

## 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]
- $\epsilon$  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()
}
```