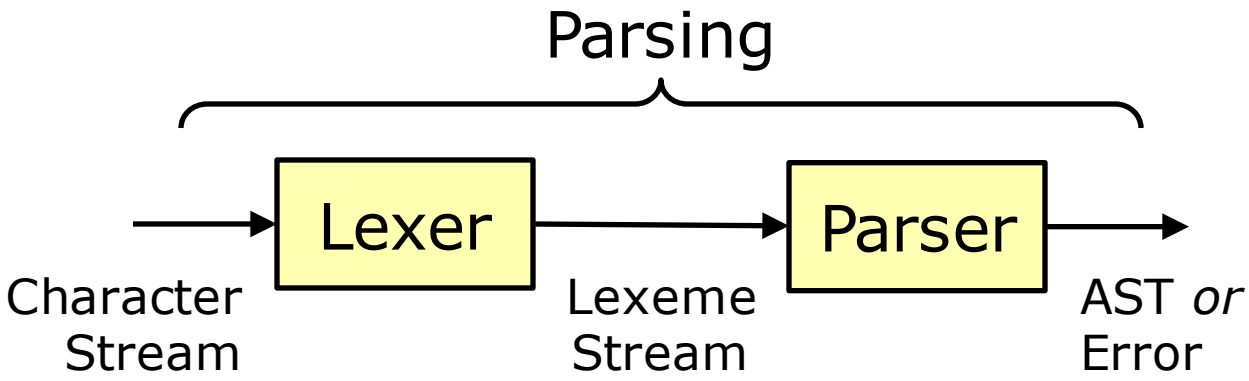
PL Category: Functional PLs
Introduction to Haskell

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, February 21, 2020

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

Review

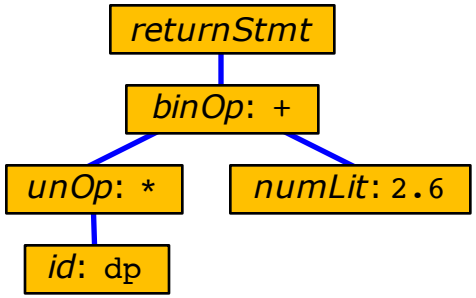
Review
Overview of Lexing & Parsing




```
return (*dp + 2.6); //x
```

```
return (*dp + 2.6); //x
```

key, op, id, op, num lit, punct, punct



Two phases:

- **Lexical analysis** (**lexing**)
- **Syntax analysis** (**parsing**)

The output of a parser is typically an **abstract syntax tree** (**AST**). Specifications of these will vary.

The Basics of Syntax Analysis — Categories of Parsers

Parsing methods can be divided into two broad categories.

Top-Down Parsers

- Go through derivation top to bottom, **expanding** nonterminals.
- Important subcategory: **LL parsers** (read input **L**eft-to-right, produce **L**eftmost derivation).
- Often hand-coded—but not always.
- Method we look at: **Predictive Recursive Descent**.

Bottom-Up Parsers

- Go through the derivation bottom to top, **reducing** substrings to nonterminals.
- Important subcategory: **LR parsers** (read input **L**eft-to-right, produce **R**ightmost derivation).
- Almost always automatically generated.
- Method we look at: **Shift-Reduce**.

Parsing Wrap-Up — Efficiency of Parsing

Counting the input size as the number of lexemes/characters:

Practical parsers (and lexers) run in linear time.

This includes state-machine lexers, Predictive Recursive-Descent parsers and Shift-Reduce parsers.

Various parsing methods are known that can handle all CFLs; these generally run in cubic time.

Among these is **Generalized LR** (**GLR**). This is of particular interest because

- it runs faster than cubic time for *some* grammars, and
- it easily handles some situations that are difficult for other methods.

GLR is thus considered a useful, practical parsing method—but only for those grammars for which it is efficient.

Parsing Wrap-Up — Parsing in Practice

In my experience, the parser used in a production compiler is generally one of the following two kinds.

- A hand-coded top-down parser using Recursive Descent—or something similar.
- An automatically generated bottom-up parser using a table-based Shift-Reduce method—or something similar (usually LALR or GLR).

Producing a parser is a very practical skill, because:

***Parsing* is making sense of input.**

And that is something that computer programs need to do *a lot*.

Review
PL Feature: Type System

The following type-related terms will be particularly important in the upcoming material.

- **Type**
- **Static typing**
- **Manifest typing** vs. **implicit typing**
- **Type inference**
- **Type annotation**
- **First-class functions**
- **Sound** type system

See *A Primer on Type Systems* for full information on this topic.

PL Category: Functional PLs

PL Category: Functional PLs
Background [1/4]

For most of us, our programming experience has largely involved **imperative programming**: writing code to tell a computer *what to do*. Running a program is thus saying to a computer, "Follow these instructions."

This is the predominant paradigm in PLs like Java, C++, and Lua.

An alternative is **declarative programming**: writing code to tell a computer *what is true*. Running a program might be thought of in terms of asking a question.

The most important declarative programming style is **functional programming**.

Later in the semester, we will look at **logic programming**, another declarative style.

PL Category: Functional PLs
Background [2/4]

Functional programming (**FP**) is a programming style that generally has the following characteristics.

- Computation is considered primarily in terms of the *evaluation* of functions—as opposed to *execution* of tasks.
- Functions are a primary concern. Rather than mere repositories for code, functions are values to be constructed and manipulated.
- Side effects* are avoided. A function's only job is to return a value.

*A function has a **side effect** when it makes a change, other than returning a value, that is visible outside the function.

PL Category: Functional PLs
Background [3/4]

One can do functional programming, in some sense, in just about any PL. However, some PLs support it better than others.

A **functional programming language** is a PL designed to support FP well. This is thus a somewhat vague term.

- No one calls C a functional PL.
- Opinions vary about JavaScript.
- Everyone agrees that Haskell is a functional PL.

PLs generally agreed to be functional include Haskell, the ML family (ML, OCaml, F#), R, and Miranda.

In addition, the Lisp family of PLs (Common Lisp, Emacs Lisp, Scheme, Clojure, Racket, Logo, Arc) offers excellent support for FP, but is usually considered as a separate category.

PL Category: Functional PLs
Background [4/4]

Functional programming and functional PLs have been around for many decades, but they remained largely the province of academia until two things happened.

- In the 1990s a solution was found to the problem of how to do interactive I/O in a context where side effects were not allowed.
- Around 2000, serious attention started to be given to the practical issues of algorithmic efficiency and resource usage in a functional context.

Today, FP is becoming increasingly mainstream. Functional PLs are being used for large projects. Constructs inspired by FP are being introduced into many PLs.

For example, lambda functions became part of C++ in the 2011 Standard.

PL Category: Functional PLs
Typical Characteristics

A typical functional programming language has the following characteristics.

- It has first-class functions.
- It offers good support for higher-order functions*.
- It offers good support for recursion.
- It has a preference for immutable** data.

A **pure** functional PL goes further, and does not support mutable data at all. There are no side effects in a pure functional PL.

*A **higher-order function** is a function that acts on functions.

A value is **mutable if it can be changed. Otherwise, it is **immutable**—like `const` values in C++.

Introduction to Haskell

Introduction to Haskell
History [1/4]

In the mid-20th century, any number of functional PLs were created, often as part of academic research projects in computer science or mathematics. Most saw very little use. *None* saw widespread use in mainstream programming.

In 1987, members of the FP community met at a conference in Portland, Oregon. Feeling that their field was too fragmented, they decided to pool their efforts. A committee was formed, led by Simon Peyton Jones of Microsoft Research, to create a PL that could form a stable platform for research, development, and the promotion of functional programming.

They named this programming language **Haskell**, after logician Haskell B. Curry (1900–1982).

History [2/4]

The initial release of Haskell came in 1990.

In the 1990s, the problem of how to do interactive I/O in a pure functional context was solved, allowing Haskell and FP to enter the programming mainstream.

Various language definitions in the 1990s culminated in a long-term standard in 1998: **Haskell 98**.

The 1998 standard had two primary implementations:

- **Hugs**, a lightweight interactive environment supported on all major platforms.
- The **Glorious Glasgow Haskell Compiler** (**GHC**), a full-featured compiler.

Hugs has since been folded into GHC; the interactive environment is now called **GHCi**.

History [3/4]

2009 saw the release of the **Haskell Platform**, a collection of libraries and tools with the goal of creating a high-quality, "batteries-included" collection of packages that all Haskell developers could have in common. A new release of the Haskell Platform comes once or twice each year.

A second Haskell standard was published in 2010: **Haskell 2010**, a.k.a. **Haskell Prime**. The is currently the most recent standard document.

Introduction to Haskell
History [4/4]

Until recently, a third standard was in the works, with the goal of a 2020 release. However, work on this standard slowed around 2016 and has now stopped. There is currently no concrete plan for a new Haskell standardization document.

Thus, in practice, Haskell is now defined by its primary implementation: GHC. This fully supports the 2010 standard. It also includes a number of language extensions (over 100), which can be optionally enabled.

Haskell is now a robust, well supported PL, suitable for large projects. However, its unusual nature means that it still meets a fair amount of resistance from traditionally minded programmers.

Characteristics — Type System [1/5]

Haskell is a pure functional PL. It has first-class functions and excellent support for higher-order functions.

Haskell has a sound static type system with sophisticated type inference. So typing is largely inferred, and thus implicit; however, we are *allowed* to write type annotations, if we wish.

Haskell's type-checking standards are difficult to place on the nominal-structural axis.

Haskell has few implicit type conversions. Support for the definition of new implicit type conversions lies somewhere between minimal and nonexistent.

Characteristics — Type System [2/5]

Like C++ and Java, Haskell does static typing of both variables and values. Unlike C++ and Java, Haskell includes sophisticated type inference (based on the **Hindley-Milner** type-inference algorithm). Thus, types usually do not need to be specified.

In C++:

```
int n = 3;
```

In Haskell:

```
n = 3
```

Of course, in Lua we can say that, too. But in contrast to Lua, every variable gets a type in Haskell. For example, above, `n` has type `Integer`; the compiler figures this out for us.

Characteristics — Type System [3/5]

Haskell still allows type annotations, if desired. We can say:

```
n :: Integer
n = 3
```

This lets us communicate our intentions to be compiler. For example, the following is legal.

```
s = "abc"
```

But this will not compile:

```
s :: Integer
s = "abc"  -- Type error: "abc" is not of type Integer
```

Characteristics — Type System [4/5]

We can also use Haskell type annotations to restrict which types are allowed. Below is a function with its natural type annotation. If this annotation were omitted, the result would be the same.

```
blug :: (Num t, Eq t) => t -> t -> Bool
blug a b = (a == b+1)
```

The above says that `blug` is a function that takes two values of type `t`, where `t` is any numeric type with the equality operator defined. And `blug` returns a `Boolean`.

But if we want `blug` to take only `Integer` values, we can do this:

```
blug :: Integer -> Integer -> Bool
blug a b = (a == b+1)
```

Characteristics — Type System [5/5]

Haskell's type system is extensible. We can create new types. We can also overload functions and operators to use them.

However, unlike many modern PLs, such extensibility is *not* facilitated via constructs that support object-oriented programming.

Arguably, Haskell has no need for OOP constructions. Problems that are solved using OOP in some PLs can be solved by other means in Haskell (closures, for example).

Characteristics — Flow of Control [1/2]

Iteration is difficult without mutable data. And, indeed, Haskell has no iterative flow-of-control constructs. To be perfectly clear: *Haskell has no loops!*

Instead of iteration, Haskell uses recursion, with tail recursion preferred. The latter will generally be optimized using **tail call optimization** (**TCO**).

Haskell implementations are required to do TCO. This means the last operation in a function is not implemented via a function call, but rather as the equivalent of a goto, never returning to the original function.

However, we often do not make recursive calls explicitly. Instead, we use functions that encapsulate recursive execution.

Characteristics — Flow of Control [2/2]

Haskell has an if … else construction, but it is often more convenient to use **pattern matching**.

Example: a recursive factorial function in C++ and Haskell.

```
int factorial(int n)  // C++
{
    if (n == 0) return 1;
    return n * factorial(n-1);
}
```

Patterns

```
factorial 0 = 1          -- Haskell
factorial n = n * factorial (n-1)
```

Characteristics — Syntax [1/2]

Haskell has a simple syntax, with less punctuation than C++—and even less than Lua.

Here are function calls in C++ and Lua.

```
foo(a, b, c);  // C++
foo(a, b, c)   -- Lua
```

Here is a more or less equivalent function call in Haskell.

```
foo a b c      -- Haskell
```

Characteristics — Syntax [2/2]

Haskell has **significant indentation**. Indenting is the usual way to indicate the start & end of a block. Here is Lua.

```
function bar(a)
    local b = 42           // Indented, but only for
    local c = 30 * b + 1   //  readability; the compiler
    return foo(a, b, c)    //  ignores indentation.
end
```

And here is the more or less equivalent Haskell.

```
bar a = foo a b c where
    b = 42
    c = 30 * b + 1         -- We MUST indent this line.
```

Characteristics — Evaluation [1/2]

By default Haskell does **lazy evaluation**: expressions are not evaluated until they need to be.

C++, Java, and Lua do the opposite, evaluating as soon as an expression is encountered; this is **eager evaluation**.

For example, here is a function in Lua, and then in Haskell.

```
function f(x, y)
    return x+1    -- y is not used
end

f x y = x+1       -- y is not used
```

We look at what eager vs. lazy evaluation means for these.

Introduction to Haskell
Characteristics — Evaluation [2/2]

```
function f(x, y)
    return x+1    -- y is not used
end
```

Lua (eager). Do "`f(g(1), g(2))`". Function `g` is called with `1`. Then `g` is called with `2`. The return values are passed to `f`.

```
f x y = x+1       -- y is not used
```

Haskell (lazy). Do "`f (g 1) (g 2)`". Function `f` is called; this uses its first parameter (`x`), so `g` is called with argument `1`, and its return value becomes `x`. Then `f` adds `1` to this and returns the result. The second call to `g` is never made.

If the return value of `f` is not used, then *no* calls to `g` are made!

Lazy evaluation has other interesting consequences, as we will see.

Introduction to Haskell
Build & Execution [1/3]

The standard filename suffix for Haskell source files is "`.hs`".

GHC is a Haskell compiler that usually generates native machine code. On the command line, GHC is used much like g++, clang, or any other command-line compiler.

```
ghc myprog.hs -o myprog
```

If there are no errors, then an executable named `myprog` will be created. Running that file will execute function `main` in module `Main` (a **module** in Haskell is much like a module in Lua).

Of course, if you are using an IDE, then things are handled differently. GHC is supported by various IDEs, including Eclipse.

Build & Execution [2/3]

GHCi is an interactive environment that interprets Haskell code. Such an environment is often called a **Read-Eval-Print Loop** (**REPL**), a term originating with Lisp.

GHCi can load source files or evaluate entered Haskell expressions. Haskell is compiled to a bytecode, which is interpreted.

After running GHCi, you are presented with a prompt. GHCi commands begin with colon (`:`). Some important commands:

`:l` *FILENAME*`.hs`

Load & compile the given source file. Afterward, calls to functions in the file may be typed in at the prompt.

`:r`

Reload the last file loaded. Useful if you change a file.

Continued …

Introduction to Haskell
Build & Execution [3/3]

Continuing with GHCi commands:

`:t` *EXPRESSION*

Get the type of a Haskell expression. The expression can involve variables and functions defined in a file that has been loaded.

`:i` *IDENTIFIER*

Get information about the identifier: its type; precedence and associativity if it is an operator; perhaps the file it is defined in.

To evaluate a Haskell expression, enter the expression.

To define a Haskell variable or function, enter the definition.

```
n = 5
```

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

See `fibo.hs`.

TO DO

- As time permits, run `fibo.hs` and other Haskell code.