# PROGRAM STRUCTURE AND INVOCATION

### **HL**.jj file contains scanner and parser specifications for language HL

### Test**HL**.java file contains the main program to use with the scanner and parser.

### Compilation:

#### 1) javacc HL.jj

####

#### This produces

##### **HL**.java: parser for HL

##### **HL**TokenManager.java: scanner for HL

##### **HL**Constants.java: constants for the scanner: Token definitions and the strings that they match

##### Token.java: definition of token class

##### SimpleCharStream.java: I/O buffering

##### TokenMgrError.java: scanning error class (extension of class Error)

##### ParseException.java: parse exception class (extension of class Exception)

#### javac \*.java

####

#### To compile all the .java files

### Running the program

#### 3) java –classpath . TestHL

# STRUCTURE OF JAVACC FILE (.jj extension)

//--------------------------------------------------------------------------------------------------

// JavaCC options for the parser and scanner

// run javacc on the command line to get list of options

options {

 IGNORE\_CASE=false;

 DEBUG \_TOKEN\_MANAGER=false;

}

//--------------------------------------------------------------------------------------------------

// Parser section

PARSER\_BEGIN(HL) // The parameter is your language name

public class HL { // The class name is your language name

// Additional Java variables and methods for the parser

}

PARSER\_END(HL)

//--------------------------------------------------------------------------------------------------

// Scanner section

TOKEN\_MGR\_DECLS :

{

 // Additional Java variables and methods for the scanner

}

SKIP : // Characters that should be skipped by scanner

{

 " " | "\t" | "\n" | "\r"

}

TOKEN : // Token definitions

{

 < ELSE: "else">

| < FOR: "for">

}

# JAVACC SCANNER COMMANDS

## Syntax

### Element in *italics* are optional

### *<state>* action : {

###  matching-expression

### | matching- expression

### …

### | matching- expression

### }

### Where a matching expression is:

### < #*token-name :* regular-expression > *{ java-code } : new-state*

## States

### Users can manually add (meta)states to the FSA. This is used to specify that the scanner should behave differently when it is in a different (meta)state, for example to scan strings and comments.

### This is optional

### The default state is <DEFAULT> it does not need to be specified but it can be.

### The state qualifier in front of an action means that the action will only be executed in that state.

## Scanner Actions

### SKIP: skips the regular expression

### TOKEN: defines a token

### SPECIAL\_TOKEN: defines a special token which is not parsed but accessed by a different process (used for separate parsing e.g. for JavaDoc documentation)

### MORE: matches the beginning of a regular expression which will continue to be matched in another command (the remainder of the regular expression will be matched later).

## #Token-name

### Is only necessary for TOKEN and SPECIAL\_TOKEN actions.

### When the # is omitted, this defines a new token.

### When the # is included, this defines a regular expression that can be used by other regular expressions as <token-name>

### (Note that these are all regular definitions)

## Regular-expression

### Regular expression that should be matched

### Syntax:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Elements and actions** | **Example** | **Matches** |
| char | Literal | "a" | "a" |
|  | Character class | ["a","b","c"] | "a" or "b" or "c"  |
|  | Ranged character class | ["a"-"z"] | Any lowercase letter |
|  | Negation | ~["a"] | Any single character other than a |
| string | Concatenation | “ab” | “ab” |
|  | Repetition | ("a"){4} | "aaaa" |
|  | Repetition range | ("a"){2,4) | "aa" or "aaa" or "aaaa" |
|  | Zero or 1 | ("a")? | Either 0 or 1 "a" |
|  | Zero or more | ("a")\* | Any number of "a"'s |
|  | One or more | ("a")+ | At least one "a" |
|  | Or | “yes” | ”no” | “yes” or “no” |

## Java-code

### Additional java code to be executed after matching the regular expression

## New-state

### Manually switches to the state after executing java code

## Conflict Resolution Rules

### When more than one regular expressions matches the input, JavaCC uses two rules to decide which regular expression to use:

### **Maximal Munch**: JavaCC uses the regular expression which consumes the largest amount of input data.

### **Order**: If two regular expressions can match exactly the same string (of same length) JavaCC uses the first one listed.