Where Are We?
PL Category: Concatenative PLs
Introduction to Forth

CS F331 Programming Languages

CSCE A331 Programming Language Concepts

Lecture Slides

Monday, March 23, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Review

Review
Haskell: Data

We made a Binary Tree type, with a data item in each node.

Such a Binary Tree either has no nodes (it is **empty**) or it has a **root** node, which contains a data item and has **left** and **right subtrees**, each of which is a Binary Tree.

The type is called the type `BT`. It has two constructors.

- `BTEmpty` gives an empty Binary Tree.
- `BTNode`, followed by an item of the value type, the left subtree, and the right subtree, constructs a nonempty tree.

```
data BT vt = BTEmpty | BTNode vt (BT vt) (BT vt)
```

The **value type**

We implemented the Treesort algorithm in Haskell, using `BT` for the Binary Search Tree.

See `data.hs`.

Review
PL Feature: Values & Variables [1/3]

Remember:

- A **value** has a **lifetime**: time from construction to destruction.
- An **identifier** has a **scope**: where in code it is accessible.

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

Review
PL Feature: Values & Variables [2/3]

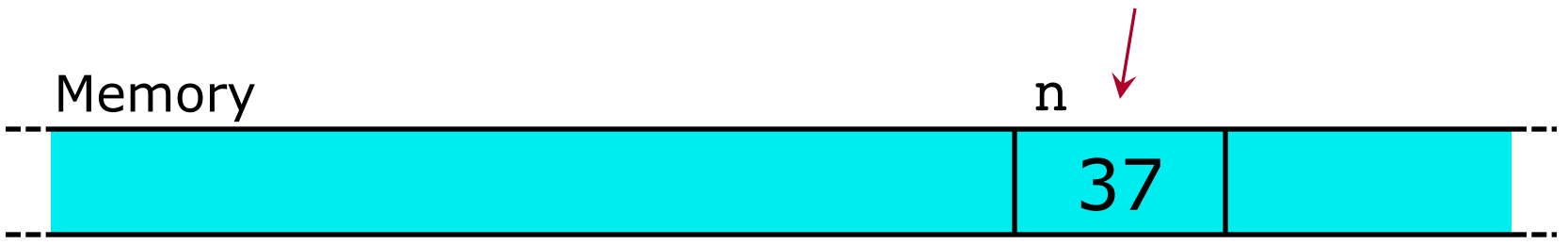
At runtime, a variable is typically implemented as a location in memory large enough to hold the internal representation of the variable's value.

When a variable's value is set, the value is computed and its representation is stored in the memory location.

```
// C++
int n = f(1) + g(2);
```

Functions `f` & `g` are called. Their return values are added. Say the result is 37.

This is stored in the memory location for variable `n`.

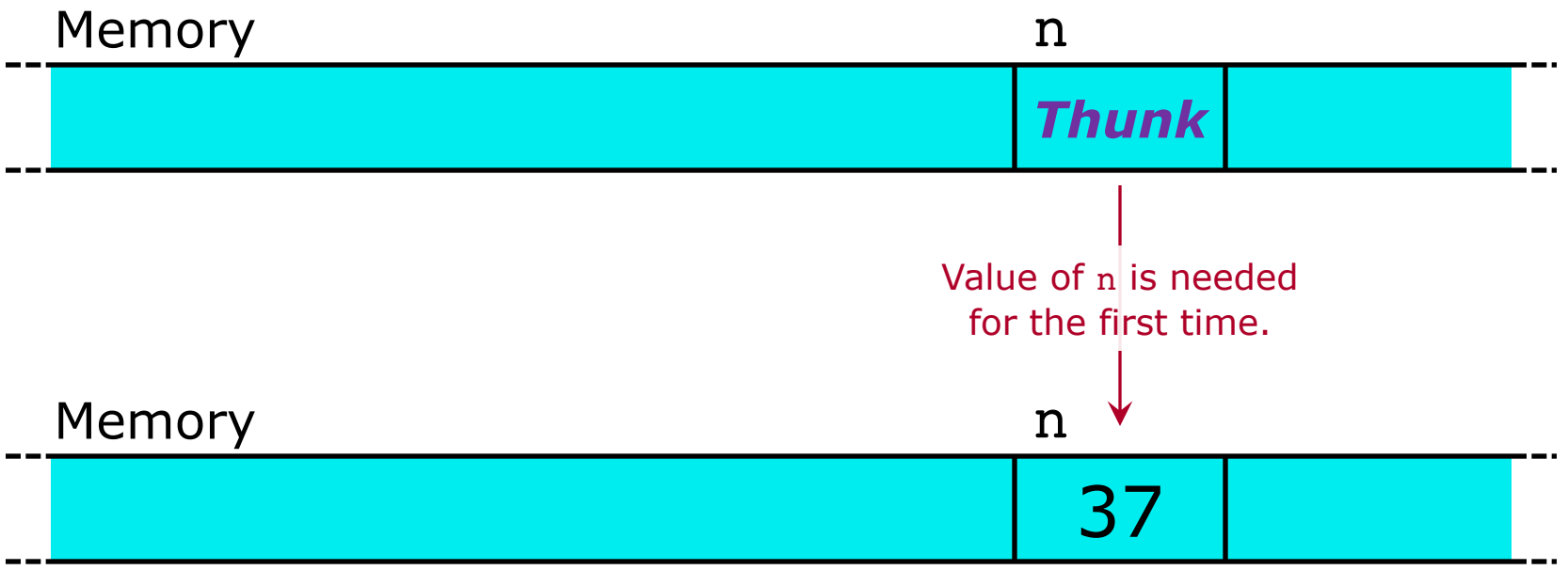


But what if we are evaluation is lazy?

Review
PL Feature: Values & Variables [3/3]

```
-- Haskell
n = (f 1) + (g 2)
```

The usual way to implement a variable when lazy evaluation is done, is to have an unevaluated variable hold a **thunk**: a reference to code whose execution computes a value.



Where Are We?

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

In this class, we study programming languages with a view toward the following.

- How programming languages are specified, and how these specifications are used.
- What different kinds of programming languages are like.
- How certain features differ between various programming languages.

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

After taking this class, you should:

- 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 following topics will be covered:

- Formal Languages & Grammars.
- PL Feature. Execution I: compilers, interpreters.
- PL #1. Lua.
- Lexing & Parsing.
- PL Feature. Type Systems.
- PL #2. Haskell.
- PL Feature. Values & variables.
- PL #3. Forth.
- Semantics & Interpretation.
- PL Feature. Reflection.
- PL #4. Scheme.
- ~~PL Feature. Execution II: execution models, flow of control.~~
- ~~PL #5. Prolog.~~

Red: Track 1 — Syntax & semantics of programming languages

Blue: Track 2 — Programming-language features & categories, and specific programming languages

We are here.

We lost a week of classes, so we will not have time to cover these.

PL Category: Concatenative PLs

PL Category: Concatenative PLs
Background [1/4]

A **concatenative** programming language is one in which the concatenation of two programs is a valid program, with the data returned by the first part being passed to the second part.

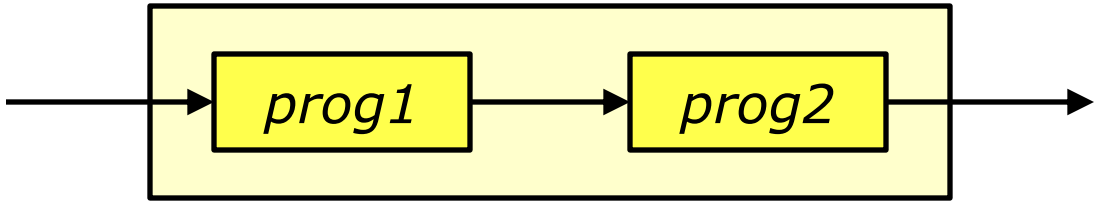
The first major concatenative PL was **Forth**, developed by Charles H. Moore in the 1960s.

PL Category: Concatenative PLs
Background [2/4]

Here is a diagram of a program *prog1* with arrows representing the data it takes and returns.

Another program *prog2*:

Put the two together to make a new program:



From a more theoretical point of view, we say that concatenative PLs build up programs using **composition** of functions.

PL Category: Concatenative PLs
Background [3/4]

As a more practical example, here is a Forth program.

- Takes: nothing, returns: the integer 5.

```
5
```

Here is another Forth program.

- Takes: an integer, returns: nothing, side effect: prints the integer.

```
.
```

Put the two programs together to get a third Forth program.

- Takes: nothing, returns: nothing, side effect: prints "5".

```
5 .
```

PL Category: Concatenative PLs
Background [4/4]

Concatenative programming languages have not formed a very fruitful branch of the PL family tree. However, there are some in common use.

The concatenative PL that is most heavily used today is probably **PostScript**, from Adobe Systems. While considered mostly as a way of specifying documents to be printed, PostScript is actually a full-featured programming language.

A concatenative PL that I think deserves more attention is **Factor**. This is a dynamic PL that grew out of a scripting language for games, beginning in 2003.

PL Category: Concatenative PLs
Typical Characteristics

A typical concatenative programming language has the following characteristics.

- It is **stack-based**: stacks are the primary means of passing around data. Values passed to a program—including function parameters—are placed on a stack. Return values are left on the same stack.
- It has a very simple syntax. Usually a program consists of a sequence of **words**, separated by whitespace.
- It blurs or eliminates the distinctions between *statements*, *blocks*, and *programs*.
  - See the Forth code a couple of slides back.
- It is **extensible**: little or no distinction is made between built-in constructs and those put together by a programmer using the PL.
  - Compare C++ or Lua.
- It is *not* statically typed. It may be dynamically typed, or it may have no type checking.
  - I speak of a *typical* concatenative PL. There are statically typed concatenative PLs.

Introduction to Forth
History [1/2]

The next programming language we will look at is **Forth**. (The name apparently began as an abbreviation of "Fourth", due to a file system with a five-character limit on filenames.) Forth was invented in the 1960s by Charles H. Moore, for his own use. The PL first saw significant use at the U.S. National Radio Astronomy Observatory.

A Forth interpreter can be very small. So Forth is easy to port—or to implement from scratch. As a result, essentially every computer platform produced since 1970 has had a Forth implementation available. But through much of the history of Forth, no two of these implementations were compatible.

For many platforms, a Forth interpreter was the first nontrivial program ever executed on the platform.

Introduction to Forth
History [2/2]

Forth became popular in the late 1970s and early 1980s, but its popularity has waned considerably since then. It remains a strong influence on some other PLs.

Forth was first standardized in 1983. In 1994, a Forth standard was issued by ANSI. Reportedly, a revised standard is in the works. A free implementation of the 1994 standard is available for most/all modern platforms from the GNU project: **Gforth**.

An endless number of dialects of Forth exist. We will follow the 1994 ANSI standard; from now on, “Forth” means ANSI standard Forth, as implemented in Gforth.

Characteristics — General

Forth is a **concatenative** programming language.

- Again, this means that the concatenation of two Forth programs is a valid Forth program, with the data returned by first part being passed to the second part.

Like most concatenative PLs, Forth is **stack-based**: values are passed and returned via a stack.

Like Java, C++, and Lua—and *unlike* Haskell—Forth is aimed at **imperative** programming: programs consist largely of instructions that tell a computer what to do.

Characteristics — Words

Forth has an extremely simple syntax. Programs consist of sequences of **words**: strings of non-space characters, separated by whitespace. For the purposes of separating words, blanks and newlines are considered identical.

- Forth does have single-line comments, which end at newline.
- There is special syntax for string literals, which may contain blanks.

Forth words are **case-insensitive**: "`ab;c1`", "`AB;C1`", and "`Ab;c1`" name the same word.

- In contrast, Java, C++, Lua, and Haskell are all **case-sensitive**. "`ab`" and "`aB`" are considered distinct identifiers in all of these PLs.

Forth syntax generally does not distinguish between numeric literals, variables, functions, and flow-of-control constructs. There are only words.

Forth is an **extensible** programming language. New functionality has equal status with previously defined functionality.

In particular, Forth allows programmers to:

- Create new flow-of-control constructs.
- Create new words for defining words.
- Redefine standard Forth words.

This extensibility has already been used to create the basic set of words available when Forth starts up. Many of the words we commonly use are not actually core Forth words; rather, they are written in Forth.

Extensibility is accomplished via an internal Forth data structure: the **dictionary**. This lists all defined words, in the order they were defined. Programmer-defined words simply come later in the dictionary than standard words.

Characteristics — Scope

Forth allows for **local** words to be defined within the definition of a word. Such local words have static, lexical scope; they cannot be used outside the word definition.

Other (global) Forth words are **dynamically scoped**. They may be used at any *time* after their definition.

When a global Forth word is called, a backwards search is done in the dictionary, so the latest version is used.

When a word is *compiled* (the usual thing to do when defining a word), references to the dictionary entries for any words it uses are maintained in the compiled code. Thus, if these words are redefined, then the compiled word still does the same thing.

Introduction to Forth
Characteristics — Dynamic?

Forth has a number of characteristics that make it feel a bit like a dynamic programming language.

- New words may be defined at runtime.
- Existing words may be redefined at runtime.
- There are few distinctions between the runtime of a compiled program and the interactive environment.

However, I do not call Forth a dynamic PL, primarily due to its type system. *See the next slide …*

Characteristics — Type System

Forth supports a very limited set of types.

The majority of Forth operations involve machine integers—like C/C++ `int`. These are used as all of the following.

- Numbers
- Boolean values (0: false, nonzero: true)
- Pointers
- Characters

There is also support for floating-point values and operations.

Forth does *not* have an extensible type system (like Lua, and unlike Java and C++).

However, although Forth arguably has a notion of *type*, it has **no type checking**. Instead, different types are passed & returned via different stacks and handled via different constructions.

Introduction to Forth
Build & Execution [1/2]

Forth source files end with the suffix "`.fs`".

As with the other PLs we have looked at, Forth supports both compilation to an executable file and an interactive environment. We will use the latter exclusively.

Starting Gforth runs the interactive environment. In the versions I have experience with, there is no prompt. Simply type words and hit <Enter>. When execution completes successfully, "`ok`" will be printed, and you may enter more words.

Introduction to Forth
Build & Execution [2/2]

To load a source file in the interactive environment, type the word `include`, and then the filename of the Forth source file.

Example:

```
include myprog.fs
```

Or, if you are using a GUI that has—say—a menu item for loading files, then you may use that instead.

Once a file is loaded, any words it defines are available for use.

Some Programming

I have written a simple example Forth program that computes Fibonacci numbers.

See `fibo.fs`.

TO DO

- As time permits, run `fibo.fs` and other Forth code.