Lecture 1: Course overview, lexical analysis

David Hovemeyer

August 26, 2024

601.428/628 Compilers and Interpreters

Welcome!

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

Logistics

- All *public* course information will be posted on the course website: `https://jhucompilers.github.io/fall2024`
  - Please check the course website frequently!
- Class is taught in person in Shaffer 2
- Q&A, course announcements, and non-public course information will be on Courselore, `https://courselore.org`
- Assignment submission using Gradescope, `https://www.gradescope.com`
- Lecture recording videos will be posted to Canvas (details TBD)
- Office hours: some will be held via Zoom, some in person, see Courselore for details

Development environment

- The target platform for all programming assignments is x86-64 Linux
  - Autograders will use Ubuntu 22.04
- You will need an x86-64 Linux development environment
- Easy option: use one of the ugrad or grad machines maintained by the CS department
  - If you are a CS or CE student, you are entitled to an account (if you don't already have one)
  - If you need a temporary account, let me know
- WSL2 in Windows is also a good option
- We don't know of any way to set up a local development environment on a Mac

Syllabus

- Syllabus is posted on course website:

  `https://jhucompilers.github.io/fall2024/syllabus.html`
- Please read it!
- This lecture just includes summary/highlights

Syllabus: communication policy

- Official course communication will use Courselore (check it regularly)
- Use Courselore 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
  - Code generated by AI (ChatGPT, Github Copilot, etc.) is not considered original work, submitting it is 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 $13.\overline{3}\%$ of course grade
  - First two exams during semester, third during scheduled final exam time

Course overview

Compilers and interpreters

- Compilers and interpreters are 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

Implementation advice

Interpreters and compilers are complex!

- In this course you will be implementing realistic interpreters and compilers
- These will be fairly complex software artifacts!
- Managing the complexity will be crucial for success
- Here is some advice

Modularity is the key

- Modularity is the key to developing complex software systems
- Your programs should be a collection of data types and objects (instances of those data types) that work together
  - Make sure there is a clear separation of responsibility between each class
- Do not cut corners by violating encapsulation, adding unrelated responsibilities to the same class, etc.
- If any the complexity of any class seems to be getting out of hand, consider refactoring it

Use C++

- All of the starter code is in C++
- You could use a different language, but I don't recommend it
- C++ gives you some powerful tools that can make your job easier

Use STL containers

- Use the C++ standard library container classes to manage data
- `std::vector`: useful for sequences
- `std::map`: useful for dictionaries (such as environments and symbol tables!)
- Others could be useful too, e.g. `std::set`, `std::deque`

Use smart pointers

- Interpreters and compilers make heavy use of dynamic memory allocation for incremental data structures
  - E.g., tree nodes
- Smart pointers such as `std::shared_ptr` and `std::unique_ptr` can take care of much of the responsibility for ensuring that dynamically allocated objects are deleted
- This is especially helpful when exceptions could be thrown

Use exceptions for error reporting

If an error (syntax error, semantic error, runtime error, etc.) prevents the program from progressing normally, throw an exception.

This allows the error handling code to be separated from the rest of the program.

General idea:

```
int main() {
  try {
    parse_input();
    generate_code();
    return 0;
  } catch (Exception &ex) {
    fprintf(stderr, "Error: %s\n", ex.what());
    return 1;
  }
}
```

Use modern C++ features to simplify your code

- Modern C++ features can make your job easier
- Always use `auto` when possible to let the compiler determine the type of a variable
- Lambdas can be useful for creating a function on the fly to pass to an STL algorithm or other generic function

```
// example: loop over a container
for (auto i = nodes.begin(); i != nodes.end(); ++i) {
  Node *node = *i;
  // do something with the node
}

// example: recursively count nodes in a tree
int count = 0;
root->preorder([&count](Node *) { ++count; });
```

Lexical analysis

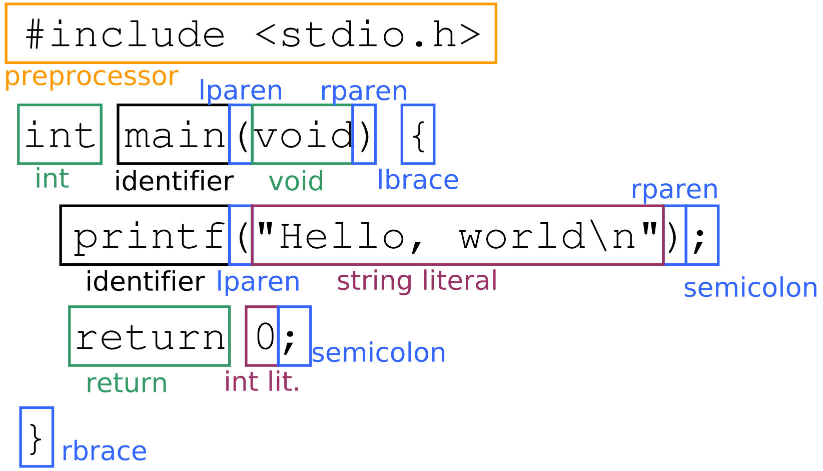
Lexical analysis

- 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>
int main(void) {
    printf("Hello, world\n");
    return 0;
}
```

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 occured:

- 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;
  std::string lexeme;
  std::string 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...

Lexical analyzer design and implementation

A prefix calculator language

- A *prefix expression* is one where operators precede their operand(s)
- Example: $\boxed{\texttt{+ - 4 1 5}}$ 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: $\boxed{\texttt{+ - * /}}$
  - Assignment: $\boxed{\texttt{=}}$ (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
```

Lexer class

```
// lexer.h
class Lexer {
private:
  // ...private fields...

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

  Node *next();
  Node *peek();

  Location get_current_loc() const;

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

Lexer operations

- `next()` consumes one token from the input (calling `next()` repeatedly will consume all tokens in the input)
- `peek()` returns the next token, *without* consuming it
  - Parsers will use this function for *lookahead*
- Note that tokens are represented using the `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

How does the lexer actually work?

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

```
Node *Lexer::next() {
  fill();
  if (m_next == nullptr)
    ...throw exception...
  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` throws an exception if at end of input, and also consumes the token

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

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

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 can be 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