Where Are We?
PL Feature: Identifiers & Values
PL Feature: Reflection

CS 331 Programming Languages

Lecture Slides

Monday, March 17, 2025

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- ✓ Haskell: functions
- ✓ Haskell: lists
- ✓ Haskell: flow of control
- ✓ Haskell: I/O
- ✓ Haskell: data

DONE

Where Are We?

Where Are We?
From the First Day of Class: Course Overview — Goals

Upon successful completion of CS 331, students are expected to:

- Understand the concepts of syntax and semantics, and how syntax can be specified.
- Understand, and have experience implementing, basic lexical analysis, parsing, and interpretation.
- Understand the various kinds of programming languages and the primary ways in which they differ.
- Understand standard programming language features and the forms these take in different programming languages.
- Be familiar with the impact (local, global, etc.) that choice of programming language has on programmers and users.
- Have a basic programming proficiency in multiple significantly different programming languages.

Where Are We?
From the First Day of Class: Course Overview — Topics

The course material will be divided into eight units:

1. Formal Languages & Grammars
2. The Lua Programming Language
   - PL Feature: Compilation & Interpretation
3. Lexing & Parsing
4. The Haskell Programming Language
   - PL Feature: Type System
5. The Scheme Programming Language
   - PL Feature: Identifiers & Values
   - PL Feature: Reflection
6. Semantics & Interpretation
7. The Prolog Programming Language
   - PL Feature: Execution Model
8. Student Presentations on Programming Languages

Track 1: Syntax & Semantics of PLs.

Track 2: PL features & categories, specific PLs.

We are **here**.

Unit Overview
The Scheme Programming Language

Our fifth unit: The Scheme Programming Language.

Topics

- PL feature: identifiers & values
- PL feature: reflection
- PL category: Lisp-family PLs
- Introduction to Scheme
- Scheme: basics
- Scheme: evaluation
- Scheme: data
- Scheme: macros

After this we will cover Semantics & Interpretation.

PL Feature: Identifiers & Values

PL Feature: Identifiers & Values
Identifiers

These slides are an incomplete summary of the reading "Identifiers, Values, and Variables".

An **identifier** is the name of something in source code.

Every identifier lies in some **namespace**.

Overloading: when a single name *in a single namespace* refers to two (or more) different entities.

The code from which an identifier is accessible forms the identifier's **scope**.

Static scope: scope is determined before runtime.

This is typically **lexical scope**: the scope consists of a fixed portion of the program's source code.

Dynamic scope: scope is determined at runtime.

PL Feature: Identifiers & Values
Values [1/2]

A **value** might be a number or a string or a Boolean or some kind of object.

An **expression** is an entity that has a value.

A **literal** is a representation of a fixed value in source code. The value itself is represented, not an identifier bound to the value, and not a computation whose result is the value.

C++ Literals	
Literal	**Type**
`42`	`int`
`42.5`	`double`
`false`	`bool`
`'A'`	`char`
`"zebra"`	`char[]`

Lua Literals	
Literal	**Type**
`42.5`	`number`
`false`	`boolean`
`"zebra"`	`string`
`[=[xy]=]`	`string`
`{ 1, 2 }`	`table`

PL Feature: Identifiers & Values
Values [2/2]

When a value comes into existence, we say it is **constructed**.

A value continues to exist until it is **destroyed**.

The time between construction and destruction of a value is the value’s **lifetime**, that is, the period during which the value exists.

PL Feature: Identifiers & Values
Variables

A **variable** is an identifier than can be associated with a value. Making this association is called **binding**: a variable is **bound** to a value.

A variable that is not bound within an expression is said to be **free** in that expression.

Remember:

- An **identifier** has a **scope**.
- A **value** has a **lifetime**.

Because a bound variable involves both an identifier and a value, scope and lifetime are both applicable.

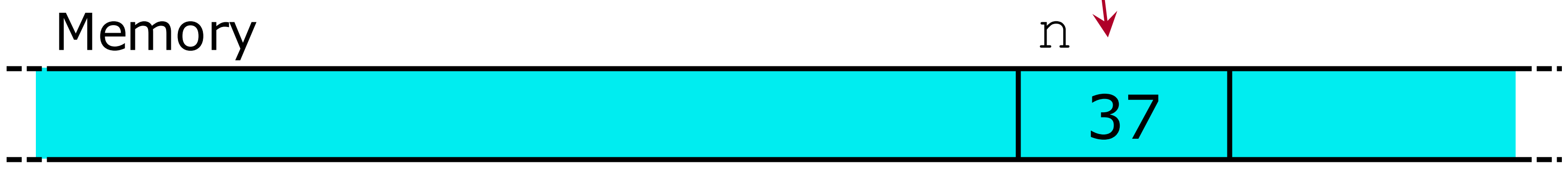
PL Feature: Identifiers & Values Implementation [1/2]

At runtime, a value might be implemented as a block of memory large enough to hold its internal representation.

When a value is set, it is computed, and its representation is stored in the memory block.

```
// C++
int n = …;
```

The right-hand side is evaluated. Say the result is 37. This is stored in the memory location for variable `n`.

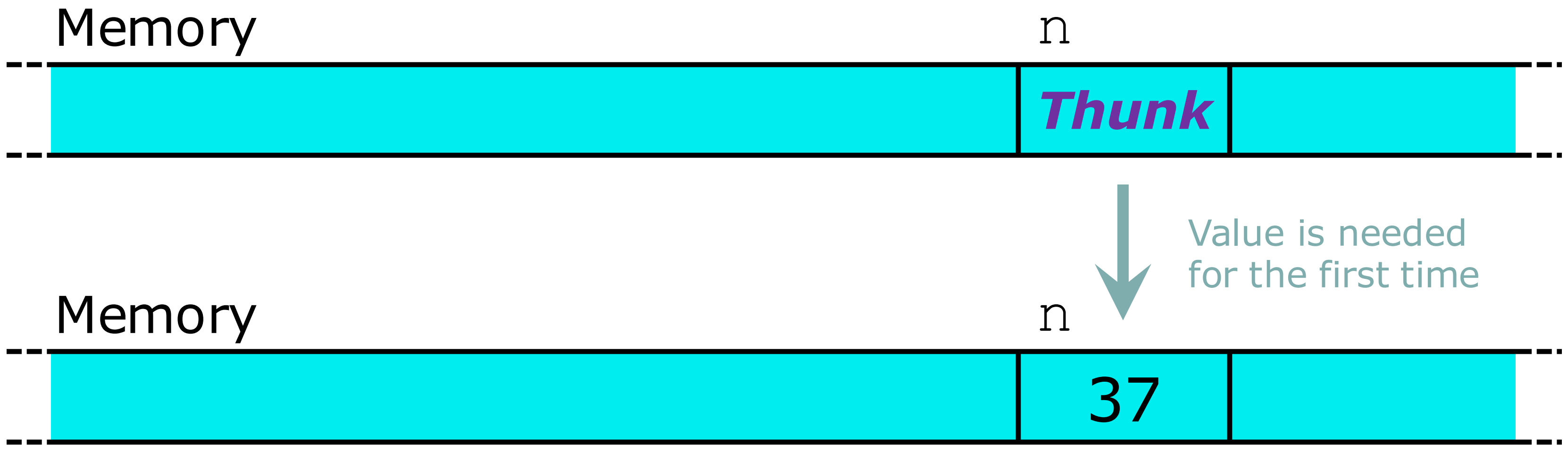


But what if we are evaluation is lazy?

PL Feature: Identifiers & Values Implementation [2/2]

```
-- Haskell
n = …
```

With lazy evaluation, an unevaluated value usually holds a **thunk**: a reference to code whose execution computes the value.



As we have seen, use of thunks allows for infinite data structures.

PL Feature: Reflection

PL Feature: Reflection
What It Is

These slides are an incomplete summary of the reading "Reflection in Programming".

Reflection in a computer program refers to the ability of the program to deal with its own code: examining the code, looking at its properties, possibly modifying it, and executing the modified code.

In practice, reflection is largely a matter of support from the programming language and associated runtime environment. Thus, an important property of a programming language is whether, and how well, it supports reflection.

PL Feature: Reflection
Support in Programming Languages [1/2]

Machine code usually supports reflection, in the form of **self-modifying** code. Today, this is generally frowned on in production software development.

Early high-level programming languages mostly did not support reflection at all, the major exception being **Lisp** (late 1950s). *More on Lisp soon.* C and C++ offer essentially no support for reflection. Haskell is similar.

Reflection was first named and studied in the early 1980s. Since then, various programming languages have included some support for reflection—often in very limited ways.

PL Feature: Reflection
Support in Programming Languages [2/2]

Dynamic PLs can have limited reflection support. For example, Lua "classes" and "objects" can be examined at runtime. All members can be found, along with their types and metatables (if any). Members can be added and removed.

Lisp has evolved into a family of programming languages: Common Lisp, Scheme, EMACS Lisp, Clojure, Logo, and others. These form the gold standard for reflection support.

In Lisp-family PLs, executable code can be made available to a program in the form of an abstract syntax tree, which can be transformed arbitrarily and then executed.

Writing code transformations is a normal part of Lisp programming; in some Lisp-family PLs, constructs called **macros** make such transformations convenient.

PL Feature: Reflection
Example

We will shortly study a programming language called **Scheme**. As a Lisp-family programming language, Scheme offers excellent support for reflection.

TO DO

- Look at an example of reflection in Scheme.

See `reflect.scm`.