

01-0: **Syllabus**

- Office Hours
- Course Text
- Prerequisites
- Test Dates & Testing Policies
- Projects
 - Teams of up to 2
- Grading Policies
- Questions?

01-1: **Notes on the Class**

- 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

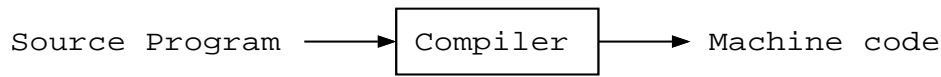
01-2: **Notes on the Class**

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

01-3: **Notes on the Class**

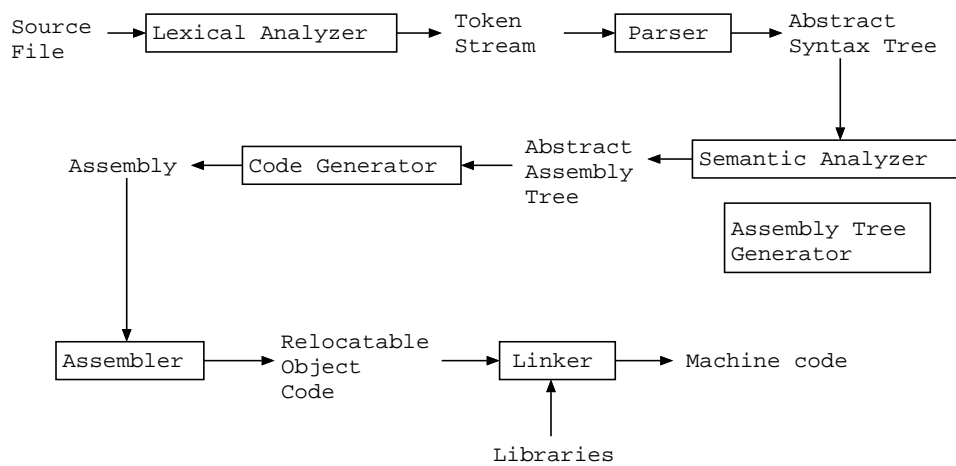
- Projects are non-trivial
 - Using new tools (JavaCC)
 - Managing a large scale project
 - Lots of complex classes & advanced programming techniques.
- *START EARLY!*
 - Projects will take longer than you think (especially starting with the semantic analyzer project)
- *ASK QUESTIONS!*

01-4: What is a compiler?



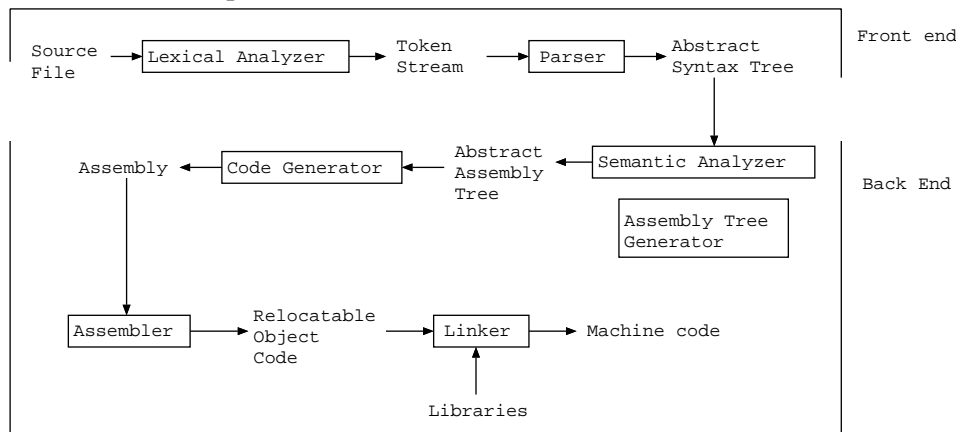
Simplified View

01-5: What is a compiler?

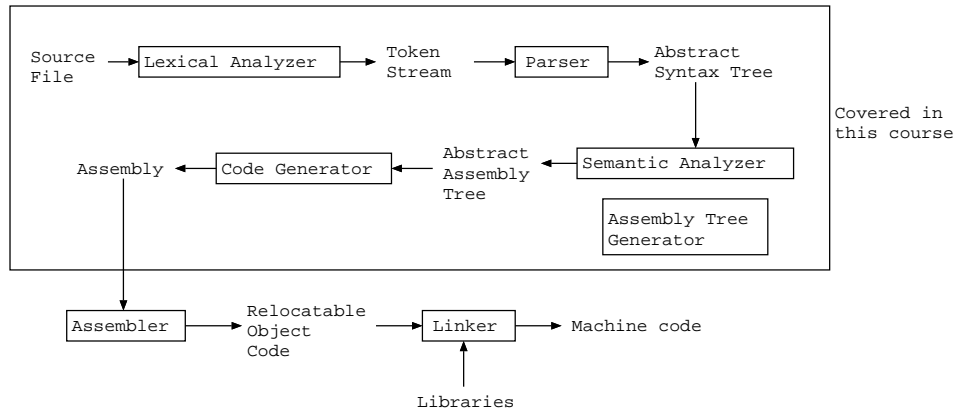


More Accurate View

01-6: What is a compiler?



01-7: What is a compiler?

**01-8: Why Use Decomposition?****01-9: 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

```
void main() {
    print(4);
}
```

01-11: 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
                  SEMICOLON
                  RIGHT-BRACE
```

01-12: Lexical Analysis

Brute-Force Approach

- Lots of nested if statements

```

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
        */
      } else if (c == 'C') {
        /* Code to handle the rest of either
           PROCEDURE or any identifier that starts
           with PROC
        */
      }
    }
  }
}
...

```

01-13: Lexical Analysis

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

01-14: Deterministic Finite Automata

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

DFA for else, end, identifiers

Combine DFA 01-15: DFAs and Lexical Analyzers

- 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

01-16: Automatic Creation of DFAs

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** Σ : Set of all possible symbols (characters) in the input file
 - Think of Σ 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** ϵ : String of length 0: $|\epsilon| = 0$
- **Formal Language**: Set of strings over an alphabet

Formal Language \neq Programming language – Formal Language is only a set of strings.

01-18: Formal Languages

Example formal languages:

- Integers $\{0, 23, 44, \dots\}$
- Floating Point Numbers $\{3.4, 5.97, \dots\}$
- Identifiers $\{\text{foo}, \text{bar}, \dots\}$

01-19: Language Concatenation

- **Language Concatenation** Given two formal languages L_1 and L_2 , the concatenation of L_1 and L_2 , $L_1L_2 = \{xy | x \in L_1, y \in L_2\}$

For example:

$\{\text{fire}, \text{truck}, \text{car}\} \{\text{car}, \text{dog}\} =$
 $\{\text{firecar}, \text{firedog}, \text{truckcar}, \text{truckdog}, \text{carcar}, \text{cardog}\}$

01-20: Kleene Closure

Given a formal language L :

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= LL \\ L^3 &= LLL \\ L^4 &= LLLL \end{aligned}$$

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots \cup L^n \cup \dots$$

01-21: Regular Expressions

Regular expressions are used to describe formal languages over an alphabet Σ :

Regular Expression	Language
ϵ	$L[\epsilon] = \{\epsilon\}$
$a \in \Sigma$	$L[a] = \{a\}$
(MR)	$L[MR] = L[M]L[R]$
$(M R)$	$L[(M R)] = L[M] \cup L[R]$
(M^*)	$L[(M^*)] = L[M]^*$

01-22: r.e. Precedence

From highest to Lowest:

Kleene Closure $*$
 Concatenation
 Alternation $|$

$$\text{ab}^* \text{c} | \text{e} = (\text{a}(\text{b}^*)\text{c}) | \text{e}$$

01-23: Regular Expression Examples

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

01-24: **Regular Expression Examples**

all strings over $\{a,b\}$	$(a b)^*$
binary integers (with leading zeroes)	$(0 1)^*$
all strings over $\{a,b\}$ that begin and end with a	$a(a b)^*a$
all strings over $\{a,b\}$ that contain aa	$(a b)^*aa(a b)^*$
all strings over $\{a,b\}$ that do not contain aa	$b^*(abb^*)^*(a \epsilon)$

01-25: **Reg. Exp. Shorthand**

$[a,b,c,d]$	=	$(a b c d)$
$[b-g]$	=	$[b,e,f,g] = (b e f g)$
$[b-f,M-O]$	=	$[b,e,f,M,N,O]$
	=	$(b e f M N O)$
$(\alpha)?$	=	Optionally α (i.e., $(\alpha \epsilon)$)
$(\alpha)^+$	=	$\alpha(\alpha)^*$

01-26: **Regular Expressions & Unix**

- Many unix tools use regular expressions
- Example: `grep '<reg exp>' filename`
 - Prints all lines that contain a match to the regular expression
 - Special characters:
 - \wedge beginning of line
 - $\$$ end of line
 - (grep examples on other screen)

01-27: **JavaCC Regular Expressions**

- All characters & strings must be in quotation marks
 - "else"
 - "+"
 - ("a" | "b")
- All regular expressions involving * must be parenthesized
 - ("a")*, not "a"*

01-28: **JavaCC Shorthand**

<code>["a","b","c","d"]</code>	=	<code>("a "b "c "d")</code>	
<code>["b"-,"g"]</code>	=	<code>["b","e","f","g"] = ("b "e "f "g")</code>	
<code>["b"-,"f","M"-,"O"]</code>	=	<code>["b","e","f","M","N","O"]</code>	
	=	<code>("b "e "f "M "N "O")</code>	
$(\alpha)?$	=	Optionally α (i.e., $(\alpha \epsilon)$)	01-29: r.e. Shorthand Examples
$(\alpha)^+$	=	$\alpha(\alpha)^*$	
$(\sim["a","b"])$	=	Any character <i>except</i> "a" or "b". Can only be used with [] notation $\sim(a(a-b)^*b)$ is not legal	

Regular Expression	Language
	{if}
	Set of legal identifiers
	Set of integer literals (leading zeroes allowed)
	Set of real literals

01-30: r.e. Shorthand Examples

Regular Expression	Language
<code>"if"</code>	{if}
<code>["a"-,"z"](["0"-,"9","a"-,"z"])*</code>	Set of legal identifiers
<code>["0"-,"9"]</code>	Set of integer literals (leading zeroes allowed)
<code>(["0"-,"9"]+)(["0"-,"9"]*) </code> <code>((["0"-,"9"])*)(["0"-,"9"]+)</code>	Set of real literals

01-31: Lexical Analyzer Generator

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

01-32: Structure of a JavaCC file

```
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

01-34: Token Rules in JavaCC

- Token rule examples:

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

```
TOKEN :
{
    <INTEGER_LITERAL: ([ "0"-"9" ])+>
}
```

01-35: Token Rules in JavaCC

- Several different tokens can be described in the same TOKEN block, with token descriptions separated by |.

```
TOKEN :
{
    <ELSE: "else">
|
    <INTEGER_LITERAL: ([ "0"-"9" ])+>
|
    <SEMICOLON: ";">
}
```

01-36: getNextToken

- 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.

01-37: getNextToken

- 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?

```
TOKEN :
{
    <ELSE: "else">
|
    <IDENTIFIER: ([ "a"-"z" ])+>
}
```

01-38: getNextToken

- 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

01-39: 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"])*>
}

```

01-42: JavaCC States

- Comments can be dealt with using SKIP rules
- How could we skip over 1-line C++ Style comments?

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

01-43: JavaCC States

- 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" >
}
```

01-44: JavaCC States

- 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

01-45: 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

01-46: JavaCC States

```
SKIP :
{
    | < " " >
    | < "\n" >
    | < "\t" >
}
SKIP :
{
    < "/*" > : IN_COMMENT
}
<IN_COMMENT>
SKIP :
{
    < "*/" > : DEFAULT
    | < "~[]" >
}
TOKEN :
{
    <ELSE: "else">
    ... (etc)
}
```

01-47: Actions in TOKEN & SKIP

- 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

01-48: Actions in TOKEN & SKIP

```

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

```

public interface simplejavaConstants {
    int EOF = 0;
    int CLASS = 8;
    int DO = 9;
    int ELSE = 10;
    ...
}

```

01-51: 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 beginLine, beginColumn, endLine, endColumn;` The location of the token in the input file

01-52: 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 String image;` The text that was matched to create the token.

01-53: Generated TokenManager

```
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));
    }
}
```

01-54: Generated TokenManager

```
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)

01-56: Project Details

- 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/>