Designing the Formula Parser

- **Scanner**
  - break input into lexical units (tokens)
  - $A11 + A12 \rightarrow \text{CELLID PLUS CELIID}$
  - uses regular expressions that specify the tokens

- **Parser**
  - create parse tree from tokens
  - represents the grammatical structure
  - represents precedence and associativity rules
  - uses (context-free) grammar to represent structure

- Can be written by hand
  - more flexible and productive to use scanner- and parser generators

Scanner Role

- source (formula)
- Scanner (aka lexical analyzer)
- token
- get next token
- parser
- symbol table
- Scanning = converting character stream into logical units (aka. tokens)
**Parser Role**

- Source
- Lexical analyzer
- Get next token
- Symbol table
- Parser
- Parse tree
- Rest of spreadsheet

Parsing = determining whether a string of tokens can be generated by a grammar

---

**Jlex: a scanner generator**

- Jlex specification
  - xxx.jlex
  - xxx.jlex.java
- xxx.jlex.java
- generated scanner
  - JLex.Main.java
- JLex.Main.java
- javac
- Yylex class
- P.main
  - P.main.java
- P.main.java
- Output of P.main
- Yylex class

---

**Creation & Invocation**

```java
class P {
    public static void main(String[] args) {
        FileReader inFile = new FileReader(args[0]);
        Yylex scanner = new Yylex(inFile);
        Symbol token = scanner.next_token();
        while (token.sym != sym.EOF) {
            switch (token.sym) {
                case sym.INTLITERAL:
                    System.out.println("INTLITERAL (" + ((IntLitTokenVal)token.value).intValue + ");
                    break;
                ...}
            token = scanner.next_token();
        }
```
Jlex Specification file
(*xxx.jlex*)

- **User code**: copied to xxx.jlex.java,
  - use it to define auxiliary classes and methods.

- **JLex directives**: macro definitions
  - use to specify what letters, digits, whitespace are.

- **Regular expression rules**:
  - specify how to divide up input into tokens,
  - regular expressions are followed by actions
  - print error messages, return token codes
  - no need to put characters back to input (done by Jlex)

---

Regular expression rules

```
regular-expression
  { action }
```

- pattern to be matched
  - code to be executed
  when the
  - pattern is matched

When `next_token()` method is called, it repeats:

- Find the longest sequence of characters in the input (starting with the current character) that matches a pattern.
- Perform the associated action (plus "consume the matched lexeme").
- until a return in an action is executed.

---

Matching rules

- If several patterns that match the same sequence of characters, then the **longest pattern** is considered to be matched.
- If several patterns that match the same (longest) sequence of characters, then the **first such pattern** is considered to be matched
  - so the order of the patterns can be important!
- If an input character is not matched in any pattern, the scanner throws an **exception**
  - make sure that there can be no unmatched characters,
  - otherwise the scanner will "crash" on bad input.
Regular expressions

- Closely follow standard conventions.
  - most characters match themselves:
    - abc
    - ==
    - while
  - characters in quotes, including special characters, except \, match themselves
    - "ab" matches ab not a or b
    - "a\b" matches a\b not a\b

Regular-expression operators

- the traditional ones, plus the ? operator

| | means "or"
| * | means zero or more instances of
| + | means one or more instances of
| ? | means zero or one instance of
| () | are used for grouping

Backslash is special escape character:

- \n newline
- \t tab
- \" double quote

To match a backslash character, put it in quotes
More operators

- ^ matches beginning of line
  - ^main matches string "main" only when it appears at the beginning of line.

- $ matches end of line
  - main$ matches string "main" only when it appears at the end of line.
  - (dot) matches any character except newline – usually used in the last rule of specification to match all "bad" characters

Character classes

- [abc]
  - matches one character (either a or b or c)

- [a-z]
  - matches any character between a and z, inclusive

- [^abc]
  - matches any character except a, b, or c.

- ^ has special meaning only at 1st position in [...] (brackets)

- [\t\\]
  - matches tab or \r

- [a bc] is equivalent to a\* "b|c
  - white-space in char class and strings matches itself

JLex directives

- specified in the second part of xxx.jlex.
  - can also specify (see the manual for details)
    - the value to be returned on end-of-file,
    - that line counting should be turned on, and
    - that the scanner will be used with the parser generator javacup.

- directives includes macro definitions (very useful):
  - name = regular-expression
    - name is any valid Java identifier
  - DIGIT= \[0-9\]
  - LETTER= [a-zA-Z]
  - WHITESPACE= \[\t\n\\]

- To use a macro, use its name inside curly braces.
  - \{LETTER\} \{\LETTER\} \{[\LETTER\}\{DIGIT\}\}
Comments

- You can include comments in the first and second parts of your JLex specification,
- in the third part, JLex would think your comments are part of a pattern.
- use Java comments // ...

A Small Example

```java
DIGIT= \[0-9\]
LETTER= \[a-zA-Z\]
WHITESPACE= \[\t\n\] // space, tab, newline
// for compatibility with java CUP
%cup
// Turn on line counting
%line
```
Another example

(DIGIT)+ { int val = (new Integer(yytext()));
    Symbol S = new Symbol(sym.INTLITERAL,
        new IntLitTokenVal(yyline+1, CharNum.num, val));
    CharNum.num += yytext().length();
    return S;
}

(WHITESPACE)+ {CharNum.num +=
    yytext().length();
}

Java CUP

Parser generator - generates a bottom up parser

What is Java CUP

- a parser generator
  - CUP = Construction of Useful Parsers
- Java successor to yacc
  - a standard UNIX parser generator
  - yacc = Yet Another Compiler Compiler

### Input to Java CUP

- The specification includes:
  - optional package and import declarations
  - optional user code
  - terminal and nonterminal declarations
  - optional precedence and associativity declarations
  - grammar rules with associated actions

### Example of calling the parser

```java
FileReader inFile = new FileReader(args[0]);
parser P = new parser(new Yylex(inFile));
Symbol root=null; // the parser will return a Symbol whose value
                   // field's type is the type associated with the
                   // root nonterminal (i.e., with the nonterminal
                   // "program")
root = P.parse(); // do the parse
```

### Declarations of terminals and nonterminals

- all terminals and nonterminals used by your grammar must be declared
  ```
  terminal type name1, name2, ... ;
  non terminal type name1, name2, ... ;
  ```
- all terminals must declare their type
  ```
  type = the type of Symbol.value returned by scanner
  ```
- non terminals with a value must also declare their type
**Precedence Declarations**

- You can resolve an ambiguous grammar like 
  \[ E \rightarrow E + E \mid E - E \mid E^* E \mid E / E \]
  by rewriting the grammar
  - introduce new nonterminals
  - or using precedence declarations
  precede left PLUS, MINUS;
  precede left TIMES, DIVIDE;
  precede nonassoc EQUALS;
  - precedence declarations resolve parsing (shift/reduce) conflicts

**Grammar Rules**

- the heart of the specification
  - (optional) declaration of the start symbol
    start with program;
  - if omitted, first nonterminal is the start symbol

```
terminal TokenVal          SEMICOLON;
terminal TokenVal          INT;
terminal IdTokenVal        ID;
non terminal FieldDeclNode fieldDecl;
non terminal TypeNode      type;
non terminal IdNode        id;
fieldDecl ::= STATIC type: t id: i SEMICOLON
             { RESULT = new FieldDeclNode(t, i); }
             ;
type ::= INT
       { RESULT = new IntNode(); }
       ;
id ::= ID:i
     { RESULT = new IdNode(i.linenum, i.charnum, i.idVal); }
     ;
```
Precedence declarations again

terminal UMINUS, LPAREN, RPAREN, _ ;
precedence left    PLUS, MINUS;
precedence left    TIMES, DIVIDE, MOD;
precedence left    UMINUS;

exp ::= exp PLUS exp
     | exp DIVIDE exp
     | MINUS exp %prec UMINUS
     | ...

CS 1530 Software Engineering Fall
2004