CSE P 501 – Compilers

Languages, Automata, Regular Expressions & Scanners
Hal Perkins
Spring 2018
Agenda

• Quick review of 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), used to specify ALGOL 60 syntax
  - Borrowed from the linguistics community (Chomsky)
Formal Languages & Automata Theory
(a review on one slide)

- **Alphabet**: a finite set of symbols and characters
- **String**: a finite, possibly empty sequence of symbols from an alphabet
- **Language**: a set of strings (possibly empty or infinite)
- **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
Language (Chomsky) hierarchy:

- Regular (Type-3) languages are specified by regular expressions/grammars and finite automata (FSAs)
  - Specs and implementation of scanners
- Context-free (Type-2) languages are specified by context-free grammars and pushdown automata (PDAs)
  - Specs and implementation of parsers
- Context-sensitive (Type-1) languages ... aren’t too interesting (for us, at least)
- Recursively-enumerable (Type-0) languages are specified by general grammars and Turing machines
Example:
Grammar for a Tiny Toy Language

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

\[ a = 1 \quad \text{if} \quad (a + 1) \quad b = 2 \]
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 at different points of a derivation
Alternative Notations

• There are several syntax notations for productions in common use; all mean the same thing

\[
\text{ifStmt ::= if ( expr ) statement}
\]
\[
\text{ifStmt \rightarrow if ( expr ) statement}
\]
\[
\text{<ifStmt> ::= if ( <expr> ) <statement>}
\]
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) / narrow interface

- Efficiency
  - Scanner recognizes regular expressions – proper subset of context free grammars
    (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
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 $\texttt{>>}$ a shift 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
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>$\emptyset$</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>$L(r)∪ L(s)$</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
- In “real” regular expression tools, need some way to “escape” literal ‘*’ or ‘|’ characters vs. operators – but don’t worry, or use different fonts, for math. regexps.
## 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 &quot;!=&quot;</td>
</tr>
<tr>
<td>xyzzy</td>
<td>5 character sequence &quot;xyzzy&quot;</td>
</tr>
<tr>
<td>(1</td>
<td>0)*</td>
</tr>
<tr>
<td>(1</td>
<td>0)(1</td>
</tr>
<tr>
<td>0</td>
<td>1(0</td>
</tr>
</tbody>
</table>
Derived Operators

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Some 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>[abcxyz]</td>
<td>(a</td>
<td>b</td>
</tr>
</tbody>
</table>
More Examples

<table>
<thead>
<tr>
<th>re</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>([abc]+) ((a/b/c))+ ((a/b/c)(a/b/c)^*)</td>
<td>all strings w/ 1 or more a's, b's, c's</td>
</tr>
<tr>
<td>([abc]*)</td>
<td>0 or more</td>
</tr>
<tr>
<td>([0-9]+) ({0/1/2/\ldots/9})+</td>
<td>all non-empty strings of decimal digits</td>
</tr>
<tr>
<td>([1-9][0-9]*)</td>
<td>that don't start w/ 0</td>
</tr>
<tr>
<td>([a-zA-Z][a-zA-Z0-9_])*</td>
<td>id</td>
</tr>
</tbody>
</table>
Abbreviations / Naming

- Many systems allow naming 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
  \[ digit ::= [0-9] \]
  \[ digits ::= digit^* \]
  \[ number ::= digits ( . digits )? \]
  \[ ( [eE] (+ | -)? digits )? \]

- How would you describe this set in English?
- What are some examples of legal constants (strings) generated by \textit{number}?
- What are the differences between these and numeric constants in YFPL? (Your Favorite Programming Language)
Recognizing REs

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - Reasonably straightforward, and can be done systematically
  - Tools like Lex, Flex, JFlex et seq do this automatically, given a set of Res
  - Same techniques used for grep, sed, other regular expression packages/tools
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
  - Slightly different in a scanner where the FSA is a subroutine that accepts the longest input string matching a token regular expression, starting at the current location in the input
- Reject if no transition possible, or no more input and not in final state (DFA)
  - Some versions (including textbook) have an explicit "error" state and transitions to it on all "no legal transition possible" input. OK to omit that for CSE 401
Example: FSA for “cat”
DFA vs NFA

• Deterministic Finite Automata (DFA)
  – No choice of which transition to take under any condition
  – No $\epsilon$ transitions (arcs)

• Non-deterministic Finite Automata (NFA)
  – Choice of transition in at least one case
  – Accept if some way to reach a 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)
- But conversion from regular expressions to NFA is easy
- Fortunately, there is a well-defined procedure for converting a NFA to an equivalent DFA (subset construction – will not cover in detail)
From RE to NFA: base cases

![Diagram showing transitions for RE to NFA conversion]
Exercise

• Draw the NFA for: $b(at|ag) | bug$
From NFA to DFA

- Subset construction
  - Construct a DFA from the NFA, where each DFA state represents a set of NFA states
- Key idea
  - 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 algorithms
Exercise

• Build DFA for $b(at | ag) | bug$, given the NFA
To Tokens

- A scanner is a DFA that finds the next token each time it is called
- Every “final” state of a DFA emits (returns) a token
- Tokens are the internal compiler names for the lexemes
  - `=` becomes `EQUAL`
  - `(` becomes `LPAREN`
  - `while` becomes `WHILE`
  - `xyzzy` becomes `ID(xyzzy)`
- You choose the names
- Also, there may be additional data ... \n might count lines; tokens might include line numbers
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 lot of tedious work
  – may have subtle differences from the language specification
DFA => Code [continued]

- Option 1a: Like option 1, but structured as a single procedure with multiple return points
  - choices implemented using if and switch statements
- Pros
  - also straightforward to write
  - faster
- Cons
  - a lot of tedious work
  - may have subtle differences from the language specification
DFA => code [continued]

- Option 2: use tool to generate table driven scanner
  - 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”
DFA => code [continued]

- Option 2a: use tool to generate scanner
  - Transitions embedded in the code
  - Choices use conditional statements, loops
- Pros
  - Convenient
  - Exactly matches specification, if tool generated
- Cons
  - “Magic”
  - Lots of code – big but potentially quite fast
    - Would never write something like this by hand, but can generate it easily enough
Example: DFA for hand-written scanner

- Idea: show a hand-written DFA for some typical programming language constructs
  - Then use to outline hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
  - Scanner knows (saves) current position in input
  - From there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token; save updated position for next time
- Disclaimer: Example for illustration only – you’ll use tools for the course project
  - & we’re abusing the DFA notation a little – not all arrows in the diagram correspond to consuming an input character, but meaning should be pretty obvious
Scanner DFA Example (1)

1. **whitespace or comments**
   - End of input

2. Accept EOF

3. Accept LPAREN

4. Accept RPAREN

5. Accept SCOLON
Scanner DFA Example (2)
Scanner DFA Example (3)
Scanner 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 will 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
    // useful extra information for debugging / diagnostics:
    public int line;
    public int column;
    // lexical classes (ancient java - better to use enums)
    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)

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;
MiniJava Scanner Generation

- We’ll use the jflex tool to automatically create a scanner from a specification file
- We’ll use the CUP tool to automatically create a parser from a specification file
- Token class defs. shared by jflex and CUP. Lexical classes are listed in CUP’s input file and it generates the token class definition.
TODO & Coming Attractions

• Homework this week: paper exercises on regular expressions & automata. Due Monday night – submit via gradescope (details on assignment)
• Find a partner for the project and fill out partner info form on web site by next week
• Next topic: parsing
  – Will do LR parsing first – we need this for the project, then LL (recursive-descent) parsing, which you should also know
  – Good time to start reading ahead