Using CookCC

Lex / Parse the Easy Way

The Old Way

Lex + Yacc

- Use *.l and *.y files.
- Proprietary file format
  - Poor IDE support
  - Do not work well for some languages

The Difficulties

- Difficult to read
  - No syntax highlighting
  - No variable / function usage highlight and analysis etc.
- Difficult to write
  - Need to hook up lexer and parser manually
  - No context sensitive hints (function call and parameter hints)
  - No instant code error checking within the IDE
  - No auto-completions (auto-casting, auto-import, etc)
  - Fragmented code
- Difficult to maintain
  - Multiple files to manage (*.l and *.y)
  - No code reformatting
  - No refactoring
- Difficult to document

Difficult Lex Code Example

```
                    BEGIN(FIRSTCCL);
                    return '[';
                    }
                }
  "{-}"          return CCL_OP_DIFF;
  "{+}"          return CCL_OP_UNION;


  /* Check for :space: at the end of the rule so we don't
   * wrap the expanded regex in '(' ')' -- breaking trailing
   * context.
   */
"{"{NAME}"}"[[:space:]]?      {
            register Char *nmdefptr;
          int end_is_ws, end_ch;

          end_ch = yytext[yyleng-1];
          end_is_ws = end_ch != '}' ? 1 : 0;

             if(yyleng-1 < MAXLINE)
                      {
             strcpy( nmstr, yytext + 1 );
              }
```

Difficult Yacc Code Example

```
                for ( i = $2; i <= $4; ++i )
                    ccladd( $1, i );

                cclsorted = cclsorted && ($2 > lastchar);
                lastchar = $4;
            }

                        }

             $$ = $1;
             }

|   ccl CHAR
             {
             ccladd( $1, $2 );
             cclsorted = cclsorted && ($2 > lastchar);
             lastchar = $2;

       /* Do it again for upper/lowercase */
       if (sf_case_ins() && has_case($2)){
           $2 = reverse_case ($2);
           ccladd ($1, $2);

           cclsorted = cclsorted && ($2 > lastchar);
           lastchar = $2;
       }
```

Partial Solutions

- Supports XML input
  - Solves the problem related to file format.
    - Better support for different languages
    - Patterns / Rules stand out more
  - Still do not solve other problems.
- Integrated lexer and parser generator
  - Unfortunately, most still require manual hook up between the generated lexer and parser.

Specify Lexr / Parser w/ Annotation

- Advantages:
  - Write code as normal and take the full advantage of the modern IDEs
    - Syntax highlighting
    - Context sensitive hints
    - Code usage and analysis
    - Refactoring
    - Auto-completion
    - Instant error checking
    - etc
- Disadvantages:
  - Need to write a language specific annotation parser
    - Java rules and nothing else matters ☺

CookCC Features

- Integrated lexer and parser (LALR (1))
  - Can specify lexer and parser separately.
  - If the lexer / parser are specified together, they are automatically hooked up.
- Supports multiple input file types
  - XML (DTD): *.xcc
  - Yacc: *.y
  - Java annotation: *.java (launched within Java APT)
- Support multiple output formats
  - Uses the powerful FreeMarker template engine for code generation.
  - Java
    - Case by case optimized lexer and parser
    - No runtime library required. Very small footprint.
    - Multiple lexer / parser DFA tables options (ecs, compressed).
  - Plain text (for table extraction)
  - XML (for input file conversion)
  - Yacc (for input file conversion)

Quick Tutorial

CookCC Java Annotation Input

Annotate Input Class

- Put cookcc.jar in the project library path.
  - This jar is only required for compile time.
  - All CookCC annotations are compile time annotations (not required for runtime).
- @CookCCOption mark a class that needs the lexer and parser.
  - Generated class would be the parent class of the current class. In this example, it would be Parser.
  - lexerTable and parserTable options are optional

```
@CookCCOption (lexerTable="compressed", parserTable="compressed")
public class Calculator extends Parser
```

Create Dummy Parent Class

- Parser is an empty class
  - CookCCByte contains functions that would appear in the generated class.
    - Generated class no longer extends CookCCByte
    - CookCCByte is not required for runtime
  - Extend CookCCByte for byte parsing and CookCCChar for unicode parsing.
  - Class scope is copied to the generated class.
  - File header (i.e. the copyright message) and class header will be copied to the generated class.

```
/* Copyright 2008 by Heng Yuan */
import org.yuanheng.cookcc.CookCCByte;

/**
 * @author Heng Yuan
 * @version $Id$
 */
public class Parser extends CookCCByte
{
}
```

Lexer Annotations

- @Shortcut specifies name of the regular expressions patterns that would be used regularly later on.
  - Can be annotated on any functions in any order.
  - Avoid creating cyclic name references.
- @Shortcuts is a collection of shortcuts.
- @Lex specific the regular expression pattern, and which states this pattern is used in. The function this annotation marked is called when the pattern is matched.
  - Usually the function scope should be either protected or package since they are called by the generated class only.
- Backslash (\) needs to be escaped inside Java string, resulting in double backslash (\\)

```
@Shortcuts ( shortcuts = {
    @Shortcut (name="nonws", pattern="[^ \\t\\n]"),
    @Shortcut (name="ws", pattern="[ \\t]")
})
@Lex (pattern="{nonws}+", state="INITIAL")
void matchWord ()
```

Lexer Function Case 1

- Function returns void
  - The lexer would call this function and then move on to match the next potential pattern.

```
@Lex (pattern = "[ \\t\\r\\n]+")
protected void ignoreWhiteSpace ()
{
}
```

Lexer Function Case 2

- Function returns a non-int value
  - @Lex needs to contain the terminal token that would be returned. The return value from the function is going to be the value associated with this terminal.

```
@Lex (pattern = "[0-9]+", token = "INTEGER")
protected Integer parseInt ()
{
    return Integer.parseInt (yyText ());
}
```

Lexer Function Case 3

- Function returns an int value
  - The lexer would return the value immediately.

```
@Lex (pattern="[(){}.]")
protected int parseSymbol ()
{
    return yyText ().charAt (0);
}
```

More Lexer Annotations

@Lexs is a collection of @Lex patterns

```
@Lexs (patterns = {
    @Lex (pattern = "[;]", token = "SEMICOLON"),
    @Lex (pattern = "[=]", token = "ASSIGN"),
    @Lex (pattern = "[+]", token = "ADD"),
    @Lex (pattern = "\\-", token = "SUB"),
    @Lex (pattern = "[*]", token = "MUL"),
    @Lex (pattern = "[/]", token = "DIV"),
    @Lex (pattern = "[<]", token = "LT"),
    @Lex (pattern = "[>]", token = "GT"),
    @Lex (pattern = ">=", token = "GE"),
    @Lex (pattern = "<=", token = "LE"),
    @Lex (pattern = "!=", token = "NE"),
    @Lex (pattern = "==", token = "EQ")
})
protected Object parseOp ()
{
    return null;
}
```

Token Annotations

- @CookCCToken marks a Enum to specify tokens shared between the lexer/parser
- @TokenGroup specify the token type. Tokens marked with @TokenGroup later on have higher precedences.
- Tokens not marked with @TokenGroup would inherit the type and precedence of the previous token.
- Symbols such as + - * / < > etc would have to have a name since they can't be fit in the enum declaration. This restriction is imposed by CookCC per se, not annotation based parsing in general.

```
/**
 * Specify the tokens shared by the lexer and parser.
 */
@CookCCToken
static enum Token
{
    // TokenGroup is used to specify the token type and precedence.
    // By default, if the type of the token is not specified, it is
    // TokenGroup.NONASSOC.
    @TokenGroup
    VARIABLE, INTEGER, WHILE, IF, PRINT, ASSIGN, SEMICOLON, LT, GT,
    @TokenGroup
    IFX,
    @TokenGroup
    ELSE,

    // specify the left associativity.
    // Can use static import to avoid typing TokenType. part.
    @TokenGroup (type = TokenType.LEFT)
    GE, LE, EQ, NE,
    @TokenGroup (type = TokenType.LEFT)
    ADD, SUB,
    @TokenGroup (type = TokenType.LEFT)
    MUL, DIV,
    @TokenGroup (type = TokenType.LEFT)
    UMINUS
}
```

Parser Annotations

- @Rule specifies a single grammar rule.
  - args specifies the arguments to be passed to the function
  - The return value from the function is the value to be assigned to the LHS non-terminal.
- Advantages:
  - No more cryptic names like $1 $2 or having to specify the types of the variable elsewhere.
  - Can fully take advantage of the IDE
  - Documentation is easier since comments (in this case JavaDoc comments) can be added to the function and extracted using generic documentation tools.

```
@Rule (lhs = "stmt", rhs = "PRINT expr SEMICOLON", args = "2")
protected Node parsePrintStmt (Node expr)
{
    return new PrintNode (expr);
}

@Rule (lhs = "stmt", rhs = "VARIABLE ASSIGN expr SEMICOLON", args="1 3")
protected Node parseAssign (String var, Node expr)
{
    return new AssignNode (var, expr);
}

@Rule (lhs = "stmt", rhs = "WHILE '(' expr ')' stmt", args = "3 5")
protected Node parseWhile (Node expr, Node stmt)
{
    return new WhileNode (expr, stmt);
}

@Rule (lhs = "stmt", rhs = "IF '(' expr ')' stmt", args = "3 5", precedence = "IFX")
protected Node parseIf (Node expr, Node stmt)
{
    return new IfNode (expr, stmt, null);
}
```

More Parser Annotations

- @Rules is a collection of @Rule productions.

```
@Rule (lhs = "action", rhs = "complete_action ACTION_CODE", args = "2")
String parseAction (String action)
{
    return action;
}

@Rules (rules = {
    @Rule (lhs = "action", rhs = ""),
    @Rule (lhs = "complete_action", rhs = "complete_action PARTIAL_ACTION"),
    @Rule (lhs = "complete_action", rhs = "")
})
String parseAction ()
{
    return null;
}
```

Execute Lexer / Parser

- setInput (…) to set up the input source
  - setBufferSize to improve lexer speed.
- yyLex () for lexer only parser
- yyParse () for parser

```
YaccParser parser = new YaccParser ();
int fileSize = (int)file.length ();
if (fileSize > 4096)
    parser.setBufferSize (fileSize);
parser.setInput (new FileInputStream (file));
if (parser.yyParse () > 0)
    Main.error ("errors in input");
```

Generate Class

- Run java annotation processing tool (APT) using CookCC as the processor to generate the class needed.
  - `apt -nocompile -s . -classpath cookcc.jar;. Calculator.java`
- Unless the lex/parse patterns/rules are changed, it is not necessary to re-generate new classes.
- Can be refactored at anytime.

Ant task

- Using Ant task is easier.
- Setup <cookcc> task

```
<target name="initcookcc">
    <taskdef name="cookcc" classname="org.yuanheng.cookcc.ant.Task" classpath="${tool}/cookcc-latest.jar"/>
</target>
```

- Execute task

```
<target name="YaccLexer.java" depends="initcookcc">
    <cookcc srcdir="${src}" src="org/yuanheng/cookcc/input/yacc/YaccParser.java"/>
</target>
```

Ant task (continued…)

- Options

```
<cookcc srcdir="${src}" src="org/yuanheng/cookcc/input/yacc/YaccParser.java" lang="xml">
    <option name="-o" value="YooParser.xcc"/>
</cookcc>
```

Acknowledgement

- Dr. Stott Parker
  - For helpful discussions
- Related Work
  - SPARK for Python
    - Uses Python doc string to specify the lexer / parser.
    - Aycock, J. "Compiling Little Languages in Python", Proceedings of the Seventh International Python Conference, p100, 1998.

CookCC Web Site

- Project home:
  - http://code.google.com/p/cookcc/
- License
  - New BSD License