CSE 401 – Compilers

Languages, Automata, Regular Expressions & Scanners
Hal Perkins
Winter 2009

Agenda
- Review basic concepts of formal grammars
- 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 several productions for a nonterminal – can choose any in different parts of derivation

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>
Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from a 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

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 recognizes regular expressions – proper subset of context free grammars
    - Much faster than general CFG parsing
  - (But still often consumes a surprising amount of the compiler’s total execution time)

But …

- Not always possible to separate cleanly
  - Example: C/C++/Java type vs identifier
  - Parser would like to know which names are types and which are identifiers, but
    - Scanner doesn’t know how things are declared …
  - So we hack around it somehow…
    - Either use simpler grammar and disambiguate later, or communicate between scanner & parser
  - Engineering issue: try to keep interfaces as simple & clean as possible

Definitions

- Pattern: a definition of a related set of lexical entities
  - Ex: all sequences of numeric characters, all sequences of alphanumeric characters starting with an alphabetic character
  - Regular expressions are used in practice to define patterns
- Lexeme: group of characters that matches a pattern
  - Ex: ‘1234’, ‘43204222’, ‘snork’, ‘f0rk’
- Token: class of lexemes matching a pattern, distinguished by an attribute
  - Ex: ‘snork’ and ‘f0rk’ are both identifier lexemes with the actual names kept as an attribute

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:
  ```
  return maybe != iffy;
  ```
  should be recognized as 5 tokens:
  ```
  RETURN ID(maybe) NEQ ID(iffy) SCOLON
  ```
  i.e., `!=` is one token, not two; "iffy" is an ID, not IF followed by ID(fy).

Lexical Complications

- Most modern languages are free-form
  - Layout doesn’t matter
  - Whitespace separates tokens
- Alternatives
  - Fortran – line oriented
  - Haskell, Python – indentation and layout can imply grouping
- And other confusions
  - In C++ or Java, is `>>` a single operator or the end of two nested templates or generic classes?

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

Language (Chomsky) hierarchy: quick reminder

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
- Context-sensitive (Type-1) languages aren’t too important
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines

Regular Expressions

- Defined over some alphabet \( \Sigma \)
  - For programming languages, alphabet is usually 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>$\varnothing$</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

### Abbreviations

The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Some examples:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r+$</td>
<td>$(r^*)$</td>
<td>0 or more occurrences</td>
</tr>
<tr>
<td>$r?$</td>
<td>$(r \mid \varepsilon)$</td>
<td>0 or 1 occurrence</td>
</tr>
<tr>
<td>$[a-z]$</td>
<td>$(a</td>
<td>b</td>
</tr>
<tr>
<td>$[a-b</td>
<td>x</td>
<td>y</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>$</td>
<td>$</td>
</tr>
<tr>
<td>$==</td>
<td>$</td>
</tr>
<tr>
<td>$!=</td>
<td>$</td>
</tr>
<tr>
<td>$&lt;=</td>
<td>$</td>
</tr>
<tr>
<td>$xyzzy$</td>
<td>5 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 or specifications easier:
  - `name ::= re`

- Restriction: abbreviations may not be circular (recursive) either directly or indirectly (else would be non-regular)
Example

Possible syntax for numeric constants

digit ::= [0-9]
digits ::= digit+
number ::= digits ( . digits )?
    ([eE] (+ | -)? digits)?

Initial MiniJava Lexical Spec.

Program ::= (Token | Whitespace)*
Token ::= ID | Integer | ReservedWord | Operator | Delimiter
ID ::= Letter (Letter | Digit)*
Letter ::= a | ... | z | A | ... | Z
Integer ::= Digit+
ReservedWord ::= class | public | static | extends |
    void | int | boolean | if | else |
    while | return | true | false | this | new |
    String | main | System.out.println
Operator ::= + | - | * | / | < | <= | >= | > | == | != | && | !
Delimiter ::= ; | . | | = | { | | } | [ | ]

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, Jlex et seq 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 Σ, or ε
- Transition can be taken if labeled with current symbol
- ε-transition can be taken at any time
- Accept when final state reached & no more input
- Scanner uses a FSA as a subroutine – accept longest match each time called, even if more input; i.e., run the FSA from the current location in the input 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
  - No ε transitions (arcs)
- 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
  - i.e., may need to guess right path or backtrack
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

From RE to NFA: base cases

<table>
<thead>
<tr>
<th>RE</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>![NFA for 'a']</td>
</tr>
<tr>
<td>ε</td>
<td>![NFA for 'ε']</td>
</tr>
</tbody>
</table>

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 NFA states that 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
To Tokens
- Every "final" state of a DFA emits a token
- Tokens are the internal compiler names for the lexemes
  - == becomes equal
  - ( becomes leftParen
  - private becomes private
- You choose the names
- Also, there may be additional data ... \e\n might include line count

DFA => Code
- Option 1: Implement by hand using procedures
  - one procedure for each token
  - each procedure reads one character
  - choices implemented using if and switch statements
- Pros
  - straightforward to write
  - fast
- Cons
  - a fair amount of tedious work
  - may have subtle differences from the language specification

DFA => code [continued]
- Option 2: use tool to generate table driven parser
  - Rows: states of DFA
  - Columns: input characters
  - Entries: action
  - Go to next state
  - Accept token, go to start state
  - Error
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - "Magic"
  - Table lookups may be slower than direct code, but switch implementation is a possible revision

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
- Disclaimer: Example for illustration only – you'll use tools for the project (see further below)

Scanner DFA Example (1)

Scanner DFA Example (2)
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; // open parenthesis
    public static final int SCOLN = 5; // semicolon
    public static final int WHILE = 6; // "while"
    // etc. etc. etc. …
}
```

Simple Scanner Example

```java
// global state and methods
static char nextch; // next unprocessed input character
void getch() { … } // advance to next input char
void skipWhitespace() { … } // skip whitespace and comments

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.SCOLN); getch(); return result;
        // etc. …
    }
    if ('!' == nextch) { // ! or !=
        if (nextch == '=') { // "not equal"
            result = new Token(Token.NEQ); getch(); return result;
        } else { // "not"
            result = new Token(Token.NOT);
            return result;
        }
    }
    if ('<' == nextch) { // < or <=
        if (nextch == '=') { // "less than or equal"
            result = new Token(Token.LEQ); getch(); return result;
        } else { // "less than"
            result = new Token(Token.LESS);
            return result;
        }
    } // etc. …
    return result;
}
```
getToken() (3)

```java
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)

```java
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;

### Automatic Scanner Generation ForMiniJava

- We use the jflex tool to automatically create a scanner from a specification file, `Scanner/minijava.jflex`
- We use the CUP tool to automatically create a parser from a specification file, `Parser/minijava.cup`
- Token classes are shared by jflex and CUP.
- The MiniJava Makefile automatically rebuilds the scanner (or parser) whenever its specification file changes

### Symbol Class

Tokens are represented as instances of class `Symbol`

```java
class Symbol {
    Int sym; // which token class?
    Object value; // any extra data for this lexeme
    ...
}
```

A different integer constant is defined for each token class in the `sym` helper class

```java
class sym {
    static int CLASS = 1;
    static int IDENTIFIER = 2;
    static int COMMA = 3;
    ...
}
```

Can use this in printing code for `Symbol`s; see `symbolToString` in `minijava.jflex`

### Token Declarations in CUP

- Declare new token classes in `Parser/minijava.cup`, using terminal declarations
- Include Java type if `Symbol` stores extra data
- Examples
  ```java
  // reserved words: */
  terminal CLASS, PUBLIC, STATIC, EXTENDS;
  ...
  // operators: */
  terminal PLUS, MINUS, STAR, SLASH, EXCLAIM;
  ...
  // delimiters: */
  terminal OPEN_PAREN, CLOSE_PAREN;
  terminal EQUALS, SEMICOLON, COMMA, PERIOD;
  ...
  // tokens with values: */
  terminal String IDENTIFIER;
  terminal Integer INT_LITERAL;
  ```

### jflex Token Specifications

- Helper definitions for character classes and re's
  ```java
  letter = [a-z A-Z]
  ```
- Simple token definitions are of the form:
  ```java
  regexp { Java stmt }
  ```
- `regexp` can be (at least):
  ```java
  • a string literal in double-quotes, e.g. "class", "<c"
  • a reference to a named helper, in braces, e.g. {letter}
  • a character list or range, in square brackets, e.g. [a-z A-Z]
  • a negated character list or range, e.g. [^\w\d]
  • (which matches any single character)
  ```
**jflex Specifications (cont.)**

- *Java stmt* (the accept action) in a token specification is typically:
  - `return symbol(sym.CLASS);` for a simple token
  - `return symbol(sym.CLASS,yytext());` for a token with extra data based on the lexeme `stringyytext()`
  - empty for whitespace

---

**Coming Attractions**

- Homework this week: paper exercises on regular expressions, etc.
- Next week: first part of the compiler assignment – the scanner
- Next topic: parsing
  - Will do LR parsing first – we need this for the project, then LL (recursive-descent) parsing