# JAVACC PARSER OPTIONS

The following JavaCC options may be useful for debugging your work:

DEBUG\_LOOKAHEAD =true;

DEBUG\_PARSER=true;

# RUDIMENTS

### Upper and lower case reversed for terminals and non-terminals.

### All non-terminals are function calls.

### After Token definitions:

void non-terminal() :

{ declarations }

{ prod

| prod

| prod

}

### Tokens: either <NAME> or "actual string" allowed

### Shorthands: | \* + ? allowed (x)? = [x]

### ε productions:

{} /\* nothing \*/

### Or-ed productions are tried in the order presented

### Example:

##### IF\_STAT 🡪 "if" COND "then" STAT "else" STAT "end"

##### IF\_STAT 🡪 "if" COND "then" STAT "end"

void if\_stat() :

{}

{ "if" cond() "then" stat() "else" stat() "end"

| "if" cond() "then" stat() "end"

}

# LL ISSUES

## Global Lookaheads

### Default: JavaCC assumes language is LL(1)

### Can be made LL(k) by setting global LOOKAHEAD(k) at top of file

##### Unacceptable as previously discussed

## Local Lookaheads

### Can use local lookahead specific to a specific point in a specific production, called a decision point.

void S() :

{}

{ "a" "b" "c"

| "a" "d" "c"

}

Decision point right before first "a"

🡪 replace by:

void S() :

{}

{ LOOKAHEAD(2) "a" "b" "c"

| "a" "d" "c"

}

### Second Example:

void S() :

{}

{ "a" "b" "0"

| "a" "b" "1"

}

Solution 1 – no factoring

void S() :

{}

{ LOOKAHEAD(3)"a" "b" "0"

| "a" "b" "1"

}

Solution 2 – partial factoring

void S() :

{}

{ "a" (LOOKAHEAD(2) "b" "0" | "b" "1")

}

Solution 3 – full factoring

void S() :

{}

{ "a" "b"("0"|"1")

}

### Compare and discuss backtracking.

## Syntactic Lookaheads

### Example:

void S() :

{}

{ ("a")+ "0"

| ("a" | "b")+ "1"

}

Don't know how many letters to look ahead

### Solution:

void S() :

{}

{ LOOKAHEAD(("a")+ “0”) ("a")+ "0"

| ("a" | "b")+ "1"

}

### How much can it lookahead?

##### Possibly the entire program

##### VERY COSTLY 🡪 AVOID!!!

##### Very few non-terminals in the assignment need them.

### In reality your program would probably look like this:

void S() :

{}

{ lots­\_of\_as\_then\_0()

| as\_and\_bs() "1"

}

void lots­\_of\_as\_then\_0 () :

{}

{ ("a")+ "0"}

void as\_and\_bs() :

{}

{ ("a" | "b")+

}

You may not notice until JavaCC tells you about a choice conflict in S.

🡪 resolution:

void S() :

{}

{ LOOKAHEAD(lots­\_of\_as\_then\_0 ()) lots­\_of\_as\_then\_0 ()

| as\_and\_bs() "1"

}

### Where to put the syntactic lookahead?

##### where you expect the shortest matching string, or the most likely string to be matched correctly so there is no need to backtrack.

## Lookahead-only Productions

### Example

void declaration() :

{}

{ LOOKAHEAD(fn\_declaration()) fn\_declaration()

| fn\_definition()

| other\_declaration()

}

void fn\_definition():

{}

{ type() <IDENTIFIER> "(" parameters() ")" "{" body() "}"

}

void fn\_declaration():

{}

{ type() <IDENTIFIER> "(" parameters() ")" ":" package() ";"

}

Don't want to read entire definition or declaration to decide which it is.

🡪 define a production simply for looking-ahead:

void fn\_decl\_lookahead():

{}

{ type() <IDENTIFIER> "(" parameters() ")" ":"

}

void declaration() :

{}

{ LOOKAHEAD(fn\_decl\_lookahead()) fn\_declaration()

| fn\_definition()

| other\_declaration()

}