**Agenda**

- Basic concepts of formal grammars (review)
- Regular expressions
- Lexical specification of programming languages
- Using finite automata to recognize regular expressions
- Scanners and Tokens

---

**Programming Language Specs**

- 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 ) stmt`
- `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`

---

**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 times

---

**Alternative Notations**

- There are several syntax notations for productions in common use; all mean the same thing
  - `ifStmt ::= if ( expr ) stmt`
  - `ifStmt → if ( expr ) stmt`
  - `<ifStmt> ::= if ( <expr> ) <stmt>`
```
Example Derivation

program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) stmt
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

a = 1 ; 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

### Parsing & Scanning
- 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

### Characters vs Tokens (review)
- 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
  ```

### Why Separate the Scanner and Parser?
- Simplicity & Separation of Concerns
  - Scanner hides details from parser (comments, whitespace, input files, 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: integer constant token will have the actual integer (17, 42, ...) as an attribute; identifiers will have a string with the actual 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
  - \texttt{return foobar != hohum;}
  - should be recognized as 5 tokens
  - \texttt{RETURN ID(foobar) NEQ ID(hohum) SCOLON}
  - not more (i.e., not parts of words or identifiers, or ! and = as separate tokens)

Formal Languages & Automata Theory (a review 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
  - (Sometimes a little cheating is needed)
- 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+s</td>
<td>L(r)∩L(s)</td>
<td>Combination (union)</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

Abbreviations

- 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>(r*)</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, given a set of REs

Finite State Automaton

- 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

FAas 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

From RE to NFA: base cases

- $a$
- $\varepsilon$
From NFA to DFA

- Subset construction
  - Construct a DFA from the NFA, where each DFA state represents a set of NFA states
- Key idea
  - The state of the DFA after reading some input is the set of all states the NFA could have reached after reading the same input
- Algorithm: example of a fixed-point computation
- If NFA has $n$ states, DFA has at most $2^n$ states
  - $\Rightarrow$ DFA is finite, can construct in finite # steps
- Resulting DFA may have more states than needed
  - See books for construction and minimization details

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
  - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

Scanner DFA Example (1)
Scanner DFA Example (2)

Accept NEQ

Accept NOT

Accept LEQ

Accept LESS

Scanner DFA Example (3)

Accept INT

Scanner DFA Example (4)

Accept ID or keyword

Scanner DFA Example (5)

Implementing a Scanner by Hand – Token Representation

A token is a simple, tagged structure

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. ...
}

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. ...
    }
}
getToken() (2)

```java
    case '!': // ! or !=
        result = new Token(Token.NEQ); return result;
    case '<': // < or <=
        result = new Token(Token.LEQ); return result;
        // etc.
```

getToken() (3)

```java
    case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            // integer constant
            result = new Token(Token.INT, Integer(num).intValue()); return result;
```

getToken (4)

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

Project Notes

- For the course project, use a lexical analyzer generator
- Suggestion: JLex (or JFlex) a Lex/Yacc-like pair of compiler tools

Coming Attractions

- Homework this week: paper exercises on regular expressions, etc.
- Next week: first part of the compiler assignment – the scanner
  - Basically the project from Ch. 2 of Appel’s book if you want to get a bit ahead
- Next topic: parsing
  - Will do LR parsing first – suggest using this for the project (thus CUP (YACC-like) instead of JavaCC)
  - Good time to start reading ch. 3.