Agenda

- Overview of language recognizers
- Basic concepts of formal grammars
- Scanner Theory
  - Regular expressions
  - Finite automata (to recognize regular expressions)
- Scanner Implementation
And the point is…

- How do execute this?

```c
int nPos = 0;
int k = 0;
while (k < length) {
    if (a[k] > 0) {
        nPos++;
    }
}
```

- How do we understand what it means?
Compilers vs. Interpreters

- Interpreter
  - A program that reads a source program and executes that program

- Compiler
  - A program that translates a program from one language (the source) to another (the target)
Interpreter

- Execution engine
- Program execution interleaved with analysis
  
  ```
  running = true;
  while (running) {
    analyze next statement;
    execute that statement;
  }
  ```
  
- May involve repeated analysis of some statements (loops, functions)
Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Should “improve” the program in some fashion
- Offline process
  - Tradeoff: compile time overhead (preprocessing step) vs execution performance
Hybrid approaches

Well-known example: Java

- Compile Java source to byte codes – Java Virtual Machine language (.class files)
- Execution
  - Interpret byte codes directly, or
  - Compile some or all byte codes to native code
    - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code

Variation: .NET

- Compilers generate MSIL
- All IL compiled to native code before execution
Compiler/Interpreter Structure

- First approximation
  - Front end: analysis
    - Read source program and understand its structure and meaning
  - Back end: synthesis
    - Execute or generate equivalent target program
Common Issues

- Compilers and interpreters both must read the input – a stream of characters – and “understand” it; *analysis*

```c
while (k < length) {
    if (a[k] > 0)
        nPos ++;
}
```
Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar

- First done in 1959 with BNF (Backus-Naur Form or Backus-Normal Form) used to specify the syntax of ALGOL 60
- Borrowed from the linguistics community (Chomsky)
Grammar for a Tiny Language

program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) statement
expr ::= id | int | expr + expr
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Context-Free Grammars

Formally, a grammar $G$ is a tuple $\langle N, \Sigma, P, S \rangle$ where

- $N$ a finite set of *non-terminal* symbols
- $\Sigma$ a finite set of *terminal* symbols
- $P$ a finite set of *productions*
  - A subset of $N \times (N \cup \Sigma)^*$
- $S$ the *start symbol*, a distinguished element of $N$
  - If not specified otherwise, this is usually assumed to be the non-terminal on the left of the first production
Productions

- The rules of a grammar are called *productions*.
- Rules contain:
  - Nonterminal symbols: grammar variables (*program*, *statement*, *id*, etc.)
  - Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, (, {, ), }, …)
- Meaning of *nonterminal* ::= <sequence of terminals and nonterminals>
  - In a derivation, an instance of nonterminal can be replaced by the sequence of terminals and nonterminals on the right of the production.
- Often, there are two or more productions for a single nonterminal – can use either at different points in a derivation.
There are several common notations for productions; all mean the same thing.

\[
\begin{align*}
ifStmt & ::= \text{if ( expr ) stmt} \\
ifStmt & \rightarrow \text{if ( expr ) stmt} \\
<ifStmt> & ::= \text{if ( <expr> ) <stmt>}
\end{align*}
\]
Example Derivation

\[ a = 1 \; ; \; \text{if} \; ( \; a \; + \; 1 \; ) \; b = 2 \; ; \]
Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from the concrete, character-by-character grammar
- In practice this is never done
In real compilers the recognizer is split into two phases

- **Scanner**: translate input characters to tokens
  - Also, report lexical errors like illegal characters and illegal symbols

- **Parser**: read token stream and reconstruct the derivation
  - Procedural interface – ask the scanner for new tokens when needed
Scanner Example

- Input text
  
  ```
  // this statement does very little
  if (x >= y) y = 42;
  ```

- Token Stream

  ```
  IF  LPAREN  ID(x)  GEQ  ID(y)  RPAREN  ID(y)  BECOMES  INT(42)  SCOLON
  ```

  □ Notes: tokens are atomic items, not character strings; comments are *not* tokens
Parser Example

- **Token Stream Input**
  
  ```
  IF LPAREN ID(x) GEQ ID(y) RPAREN ID(y) BECOMES INT(42) SCOLON
  ```

- **Abstract Syntax Tree**
  
  ![Diagram of an abstract syntax tree representing the token stream input.](image-url)
Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
  - Scanner hides details from parser (comments, whitespace, etc.)
  - Parser is easier to build; has simpler input stream (tokens)

- Efficiency
  - Scanner can use simpler, faster design
    - (But still often consumes a surprising amount of the compiler’s total execution time)
Tokens

- Idea: we want a distinct token kind (lexical class) for each distinct terminal symbol in the programming language
  - Examine the grammar to find these

- Some tokens may have attributes. Examples:
  - All integer constants are one kind of token, but the actual value (17, 42, …) will be an attribute
  - Identifier tokens will carry a string with the id
Typical Tokens in Programming Languages

- **Operators & Punctuation**
  - `+ - * / ( ) { } [ ] ; : :: < <= == = != ! …`
  - Each of these is a distinct lexical class

- **Keywords**
  - `if while for goto return switch void …`
  - Each of these is also a distinct lexical class (not a string)

- **Identifiers**
  - A single ID lexical class, but parameterized by actual id

- **Integer constants**
  - A single INT lexical class, but parameterized by int value

- Other constants, etc.
Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice.

**Example**

```c
return iffy != todo;
```

should be recognized as 5 tokens

```
RETURN ID(iffy) NEQ ID(todo) SCOLON
```

not more (i.e., not parts of words or identifiers, or ! and = as separate tokens)
Formal Languages & Automata Theory (in one slide)

- **Alphabet**: a finite set of symbols
- **String**: a finite, possibly empty sequence of symbols from an alphabet
- **Language**: a set, often infinite, of strings
- Finite specifications of (possibly infinite) languages
  - **Automaton** – a *recognizer*; a machine that accepts all strings in a language (and rejects all other strings)
  - **Grammar** – a *generator*; a system for producing all strings in the language (and no other strings)
- A particular language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language
Regular Expressions and FAs

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  - Aside: Difficulties with Fortran, among others

- Tokens can be recognized by a deterministic finite automaton
  - Can be either table-driven or built by hand based on lexical grammar
Regular Expressions

- Defined over some alphabet $\Sigma$
  - For programming languages, commonly ASCII or Unicode
- If $re$ is a regular expression, $L(re)$ is the language (set of strings) generated by $re$
## Fundamental REs

<table>
<thead>
<tr>
<th>re</th>
<th>( L(re) )</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{ a }</td>
<td>Singleton set, for each a in ( \Sigma )</td>
</tr>
<tr>
<td>( \varepsilon )</td>
<td>{ \varepsilon }</td>
<td>Empty string</td>
</tr>
<tr>
<td>( \emptyset )</td>
<td>{ }</td>
<td>Empty language</td>
</tr>
</tbody>
</table>
## Operations on REs

<table>
<thead>
<tr>
<th>re</th>
<th>( L(re) )</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>( L(r)L(s) )</td>
<td>Concatenation</td>
</tr>
<tr>
<td>r</td>
<td>s</td>
<td>( L(r) \cup L(s) )</td>
</tr>
<tr>
<td>r*</td>
<td>( L(r)^* )</td>
<td>0 or more occurrences (Kleene closure)</td>
</tr>
</tbody>
</table>

- Precedence: * (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed
The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Typical examples:

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>r+</td>
<td>(rr*)</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>(r</td>
<td>ε)</td>
</tr>
<tr>
<td>[a-z]</td>
<td>(a</td>
<td>b</td>
</tr>
<tr>
<td>[abxyz]</td>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>
## Examples

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>single + character</td>
</tr>
<tr>
<td>!</td>
<td>single ! character</td>
</tr>
<tr>
<td>=</td>
<td>single = character</td>
</tr>
<tr>
<td>!=</td>
<td>2 character sequence</td>
</tr>
<tr>
<td>&lt;=</td>
<td>2 character sequence</td>
</tr>
<tr>
<td>hogwash</td>
<td>7 character sequence</td>
</tr>
</tbody>
</table>
More Examples

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[abc]+</td>
<td></td>
</tr>
<tr>
<td>[abc]</td>
<td></td>
</tr>
<tr>
<td>[0-9]+</td>
<td></td>
</tr>
<tr>
<td>[1-9][0-9]*</td>
<td></td>
</tr>
<tr>
<td>[a-zA-Z][a-zA-Z0-9_]*</td>
<td></td>
</tr>
</tbody>
</table>
Abbreviations

- Many systems allow abbreviations to make writing and reading definitions easier.
  
  name ::= re

  - Restriction: abbreviations may not be circular (recursive) either directly or indirectly.
Example

- Possible syntax for numeric constants

\[
digit ::= [0-9]  
digits ::= digit^*  
number ::= digits ( . digits )?  
        ( [eE] (+ | -)? digits ) ?
\]
Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Not totally straightforward, but can be done systematically
  - Tools like Lex, Flex, and JLex do this automatically from a set of REs read as input
  - Even if you don’t use a FA explicitly, it is a good way to think about the problem
Finite State Automaton (FSA)

- A finite set of states
  - One marked as initial state
  - One or more marked as final states
  - States sometimes labeled or numbered
- A set of transitions from state to state
  - Each labeled with symbol from $\Sigma$, or $\varepsilon$
- Operate by reading input symbols (usually characters)
  - Transition can be taken if labeled with current symbol
  - $\varepsilon$-transition can be taken at any time
- Accept when final state reached & no more input
  - Scanner slightly different – accept longest match each time called, even if more input; i.e., run the FSA each time the scanner is called
- Reject if no transition possible or no more input and not in final state (DFA)
Example: FSA for “cat”
DFA vs NFA

- **Deterministic Finite Automata (DFA)**
  - No choice of which transition to take under any condition

- **Non-deterministic Finite Automata (NFA)**
  - Choice of transition in at least one case
  - Accept - if *some way* to reach final state on given input
  - Reject - if *no possible* way to final state
FAs in Scanners

- Want DFA for speed (no backtracking)
- Conversion from regular expressions to NFA is easy
- There is a well-defined procedure for converting a NFA to an equivalent DFA
  - See formal language or compiler textbooks for details (RE to NFA to DFA to minimized DFA)
Example: DFA for hand-written scanner

- **Idea**: show a hand-written DFA for some typical programming language constructs
  - Then use to construct hand-written scanner

- **Setting**: Scanner is called whenever the parser needs a new token
  - Scanner stores current position in input file
  - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token, and update the “current position”
Scanner DFA Example (1)

0 as the starting state.

- Transition to state 1 on whitespace or comments.
- Transition to state 2 on '('.
- Transition to state 3 on ')'.
- Transition to state 4 on ';'.

State 1 accepts EOF (end of input).
State 2 accepts LPAREN.
State 3 accepts RPAREN.
State 4 accepts SCOLON.
Scanner DFA Example (2)

![DFA Diagram]

- **State 5**: Transition on `!` to `6` (Accept NEQ)
- **State 6**: Transition on `=` to `7` (Accept NOT)
- **State 7**: Transition on `other` to `7` (Accept NOT)
- **State 8**: Transition on `<` to `9` (Accept LEQ)
- **State 9**: Transition on `=` to `10` (Accept LESS)
- **State 10**: Transition on `other` to `10` (Accept LESS)

**States and Transitions**:
- **5**: Transition on `!` to `6` (Accept NEQ)
- **6**: Transition on `=` to `7` (Accept NOT)
- **7**: Transition on `other` to `7` (Accept NOT)
- **8**: Transition on `<` to `9` (Accept LEQ)
- **9**: Transition on `=` to `10` (Accept LESS)
- **10**: Transition on `other` to `10` (Accept LESS)
Scanner DFA Example (3)
Scanners DFA Example (4)

- Strategies for handling identifiers vs keywords
  - **Hand-written scanner**: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - **Machine-generated scanner**: generate DFA with appropriate transitions to recognize keywords
    - Lots 'o states, but efficient (no extra lookup step)
Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure
  ```java
  public class Token {
    public int kind;       // token’s lexical class
    public int intVal;     // integer value if class = INT
    public String id;      // actual identifier if class = ID
    // lexical classes
    public static final int EOF = 0;  // “end of file” token
    public static final int ID   = 1;  // identifier, not keyword
    public static final int INT  = 2;  // integer
    public static final int LPAREN = 4;
    public static final int SCOLN = 5;
    public static final int WHILE = 6;
    // etc. etc. etc. …          // use enums if you’ve got ‘em
  }
  ```
Simple Scanner Example

// global state and methods

static char nextch;  // next unprocessed input character

// advance to next input char
void getch() { ... }

// skip whitespace and comments
void skipWhitespace() { ... }
Scanner getToken() method

// return next input token
public Token getToken() {
    Token result;

    skipWhiteSpace();

    if (no more input) {
        result = new Token(Token.EOF); return result;
    }

    switch(nextch) {
    case '(' : result = new Token(Token.LPAREN); getch(); return result;
    case ')' : result = new Token(Token.RPAREN); getch(); return result;
    case ';' : result = new Token(Token.SCOLON); getch(); return result;
    // etc. …
    }

    // etc. …
getToken() (2)

case '!': // ! or !=
    getch();
    if (nextch == '=') {
        result = new Token(Token.NEQ); getch(); return result;
    } else {
        result = new Token(Token.NOT); return result;
    }

case '<': // < or <=
    getch();
    if (nextch == '=') {
        result = new Token(Token.LEQ); getch(); return result;
    } else {
        result = new Token(Token.LESS); return result;
    }
// etc. …
getToken() (3)

case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
    // integer constant
    String num = nextch;
    getch();
    while (nextch is a digit) {
        num = num + nextch; getch();
    }
    result = new Token(Token.INT, Integer(num).intValue());
    return result;

...
getToken (4)

case 'a': ... case 'z':
case 'A': ... case 'Z':  // id or keyword
    string s = nextch; getch();
    while (nextch is a letter, digit, or underscore) {
        s = s + nextch; getch();
    }
    if (s is a keyword) {
        result = new Token(keywordTable.getKind(s));
    } else {
        result = new Token(Token.ID, s);
    }
    return result;
Alternatives

- Use a tool to build the scanner from the (regexp) grammar
  - Often can be *more* efficient than hand-coded!
- Build an ad-hoc scanner using regular expression package in implementation language
  - Ruby, Perl, Java, many others
  - Fine to use for our project