Lecture 3 Outline
- Informal introduction to lexical analysis
- Examples of problems in lexical analysis
  - look ahead
  - ambiguities
- Specification: Regular expressions
  - Examples of regular expressions
- Implementation
  - specify lexical structures using regular expression
  - using finite state automata

Reading Assignment:
- Section 3.5

Lexical Analysis
- What do we want to do? Example:
  if \( i == j \)
  \( Z = 0; \)
  else
  \( Z = 1; \)
- The input is just a string of characters:
  \( \text{\textbackslash if} \ (i == j) \text{\textbackslash i} Z = 0; \text{\textbackslash else} \text{\textbackslash i} Z = 1; \)
- Goal: Partition input string into substrings
  where the substrings are tokens

What's a Token?
- A syntactic category
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - Identifier, Integer, Keyword, Whitespace, ...
- Tokens correspond to sets of strings.
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: "else" or "if" or "begin" or ...
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs
What are Tokens For?

- Classify program substrings according to role.
- Output of lexical analysis is a stream of tokens...
- ...which is input to the parser.
- Parser relies on token distinctions.
  - An identifier is treated differently than a keyword.

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens.
  - Tokens describe all items of interest.
  - Choice of tokens depends on language, design of parser.
- Recall:
  \[ \text{if} \text{ if} \text{ else} = \text{ if} \text{ else} \text{ if} \text{ = } 1; \]
- Useful tokens for this expression:
  - Integer, Keyword, Relation, Identifier, WhiteSpace, { }, *, ;
- { }, *, ; are tokens, not characters, here.

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token.
- Recall:
  - Identifier: strings of letters or digits, starting with a letter.
  - Integer: a non-empty string of digits.
  - Keyword: "else" or "if" or "begin" or ..
  - WhiteSpace: a non-empty sequence of blanks, newlines, and tabs.
Lexical Analyzer: Implementation

- An implementation must do two things:

  1. Recognize substrings corresponding to tokens

     2. Return the value or lexeme of the token

        - The lexeme is the substring

Example

- Recall:

  ```
  \#define \n  \n  
  \n  = 0; \n  \n  \n  = 1;
  ```

  - Token-lexeme groupings:
    - Identifier: i, j, z
    - Keyword: if, else
    - Relation: ==
    - Integer: 0, 1
    - ( ) [ ], : single character of the same name

Lexical Analyzer: Implementation

- The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.

- Examples: Whitespace, Comments

- Sounds easy but there are problems!!
Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant - rule motivated by inaccuracy of punch card operators
- Consider
  - DO 5 I = 1, 25
  - DO 5 I = 1.25
- The first is DO 5 I = 1 , 25
- The second is DO5 = 1.25
- Reading left-to-right, cannot tell if DO5 is a variable or DO stmt. until after ",," is reached

Lexical Analysis in FORTRAN (Cont.)

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-write, recognizing one token at a time
  2. "Lookahead" may be required to decide where one token ends and the next token begins

Lexical Analysis in PL/I

- PL/I keywords are not reserved
  IF ELSE THEN THEN = ELSE ELSE ELSE = THEN
Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:
  DECLARE (ARG1, ..., ARGN)

- Can't tell whether DECLARE is a keyword or array reference until after the `);`
  - Requires arbitrary lookahead

- More on PL/I's quirks later in the course . . .

Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax:
  ```cpp
  Foo<Bar>
  ```

- C++ stream syntax:
  ```cpp
  cin >> var;
  ```

- But there is a conflict with nested templates:
  ```cpp
  Foo<Bar<Bazz>>
  ```

Regular languages

- We still need
  - A way to describe the lexemes of each token

- There are several formalisms for specifying tokens

- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations
Languages

Def. Let $\Sigma$ be a set of characters. A language over $\Sigma$ is a set of strings of characters drawn from $\Sigma$.

Examples of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string on English characters is an English sentence

- Alphabet = ASCII
- Language = C programs
- Note: ASCII character set is different from English character set

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want.
- The standard notation for regular languages is regular expressions.
Atomic Regular Expressions

- Single character
  \[ c = \{ c \} \]
- Epsilon
  \[ \varepsilon = \{ \varepsilon \} \]

Compound Regular Expressions

- Union
  \[ A + B = \{ s \mid s \in A \text{ or } s \in B \} \]
- Concatenation
  \[ AB = \{ ab \mid a \in A \text{ and } b \in B \} \]
- Iteration
  \[ A^* = \bigcup_{i=0}^\infty A^i \text{ where } A^i = A \ldots \text{i times} \ldots A \]

Regular Expressions

- Def. The regular expressions over \( \Sigma \) are the smallest set of expressions including
  - \( \varepsilon \)
  - \( c \) where \( c \in \Sigma \)
  - \( A + B \) where \( A, B \) are rexp over \( \Sigma \)
  - \( AB \) where \( A, B \) are rexp over \( \Sigma \)
  - \( A^* \) where \( A \) is a rexp over \( \Sigma \)
Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics.
  \[ L(e) = \{ "e" \} \]
  \[ L(c) = \{ "c" \} \]
  \[ L(A + B) = L(A) \cup L(B) \]
  \[ L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \} \]
  \[ L(A^+) = \bigcup_{i \geq 0} L(A^i) \]

Example: Keyword

Keyword: "else" or "if" or "begin" or ...
  'else' + 'if' + 'begin' + ...

Note: 'else' abbreviates 'e"l"s"e'

Example: Integers

Integer: a non-empty string of digits
  digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
  integer = digit digit'

Abbreviation: \( A^+ - AA^0 \)
Example: Identifier

Identifier: strings of letters or digits, starting with a letter

letter = 'A' +...+ 'Z' + 'a' +...+ 'z'
identifier = letter (letter + digit)∗

Is (letter + digit) the same?

Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

(' ' + \n' + \t)∗

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (412)–624–8425

Σ = digits ∪ {.,(,)}
exchange = digit
phone = digit
area = digit
phone_number = '('area ')' exchange '-' phone
Example: Email Addresses

Consider soffe@cs.pitt.edu

\[ \Sigma = \text{letters } \cup \{,.@\} \]

name = letter^*

address = name '@' name '.' name '.' name

Implementation of lexical analysis (tool then theory)

Can implement lexical analysis using a tool lex (for C), flex (can be used for C & C++) and jflex (for Java)

- Specify tokens using a form of regular expressions
- Tool generates source code for the lexical analysis

Can use regular expressions to develop a finite state automaton

- Write code to express the recognition of tokens
- Table driven

Implementation of lexical analysis

Can implement lexical analysis using a tool lex (for C), flex (can be used for C & C++) and jflex (for Java)

- Specify tokens using a form of regular expressions
- Tool generates source code for the lexical analysis

Can use regular expressions to develop a finite state automaton

- Write code to express the recognition of tokens
- Table driven
Language for Specifying Lexical Analysis

(flex)+ tool that uses specifications to automatically produce lexical analyzer

1. Tabular representation of a transition diagram constructed from regular expressions of flex
2. Standard routine that uses the table to recognize lexemes
3. Actions associated with regular expressions in input

Flex Specifications

declarations
decides variables, constants & regular definitions, e.g.
digit [0-9]

% %
transition rules
regular expressions and actions
R1 (action)
R2 (action)
where actions are program fragments written in C++ actions that flex should take when pattern Ri matches lexeme
% %
auxiliary procedures
needed by the actions

% } extern int curr_line, yycolumn, YYSTYPE; yylval; # include token.h
% )
digit [0-9]

number [0-9];
% %
[number] (yylval.symbol = int_table.add_string(yylex, yyleng);
return(INT_CONST)

[a-z][a-z 0-9];

(yylval.symbol = id_table.add_string(yylex, yyleng);
return(IDENTID))
% %
Implementation Notes:
Write regular expressions for all/some of tokens

- comments: keep track of nesting level
- String table written for you
- yynline, yyncolumn maintained by you

yytext and yynlength maintained by flex

Special characters:
- \" - newline
- \t - tab
- \' - single quote
- \'\\ - backslash

Regular Languages
- many applications
- weakest formal languages used
- many languages are not regular
- strings of balanced parentheses
  (((((((...))))))
- no regular language for this language
- finite automaton cannot remember # of times
- it has been to a particular state

Need language for describing valid strings in a language
Another type of language: Context-free language

Parser:
Input: sequence of tokens from lexical analysis
Output: a parse tree of the program (or something like it)

Summarizing
- Regular expressions describe many useful languages
- Regular languages are a language specification
  - we still need an implementation
  
Now: Given a string $s$ and a regex $R$, is

$s \in L(R)$?
Implementation

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NDFAs)
- Implementation of regular expressions
  \[ \text{RegExp} \Rightarrow \text{NFA} \Rightarrow \text{DFA} \Rightarrow \text{Tables} \]

Notation

- There is a variation in regular expression notation
  - Union: \( A \mid B \) \( \Rightarrow \) \( A \lor B \)
  - Option: \( A + \epsilon \) \( \Rightarrow \) \( A^* \)
  - Range: \( 'a'+'b'+...+'z' \) \( \Rightarrow \) \( [a-z] \)
  - Excluded range:
    - complement of \([a-z]\) \( \Rightarrow \) \([^a-z]\)

Regular Expressions in Lexical Specification

- A specification for the predicate \( s \in L(R) \)
  - But a yes/no answer is not enough!
  - Instead: partition the input into tokens
- We will adapt regular expressions to this goal
Regular Expressions => Lexical Spec. (1)

1. Write a rexp for the lexemes of each token
   - Number = digit +
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenFor = '{'
   - ...

Regular Expressions => Lexical Spec. (2)

2. Construct R, matching all lexemes for all tokens
   \[ R = \text{Keyword} \cdot \text{Identifier} \cdot \text{Number} + \ldots \]
   \[ = R_1 \cdot R_2 \cdot \ldots \]

Regular Expressions => Lexical Spec. (3)

3. Let input be \( X_1 \ldots X_n \)
   - For \( 1 < i < n \) check
     \( X_{i-1} \cdot X_i \in L(R) \)

4. It must be that
   \( X_{j-1} \cdot X_j \in L(R) \) for some \( j \)

5. Remove \( X_{j-1} \cdot X_j \) from input and go to (3)
Ambiguities (1)

- There are ambiguities in the algorithm
- How much input is used? What if
  - $x_1...x_k \in L(R)$ and also
  - $x_1...x_k \in L(R')$
  - Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"

Ambiguities (2)

- Which token is used? What if
  - $x_1...x_k \in L(R)$ and also
  - $x_1...x_k \in L(R')$
  - Rule: use rule listed first (j if $j < k$)
  - Treats "if" as a keyword not an identifier

Error Handling

- What if
  - No rule matches a prefix of input?
- Problem: Can't just get stuck ...
- Solution:
  - Write a rule matching all "bad" strings
  - Put it last
Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $s_0$
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\delta: S \times \Sigma \rightarrow S$

Finite Automata

- Transition $s_1 \xrightarrow{a} s_2$
- Is read
  - In state $s_1$ on input "a" go to state $s_2$
- If end of input (or no transition possible)
  - If in accepting state $\Rightarrow$ accept
  - Otherwise $\Rightarrow$ reject
Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition

A Simple Example

- A finite automaton that accepts only "1"

Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: \( \{0, 1\} \)
And Another Example

- Alphabet \((0, 1)\)
- What language does this recognize?

![Diagram of a finite automaton]

Epsilon Moves

- Another kind of transition: e-moves

![Diagram of a transition with epsilon]

- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No e-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have e-moves
- Finite automata have finite memory
  - Need only to encode the current state
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make $\epsilon$-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states

![Diagram of NFA]

- Input: 1 0 0
- Rule: NFA accepts if it can get in a final state

NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are faster to execute
  - There are no choices to consider
NFA vs. DFA (2)

- For a given language NFA can be simpler than DFA
- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

- High-level sketch
  - Regular expressions
  - Lexical Specification
- NFA
- DFA
- Table-driven Implementation of DFA

NFA to DFA. The Trick

- Simulate the NFA
- Each state of DFA
  - a non-empty subset of states of the NFA
- Start state
  - the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$, considering ε-moves as well
Example NFA to DFA

NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?
- \(2^N - 1\) = finitely many

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "state"
  - Other dimension is "input symbol"
  - For every transition \(S_i \rightarrow S_k\), define \(T[S_i, a] = k\)
- DFA "execution"
  - If in state \(S_i\) and input \(a\), read \(T[S_i, a] = k\) and skip to state \(S_k\)
  - Very efficient
Implementation (Cont.)

- NFA $\rightarrow$ DFA conversion is at the heart of tools such as flex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations