserving the meaning or *semantics* of the program

Making progress with the interpreter

- Building the interpreter will take time ..
- Better to work steadily rather than in bursts
- If you have a partner, be sure of their schedule
- Plan your work and your time. See one of us if you'd like advice
- Fix the parser
- If you part of it is working, and you are sure of your design..
- ... you may prefer to work on the interpreter in order to pass the simplest public tests
- To do this, determine which aspects of the language are *orthogonal*
- For example: does expression evaluation require that you have a working variable name resolution?
 $3 + 4$ $\$a + \b
- Do a case analysis

A workflow

3

3 + 4

“string”

if (3 < 4) then true else false

concat(“AB”, “CD”)

declare function plus(\$a,\$b) {\$a + \$b}; plus(3,4)

declare function plus(\$a as onyx.types.Integer,\$b) {\$a + \$b};
plus(3,4)

let \$a :=3

return \$a

let \$a :=3

return \$a + 4

(1,3,5) ((5,6,7),8)

length((1,2,3)) length(1)

1 to 3

for \$i in (1,3,5)

return \$i / return 2*\$i

for \$i in (1 to 9)

where \$i > 3

return 2*\$i

Interpreters and compilers

- A *translator* is a program that transforms input source into a form that is consistent with the semantics of the language
- Often, the result is a representation which can be run by the hardware
- The translation process can have different forms
- The parser can directly perform the actions required by the semantics of the input program text it is parsing: this is called an *interpreter*
- The parser can output semantics-preserving machine code to a file for later execution: this is called a *compiler*
- It can also generate an AST, which may be used by a compiler or interpreter

Syntax and semantics

- The syntax of an input language is defined with context-free grammar rules
- But translation involves more than meeting a context-free definition
- What remain are considered to be the semantics of the input
- Semantic features of the input language can be defined with semantic rules
- We need to link the semantic and syntactic rules

Error checking

- A translator can perform *static checks* of program correctness, without running the program
 - All syntactic checks
(unless we have run time code generation)
 - Some semantic checks
- All other semantic checks must be handled dynamically while the program runs
 - Division by zero
 - File not found

Examples of static checks

- Scope checks: a variable must not be used before it is declared
- Type checks: the types of expressions used as arguments to functions or operators must be compatible with the requirements

1 + "string"

Software Design

- Design is an important part of any large software project
- A good design makes implementation, debugging, and maintenance easier
- Here we will sketch the design of some components of an Onyx interpreter

Design of an Onyx Interpreter

- The design sketch here is not the only possible one, but almost any reasonable design will have features in common with this one
- For example this design mentions AST construction and traversal, though much of it could be used with a direct SDT approach we will discuss in detail later on

AST approach to an interpreter

- The parser takes constructs an AST
- Then the AST is traversed, typically in a simple top-down, depth-first order (i.e. postorder)
- Perform computations according to the semantics of the input program during traversal

SDT approach to an interpreter

- The parser takes an input program and while parsing...
- ... either directly does the computations needed according to the semantics of the input program,
- ... or emits code which when interpreted carries out the computations needed according to the semantics of the input program
- We will return to this later on

Tradeoffs

- The AST approach requires multi-pass translation
- The separation of AST traversal from parser actions makes for a cleaner design
- Some operations become easier when we have an explicit AST, such as optimization
- Time and space costs associated with multi-pass techniques
- However, memory and CPU cycles are cheap

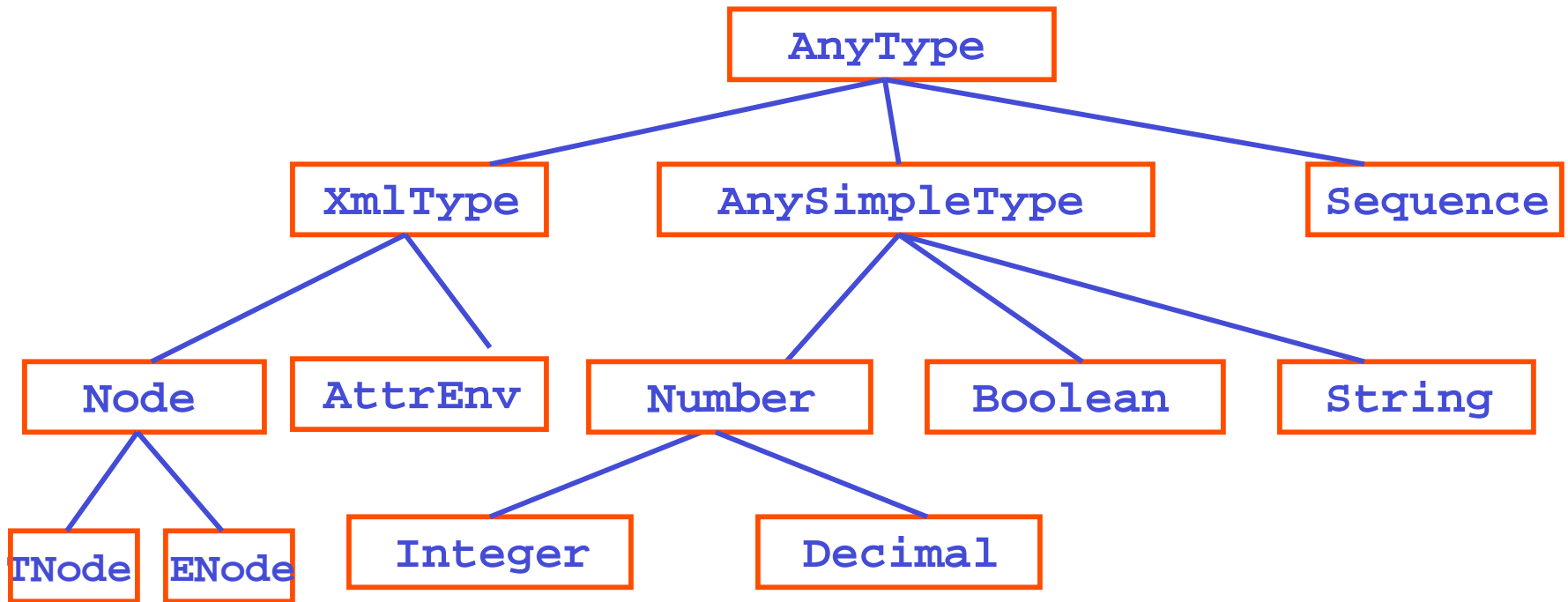
Fundamental semantics of Onyx

- Onyx is an expression language
- Every Onyx expression has a value
- An Onyx program is just a sequence of expressions, and its value is a sequence of values
- The interpreter will compute the values in that sequence and print them out
- The interpreter needs a way to represent those values so they can be used in further computations, or can be printed out

Types in Onyx

- Onyx is a strongly typed language: every value is of a definite type
- Value types determine how values interact to produce other values in expressions, and determine what expressions are “legal”
- So, the interpreter needs to represent not only values, but also types

Type hierarchy in Onyx



Representing values

- A representation of an Onyx value needs to represent the value's type as well as the value's value
- This can be done in different ways
 - Represent an Onyx value as a pair: A value, and a representation of its type
 - In an implementing language with run-time type information (Java, C++), represent an Onyx value as an appropriate run-time typed value

Representing types

- If each type has a unique name, the name (in the form of a String, say) can represent the type
- In Java, we can represent a type using an object that is an instance of the class **Class**
- We can get the **Class** object representing the type of any object by using the object's **getClass()** instance method
- We can get the **Class** object corresponding to a class by accessing the **class** static variable of the class; the following is **true**
"hi".getClass() == java.lang.String.class
- The **instanceof()** operator checks the membership of the object

Onyx and Java types

- We can use instances of Java types to represent Onyx values
- The Java standard libraries define many types that are a good fit with Onyx types...
- ...except that they have different names, and do not have the same hierarchical relationship
- Still they are useful in implementing the interpreter

Correspondence of Onyx and Java types

- A reasonable correspondence between Onyx and standard Java types would be:

Onyx	Java
<code>AnyType</code>	<code>Object</code>
<code>Boolean</code>	<code>Boolean</code>
<code>Decimal</code>	<code>java.math.BigDecimal</code>
<code>Integer</code>	<code>java.math.BigInteger</code>
<code>String</code>	<code>String</code>
<code>Sequence</code>	<code>java.util.List</code>
<i><code>Node and AttrEnv</code></i>	<i><code>onyx.xml</code></i>

Evaluating expressions

- The value (and type) of an expression is determined by semantic rules applied to the expression
- These rules are stated in terms of
 - the syntax of the language
 - the type system of the language
 - the semantics of the operations defined in the language

Evaluating literal constants

- The semantic rules for expression evaluation are defined recursively, following the recursive structure of the language's grammar rules
- This recursion terminates with a literal constant, whose value is explicitly available in the input program
- Constructing a representation of the value of a literal constant is the most basic thing the interpreter does

Evaluating literal constants

- In Onyx, literal constants have a distinctive lexical form
- Therefore the lexical analyzer can detect them in the input and provide the parser with the matching lexeme and the type of the constant
- The parser/interpreter can then convert the lexeme into the corresponding value of the appropriate type
- These values of literal constants then let the computation get off the ground
- Constants are the fundamental construct in Onyx, since they are at the leaves of the AST

Values and Sequences of Values

- In Onyx, a value can be a sequence of values
- With these additional points:
 - A sequence of length one (a *singleton sequence*) is identical to the element it contains, and vice versa
 - A sequence cannot contain any sequences (only values of `AnySimpleType`)

Distinguishing Singletons and Sequences

- Singleton values and sequences of values are different in Onyx... but are also related
- A sequence of one value needs to be treated exactly the same as the one value alone
- This relationship is not captured in the Onyx type hierarchy, but it does have to be maintained by the interpreter

Values and Sequences of Values

- Useful operations on values and sequences:
 - Convert from a singleton sequence to the value contained in that sequence
 - Convert from a non-sequence value to the sequence containing that value
 - Add values to an existing sequence:
 - adding a sequence to an existing sequence adds the values in the sequence, not the sequence itself
- These operations need to be implemented as part of a support library

Evaluating variables

- After literal constants, variables are the next most basic building blocks of expressions
- The lexical analyzer can recognize the lexical form of a variable...
- ... but unlike literal constants, the value of a variable will be determined by a lookup performed at runtime
- This lookup requires a *symbol table* that maps the variable name to its value

Symbol tables

- A symbol table entry for a variable can contain such information as:
 - the name of the variable
 - the type of the variable
 - means for computing the value of the variable

Evaluating variables

- The symbol table maintains a mapping (“binding”) between variable names and values
- It needs to do this in a way that respects the *scope rules* of the language
 - When a symbol’s binding should be entered in the table, and when it should be removed, depends on the scope rules
 - This can be done at compile time, for static scope and type checking; and/or at run time, for dynamic scope checking and variable value lookup
 - A stack-oriented table is often appropriate

let \$a := 3,

let \$b := 4

return \$a + \$b

let \$a := 3, \$b := 4, \$a := 7

return \$a + \$b

Symbol table operations

- A symbol table should provide these operations:
 - add a new entry for a name
 - add/update information for a name (e.g., type or value information)
 - find information about a name
 - remove an entry for a name
- For example, to keep track of which declaration of a variable is in scope

Entering a scope

- When a new declaration is encountered in compiling or evaluating a program, this is the beginning of a new scope
- Perform these actions:
 - lookup the declared name in the symbol table; if it already exists, this may be an error (depends on language rules and where this declaration occurs)
 - add a new entry so that lookups for the name will now find this entry
 - add information to the entry as it becomes available (type and value)

Leaving a scope

- Scopes begin with the introduction of a name
- The end of the scope depends on the semantics of the language, but usually at the end of the construct in which the name is introduced, e.g.:
 - formal parameter: end of the function definition body
 - local variable: end of the block in which it is declared
- When reaching the end of scope in a compiler or interpreter...
- ... remove all entries from the table that were added in this scope, leaving previous entries alone
- This suggests stack-like behavior: push when entering scope, pop when leaving

Stacks for symbol tables

- Maintain a stack of symbol table entries
- When encountering a variable declaration, push an entry on the stack
- When encountering a use of a variable, look for an entry with that name starting at the top of the stack
 - the first one found is the one in scope
 - if none found, probably a “variable not declared” error
- When leaving scope, pop all entries added in that scope off the stack

Binding variables

- In Onyx, variables are bound to values in 4 ways:
 - in a query prolog variable or function declarations
 - in a function call
 - in a let-clause in a FLWR expression
 - in a for-clause in a FLWR expression
- Let's look at function calls and function definitions first, FLWR expressions later

Operators and functions

- Onyx permits user-defined functions in the query prolog of an Onyx program
- It also has a number of *built-in* predefined functions and operators, e.g.
*+, -, *, or, <, >, div, and*
- These can both be treated with a *function table*

Function tables

- A function table entry can contain:
 - ▶ the name of the function
 - ▶ the return type of the function
 - ▶ the number and type of formal parameters
 - ▶ the logic for computing the function, given values for its parameters

Function table design

- A function table has a different design than a symbol table
- The information needed to represent a function definition is different from what is needed to represent a variable binding
- Onyx function definitions cannot be nested, so a stack structure is not required

Function table entries

- A function definition in Onyx looks like:
`declare function f($a as onyx.types.Integer, $b as onyx.types.Integer)
 as onyx.types.Integer {$a + $b};`
- The function table entry contains
 - the name of the function: `f`
 - its return type: `onyx.types.Integer`
 - the names of the formal parameters: `$a, $b`
 - their types: `onyx.types.Integer, onyx.types.Integer`
 - the body of the function, i.e. the expression that will be evaluated to determine the return value: `$a + $b`
 - A `call()` method

Operators and overloading

- The information should be entered in the function table when the function is declared
- Pre-defined, built-in functions should be entered in the function table when the interpreter starts
 - `concat` `document` `isNode`
- Operators have a special prefix `op:numeric-add`
- Onyx permits function overloading: you can declare a function with the same name but different number of arguments and types as an existing one
- See the semantic spec about name resolution
- Function overriding (i.e. redefinition) is not allowed

Representing function bodies

- In an AST approach to the interpreter, the function body in a function table entry can just be a pointer to the root of the AST that is the translation of the function definition body expression
- But this does not work in general for predefined built-in functions: addition is defined in terms of a more primitive method (e.g. Java code)

Entries for builtin functions

- Suppose `FunctionTableEntry` is a class
- To declare a function, an instance of `FunctionTableEntry` is created and added to the function table
- For each builtin function, define a subclass of `FunctionTableEntry` that overrides `call()` to implement the function appropriately (including type checks and type coercions)

Evaluating a function call

- To evaluate a function call...
- Evaluate the actual arguments, left to right; the result is a list of values
- Get the `FunctionTableEntry` object from the function table for a function with the given name and number of arguments
- Invoke its `call()` method, passing the actual argument value list, returning the result
 - binds formal to actual parameters (push on stack)
 - evaluates function body in the context of those bindings
 - cleans up stack and returns result of function body

Scope

- There may be independent definitions of the same name in different parts of a program
- The *scope rules* of a language determine which introduction of a name applies when a name is used somewhere in the program text
- The portion of the program text in which a particular name declaration applies is called the *scope* of that name

Symbol table stack in action

```
declare function f($j,$k)
{ $j + $k }
```

```
let $m := 2, ←
```

```
let $k := f($m,3)
```

```
return $k
```

name	value
\$m	2

Symbol table stack in action

```
declare function f($j,$k)
{ $j + $k }

let $m := 2
let $k
  := f($m,3) ←
return $k
```

name	value
\$m	2

Symbol table stack in action

```
declare function f($j,$k) ←  
{ $j + $k }  
  
let $m := 2  
let $k  
  := f($m,3)  
  
return $k
```

name	value
\$k	3
\$j	2
\$m	2

Symbol table stack in action

```
declare function f($j,$k)
{ $j + $k } ←
let $m := 2
let $k
  := f($m,3)
return $k
```

name	value
\$k	3
\$j	2
\$m	2

Symbol table stack in action

```
declare function f($j,$k)
{ $j + $k }

let $m := 2
let $k ←
    := f($m,3)

return $k
```

name	value
\$k	5
\$m	2

Symbol table stack in action

```
declare function f($j,$k)
{ $j + $k }

let $m := 2
let $k
  := f($m,3)

return $k ←
```

name	value
\$k	5
\$m	2

Next time

- Midterm return!

Midterm return

- If you have a regrade request, see the Regrading Policy on the front of the exam
 - DO NOT write on the exam
 - Attach another sheet of paper to the front explaining your request
 - Hand it in **before you leave lecture** today
- **Once you remove your exam from this room, you may no longer request a regrade**
 - Review your exam thoroughly right now. If you need additional time, RETURN the exam to the staff, and then come to office hours to retrieve it. You can review it during office hours or section
- Anyone not claiming their exam today should pick it up within one week (by Friday, March 2nd) either in office hours or section