

Lecture 1: Course overview, lexical analysis

David Hovemeyer

August 30, 2021

601.428/628 Compilers and Interpreters



Welcome!

- ▶ Welcome to Compilers and Interpreters!
- ▶ Today:
 - ▶ Policies and syllabus
 - ▶ Course overview
 - ▶ Lexical analysis

Where to find information

- ▶ All *public* course information will be posted on the course website:
<https://jhucompilers.github.io/fall2021>
- ▶ Please check the course website frequently!
- ▶ Q&A, course announcements, and non-public course information will be on Campuswire, <https://campuswire.com>
- ▶ Assignment submission using Gradescope,
<https://www.gradescope.com>
- ▶ Videos will be posted to Blackboard (my.jh.edu → Education → Blackboard)

- ▶ Slides and public materials posted on course website
- ▶ Class is taught in person in Hackerman B17
- ▶ There is a synchronous Zoom meeting for remote participation (see Campuswire for URL)
- ▶ Lecture recordings will be posted to Panopto (accessible using Blackboard)
- ▶ Use Campuswire for Q&A!
- ▶ Office hours: will likely be via Zoom (see Campuswire for URLs)

Syllabus

- ▶ Syllabus is posted on course website:

<https://jhucompilers.github.io/fall2021/syllabus.html>

- ▶ Please read it!
- ▶ This lecture just includes summary/highlights

Syllabus: communication policy

- ▶ Official course communication will use Piazza (check it regularly)
- ▶ Use Piazza for questions
 - ▶ If possible, make questions public (if they would benefit other students, and don't contain assignment code or personal information)
 - ▶ Please answer public questions if you can!
 - ▶ Otherwise, private questions are fine
 - ▶ We will make every effort to respond in a timely manner (usually the same day)
- ▶ Please check your email regularly
- ▶ Email me (daveho@cs.jhu.edu) if you have any concerns

Syllabus: academic ethics

- ▶ Follow the CS Academic Integrity Code:
<https://www.cs.jhu.edu/academic-integrity-code/>
- ▶ Assignment submissions and exams must be entirely your work!
Submitting someone else's work or allowing someone else to submit yours constitute a violation of academic ethics
- ▶ Cite all sources used
- ▶ If you aren't sure what is allowed and what isn't, ask!

Syllabus: grading

- ▶ Assignments: 60%
 - ▶ Series of projects to build interpreters and a compiler
- ▶ Exams: 40%
 - ▶ Three in-class exams, each worth $13.\bar{3}\%$ of course grade
 - ▶ First two exams during semester, third during scheduled final exam time

Course overview

Compilers and interpreters

- ▶ Compilers and interpreters are frequently-used strategies for implementing programming languages
- ▶ This course: practical techniques for implementing compilers and interpreters

What is a compiler?

- ▶ A compiler translates a program (or partial program) from a *source language* to a *target language*
- ▶ Source language is often a “high-level” language
 - ▶ E.g., C, C++
- ▶ Target language is often *assembly language* which can be translated into directly-executable *machine language*
 - ▶ E.g., x86-64 assembly language

What is an interpreter?

- ▶ An interpreter analyzes a source language program and carries out the computation it embodies
- ▶ The source program is represented as a data structure
 - ▶ Represents the program in “ready-to-execute” form
- ▶ The interpreter *evaluates* this data structure

Compilers vs. interpreters

- ▶ Compilation and interpretation are both useful ways to implement a programming language
- ▶ The “front-end” of a programming language implementation — the components which recognize and analyze the source program — are similar in both interpreters and compilers
- ▶ Interpreters tend to be less effort to implement
- ▶ Compilers tend to allow the program to execute at closer to machine-level performance
- ▶ Hybrid strategies such as virtual machines and just-in-time compilation are possible

Rough course outline

- ▶ Lexical analysis: recognizing the lexical units (“words”) of a source program
- ▶ Parsing: recognizing the syntax (constructs) of a source program
- ▶ High-level intermediate representations: parse trees and abstract syntax trees
- ▶ Interpretation
- ▶ Semantic analysis and type checking
- ▶ Lower-level intermediate representations (e.g., control-flow graphs)
- ▶ Code generation
- ▶ Code optimization

Why is this course useful?

- ▶ Gain a deeper understanding of how programming languages are implemented
 - ▶ Know how the tools you are using work “under the hood”
- ▶ Compilation techniques can be used to create interesting tools for software engineering (static analyzers, instrumentation tools)
- ▶ Lexical analysis and parsing techniques can be applied to all kinds of structured data, not just source code

Lexical analysis

Lexical analysis

- ▶ Source code is generally represented as text: in other words, a sequence of characters
- ▶ *Lexical analysis* (also known as *scanning*) refers to the task of grouping sequences of input characters into *lexical units*, also known as *tokens*
- ▶ Tokens are the “words” of a source program

Hello, world

```
#include <stdio.h>

int main(void) {

    printf("Hello, world\n");

    return 0;

}
```

Hello, world (lexical structure)

```
#include <stdio.h>
```

preprocessor

```
int main(void) {
```

int identifier void lbrace

```
printf("Hello, world\n");
```

identifier lparen string literal rparen semicolon

```
return 0;
```

return int lit. semicolon

```
}
```

rbrace

What is a token?

- ▶ Token kind: value (usually integer or enumerated) representing what kind of token it is
- ▶ Lexeme: the exact text of the token in the source code
 - ▶ Some kinds of token can only ever have one lexeme (keywords, punctuation, etc.)
 - ▶ Some kinds of token can have a variety of lexemes (identifiers, literal values)

Source information

It is also useful to represent where in the source code the token occurred:

- ▶ Source file
- ▶ Line number
- ▶ Column number

Keeping track of this information helps the compiler or interpreter generate useful error messages

Example token representation

```
enum TokenKind {  
    TOK_INT_KEYWORD,  
    TOK_RETURN_KEYWORD,  
    TOK_IDENTIFIER,  
    TOK_LPAREN,  
    // etc. for other kinds of tokens  
};
```

```
struct Token {  
    enum TokenKind kind;  
    char *lexeme;  
    const char *filename;  
    int row, col;  
}
```

Lexical analyzer design

- ▶ The job of a *lexical analyzer* is to break down source code text into a sequence of tokens
- ▶ Typical approach: lexical analyzer produces one token at a time, on demand
 - ▶ The *parser* will consume the scanned tokens, more about this soon...

Interfaces and implementations

Interfaces and implementations

- ▶ Compilers and interpreters are complex software artifacts
- ▶ To manage the complexity, we need to design and implement them in a modular fashion
- ▶ One approach: *interfaces* and *implementations* using *opaque data types*
- ▶ By following this approach, we help ensure that modules can be modified independently of each other

Interfaces

An interface defines an *opaque data type*, and declarations of functions to

- ▶ create instances
- ▶ destroy instances
- ▶ do operations on instances

Opaque data types in C

When a struct data type is defined as a *forward reference* (without a full definition), code *using* the type

- ▶ Can declare pointers and references to instances, and pass them to functions
 - ▶ This means that instances must be dynamically allocated
- ▶ *Cannot* access fields directly → encapsulation is enforced

Example interface: tree node data type

```
// node.h  
  
struct Node;  // forward declaration  
  
struct Node *node_alloc(int tag);    // create a Node instance  
void node_destroy(struct Node *n);   // destroy a Node instance  
  
// operations/accessors  
int node_get_tag(struct Node *n);  
int node_get_num_kids(struct Node *n);  
void node_add_kid(struct Node *n, struct Node *kid);  
struct Node *node_get_kid(struct Node *n, int index);  
// etc...
```

Implementation of an interface

An implementation of an interface is simply a source module where

- ▶ The opaque data type is defined concretely
- ▶ The functions associated with the data type are defined

The implementation module is the only module in which the data type is not opaque!

Example implementation: tree node data type

```
// node.c
#include "node.h"

struct Node {
    int tag, num_kids, capacity,
    struct Node **kids;
    // ...other fields...
};

struct Node *node_alloc(int tag) {
    struct Node *n = malloc(sizeof(struct Node));
    n->tag = tag;
    n->num_kids = 0;
    n->capacity = 1;
    n->kids = malloc(sizeof(struct Node *) * n->capacity);
    return n;
}

// ...implementations of other functions...
```

Using C++ data types

- ▶ Ultimately, it's up to you to decide how to approach design and implementation
- ▶ Using C++ classes with private data members is also a perfectly valid way to enforce encapsulation
 - ▶ C++ also offers *huge* conveniences in string representation, vectors, maps, etc.
 - ▶ Having taught this class once before, I'd recommend just writing clean C++ code

Lexical analyzer design and implementation

A prefix calculator language

- ▶ A *prefix expression* is one where operators precede their operand(s)
- ▶ Example: `+ - 4 1 5` means $(4 - 1) + 5$
- ▶ The “prefix calculator language” accepts inputs which are a series of prefix expressions, each terminated by a semicolon (;)
 - ▶ Primary expressions: literal integers and identifiers
 - ▶ Numeric operators: `+ - * /`
 - ▶ Assignment: `=` (first operand must be an identifier)
 - ▶ Result of evaluation is the result of the last expression
- ▶ Code: <https://github.com/daveho/pfxcalc>

Running the prefix calculator

```
$ ./pfxcalc  
= a 1;  
= b 3;  
* + a b 6;  
Result: 24
```

Lexical analyzer (“lexer”) interface

```
// lexer.h
#include <stdio.h>
#include "node.h"

struct Lexer;

struct Lexer *lexer_create(FILE *in, const char *filename);
void lexer_destroy(struct Lexer *lexer);

struct Node *lexer_next(struct Lexer *lexer);
struct Node *lexer_peek(struct Lexer *lexer);
```

Lexer operations

- ▶ `lexer_next` consumes one token from the input (calling `lexer_next` repeatedly will consume all tokens in the input)
- ▶ `lexer_peek` returns the next token, *without* consuming it
 - ▶ Parsers will use this function for *lookahead*
- ▶ Note that tokens are represented using the struct `Node` data type
 - ▶ This is useful for building parse trees, more about this soon

Token kinds

```
// token.h
enum TokenKind {
    TOK_IDENTIFIER,
    TOK_INTEGER_LITERAL,
    TOK_PLUS,
    TOK_MINUS,
    TOK_TIMES,
    TOK_DIVIDE,
    TOK_ASSIGN,
    TOK_SEMICOLON,
};
```

These will be used as the “tag” values for the struct Node instances representing tokens

Lexer implementation

```
// lexer.cpp
#include <string>
#include "token.h"
#include "lexer.h"

struct Lexer {
private:
    // ... private fields ...

public:
    Lexer(FILE *in, const std::string &filename);
    ~Lexer();
    struct Node *next();
    struct Node *peek();

private:
    // ... private member functions...
};
```

Module implementation in C++

- ▶ Note that `struct Lexer` is implemented as a C++ type with constructor, destructor, member functions, etc.
- ▶ C code can use instances of this type by calling the functions defined by the interface
- ▶ This follows the “Interfaces and Implementations” approach using opaque data types
- ▶ But, as I mentioned, just using C++ types directly is fine as long as data is private

Lexer implementation (functions)

```
struct Lexer *lexer_create(FILE *in, const char *filename) {  
    return new Lexer(in, filename);  
}
```

```
void lexer_destroy(struct Lexer *lexer) {  
    delete lexer;  
}
```

```
struct Node *lexer_next(struct Lexer *lexer) {  
    return lexer->next();  
}
```

```
struct Node *lexer_peek(struct Lexer *lexer) {  
    return lexer->peek();  
}
```


Function implementation

- ▶ The interface functions (`lexer_create`, `lexer_next`, etc.) are implemented in C++ and can call member functions of the struct `Lexer` data type
- ▶ Because these functions have `extern "C"` linkage, they can be called from C code

How does the lexer actually work?

Let's look at the next and peek member functions:

```
struct Node *Lexer::next() {  
    fill();  
    Node *tok = m_next;  
    m_next = nullptr;  
    return tok;  
}
```

```
struct Node *Lexer::peek() {  
    fill();  
    return m_next;  
}
```

- ▶ fill is a private member function that calls the read_token private member function if a token object is not available
- ▶ m_next is a pointer to the available token object
- ▶ next and peek are similar; the main difference is that next sets m_next to null

fill function

```
void Lexer::fill() {  
    if (!m_eof && !m_next) {  
        m_next = read_token();  
    }  
}
```

- ▶ `m_eof` is a boolean member variable that is set to `true` when end of file is reached
- ▶ The `read_token` private member function does the actual work of reading a token

read_token function

```
struct Node *Lexer::read_token() {  
    // ... lots of code, read it yourself on Gitub ...  
}
```

Ad-hoc lexical analysis

Basic idea for implementing ad-hoc lexical analysis (as in the prefix calculator's `read_token` member function):

- ▶ Skip whitespace (if any)
- ▶ Read a character; if EOF is reached, then there are no more tokens
- ▶ Based on what character is read, start scanning a particular kind of token
 - ▶ E.g., if an alphabetic character was read, scan an identifier
- ▶ Keep reading characters that are a valid continuation of the current lexeme
- ▶ When a character that isn't a valid continuation is read, or EOF is reached, create the token object

Disadvantages of ad-hoc lexical analysis

- ▶ Ad-hoc lexical analyzers are somewhat tedious to implement
- ▶ Would be nice to have a declarative way to do lexical analysis:
 - ▶ Specify regular expression patterns for each kind of token
 - ▶ Have a tool generate a custom lexical analyzer from this specification
- ▶ Good news: *lexical analyzer generators* exist, and this is precisely what they do!
 - ▶ We will cover these soon

Next time

Next time we will discuss grammars and parsing techniques