More Finite Automata/ Lexical Analysis /Introduction to Parsing

Lecture 7
Programming a lexer in Lisp “by hand”

• (actually picked out of comp.lang.lisp when I was teaching CS164 3 years ago, an example by Kent Pitman).

• Given a string like "foo+34-bar*g(zz)" we could separate it into a lisp list of strings:
  ("foo" "+" "34" ...) or we could try for a list of Lisp symbols like (foo + 34 - bar * g | ( | zz | ) |).

  Huh? What is |(|? It is the way lisp prints the symbol with printname "(" so as to not confuse the Lisp read program, and humans too.
Set up some data and predicates

(defvar *whitespace* '#\Space #\Tab #\Return #\Linefeed)

(defun whitespace? (x) (member x *whitespace*))

(defvar *single-char-ops* '#\+ #\- #\* #\/ #\( #\) #\. #\, #\=)

(defun single-char-op? (x) (member x *single-char-ops*))
Tokenize function...

(defun tokenize (text) ;; text is a string "ab+cd(x)"
  (let ((chars '()) (result '()))
    (declare (special chars result)) ;; explain scope
    (dotimes (i (length text))
      (let ((ch (char text i))) ;; pick out ith character of string
        (cond ((whitespace? ch)
                  (next-token))
              ((single-char-op? ch)
               (next-token)
               (push ch chars)
               (next-token))
              (t
               (push ch chars))))
      (next-token)
    (nreverse result))
)
Next-token / two versions

(defun next-token () ;; simple version
  (declare (special chars result))
  (when chars
    (push (coerce (nreverse chars) 'string) result)
    (setf chars '())))

(defun next-token () ;; this one "parses" integers magically
  (declare (special chars result))
  (when chars
    (let((st (coerce (reverse chars) 'string))) ; keep chars around to test
      (push (if (every #'digit-char-p chars)
                  (read-from-string st)
                (intern st))
          result))
    (setf chars '())))
Example

- (tokenize "foo(-)+34") ➔ (foo | (| - |) | + 34)

- (Much) more info in file: pitmantoken.cl

- Missing: line/column numbers, 2-char tokens, keyword vs. identifier distinction. Efficiency here is low (but see file for how to use hash tables for character types!)

- Also note that Lisp has a programmable read-table so that its own idea of what delimits a token can be changed, as well as meanings of every character.
Introduction to Parsing
Outline

• Regular languages revisited

• Parser overview

• Context-free grammars (CFG’s)

• Derivations
Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages

- Regular languages
  - The weakest class of formal languages widely used
  - Many applications

- We will also study context-free languages
Limitations of Regular Languages

• Intuition: A finite automaton with N states that runs N+1 steps must revisit a state.
• Finite automaton can’t remember # of times it has visited a particular state. No way of telling how it got here.
• Finite automaton can only use finite memory.
  - Only enough to store in which state it is
  - Cannot count, except up to a finite limit
• E.g., language of balanced parentheses is not regular: \{ (^i )^i \mid i > 0 \}
Context Free Grammars are more powerful

- Easy to parse balanced parentheses and similar nested structures
- A good fit for the vast majority of syntactic structures in programming languages like arithmetic expressions.
- Eventually we will find constructions that are not CFG, or are more easily dealt with outside the parser.
The Functionality of the Parser

- **Input**: sequence of tokens from lexer
- **Output**: parse tree of the program
Example

- **Program Source**

  ```
  if (x < y) a = 1; else a = 2;
  ```

  Lex output = parser input (simplified)

  ```
  IF lpar ID < ID rpar ID = ICONST; ID = ICONST ICONST
  ```

- **Parser output (simplified)**

  ```
  IF-THEN-ELSE
      IF  ID < ID ASSIGN ID ASSIGN
          < ID ID
  ```
Example

• **MJ Source**

```plaintext
if (x<y) a = 1; else a = 2;
```

• Actual lex output (from lisp...)

```plaintext
(fstring " if (x<y) a=1; else a=2;") →
```

```plaintext
(if if (1 . 10))
(\( 1 . 12))
(id x (1 . 13))
(\< 1 . 14))
(id y (1 . 15))
(\) (1 . 16))
(id a (1 . 18))
(\= (1 . 19))
(iconst 1 (1 . 20))
(\; (1 . 21))
(else else (1 . 26)) ...
```
**Example**

- **MJSource**
  
  ```
  if (x < y) a=1; else a=2;
  ```

- **Actual Parser output ; lc = line&column**
  ```
  (If (LessThan (IdentifierExp x) (IdentifierExp y))
   (Assign (id a lc) (IntegerLiteral 1))
   (Assign (id a lc) (IntegerLiteral 2)))
  ```

  - Or cleaned up by taking out “extra” stuff...
    ```
    (If (< x y) (assign a 1)(assign a 2))
    ```
## Comparison with Lexical Analysis

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexer</td>
<td>Sequence of characters</td>
<td>Sequence of tokens</td>
</tr>
<tr>
<td>Parser</td>
<td>Sequence of tokens</td>
<td>Parse tree</td>
</tr>
</tbody>
</table>
The Role of the Parser

• Not all sequences of tokens are programs . . .
• . . . Parser must distinguish between valid and invalid sequences of tokens
• Some sequences are valid only in some context, e.g. MJ requires framework.

• We need
  - A formal technique $G$ for describing exactly and only the valid sequences of tokens (i.e. describe a language $L(G)$)
  - An “implementation” of a recognizer for $L$, preferably based on automatically transforming $G$ into a program. $G$ for grammar.
A test framework for trivial MJ line of code

class Test {
    public static void main(String[] S){
        {}
    }
}

class fooClass {
    public int aMethod(int value) {
        int a;
        int x;
        int y;

        if (x<y) a=1; else a=2;
        return 0;
    }
}
Context-Free Grammars: Why

- Programming language constructs often have an underlying recursive structure

- An \texttt{EXPR} is \texttt{EXPR + EXPR}, ..., or
  A statement is if \texttt{EXPR} statement; else statement, or
  while \texttt{EXPR} statement

- Context-free grammars are a natural notation for this recursive structure
Context-Free Grammars: Abstractly

- A CFG consists of
  - A set of *terminals* \( T \)
  - A set of *non-terminals* \( N \)
  - A *start symbol* \( S \) (a non-terminal)
  - A set of *productions*, or PAIRS of \( N \times (N \cup T)^* \)

Assuming \( X \in N \)

\[
X \rightarrow \varepsilon \quad \text{, or} \quad X \rightarrow Y_1 Y_2 \ldots Y_n
\]

where \( Y_i \in N \cup T \)
Notational Conventions

• In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production
    $\epsilon$ production; vaguely related to same symbol in RE.
    $X \rightarrow \epsilon$ means there is a rule by which $X$ can be replaced by “nothing”
Examples of CFGs

A fragment of MiniJava

STATE → if ( EXPR ) STATE;
STATE → LVAL = EXPR
EXPR → id
Examples of CFGs

A fragment of MiniJava

STATE → if ( EXPR ) STATE;
  |  LVAL = EXPR

EXPR → id

*Shorthand notation with /.*
Examples of CFGs (cont.)

Simple arithmetic expression language:

\[ E \rightarrow E \ast E \]
\[ \mid E + E \]
\[ \mid (E) \]
\[ \mid \text{id} \]
The Language of a CFG

Read productions as replacement rules in generating sentences in a language:

\[ X \rightarrow Y_1 ... Y_n \]

Means \( X \) can be replaced by \( Y_1 ... Y_n \)

\[ X \rightarrow \varepsilon \]

Means \( X \) can be erased (replaced with empty string)
Key Idea

1. Begin with a string consisting of the start symbol “S”
2. Pick a non-terminal \( X \) in the string by a right-hand side of some production e.g. \( X \rightarrow YZ \)
   \( \ldots \text{string1} \ X \ \text{string2}\ldots \Rightarrow \ldots \text{string1} \ YZ \ \text{string2} \ \ldots \)

1. Repeat (2) until there are no non-terminals in the string. i.e. do \( \Rightarrow^* \)
The Language of a CFG (Cont.)

More formally, write

\[ X_1 \ldots X_i \ldots X_n \Rightarrow X_1 \ldots X_{i-1} Y_1 Y_2 \ldots Y_m X_{i+1} \ldots X_n \]

if there is a production

\[ X_i \rightarrow Y_1 Y_2 \ldots Y_m \]

Note, the double arrow denotes rewriting of strings is \( \Rightarrow \)
The Language of a CFG (Cont.)

Write \( u \Rightarrow^* v \)

If \( u \Rightarrow \ldots \Rightarrow v \)

in 0 or more steps
The Language of a CFG

Let $G$ be a context-free grammar with start symbol $S$. Then the language of $G$ is:

$$\{a_1 \ldots a_n \mid S \Rightarrow a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal symbol}\}$$
Terminals

Terminals are called that because there are no rules for replacing them. (terminated..)

- Once generated, terminals are permanent.

- Terminals ought to be tokens of the language, numbers, ids, not concepts like “statement”.

Prof. Fateman  CS 164  Lecture 7
Examples

$L(G)$ is the language of CFG $G$

Strings of balanced parentheses $\left\{ (i)^i \mid i \geq 0 \right\}$

A simple grammar:

\[
S \rightarrow (S)
\]
\[
S \rightarrow \varepsilon
\]
To be more formal..

- The alphabet $\Sigma$ for $G$ is $\{ ( , ) \}$, the set of two characters left and right parenthesis. This is the set of terminal symbols.
- The non-terminal symbols, $N$ on the LHS of rules is here, a set of one element: $\{S\}$
- There is one distinguished non-terminal symbol, often $S$ for “sentence” or “start” which is what you are trying to recognize.
- And then there is the finite list of rules or productions, technically a subset of $N \times (N \cup \Sigma)^*$
Let’s produce some sentential forms of a MJgrammar

A fragment of a Tiger grammar:

\[
\text{STATE } \rightarrow \text{ if ( EXPR ) STATE ; else STATE} \\
| \text{ while EXPR do STATE} \\
| \text{ id}
\]
MJ Example (Cont.)

Some *sentential forms* of the language

id

if (expr) state; else state

while id do state;

if if id then id else id then id else id
Arithmetic Example

Simple arithmetic expressions:

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid id \]

Some elements of the language:

\[ \begin{align*} 
\text{id} & \quad \text{id + id} \\
(id) & \quad \text{id * id} \\
(id) \ast \text{id} & \quad \text{id * (id)} 
\end{align*} \]
The CFG idea for describing languages is a powerful concept. Understanding its complexities can solve many important Programming Language problems.

- **Membership in a CFG’s language is “yes” or “no”**.
- **But to be useful to us, a CFG parser**
  - Should show how a sentence corresponds to a parse tree.
  - Should handle non-sentences gracefully (pointing out likely errors).
  - Should be easy to generate from the grammar specification “automatically” (e.g., YACC, Bison, JCC, LALR-generator)
More Notes

- Form of the grammar is important
  - Different grammars can generate the identical language
  - Tools are sensitive to the form of the grammar
  - Restrictions on the types of rules can make automatic parser generation easier
Simple grammar  (3.1 in text)

1:  S → S ; S
2:  S → id := E
3:  S → print (L)
4:  E → id
5:  E → num
6:  E → E + E
7:  E → (S , E)
8:  L → E
9:  L → L , E
A derivation is a sequence of sentential forms starting with S, rewriting one non-terminal each step. A left-most derivation rewrites the left-most non-terminal.

Using rules

- $S$  
- id := E  
- id := E + E  
- id := num + E  
- id := num + num

The sequence of rules tells us all we need to know! We can use it to generate a tree diagram for the sentence.
Building a Parse Tree

- Start symbol is the tree's root
- For a production $X \rightarrow y_1 y_2 y_3$ we draw

```
     X
   /   \
  /     \
y1     y2
  |      |   \
  y3     y3
```
Another Derivation Example

• Grammar Rules

\[ E \rightarrow E + E \mid E \times E \mid (E) \mid id \]

• Sentential Form (input to parser)

\[ id \times id + id \]
Derivation Example (Cont.)

\[ E \]

\[ \rightarrow E + E \]

\[ \rightarrow E \ast E + E \]

\[ \rightarrow id \ast E + E \]

\[ \rightarrow id \ast id + E \]

\[ \rightarrow id \ast id + id \]
Left-Most Derivation in Detail (1)
Derivation in Detail (2)

\[ E \rightarrow E + E \]
Derivation in Detail (3)

E → E+E
E → E*E+E
Derivation in Detail (4)

\[
E 
\rightarrow E + E
\]
\[
E \rightarrow E \ast E + E
\]
\[
E \rightarrow id \ast E + E
\]
Derivation in Detail (5)

\[ E \]

\[ \rightarrow E + E \]

\[ \rightarrow E \ast E + E \]

\[ \rightarrow \text{id} \ast E + E \]

\[ \rightarrow \text{id} \ast \text{id} + E \]

Diagram:

```
      E
     / \
    /   \
   E   +
    |   |
   /   |
  E   E
```

\[ \text{id} \]

\[ \text{id} \]
Derivation in Detail (6)

E → E + E
E → E * E + E
id * E + E → id * id + E
id * id + E → id * id + id
Notes on Derivations

• A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

• An in-order traversal of the leaves is the original input

• The parse tree shows the association of operations, even if the input string does not
What is a Right-most Derivation?

- Our examples were left-most derivations
  - At each step, replace the left-most non-terminal

- There is an equivalent notion of a right-most derivation

\[
\begin{align*}
E & \rightarrow E+E \\
& \rightarrow E+id \\
& \rightarrow E*E + id \\
& \rightarrow E*id + id \\
& \rightarrow id*id + id
\end{align*}
\]
Right-most Derivation in Detail (1)
Right-most Derivation in Detail (2)

\[ E + E \rightarrow E + E \]
Right-most Derivation in Detail (3)

\[
E 
\rightarrow E + E
\rightarrow E + \text{id}
\]
Right-most Derivation in Detail (4)

\[ E \rightarrow E + E \]
\[ \rightarrow E + id \]
\[ \rightarrow E * E + id \]
Right-most Derivation in Detail (5)

\[ E \rightarrow E + E \]
\[ E + id \rightarrow E * E + id \]
\[ E * id + id \]
Right-most Derivation in Detail (6)

$E$  $\rightarrow$  $E + E$
$\quad$  $\rightarrow$  $E + id$
$\quad$  $\rightarrow$  $E * E + id$
$\quad$  $\rightarrow$  $E * id + id$
$\quad$  $\rightarrow$  $id * id + id$
Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree

- The difference is the order in which branches are added
Summary: Objectives of Parsing

• We are not just interested in whether $s \in L(G)$
  - We need a parse tree for $s$

• A derivation defines a parse tree
  - But one parse tree may have many derivations

• Left-most and right-most derivations are important in parser implementation
The simplest way of handling this is to write a program to just suck up characters looking for *//, and “count backwards”.

Here’s an attempt at a grammar

- \( C \rightarrow */A */ \)
- \( C \rightarrow */ A C A */ \)
- \( A \rightarrow a | b | c | 0 | \ldots 9 | \ldots \text{all chars not } / \)
- \( B \rightarrow a | b | c | 0 | \ldots 9 | \ldots \text{all chars not } * \)
- \( A \rightarrow A B1 | A1 B1 A B1 A1 \mid \varepsilon \)

--To make this work, you’d need to have a grammar that covered both “real programs” and comments concatenated.