CSE 413
Programming Languages & Implementation

Hal Perkins
Autumn 2012
Grammars, Scanners & Regular Expressions
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 (review)

- **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

- 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 ) { <nl>
    if ( a[ k ] > 0
        ) <nl> <tab> <tab>{ nPos ++ ; } <nl> <tab> } 
```
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
  - Adapted 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 $<N, \Sigma, P, S>$ 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
  
  $\text{nonterminal} ::= <\text{sequence of terminals and nonterminals}>$
  – 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
Alternative Notations

- There are several common notations for productions; all mean the same thing

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

\[
\begin{align*}
\text{program} & ::= \text{statement} \mid \text{program statement} \\
\text{statement} & ::= \text{assignStmt} \mid \text{ifStmt} \\
\text{assignStmt} & ::= \text{id} = \text{expr} \\
\text{ifStmt} & ::= \text{if} ( \text{expr} ) \text{ statement} \\
\text{expr} & ::= \text{id} \mid \text{int} \mid \text{expr} + \text{expr} \\
\text{id} & ::= \text{a} \mid \text{b} \mid \text{c} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{n} \mid \text{x} \mid \text{y} \mid \text{z} \\
\text{int} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

\[
\begin{align*}
a & = 1 \; ; \quad \text{if} \ ( \ a \ + \ 1 \ ) \quad b & = 2 \; ;
\end{align*}
\]
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
- Typically a procedural interface – parser asks the scanner for new tokens when needed
Scanner Example

• Input text

```c
// 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
```

- Tokens are atomic items, not character strings
- Comments and whitespace are not tokens in most programming languages
Parser Example

• Token Stream Input

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

• Abstract Syntax Tree

```
ifStmt
  >=
  ID(x) ID(y) INT(42)
assign
```

```python
if ID(x) >= ID(y):
    ID(y) = INT(42)
```
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 if you’re not careful)
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 a single kind of token, but the actual value (17, 42, …) will be an attribute
  - Identifier tokens carry a string with the id
Typical Programming Language Tokens

• Operators & Punctuation
  – \+ - \* / ( ) { } [ ] ; :: < \leq = \equiv = != ! \ldots
  – Each of these is a distinct lexical class

• Keywords
  – \texttt{if while for goto return switch void} \ldots
  – 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 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, some 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) ∪ 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. 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><em>re</em></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

\[
\text{name ::= re}
\]

– Restriction: abbreviations may not be circular (recursive) either directly or indirectly (otherwise it would no longer be a regular expression – would be a context-free grammar)
Example

- Possible syntax for numeric constants

  \[
  \begin{align*}
  \text{digit} &::= \text{[0-9]} \\
  \text{digits} &::= \text{digit}^+ \\
  \text{number} &::= \text{digits} ( \ . \ \text{digits} )? \\
  &\quad ( [eE] ( + | - )? \ \text{digits} ) ?
  \end{align*}
  \]

33
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 $\epsilon$
• Operate by reading input symbols (usually characters)
  – Transition can be taken if labeled with current symbol
  – $\epsilon$-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 any 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 the DFA to construct a 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)

whitespace or comments

end of input

Accept EOF

Accept LPAREN

Accept RPAREN

Accept SCOLON
Scanner DFA Example (2)

- State 5: Transition on '!' to State 6, Accepts NEQ
- State 5: Transition on '=' to State 7, Accepts NOT
- State 7: Transition on 'other' to State 8, Accepts NOT
- State 8: Transition on '<' to State 9, Accepts LEQ
- State 8: Transition on 'other' to State 10, Accepts LESS
- State 6: Accepts NEQ
- State 7: Accepts NOT
- State 9: Accepts LEQ
- State 10: Accepts LESS
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 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. Something like:
  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. … // but 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 'b'): result = new Token(Token.RPAREN); getch(); return result;
        case 'b';: 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. …
```java
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
  – Suggest you use this for our project (good excuse to learn the Ruby regexp package)