CSE 413
Programming Languages & Implementation

Hal Perkins
Winter 2019
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 we execute this?

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

• Or, more concretely, how do we program a computer to understand and carry out a computation described in a programming language?
Compilers vs. Interpreters (recall)

• **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)

• For both of these we need to represent the program in some suitable data structure (usually a tree)
  – With MUPL we started with the tree and didn’t worry about where it came from
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)
  - MUPL was a special case of this – a function to evaluate expressions under a given environment
Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - Presumably easier to execute or more efficient
  - Usually “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 (interpreter in JVM), 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 when method is called
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

```plaintext
while ( k < length ) { <nl>
  if ( a[ k ] > 0 ) <nl> <tab>
  { nPos ++ ; } <nl> <tab> }
```
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

\[
\begin{align*}
\text{program} &::= \text{statement} \mid \text{program statement} \\
\text{statement} &::= \text{assignStmt} \mid \text{ifStmt} \\
\text{assignStmt} &::= \text{id} = \text{expr} ; \\
\text{ifStmt} &::= \text{if} ( \text{expr} ) \text{statement} \\
\text{expr} &::= \text{id} \mid \text{int} \mid \text{expr} + \text{expr} \\
\text{id} &::= a \mid b \mid c \mid i \mid j \mid k \mid n \mid x \mid y \mid z \\
\text{int} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]
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 $N \times (N \cup \Sigma)^*$
  
  (can think of these as rules from $N \rightarrow (N \cup \Sigma)^*$)

- $S$ the start symbol, a distinguished element of $N$

  If not otherwise specified, 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 production
  
  \[
  \text{nonterminal ::= <sequence of terminals and nonterminals>}
  \]
  
  In a derivation, any 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 any at different points in a derivation
Alternative Notations

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

\[
\text{ifStmt} ::= \text{if ( expr ) stmt}
\]

\[
\text{ifStmt} \rightarrow \text{if ( expr ) stmt}
\]

\[
<\text{ifStmt}> ::= \text{if ( <expr> ) <stmt>}
\]
Example Derivation

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

\[
\begin{align*}
\text{program} &::= \text{statement} \mid \text{program statement} \\
\text{statement} &::= \text{assignStmt} \mid \text{ifStmt} \\
\text{assignStmt} &::= \text{id} = \text{expr} ; \\
\text{ifStmt} &::= \text{if} ( \text{expr} ) \ \text{statement} \\
\text{expr} &::= \text{id} \mid \text{int} \mid \text{expr} + \text{expr} \\
\text{id} &::= a \mid b \mid c \mid i \mid j \mid k \mid n \mid x \mid y \mid z \\
\text{int} &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]
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
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
  ```
  // 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
  - But sometimes whitespace does matter
    Examples: Python indentation, Ruby newlines
Parser Example

- Token Stream Input

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

- Abstract Syntax Tree

![Diagram of Abstract Syntax Tree]

```
ifStmt

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

CSE413 Winter 2019
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) to represent 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 the actual identifier string as an attribute of the token “identifier”
Typical Programming Language Tokens

- 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 (doubles, strings, …), 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 != dowhile;
```

should be recognized as 5 tokens:

- `RETURN`
- `ID(iffy)`
- `NEQ`
- `ID(dowhile)`
- `SCOLON`

not more (i.e., not parts of words or identifiers, not `!` 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
  – Not always, e.g., FORTRAN and some others, but can usually cheat in the unusual corner cases
• Tokens can be recognized by a deterministic finite automaton (DFA)
  – Can be either table-driven or built by hand based on lexical grammar

• Facts (er, theorems): any language that can be generated by a regexp can be recognized by the corresponding DFA; for every DFA, there is a set of regular expressions that generate the language it recognizes
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
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*r)</td>
<td>1 or more occurrences</td>
</tr>
<tr>
<td>r?</td>
<td>(r</td>
<td>(\varepsilon))</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><code>re</code></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

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

– 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

  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 recognition 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
  - Difference in a scanner: start scan in initial state at previous point in input. When a final state is reached, recognize the token corresponding to that final state
- 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 (subset construction)
  - See any formal language or compiler textbook 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 remembers current position in input file
  - Starting there, use a DFA to recognize the longest possible input sequence that makes up a token, update the “current position”, and return that token
Scanner DFA Example (1)

0

whitespace or comments

end of input

1

Accept EOF

2

Accept LPAREN

3

Accept RPAREN

4

Accept SCOLON
Scanner DFA Example (2)

![Diagram showing the DFA for 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 with appropriate transitions to recognize keywords
    - Lots ’o states, but efficient (no extra lookup step)
A token is a simple, tagged structure. Something like:

```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 (should really be an enum type)
    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() pseudocode

// 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 !=
        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. ...
```
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)