Scanners

Sometimes called lexers
Recall: scanners break input stream up into a set of tokens
- Identifiers, reserved words, literals, etc.
- What do we need to know?
  - How do we define tokens?
  - How can we recognize tokens?
  - How do we write scanners?

Regular expressions

- Regular sets: set of strings defined by regular expressions
- Strings are regular sets (with one element): `purdue 3.14159`
  - So is the empty string: `λ` (sometimes use `ε` instead)
- Concatenations of regular sets are regular: `purdue 3.14159`
  - To avoid ambiguity, can use `()` to group regexps together
- A choice between two regular sets is regular: `purdue | 3.14159`
- 0 or more of a regular set is regular, using `*`: `(purdue) *`
- Some other notation used for convenience:
  - Use `Not` to accept all strings except those in a regular set
  - Use `?` to make a string optional: `x ?` equivalent to `(x | λ)`
  - Use `+` to mean 1 or more strings from a set: `x +` equivalent to `x x *`
  - Use `[]` to present a range of choices: `[1-3]` equivalent to `(1 | 2 | 3)`

Examples of regular expressions

- Numbers: `D = [0-9] +`
- Words: `L = [A-Za-z] +`
- Literals (integers or floats): `-? D + (.D |) ?`
- Identifiers: `(_ | L) (_.L | D) *`
- Comments (as in Micro): `-- Not(\n) * \n`
- More complex comments (delimited by `##`, can use `#` inside comment): `##(##(\# | Not(\#)) ##``##`

Finite automata

- Finite state machine which will only accept a string if it is in the set defined by the regular expression
- Transitions between states that aren’t triggered by seeing another character
  - Can optionally take the transition, but do not have to
  - Can be used to link states together

\( \lambda \) transitions
Non-deterministic FA

- Note that if a finite automaton has a \(\lambda\)-transition in it, it may be non-deterministic (do we take the transition? or not?)
- More precisely, FA is non-deterministic if, from one state reading a single character could result in transition to multiple states
- How do we deal with non-deterministic finite automata (NFAs)?

“Running” an NFA

- Intuition: take every possible path through an NFA
- Think: parallel execution of NFA
- Maintain a “pointer” that tracks the current state
- Every time there is a choice, “split” the pointer, and have one pointer follow each choice
- Track each pointer simultaneously
  - If a pointer gets stuck, stop tracking it
  - If any pointer reaches an accept state at the end of input, accept

Example

- How does this NFA handle the string “aba”?

Building a FA from a regexp

<table>
<thead>
<tr>
<th>Expression</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>![FA Diagram for a]</td>
</tr>
<tr>
<td>(\lambda)</td>
<td>![FA Diagram for (\lambda)]</td>
</tr>
<tr>
<td>AB</td>
<td>![FA Diagram for AB]</td>
</tr>
<tr>
<td>A(\mid)B</td>
<td>![FA Diagram for A(\mid)B]</td>
</tr>
<tr>
<td>A(\ast)</td>
<td>![FA Diagram for A(\ast)]</td>
</tr>
</tbody>
</table>

Mini-exercise: how do we build an FA that accepts Not(A)?

NFAs to DFAs

- Can convert NFAs to deterministic finite automata (DFAs)
- No choices — never a need to “split” pointers
- Initial idea: simulate NFA for all possible inputs, any time there is a new configuration of pointers, create a state to capture it
- Pointers at states 1, 3 and 4 \(\rightarrow\) new state \(\{1, 3, 4\}\)
- Trying all possible inputs is impractical; instead, for any new state, explore all possible next states (that can be reached with a single character)
- Process ends when there are no new states found
- This can result in very large DFAs!

Example

- Convert the following into a DFA

Monday, August 26, 13
DFA reduction

- DFAs built from NFAs are not necessarily optimal
- May contain many more states than is necessary

\[(ab)^+ = (ab)(ab)^*\]

DFA reduction

- Intuition: merge equivalent states
  - Two states are equivalent if they have the same transitions to the same states
  - Basic idea of optimization algorithm
    - Start with two big nodes, one representing all the final states, the other representing all other states
    - Successively split those nodes whose transitions lead to nodes in the original DFA that are in different nodes in the optimized DFA

Example

- Simplify the following

Transition tables

- Table encoding states and transitions of FA
  - 1 row per state, 1 column per possible character
  - Each entry: if automaton in a particular state sees a character, what is the next state?

Finite automata program

- Using a transition table, it is straightforward to write a program to recognize strings in a regular language

```c
state = initial_state; //start state of FA
while (true) {
    next_char = getc();
    if (next_char == EOF) break;
    next_state = T[state][next_char];
    if (next_state == ERROR) break;
    state = next_state;
}
if (is_final_state(state)) //recognized a valid string
else
    handle_error(next_char);
```
Alternate implementation

- Here’s how we would implement the same program “conventionally”

```c
next_char = getc();
while (next_char == 'a') {
    next_char = getc();
    if (next_char != 'b') handle_error(next_char);
    next_char = getc();
    if (next_char != 'c') handle_error(next_char);
    while (next_char == 'c') {
        next_char = getc();
        if (next_char == EOF) return; //matched token
        if (next_char != 'a') break;
        if (next_char != 'c') handle_error(next_char);
    }
}
handle_error(next_char);
```

Practical Considerations

Or: what do I have to worry about if I’m actually going to write a scanner?

Handling reserved words

- Keywords can be written as regular expressions. However, this leads to a big blowup in FA size
- Consider writing a regular expression that accepts identifiers which cannot be if, while, do, for, etc.
- Usually better to specify reserved words as “exceptions”
- Capture them using the identifier regex, and then decide if the token corresponds to a reserved word

Lookahead

- Up until now, we have only considered matching an entire string to see if it is in a regular language
- What if we want to match multiple tokens from a file?
- Distinguish between `int a` and `inta`
- We need to look ahead to see if the next character belongs to the current token
- If it does, we can continue
- If it doesn’t, the next character becomes part of the next token

Multi-character lookahead

- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens
- Examples
  - Fortran: `DO I = 1,100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Pascal: `23.85` (literal) vs. `23..85` (range)
- 2 solutions: Backup or special “action” state

Multi-character lookahead

- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens
- Examples
  - Fortran: `DO I = 1,100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Pascal: `23.85` (literal) vs. `23.85` (range)
- 2 solutions: Backup or special “action” state
General approach

- Remember states (T) that can be final states
- Buffer the characters from then on
- If stuck in a non-final state, back up to T, restore buffered characters to stream
- Example: 12.3e+q

Why can’t we do this?

- Just build an FA which recognizes the string
  \[D+(\lambda|D+)[.]D+(\lambda|D+)\] and recognize the final state we are in to determine the token type?
- Note that this will recognize tokens of the form 12.3 and 12..3

Error Recovery

- What do we do if we encounter a lexical error (a character which causes us to take an undefined transition)?

  Two options
  - Delete all currently read characters, start scanning from current location
  - Delete first character read, start scanning from second character

  This presents problems with ill-formatted strings (why?)
  - One solution: create a new regexp to accept runaway strings

Scanner Generators

- Lex (Flex)
  - Commonly used Unix scanner generator (superseded by Flex)
  - Flex is a domain specific language for writing scanners
  - Features:
    - Character classes: define sets of characters (e.g., digits)
    - Token definitions: regex {action to take}

Scanner generators

- Essentially, tools for converting regular expressions into finite automata
- Two popular scanner generators
  - Lex (Flex): generates C/C++ scanners
  - ANTLR: generates Java scanners
Lex (Flex)

```c
DIGIT  {0-9}
ID    {a-zA-Z}[a-zA-Z0-9]*
%

{DIGIT}+ {
    printf( "An integer: %s (%d)\n", yytext, atoi( yytext ) );
}

{DIGIT}+.{DIGIT}+ {
    printf( "A float: %s (%g)\n", yytext, atof( yytext ) );
}

if|then|begin|end|procedure|function {
    printf( "A keyword: %s\n", yytext );
}

(ID)  printf( "An identifier: %s\n", yytext );
```

Lex (Flex)

- The order in which tokens are defined matters!
- Lex will match the longest possible token
- "ifa" becomes ID(ida), not IF ID(a)
- If two regexes both match, Lex uses the one defined first
- "if" becomes IF, not ID(if)
- Use action blocks to process tokens as necessary
- Convert integer/float literals to numbers
- Remove quotes from string literals

Lex (Flex)

- Compile lex file to C code
  - Example of compiling high-level language to another high-level language!
- Compile generated scanner to produce working scanner
- Combine with yacc/bison to produce parser

ANTLR

- More powerful tool than Lex (can generate parsers, too, not just scanners)
- Same basic principles
  - Tokens:
    - Token definition: tokenName : regex1 | regex2 | ...
  - Character classes:
    - Look similar to token definitions
    - fragment characterClassName : regex1 | regex2 ...
    - Can use character classes when defining tokens

Next Time

- We’ve covered how to tokenize an input program
- But how do we decide what the tokens actually say?
  - How do we recognize that
    IF ID(a) OP(<) ID(b) { ID(a) ASSIGN LIT(5) ; }
    is an if-statement?
  - Next time: Parsers