PL Category: Functional PLs
Introduction to Haskell

CS 331 Programming Languages

Lecture Slides

Friday, February 21, 2025

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

Unit Overview
Lexing & Parsing

Topics

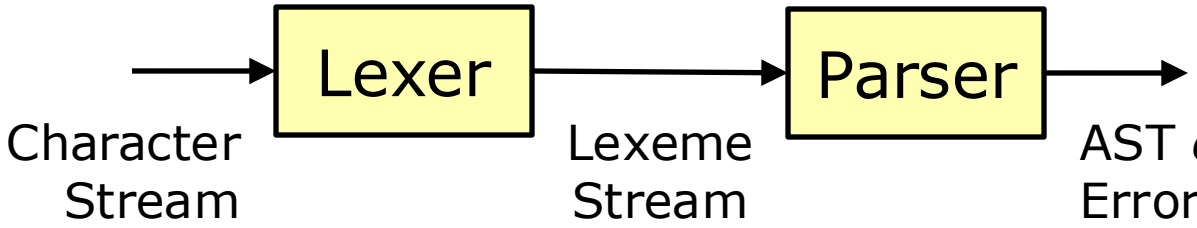
- ✓ Introduction to lexing & parsing
- ✓ The basics of lexical analysis
- ✓ State-machine lexing
- ✓ The basics of syntax analysis
- ✓ Recursive-descent parsing
- ✓ Shift-reduce parsing
- ✓ Parsing wrap-up

Lexical Analysis (Lexing)

Syntax Analysis (Parsing)

Review

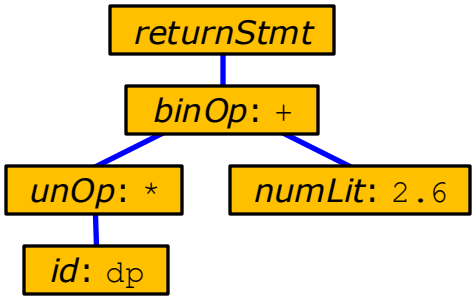
Review
Introduction to Lexing & Parsing




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

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

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



Two steps:

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

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

Parsing Wrap-Up — Efficiency of Parsing

To analyzing lexers & parsers, we can:

- Consider the input size (n) to be the number of characters, OR
- Consider the input size to be the number of lexemes, and count lexeme operations (read, copy, compare) as basic operations.

In either case:

Practical lexers and parsers run in linear time.

This includes State-Machine lexers, Predictive Recursive-Descent parsers, Shift-Reduce parsers, and others used in practice.

A practical method is generally only able to handle a restricted class of CFLs. Various parsing methods are known that can handle all CFLs; these mostly have a worst case of $\Theta(n^3)$.

Parsing Wrap-Up — Parsing in Practice [1/2]

In recent years, the world of parsing has been branching out, with methods that previously saw little use gaining traction.

In particular, a **Generalized LR** (**GLR**) parser runs something like a Shift-Reduce automaton. Roughly speaking, it allows multiple actions for a single state-input combination, and tries them all.

GLR can handle all CFLs. It is cubic-time for many grammars. However:

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

For these reasons GLR is seeing increasing use.

Parsing Wrap-Up — Parsing in Practice [2/2]

The lexing & parsing methods we have covered are all appropriate for use in production code—but Lua is generally not the best PL to write them in.

It appears to me that production parsers today are mostly either Recursive-Descent parsers or automatically generated bottom-up parsers (LALR, 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*.

Knowing how to produce a parser can be a useful addition to your personal toolbox.

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

Review
PL Feature: Type System [1/2]

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

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

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

PL Feature: Type System [2/2]

A PL or PL construct is **type-safe** if it forbids operations that are incorrect for the types on which they operate.

Two unfortunate terms are often used in discussions of type safety: **strong typing** (or **strongly typed**) and **weak typing** (or **weakly typed**). But these terms have no standard definitions. They are used in different ways by different people.

Therefore:

Avoid using the terms "strong" and "weak" typing, or "strongly typed" and "weakly typed".

PL Category: Functional PLs

PL Category: Functional PLs
Background [1/5]

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 common declarative programming style is **functional programming**.

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

PL Category: Functional PLs
Background [2/5]

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

When we talk about a side effect, we are *not* suggesting that the change made is accidental or undesirable.

In C++, adding `int`s has no side effect.

```
int aa, bb;
int n = aa + bb;  // operator+ has no side effect
```

On the other hand, the `+=` operator does have a side effect.

```
n += bb;          // operator+= side effect: changing n
```

Why are we talking about side effects? *See the next slide.*

PL Category: Functional PLs
Background [3/5]

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

- Side effects are avoided. Once a variable is set, its value is generally not changed. A function's only job is to return a value.
- Computation is considered primarily in terms of the *evaluation* of functions—as opposed to *execution* of tasks.
- Functions are of primary interest. Rather than mere repositories for code, functions are values to be constructed and manipulated.

PL Category: Functional PLs
Background [4/5]

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 PLs 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 often considered as a separate category.

PL Category: Functional PLs
Background [5/5]

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.

In recent decades, FP has become increasingly mainstream. Functional PLs are now being used for large projects.

Furthermore, constructs inspired by FP are being introduced into many PLs. For example, lambda functions became part of C++ in the 2011 Standard. And C++ got *fold expressions* in the 2017 Standard. (What are those? Look it up!).

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 [1/3]

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 formed a committee to create a single pure functional 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). (Why not "Curry"? Probably because that was already used for something else, as we will see.)

Introduction to Haskell
History [2/3]

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:

- The **Haskell User's Gofer System** (**Hugs**), a lightweight interactive environment.
- The **Glorious Glasgow Haskell Compilation System**, a.k.a. the **Glasgow Haskell Compiler** (**GHC**), a full-featured compiler.

Hugs was eventually folded into GHC. The interactive environment was renamed **GHCi**.

Introduction to Haskell
History [3/3]

A second Haskell standard was released in 2010: **Haskell 2010**, a.k.a. **Haskell Prime**. This is the most recent published standard.

Efforts to produce a third official standard have stalled. In practice, Haskell is now defined by its primary implementation: GHC. This supports Haskell Prime. It also includes optional language extensions—well over 100 of them.

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

Characteristics — Overall + Syntax [1/2]

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

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

Below are more or less equivalent function calls in C++, Lua, and Haskell.

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

Characteristics — Overall + Syntax [2/2]

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

Example: more or less equivalent functions in Lua and Haskell.

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

In Lua, the end of the block is marked with "`end`".

In Haskell, it is marked by a return to the previous indentation level.

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

Characteristics — Type System [1/5]

Haskell has a sound static type system with sophisticated type inference (based on the *Hindley-Milner* type-inference algorithm). So typing is largely 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 variables. Unlike C++ and Java, Haskell includes sophisticated type inference, so types usually do not need to be specified.

```
int n = 3;   // C++
n = 3        -- Haskell
```

The above Haskell code is legal in Lua, too. However "`n = 3`" is an executable statement in Lua, while in Haskell it is not. In Haskell it is something *true*, not something that *happens* at runtime.

Furthermore, in Lua a variable does not have a type. Only values have types; a variable is merely a reference to a value. But in Haskell every variable has a type. When it is not specified, the compiler can usually figure out this type. (Above, the type of `n` actually depends on context; it will typically be `Integer`.)

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, this following is legal.

```
s = "abc"
```

But the following will not compile:

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

Characteristics — Type System [4/5]

Haskell type annotations let us 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 :: (Eq a, Num a) => a -> a -> Bool
blug x y = (x == y+1)
```

The above says that `blug` is a function that takes two values of type `a`, where `a` 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 x y = (x == y+1)
```

Introduction to Haskell
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.

However, we often do not make recursive calls explicitly. Instead, we use functions that encapsulate recursive execution. *More on this at another time.*

When the recursion is tail recursion, it can be optimized using **tail call optimization** (**TCO**): 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.

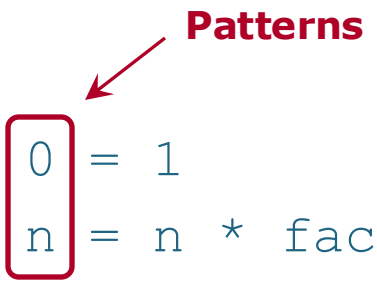
Haskell implementations are *required* to do TCO.

Characteristics — Flow of Control [2/2]

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

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

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




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

Characteristics — Evaluation [1/2]

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

In contrast, C++, Java, and Lua evaluate an expression as soon as it is encountered during execution; this is **eager evaluation**.

Here are functions in Lua and 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.

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 plug-ins for various IDEs, including Visual Studio, Xcode, and Eclipse.

Introduction to Haskell
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 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
```

Older versions of GHCi require "`let`" first:

```
let n = 5
```

Some Programming [1/2]

TO DO

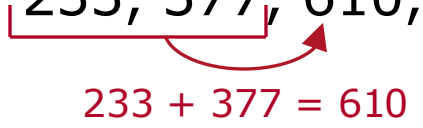
- Try out the Haskell interactive environment.
- Write a hello-world program in Haskell and execute it in various ways.

Done. See `hello.hs`.

Introduction to Haskell
Some Programming [2/2]

Recall the **Fibonacci numbers**:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …



I have written a Haskell program that computes and prints Fibonacci numbers: `fibo.hs`.

TO DO

- Run `fibo.hs`.

See `fibo.hs`.