01-0: Syllabus

- Office Hours
- Course Text
- Prerequisites
- Test Dates & Testing Policies
- Projects
  - Teams of up to 2
- Grading Policies
- Questions?
Don’t be afraid to ask me to slow down!

We will cover some pretty complex stuff here, which can be difficult to get the first (or even the second) time. **ASK QUESTIONS**

While specific questions are always preferred, “I don’t get it” is always an acceptable question. I am always happy to stop, re-explain a topic in a different way.

- If you are confused, I can *guarantee* that at least one other person in the class would benefit from more explanation.
01-2: Notes on the Class

- Projects are non-trivial
  - Using new tools (JavaCC)
  - Managing a large scale project
  - Lots of complex classes & advanced programming techniques.
Projects are non-trivial
- Using new tools (JavaCC)
- Managing a large scale project
- Lots of complex classes & advanced programming techniques.

START EARLY!
- Projects will take longer than you think (especially starting with the semantic analyzer project)

ASK QUESTIONS!
01-4: What is a compiler?

Source Program → Compiler → Machine code

Simplified View
01-5: What is a compiler?

More Accurate View
01-6: What is a compiler?

Source File → Lexical Analyzer → Token Stream → Parser → Abstract Syntax Tree

Front end

Assembly → Code Generator → Abstract Assembly Tree → Semantic Analyzer → Assembly Tree Generator

Back End

Assembler → Relocatable Object Code

Linker → Machine code

Libraries
01-7: What is a compiler?

- Source File
- Lexical Analyzer
- Token Stream
- Parser
- Abstract Syntax Tree
- Semantic Analyzer
- Assembly
- Code Generator
- Abstract Assembly Tree
- Semantic Analyzer
- Assembly Tree Generator
- Assembler
- Relocatable Object Code
- Linker
- Machine code
- Libraries

Covered in this course
01-8: Why Use Decomposition?
Why Use Decomposition?

Software Engineering!

- Smaller units are easier to write, test and debug
- Code Reuse
  - Writing a suite of compilers (C, Fortran, C++, etc) for a new architecture
  - Create a new language – want compilers available for several platforms
01-10: **Lexical Analysis**

- Converting input file to stream of tokens

```c
void main() {
    print(4);
}
```
Lexical Analysis

- Converting input file to stream of tokens

void main() {
    IDENTIFIER(void)
    print(4);
    IDENTIFIER(main)
} LEFT-PARENTHESIS
RIGHT-PARENTHESIS
LEFT-BRACE
IDENTIFIER(print)
LEFT-PARENTHESIS
INTEGER-LITERAL(4)
RIGHT-PARENTHESIS
SEMICOLO
RIGHT-BRACE
Brute-Force Approach

- Lots of nested if statements

```c
if (c = nextchar() == 'P') {
    if (c = nextchar() == 'R') {
        if (c = nextchar() == '0') {
            if (c = nextchar() == 'G') {
                /* Code to handle the rest of either
                 * PROGRAM or any identifier that starts
                 * with PROG
                */
                /* Code to handle the rest of either
                 * PROCEDURE or any identifier that starts
                 * with PROC
                */
            } else if (c == 'C') {
                /* Code to handle the rest of either
                 * PROGRAM or any identifier that starts
                 * with PROG
                */
                /* Code to handle the rest of either
                 * PROCEDURE or any identifier that starts
                 * with PROC
                */
            }
        } else if (c == 'C') {
            /* Code to handle the rest of either
             * PROGRAM or any identifier that starts
             * with PROG
            */
            /* Code to handle the rest of either
             * PROCEDURE or any identifier that starts
             * with PROC
            */
        }
    }
}
...
Brute-Force Approach

- Break the input file into words, separated by spaces or tabs
  - This can be tricky – not all tokens are separated by whitespace
- Use string comparison to determine tokens
Deterministic Finite Automata

- Set of states
- Initial State
- Final State(s)
- Transitions

DFA for else, end, identifiers

Combine DFA
Given a DFA, it is easy to create C code to implement it.

DFAs are easier to understand than C code.
  - Visual – almost like structure charts

However, creating a DFA for a complete lexical analyzer is still complex.
We’d like a tool:

- Describe the tokens in the language
- Automatically create DFA for tokens
- Then, automatically create C code that implements the DFA

We need a method for describing tokens
01-17: Formal Languages

- **Alphabet** \( \Sigma \): Set of all possible symbols (characters) in the input file
  - Think of \( \Sigma \) as the set of symbols on the keyboard
- **String** \( w \): Sequence of symbols from an alphabet
- **String length** \( |w| \): Number of characters in a string: \( |\text{car}| = 3, |\text{abba}| = 4 \)
  - Empty String \( \varepsilon \): String of length 0: \( |\varepsilon| = 0 \)
- **Formal Language**: Set of strings over an alphabet

Formal Language \( \neq \) Programming language – Formal Language is only a set of strings.
Example formal languages:

- Integers \{0, 23, 44, \ldots\}
- Floating Point Numbers \{3.4, 5.97, \ldots\}
- Identifiers \{foo, bar, \ldots\}
Language Concatenation

Given two formal languages $L_1$ and $L_2$, the concatenation of $L_1$ and $L_2$, $L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$

For example:
\{fire, truck, car\} \{car, dog\} = 
\{firecar, firedog, truckcar, truckdog, carcar, cardog\}
Given a formal language $L$:

$L^0 = \{\epsilon\}$

$L^1 = L$

$L^2 = LL$

$L^3 = LLL$

$L^4 = LLLL$

$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots \cup L^n \cup \ldots$
Regular expressions are used to describe formal languages over an alphabet $\Sigma$:

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>$L[\epsilon] = {\epsilon}$</td>
</tr>
<tr>
<td>$a \in \Sigma$</td>
<td>$L[a] = {a}$</td>
</tr>
<tr>
<td>$(MR)$</td>
<td>$L[MR] = L[M]L[R]$</td>
</tr>
<tr>
<td>$(M</td>
<td>R)$</td>
</tr>
<tr>
<td>$(M^*)$</td>
<td>$L[(M^<em>)] = L[M]^</em>$</td>
</tr>
</tbody>
</table>
From highest to Lowest:

Kleene Closure *
Concatenation
Alternation |

\[ ab^*c | e = (a(b^*)c) | e \]
all strings over \{a,b\}
binary integers (with leading zeroes)
all strings over \{a,b\} that
  begin and end with a
all strings over \{a,b\} that
  contain aa
all strings over \{a,b\} that
  do not contain aa
all strings over \{a,b\} \quad (a|b)^* \\
binary integers (with leading zeroes) \quad (0|1)(0|1)^* \\
all strings over \{a,b\} that \quad a(a|b)^*a \\
begin and end with a \\
all strings over \{a,b\} that \quad (a|b)^*aa(a|b)^* \\
contain aa \\
all strings over \{a,b\} that \quad b^*(abb^*)^*(a|\epsilon) \\
do not contain aa
01-25: Reg. Exp. Shorthand

\[ [a,b,c,d] = (a|b|c|d) \]
\[ [d-g] = [d,e,f,g] = (b|e|f|g) \]
\[ [d-f,M-O] = [d,e,f,M,N,O] \]
\[ = (d|e|f|M|N|O) \]
\[ (\alpha)? = \text{Optionally } \alpha \text{ (i.e., } (\alpha \mid \epsilon)) \]
\[ (\alpha)^+ = \alpha(\alpha)^* \]
Many unix tools use regular expressions.

Example: `grep '〈reg exp〉' filename`

- Prints all lines that contain a match to the regular expression.
- Special characters:
  - `^` beginning of line
  - `$` end of line
- (grep examples on other screen)
All characters & strings must be in quotation marks

- "else"
- "+
- ("a" | "b")

All regular expressions involving * must be parenthesized

- ("a")*, not "a"*
JavaCC Shorthand

"a","b","c","d"] = ("a" | "b" | "c" | "d")
"d"-"g"] = ["d","e","f","g"] = ("b" | "e" | "f" | "g")
"d"-"f", "M"-"O"] = ["d","e","f","M","N","O"]
= ("d" | "e" | "f" | "M" | "N" | "O")
(\(\alpha\))? = Optionally \(\alpha\) (i.e., (\(\alpha\) | \(\epsilon\)))
(\(\alpha\))+ = \(\alpha(\alpha)^{+}\)
(~["a","b"])) = Any character except “a” or “b”. Can only be used with [] notation
\(~(a(a|b)^{+}b)\) is not legal
<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{if}</code></td>
<td>Set of legal identifiers</td>
</tr>
<tr>
<td></td>
<td>Set of integer literals</td>
</tr>
<tr>
<td></td>
<td>(leading zeroes allowed)</td>
</tr>
<tr>
<td></td>
<td>Set of real literals</td>
</tr>
<tr>
<td>Regular Expression</td>
<td>Language</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------</td>
</tr>
<tr>
<td>&quot;if&quot;</td>
<td>{if}</td>
</tr>
<tr>
<td><a href="%5B%220%22-%229%22,%22a%22-%22z%22%5D">&quot;a&quot;-&quot;z&quot;</a>* [&quot;0&quot;-&quot;9&quot;]</td>
<td>Set of legal identifiers (leading zeroes allowed)</td>
</tr>
<tr>
<td>([&quot;0&quot;-&quot;9&quot;]+.&quot;([&quot;0&quot;-&quot;9&quot;]*))</td>
<td>Set of real literals</td>
</tr>
<tr>
<td>((([&quot;0&quot;-&quot;9&quot;])*&quot;.&quot;[&quot;0&quot;-&quot;9&quot;])+)</td>
<td></td>
</tr>
</tbody>
</table>
JavaCC is a Lexical Analyzer Generator and a Parser Generator

- Input: Set of regular expressions (each of which describes a type of token in the language)
- Output: A lexical analyzer, which reads an input file and separates it into tokens
options{
    /* Code to set various options flags */
}

PARSER_BEGIN(foo)

public class foo {
    /* This segment is often empty */
}

PARSER_END(foo)

TOKEN_MGR_DECLS :
{
    /* Declarations used by lexical analyzer */
}

/* Token Rules & Actions */
01-33: Token Rules in JavaCC

- Tokens are described by rules with the following syntax:

  ```
  TOKEN :
  {
   <TOKEN_NAME: RegularExpression>
  }
  ```

- `TOKEN_NAME` is the name of the token being described
- `RegularExpression` is a regular expression that describes the token
• Token rule examples:

```java
TOKEN :
{
    <ELSE: "else">
}

TOKEN :
{
    <INTEGER_LITERAL: (["0"-"9"])>}
```
Several different tokens can be described in the same TOKEN block, with token descriptions separated by |.

```
TOKEN :
{
  <ELSE: "else">
  | <INTEGER_LITERAL: (["0"-"9"])>*>
  | <SEMICOLOMN: ";" critiques>
}
```
When we run javacc on the input file foo.jj, it creates the class fooTokenManager.

The class fooTokenManager contains the static method getNextToken().

Every call to getNextToken() returns the next token in the input stream.
When `getNextToken` is called, a regular expression is found that matches the next characters in the input stream.

What if more than one regular expression matches?

```plaintext
TOKEN :
{
  <ELSE: "else">
  | <IDENTIFIER: (["a"-"z"])+>
}
```
When more than one regular expression matches the input stream:

- Use the longest match
  - "elsed" should match to IDENTIFIER, not to ELSE followed by the identifier "d"
- If two matches have the same length, use the rule that appears first in the .jj file
  - "else" should match to ELSE, not IDENTIFIER
JavaCC Example

PARSER_BEGIN(simple)
public class simple {

}
PARSER_END(simple)

TOKEN :
{
  <ELSE: "else">
  | <SEMICOLON: ";"> 
  | <FOR: "for">
  | <INTEGER_LITERAL: ("0"-"9")+>
  | <IDENTIFIER: ["a"-"z"](["a"-"z","0"-"9"])*> 
}

else;ford for
01-40: SKIP Rules

- Tell JavaCC what to ignore (typically whitespace) using SKIP rules
- SKIP rule is just like a TOKEN rule, except that no TOKEN is returned.

SKIP:
{
  < regularexpression1 >
  | < regularexpression2 >
  | ...
  | < regularexpressionn >
}
01-41: Example SKIP Rules

PARSER_BEGIN(simple2)
public class simple2 {
}
PARSER_END(simple2)

SKIP :
{
  < " " >
| < "\n" >
| < "\t" >
}

TOKEN :
{
  <ELSE: "else">
| <SEMICOLON: ";" >
| <FOR: "for">
| <INTEGER_LITERAL: ("0"-"9")]>
| <IDENTIFIER: ["A"-"Z"](["A"-"Z","0"-"9"])>
}
Comments can be dealt with using SKIP rules

How could we skip over 1-line C++ Style comments?

// This is a comment
Comments can be dealt with using SKIP rules

How we could skip over 1-line C++ Style comments:

```
// This is a comment
```

Using a SKIP rule

```
SKIP :
{
    < "//" (~["\n"])\* "\n" >
}
```
Writing a regular expression to match multi-line comments (using /* and */) is much more difficult.
Writing a regular expression to match nested comments is impossible (take Automata Theory for a proof :) )
What can we do?
  • Use JavaCC States
We can label each TOKEN and SKIP rule with a “state”

Unlabeled TOKEN and SKIP rules are assumed to be in the default state (named DEFAULT, unsurprisingly enough)

Can switch to a new state after matching a TOKEN or SKIP rule using the : NEWSTATE notation
JavaCC States

SKIP :
{
    < " " >
| < "\n" >
| < "\t" >
}

SKIP :
{
    < "/*" > : IN_COMMENT
}

<TOKEN :>
	<ELSE: "else">
	... (etc)

We can add Java code to any SKIP or TOKEN rule.
That code will be executed when the SKIP or TOKEN rule is matched.
Any methods / variables defined in the TOKEN_MGR_DECLS section can be used by these actions.
PARSER_BEGIN(remComments)
public class remComments { }
PARSER_END(remComments)

TOKEN_MGR_DECLS :
{
    public static int numcomments = 0;
}

SKIP :
{
    < "/*" > : IN_COMMENT
}

SKIP :
{
    < "//" (~["\n"])* "\n" > { numcomments++; }
}
01-49: Actions in TOKEN & SKIP

<IN_COMMENT>
SKIP :
{
    < "*/" > { numcomments++; SwitchTo(DEFAULT);}
}

<IN_COMMENT>
SKIP :
{
    < ~[] >
}

TOKEN :
{
    <ANY: ~[]>
}
01-50: Tokens

- Each call to getNextToken returns a “Token” object.
- Token class is automatically created by javaCC.
- Variables of type Token contain the following public variables:
  - `public int kind;` The type of token. When javacc is run on the file foo.jj, a file fooConstants.java is created, which contains the symbolic names for each constant.

```java
public interface simplejavaConstants {
    int EOF = 0;
    int CLASSS = 8;
    int DO = 9;
    int ELSE = 10;
    ...
```
• Each call to getNextToken returns a "Token" object
• Token class is automatically created by javaCC.
• Variables of type Token contain the following public variables:
  • public int beginLine, beginColumn, endLine, endColumn; The location of the token in the input file
Each call to getNextToken returns a “Token” object
Token class is automatically created by javaCC.
Variables of type Token contain the following public variables:
  - public String image; The text that was matched to create the token.
class TokenTest {
    public static void main(String args[]) {
        Token t;
        Java.io.InputStream infile;
        pascalTokenManager tm;
        boolean loop = true;

        if (args.length < 1) {
            System.out.print("Enter filename as command line argument");
            return;
        }
        try {
            infile = new Java.io.FileInputStream(args[0]);
        } catch (Java.io.FileNotFoundException e) {
            System.out.println("File " + args[0] + " not found.");
            return;
        }
        tm = new sjavaTokenManager(new SimpleCharStream(infile));
t = tm.getNextToken();
while(t.kind != sjavaConstants.EOF) {
    System.out.println("Token : " + t + " : ");
    System.out.println(pascalConstants.tokenImage[t.kind]);
}
}
01-55: **Lexer Project**

- Write a .jj file for simpleJava tokens
- Need to handle all whitespace (tabs, spaces, end-of-line)
- Need to handle nested comments (to an arbitrary nesting level)
JavaCC is available at https://javacc.dev.java.net/

To compile your project
% javacc simplejava.jj
% javac *.java

To test your project
% java TokenTest <test filename>

To submit your program: Create a branch:
https://www.cs.usfca.edu/svn/<<username>>/cs414/lexer/