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.

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
      <
      Identifier a
      Integer 39
     Statement
   Statement
```
  • 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: $f$, $i$, $p$
  - 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

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>
<thead>
<tr>
<th>Terminal</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>floatdcl</td>
<td>&quot;f&quot;</td>
</tr>
<tr>
<td>intdcl</td>
<td>&quot;i&quot;</td>
</tr>
<tr>
<td>print</td>
<td>&quot;p&quot;</td>
</tr>
<tr>
<td>id</td>
<td>[a – e]</td>
</tr>
<tr>
<td>assign</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>plus</td>
<td>&quot;+&quot;</td>
</tr>
<tr>
<td>minus</td>
<td>&quot;-&quot;</td>
</tr>
<tr>
<td>inum</td>
<td>[0 – 9]^+</td>
</tr>
<tr>
<td>fnum</td>
<td>[0 – 9]+.([0 – 9]+</td>
</tr>
<tr>
<td>blank</td>
<td>(&quot; &quot;+)</td>
</tr>
</tbody>
</table>
```

### 9 Formal language hierarchy

<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>

### 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**: [Figure 2.7 page 42]
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.

• Given the grammar in Figure 2.1 page 33, notes p. 6, trace the parse of

\[ f \ b \ i \ a \ a = 5 \ b = a + 3.2 \ p \ b \]

11 What scanning and parsing cannot do

• enforce strong typing constraints

• disambiguate the meaning of some constructs, like \(x.y.z\) in Java, which might be package-class-field or variable-field-field or many other possibilities.

• determine the meaning of an overloaded operator.
12 Abstract syntax trees

- Instead of using the parse tree, we prefer an abstraction of the parse tree: the abstract syntax tree.
- It omits punctuation.
- Declarations store the identifier and its type in a single node.
- It represents the order of executable statements and expressions.
- Assignment nodes have two children: the identifier (the left-hand side) and the expression (the right-hand side).
- Binary operations have two children.
- The print statement is a single node that includes the name of the identifier to be printed.
- Compare the parse tree Figure 2.4 page 37, notes p. 6 with this AST (Figure 2.9 on page 44).

More appropriate in a Java implementation: the program has two children, Decls (declared to be a List of declarations) and Statements (declared to be a List of statements).
13 Semantic analysis

- Construct a symbol table for declarations and name scopes. In the case of ac, it can be very simple: an array indexed by ‘a’ . . . ‘z’. Each element has a type field, initialized to unknown.
- Enforce type consistency.
  - Walk the tree recursively, using visitor methods as shown in Figure 2.12 on page 49.
  - Insert to and query the symbol table as necessary.
  - Modify the type field to nodes as a declaration is visited.
  - Modify the AST to introduce type conversion (in our case, widening) nodes.
class Declaration {
    Id id; Type type;
    void check() {
        Symb symb = lookup(Id.name);
        if (symb != null) error("redeclaration");
        insert(Id.name, type); // put new symb in symbol table
    }
}

class Expr {
    Type type;
    abstract void check();
}

class Operation extends Expr {
    Expr op1, op2;
    void check() {
        op1.check();
        op2.check();
        if (op1.type == op2.type) {
            // no conversion
        } else if (op1.type == int) {
            op1 = new ToFloat(op1);
        } else {
            op2 = new ToFloat(op2);
        }
        type = op1.type;
    }
}

class Id extends Expr {
    char name;
    void check() {
        Symb symb = lookup(name);
        if (symb == null) error("undeclared\_variable");
        type = symb.type;
    }
}

14 Generating code

- Class 5, 9/1/2020
- In our case, the code is calculator buttons.
  - The calculator has registers; each is a single letter, such as a.
  - One can load or store a register with the L and S buttons.
  - One sets the precision with the K button.
  - One prints with the P button.
- We visit the AST recursively to generate code, invoking `codeGen()` at each node. [Figure 2.14 on page 52]
class Program {
    List<Declaration> Decls;
    List<Statement> Statements;
    void codeGen() {
        for (Statement statement : Statements) {
            statement.codeGen();
        }
    }
} // codeGen
} // Program

class Assign extends Statement {
    Id lhs; Expr rhs;
    void codeGen() {
        rhs.codeGen();
        emit("S"); // store
        emit(lhs.name);
        emit("0K"); // to integer mode
    } // codeGen
} // Assign

class Operation extends Expr {
    Expr op1, op2; char operation;
    void codeGen() {
        op1.codeGen();
        op2.codeGen();
        emit(operation);
    } // codeGen
} // Operation

class Id extends Expr {
    char name;
    void codeGen() {
        emit("L"); // load
        emit name;
    } // codeGen
} // Id

class Constant extends Expr {
    String value;
    void codeGen() {
        emit(value);
    } // codeGen
} // Constant

class ToFloat extends Operation {
    Expr operand;
    void codeGen() {
        operand.codeGen();
        emit("5k"); // 5 significant figures
    } // codeGen
} // ToFloat
15 **Overview of scanner: Chapter 3**

- This chapter introduces a formal, systematic approach to building scanners, instead of the hard-coded version of Chapter 2.
- Short story: tokens are defined by regular expressions, which are encoded into a non-deterministic finite automaton (NDFA), which can be automatically converted to a deterministic finite automaton (DFA), which can be described as a table of state $\times$ input $\rightarrow$ action $\times$ state, which can be executed by a simple program.
- Shorter story: write a set of regular expressions and let a [scanner generator](#) do the rest of the work. This method is an example of [declarative programming](#).
- There are some complexities.
  - Escaped double-quote within a string literal.
  - Over-eagerness leading to error, such as 3..4 in Pascal, or ‘a’ in Ada, or DO 200 I = 1.10 in Fortran.
  - The scanner needs to be very fast. Scanning tends to be the most time-consuming step of compilation, partly because of the cost of reading the source code (with all its inclusions).

16 **Regular expressions**

- This material should be a review.
- A regular expression defines a [language](#), which is a set (possibly infinite) of strings over some [alphabet](#) $\Sigma$.
- A regular expression is built recursively on the following components.
  - $\emptyset$.
  - $\lambda$.
  - Individual letters in $\Sigma$. Example: $a$.
  - The concatenation of regular expressions. The concatenation operator is usually omitted. Example: $abca$. 
• the alternation of regular expressions. The alternation operator is written $\mid$.
• closure operations: the Kleene closure $\ast$ and the positive closure $\oplus$.
• parentheses for grouping.
• If you want to use a metacharacter such as $\mid, \ast, \oplus, (,)$ in a regular expression, use some sort of escape character (typically \ ) before it.

• A regular expression generates a set of strings. That set is called the language generated by the regular expression.

• $\emptyset$ generates no strings at all.
• $\lambda$ generates the empty string.
• An individual letter generates the string containing just that letter.
• The concatenation of two regular expressions $A$ and $B$ generates all two-part strings, whose first part is a string generated by $A$ and whose second part is a string generated by $B$.
• The alternation of two regular expressions $A$ and $B$ generates all strings generated by $A$ and all strings generated by $B$.
• The expression $A\ast$ generates the empty string and (recursively) all strings generated by $AA\ast$. The expression $A\oplus$ generates all strings generated by $AA\ast$.

• Useful facts

• The set of strings generated by a regular expression is called a regular set. Every regular set can be generated by some regular expression.
• Every finite set of strings is a regular set. At worst, one can just build a regular expression that enumerates them with alternations.
• Any regular set has multiple regular expressions that generate it. For instance, $(ab)^\ast$ can also be written $\lambda\mid ab\mid abab(ab)^\ast$.

• Notations

• If $A$ is a set of characters, we use $\text{not}(A)$ to denote $\Sigma - A$, the characters not in $A$. 
• If $S$ is a set of strings, we use $\text{not}(S)$ to denote all (finite) strings except those in $S$. It turns out that if $S$ is a regular set, so is $\text{not}(S)$.

• If $k \geq 0$ is a constant integer and $S$ is a set of strings, then $S^k$ is the set of strings formed by concatenating $k$ strings (possibly different) from $S$. If $S$ is a regular set, so is $S^k$.

### 17 Useful examples

- **Class 6, 9/3/2020**

- a Java comment that goes to the end of the line: \(/ \text{not} \leftarrow\)  
  (Here, $\Sigma$ is the set of all 16-bit Unicode characters and $\langle \rangle$ is a line separator, which is platform-dependent.)

- a decimal literal: $D^+.D^+$ where $D$ is shorthand for $(0|1|2|3|4|5|6|7|8|9)$.

- an integer literal, optionally signed: $(+|−|\lambda)D^+$.

- a comment delimited by `##` markers: `##((\#|\lambda)\text{not}(\#))##`

- a Fortran-like real literal, which requires digits only on one side (either one) of the decimal point: $(D^+.D^*)|(D^+)$

- an identifier, with underscores, but not adjacent, frontal, or terminal ones: $L(L|D)^*|(L|D)^+)^*$, or (Daniel Michler) $L((\lambda)(L|D)^*)^*$.

### 18 Hashing

- Very popular data structure for searching.

- Cost of insertion and of search is $O(\log n)$, but only because $n$ distinct values must be $\log n$ bits long, and we need to look at the entire key. If we consider looking at the key to be $O(1)$, then hashing is expected to be $O(1)$.

- Java provides an interface `Map<K, V>` with several implementations: `HashMap<K, V>` (recommended), `Hashtable<K, V>` (synchronized, so more expensive) and others for specialty purposes. The key type $K$ and value type $V$ can be any classes, although `String` and `String` are typical.

- Idea: find the value associated with key $k$ at $A[h(k)]$, where
• $h()$ maps keys to integers in $0..s - 1$, where $s$ is the size of A[].
• $h()$ is “fast”. (It generally needs to look at all of $k$, though.)

• Example
  • $k$ = student in class.
  • $h(k)$ = $k$’s birthday (a value from 0 .. 365).

• Difficulty: collisions
  • Birthday paradox: \( \text{Prob(no collisions with } j \text{ people)} = \frac{365!}{(365-j)!365^j} \)
  • This probability goes below $\frac{1}{2}$ at $j = 23$.
  • At $j = 50$, the probability is 0.029.

• Moral: One cannot in general avoid collisions. One has to deal with them.

• A good hash function
  • Desiderata
    • Uniform: Equally likely to give any value in $0..s - 1$.
    • Spreading: similar inputs $\rightarrow$ dissimilar outputs, to prevent clustering. Only important for open-addressing conflict resolution.
    • Fast.

• Several suggestions, assuming that $k$ is a multi-word data structure, such as a string.
  • Add (or multiply) all the words of $k$, discarding overflow, then mod by $s$. It helps if $s = 2^j$, because mod is then masking with $2^j - 1$.
  • XOR the words of $k$, shifting left by 1 after each, followed by mod $s$.

• Wisdom: it doesn’t make much difference what hash function you choose. It is not even necessary to look at all of $k$. Just make sure that $h(k)$ is not constant (except for testing collision resolution).

• Dealing with collisions: open addressing
  • Overview
• The following methods store all items in A[] and use a probe sequence. If the desired position is occupied, some other position is open to consider instead.
• They tend to suffer from clustering.
• Deletion is hard, because removing an element can damage unrelated searches. Deletion by **marking** is the only reasonable approach.

• Perfect hashing: if you know all \( n \) values in advance, you can look for a non-colliding hash function \( h \). Finding such a function is in general quite difficult, but compiler writers do sometimes use perfect hashing to detect keywords in the language (like **begin** and **for**).

• Additional hash functions. Have a series of hash functions, \( h_1(), h_2(), \ldots \)
  • insertion: key probing with different functions until an empty slot is found.
  • searching: probe with different functions until you find the key (success) or an empty slot (failure).
  • You need a **family** of independent hash functions.
  • The method is very expensive when A[] is almost full.

• Linear probing. Probe \( p \) is at \( h(k) + p \mod s \), for \( p = 0, 1, \ldots \)
  • Terrible behavior when A[] is almost full, because chains coalesce. This problem is called “primary clustering”.

• Quadratic probing. Probe \( p \) is at \( h(k) + p^2 \mod s \), for \( p = 0, 1, \ldots \)
  • When does this sequence hit all of A[]? Certainly it does if \( s \) is prime.
  • We still suffer “secondary clustering”: if two keys have the same hash value, then the sequence of probes is the same for both.

• Add-the-hash rehash. Probe \( p \) is at \((p + 1) \cdot h(k) \mod s\).
  • This method avoids clustering.
  • Warning: \( h(k) \) must never be 0.

• Double hashing. Use two has functions, \( h_1() \) and \( h_2() \). Probe \( p \) is at \( h_1(k) + p \cdot h_2(k) \).
  • This method avoids clustering.
• Warning: \( h_2(k) \) must never be 0.

• Dealing with collisions: chaining
  • Each element in A is a pointer, initially null, to a bucket, which is a linked list of nodes that hash to that element; each node contains \( k \) and any other associated data.
  • insert: place \( k \) at the head of \( A[h(k)] \).
  • search: look through the list at \( A[h(k)] \).
    • optimization: When you find, promote the node to the start of its list.
  • average list length is \( s/n \). So if we set \( s \approx n \) we expect about 1 element per list, although some may be longer, some empty.
  • Instead of lists, we can use something fancier (such as 2-3 trees), but it is generally better to use a larger \( s \).

19 Finite-state automata

• A finite-state automaton (FSA) is an idealization of a very simple computer, composed of
  • A finite set of states, usually depicted by circles.
    • One of the states is called the start state. It can be depicted by a circle with an arrow from nowhere pointing to it.
    • One or more of the states are called final (or accepting) states. They are usually depicted by double circles.
  • A finite alphabet, denoted \( \Sigma \). We’ll call the elements of the alphabet letters.
  • A set of transitions between states, usually depicted by labelled arrows. The labels are letters.