1 Intro

- Class 1, 8/18/2020
- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Plagiarism — read aloud from handout 1
- Assignments on web. The first is very easy, the rest not, so start immediately.
- E-mail list: cs541001@cs.uky.edu; instructor uses to reach students.
- All students have MultiLab accounts, although you may use any computer you like to do assignments.
- Textbook — It is important that you read ahead.
- Undergraduates — Send me email; grading is 5% more lenient.

2 Overview of compilers: Chapter 1

- A compiler language is an example of a software tool.
• The compiler’s job.

![Diagram of the compilation process](image)

- **Compiler outputs**
  - Pure machine code: specific to a given architecture, no runtime linking. Example: Linux kernel.
  - Augmented machine code: specific to a given architecture and operating system. Example: C programs written for Linux, which may make OS calls.
  - Virtual machine code, interpreted or compiled on the fly during execution. Examples: Java (JVM), C# (.NET). Advantages: portability, code size **Our assignments use this output type.**

- **Output representations**
  - Assembler: good for cross-compilation; avoids having the compiler resolve all references. Modular compilation. **Our assignments use this output format.**
  - Relocatable binary: defers resolving external references. Modular compilation. Very common; used by Java and C.
  - Absolute binary: all references resolved.

3 The organization of a compiler

• [Figure 1.4 page 15](#)
• Scanner: reads the source program and constructs a stream of tokens, removing comments, and processing directives such as listing.

  • Example: `if (a < 39) {` is an input string of characters. The associated output tokens are `if:reserved, (,:symbol, a:identifier, <:operator, 39:integer, ):symbol, {,:symbol.`
  • The scanner can discover and report errors, such as `39f`.
  • We describe tokens by regular expressions.
  • We recognize tokens by using a deterministic finite automaton (DFA). That automaton is built for us by a scanner generator tool such as `lex`, `flex`, or `jflex`. Our assignments use `jflex`.

• Parser: reads the token stream and creates an abstract syntax tree (AST), verifying syntax and possibly repairing syntax errors.

  • Example: given the tokens above, the tree fragment would be:

```
            Statement
             if
               /   \
              /     \
            Expression  Statement
             <        Statement
               /   \
              /     \
          Identifier  Integer
            a       39
```

  • The parser can discover and report errors, such as `]` instead of `)` in the example.
  • We describe the syntax by a context-free grammar (CFG).
  • The table that drives the scanner is built for us by a parser generator tool such as `yacc`, `bison`, or `javaCUP`. Our assignments use `javaCUP`.

• Semantics checker: navigates through the AST and verifies that variables are declared, that types are used consistently, and that other semantic constraints (reachability, consistent use of exceptions) are met.

  • For instance, if `a` in the example is not of a numeric type, the type checker can report an error.
• It can also modify the AST, for instance, introducing type-conversion nodes, if, for instance, \( a \) is a short integer, in which case it might be converted to a regular integer.

• Code generator: navigates through the AST and generates either an intermediate representation (IR) or some other representation of executable code. **Our IR will be assembler for Java bytecode.**

• Optimizer: Analyzes the IR to improve the code. There are many forms of optimization, such as simplifying expressions, moving code, re-using values, eliminating trivial arithmetic, replacing sequences of instructions. **We will not cover optimization in this class.**

• Code generator: Maps the IR to target machine code. **Our assignments use Jasper to generate the target machine code: Java bytecode.**

### 4 Programming language considerations

• **Class 2, 8/20/2020**

• Successful designers of programming languages often have strong backgrounds in constructing compilers. If it can’t be compiled, it’s not very useful.

• Many features of modern languages require special care.
  
  • passing by name (obsolete since Algol 60; requires thunks)
  • dynamic-sized arrays (requires runtime type descriptors)
  • nested name scopes (require static chains)
  • anonymous functions, first-class functions (as in Python, requiring closures)
  • multiple-yield iterators (as in Python, require special stack manipulation)
  • automatic reclamation of object store (requires garbage collection).
5 Computer architecture considerations

- How many registers? What operations use them? How many register classes?
- Some operations can be very expensive: virtual method dispatch, dynamic heap access, reflective programming, exceptions, threads.
- The effect of memory architecture, such as paging and caches, is difficult to present to programmers but is significant.

6 Specialty compilers

- Debugging support, including participation in an integrated development environment (IDE).
- Highly optimizing compilers.
- Retargetable compilers.

7 The ac (adding calculator) language: Chapter 2

- This is a very simple language that lets us explore the components of a compiler.
- Components
  - Types: integer and float
  - Keywords: \texttt{f}, \texttt{i}, \texttt{p}
  - Variables: lowercase Roman single letters, excluding keywords
- Context-free grammar (CFG), expressed in Backus-Naur Form (BNF) [Figure 2.1 page 33]
8 The scanner

- Translates a stream of characters (as above) into a stream of tokens.
- A token has a type (such as operator or reserved) and a semantic value (such as plus or print).
- It’s a matter of choice whether each operator has its own type, in
which case there is no need for semantic values.

- Likewise, one can choose (1) reserved words each have their own type, or (2) they are of type reserved with a semantic value (their spelling), or (3) that they are of type id with a semantic value.

- **Class 3, 8/25/2020**

- Hard-coded example [Figure 2.5 page 40](#) uses peek() and advance()

```java
Token scanner(Stream<char> cs) throws LexicalException {
    while (isSpace(peek(cs))) advance(cs);
    if (eof(cs)) return (eof);
    if (isDigit(peek(cs))) return (scanDigits(cs));
    char c = advance(cs);
    switch (c) {
        case {'a'..'z'} - {'i', 'f', 'p'}:
            return (Token(id, c));
            break;
        case 'f': return (floatDecl); break;
        case 'i': return (intDecl); break;
        case 'p': return (print); break;
        case '=': return (assign); break;
        case '+': return (plus); break;
        case '-': return (minus); break;
        default: throw LexicalException;
    } // switch
} // scanner
```

```java
Token scanDigits(Stream<char>cs) {
    // the returned value is a string.
    Token answer = Token(inum, "");
    while (isDigit(peek(cs))) answer.value += advance(cs);
    if (peek(cs) != ".") return (answer);
    answer.type = fnum;
    answer.value += advance(cs);
    while (isDigit(peek(cs))) answer.value += advance(cs);
    return (answer);
} // scanDigits
```

- Production-quality scanners are constructed automatically from
regular expressions. We will discuss them in the next chapter.

- This parse requires that we specify the syntax of tokens.

### Table 2.3 page 36

<table>
<thead>
<tr>
<th>Language type</th>
<th>Formalism</th>
<th>Automaton</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>Regular expressions</td>
<td>Finite-state automaton (FSA)</td>
</tr>
<tr>
<td>Context-free</td>
<td>CFG (like BNF)</td>
<td>Push-down automaton (PDA)</td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>CSG</td>
<td>Linear-bounded automaton (LBA)</td>
</tr>
<tr>
<td>Type 0</td>
<td>various</td>
<td>Turing machine</td>
</tr>
</tbody>
</table>

### 9 Formal language hierarchy

### 10 The parser

- Translates a stream of tokens into an **abstract syntax tree (AST)**

- The simplest method is **recursive descent**. Each nonterminal has its own procedure. By looking ahead (using `peek()`), each procedure can decide which other procedures to call.

- Parsing statements in `ac`:
```java
void stmt(Stream<Token> ts) throws ParserException {
    if (peek(ts) == id) {
        match(ts, id);
        match(ts, assign);
        val();
        expr();
    } else if (peek(ts) == print) {
        match(ts, print);
        match(ts, id);
    } else {
        default: throw ParserException;
    } // stmt
}
```

- One needs to discover the **predict sets** for each alternative production that has the same left-hand side. For `Stmt`, the predict set for assignment is `{id}`.

- One needs to discover the **follow sets** for some productions that can derive `\lambda` in order to compute the predict set for their parent productions.