Grammar School

Parsing is hard but it’s a lot easier with the right tools

John Haldeman – Hackforge – September 2017

Parsing

Using code to understand user or other external input

Expressiveness

Regular Languages

Context-Free Languages

Context-Sensitive Languages

Recursively Enumerable Languages

Expressiveness

Regular Languages } Regular Expressions/ Finite Automata

Context-Free Languages

Context-Sensitive Languages } We'll focus on this area

Recursively Enumerable Languages } Turing Complete Languages

Tools

Regular Languages } Regular Expression Libraries

Context-Free Languages } ANTLR, yacc, bison, etc. (although only a subset of context-sensitive languages are expressible)

Context-Sensitive Languages

Recursively Enumerable Languages } Turing Complete Languages (Java, C, C++, Lisp, Haskell, Prolog, etc.)

ANTLR

- We'll be looking at ANTLR
  - LL (top-down) parsing
  - You build a context free grammar and then interact with the generated parser code using visitors and listeners
  - In addition to all context free languages, you can parse some, but not all, context-sensitive languages using the framework

Let’s Make a Language

- Hackforge Event Query Language (HeQL – Pronounced “heckle”)
  - Cool code name: HeliumQL
- SQL-Like language to query non-relational data (because that’s fashionable)
- Allows you to query hackforge events – the hackfeventbot will interpret our new language

Examples – Select Next Event

```
Hey @hackforgeeventbot can you get me the next
event
<?heql
SELECT NEXT title, content, excerpt,
start_date_time, end_date_time, meetup_link,
facebook_link, group
FROM events?> | pretty
```

Examples – Select Next Software Guild Event About Grammars

```
Hey @hackforgeeventbot can you get me the next event
for software guild that has to do with grammars


<?heql
     SELECT NEXT title, content, excerpt,
start_date_time, end_date_time, meetup_link,
facebook_link, group
FROM events
WHERE group = "Software Guild" and title LIKE
"%Grammar%";
?> | json
```

Examples – Make a hackforge Hammer say Some Links

```
Hey @hackforgeeventbot can you get me those links
but make it look like a hammer is saying it because
that’s awesome


<?heql
     SELECT NEXT title, meetup_link, facebook_link
FROM events
WHERE group = "Software Guild" and title LIKE
"%Grammar%";
?> | hammersay
```

Examples – Return Available Groups

```
Hey @hackforgeeventbot what groups are there to
choose from?

<?heql
      SELECT DISTINCT GROUP
      FROM events;
?> | json
```

This clearly isn't anywhere near all of SQL – Our goal here is to give you a taste of what parsing with ANTLR is like, not to be complete

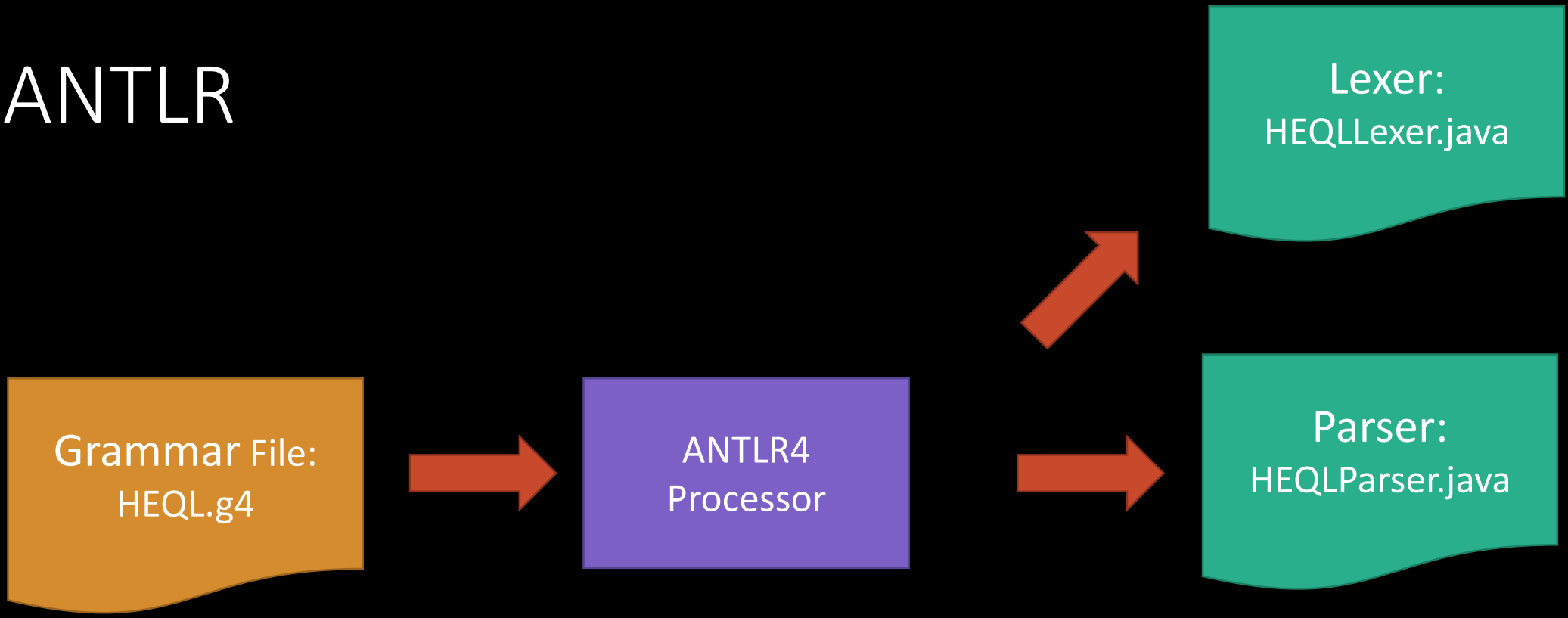
Breaking the Language Down

```
introduction text (with hackfeventbot mention)
<?heql SELECT NEXT field1, field2, …, field3
FROM events
[WHERE
        field (LIKE | =) string_condition
        [(AND | OR) field (LIKE | =) string_condition] …
]
?> | (pretty|json|hammersay)
```

Breaking the Language Down

```
introduction text (with hackfeventbot mention)
<?heql SELECT DISTINCT field
FROM events
[WHERE field (LIKE | =) string_condition]
       [(AND | OR) field (LIKE | =) string_condition] …
?> | (pretty|json|hammersay)
```

ANTLR



We'll be using the Java target but these targets are also available:
C# (and an alternate C# target)
Python (2 and 3)
JavaScript
Go
C++
Swift

Constructing the Grammar (bottom to top) – Fragments – Used Later for Case Insensitivity

```
fragment A: [aA];
fragment B: [bB];
fragment C: [cC];
fragment D: [dD];
fragment E: [eE];
fragment F: [fF];
fragment G: [gG];
fragment H: [hH];
fragment I: [iI];
fragment J: [jJ];
fragment K: [kK];
fragment L: [lL];
fragment M: [mM];
```

```
fragment N: [nN];
fragment O: [oO];
fragment P: [pP];
fragment Q: [qQ];
fragment R: [rR];
fragment S: [sS];
fragment T: [tT];
fragment U: [uU];
fragment V: [vV];
fragment W: [wW];
fragment X: [xX];
fragment Y: [yY];
fragment Z: [zZ];
```

Constructing the Grammar (bottom to top) – Special Tokens

- Tokens are always capitalized
- Skip Whitespace
- Any other undefined token - makes all errors parser errors instead of mix of parser/lexer

```
WS : [ \t\r\n]+ -> skip ;        // skip spaces, tabs, newlines

UNEXPECTED_CHAR: .;        // In case we don't define a token
```

The Rest of the Tokens

- Keywords
- Operators
- Objects
- Literals

```
HEQL: H E Q L;

AND: A N D;

OR: O R;

OPERATOR: (L I K E) | ('=');

DISTINCT: D I S T I N C T;

FROM: F R O M;

WHERE: W H E R E;

EVENTS: E V E N T S;

NEXT: N E X T;

SELECT: S E L E C T;

OBJECT: [a-zA-Z_]+;

LITERAL: '\'' ( ~'\'' | '\\\'' )* '\'';
```

Parser Rules

- Note: Many of these parser rules just have a single token
- This was done to make constructing a parser listener (used later) easier, but is not strictly necessary or common

```
logicalOr: OR;

logicalAnd: AND;

leftParen: '(';

rightParen: ')';

selField: OBJECT;

condField: OBJECT;

fieldList: (selField (',' selField)*) | '*';

operator: OPERATOR;

literal: LITERAL;

format: FORMAT;
```

Parser Rules

- Condition – A self referencing parser rule for the WHERE block

```
condition: (condField|literal) operator (condField|literal)  # condComparison
| condition logicalAnd condition                              # condLogicalAnd
| leftParen condition rightParen                             # condParen
| condition logicalOr condition                              # condLogicalOr
;
```

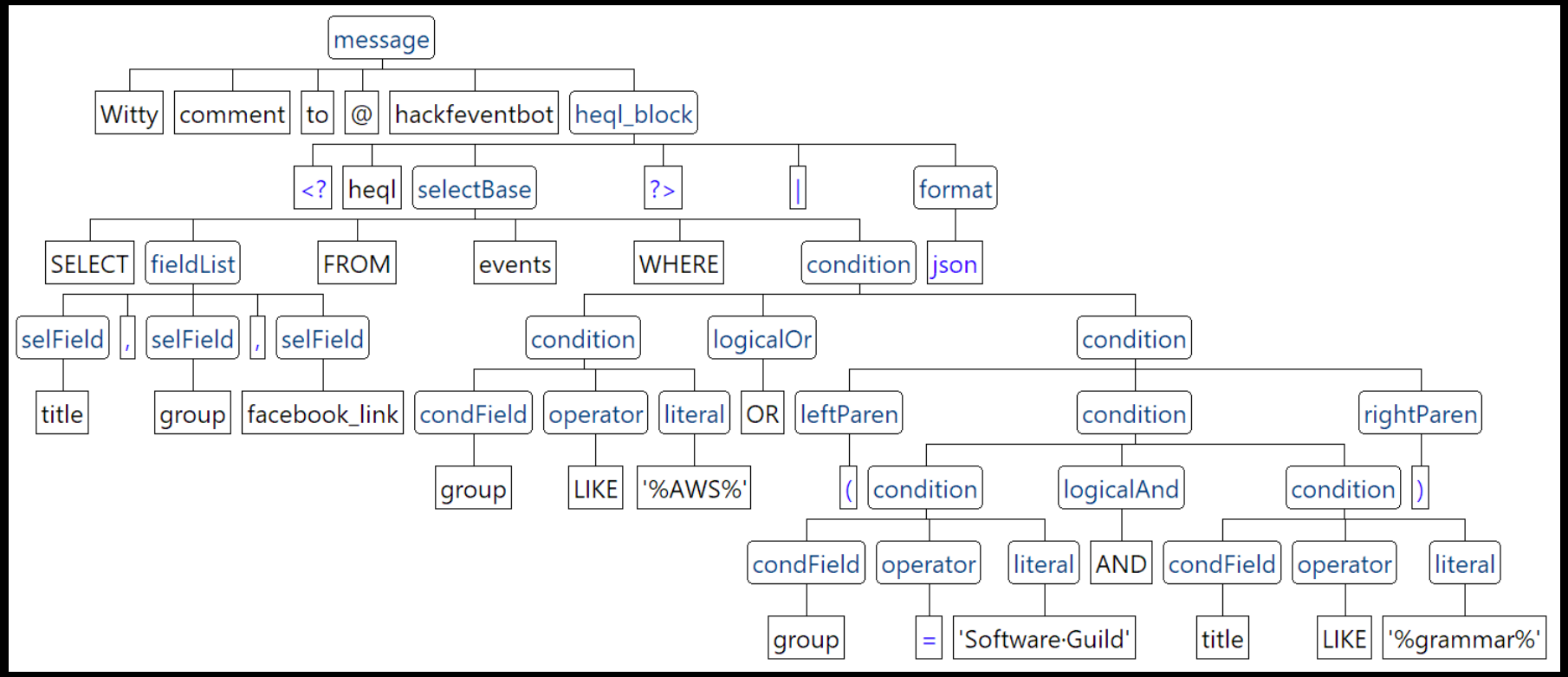
indicators are used later in the listener

Remaining Parser Rules

```
message: .*? heql_block .*?;

heql_block: '<?' HEQL (selectNext|selectDistinct|selectBase) '?>' ('|' format)?;

selectBase: SELECT fieldList FROM EVENTS (WHERE condition)? ';'?;

selectNext: SELECT NEXT fieldList FROM EVENTS (WHERE condition)? ';'?;

selectDistinct: SELECT DISTINCT selField FROM EVENTS (WHERE condition)? ';'?;
```

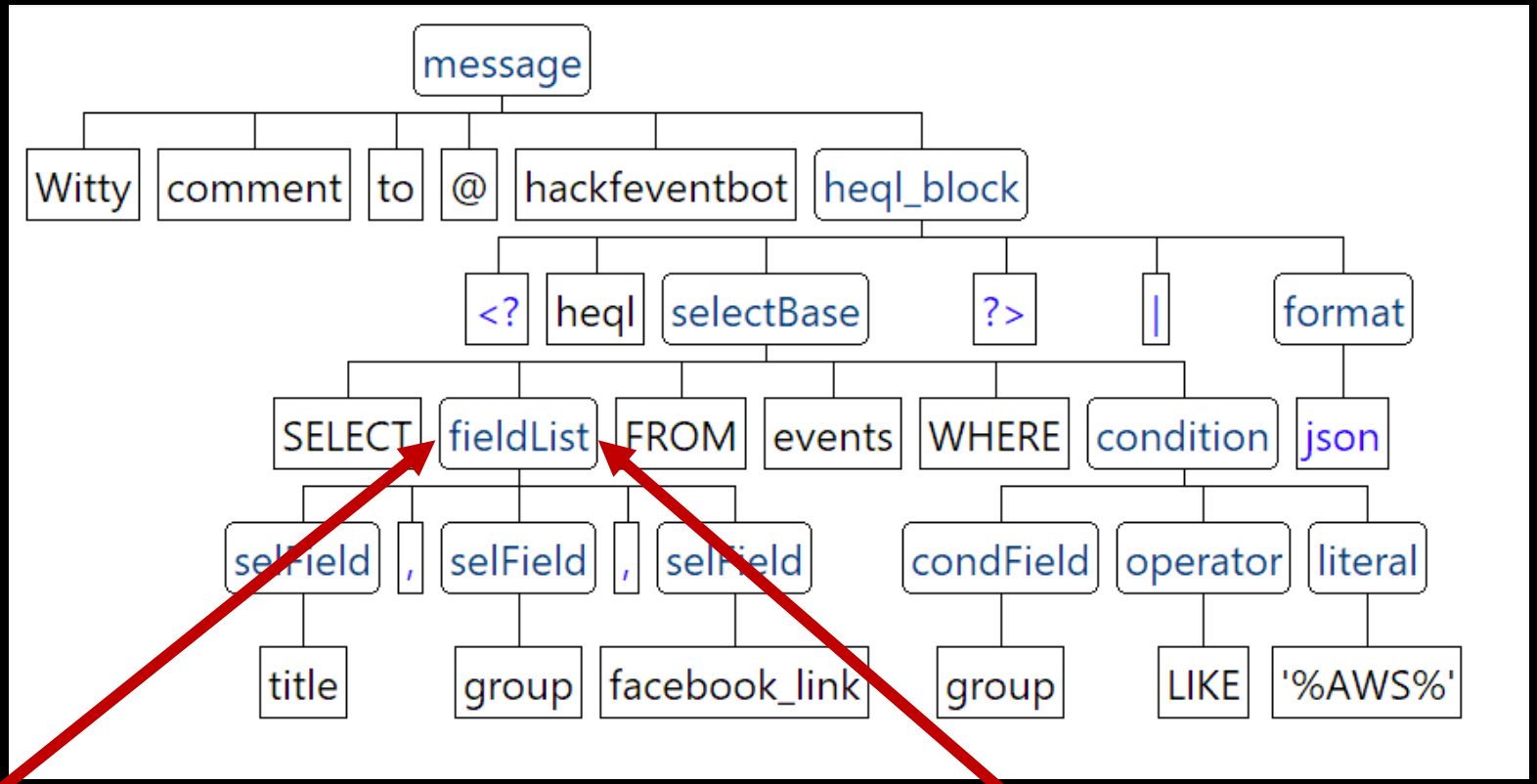
Result – Parse Tree!

```
Witty comment to @hackfeventbot
<?heql
SELECT title, group, facebook_link FROM events
WHERE group LIKE '%AWS%' OR (group = 'Software Guild' AND title LIKE
'%grammar%')
?> | json
```



Now we need to do something with it…

- Listeners let you define methods that are triggered when each node in the parse tree is visited
- Order of traversal is depth first



Method enterFieldList triggered

Method exitFieldList triggered

Listener – Setting Command Type

```
@Override
public void enterSelectDistinct(SelectDistinctContext ctx) {
   super.enterSelectDistinct(ctx);

   type = TYPE_SELECT_DISTINCT;
}

@Override
public void enterSelectNext(SelectNextContext ctx) {
   super.enterSelectNext(ctx);

   type = TYPE_SELECT_NEXT;
}

@Override
public void enterSelectBase(SelectBaseContext ctx) {
   super.enterSelectBase(ctx);

   type = TYPE_BASE_SELECT;
}
```

Listener – Building a list of fields from the SELECT field list

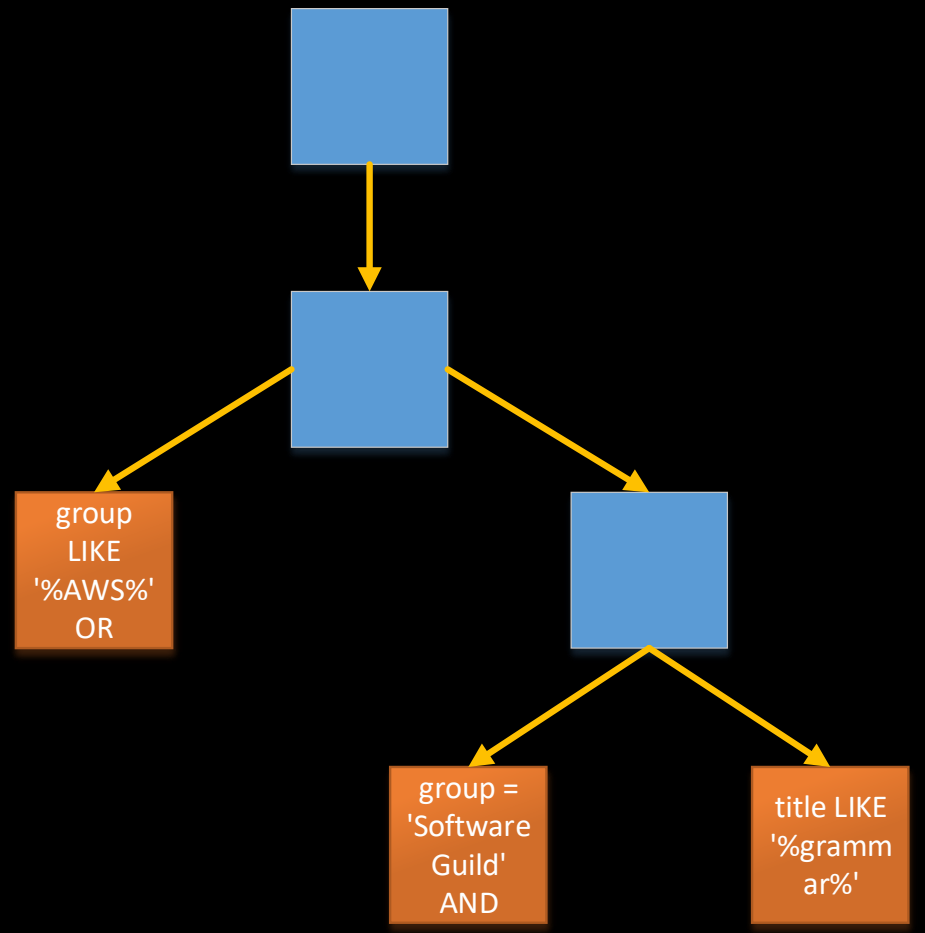
```
private Vector<String> fieldList = new Vector<String>();
```

```
@Override
public void enterSelField(SelFieldContext ctx) {
    super.enterSelField(ctx);
    String fieldText = ctx.getStart().getText();
    fieldList.add(fieldText);
}
```

Conditions – Evaluation

Build an expression tree:

```
WHERE group LIKE '%AWS%'
OR (
   group = 'Software Guild'
   AND title LIKE '%grammar%'
)
```



Evaluate the tree with a depth first traversal
(but being careful to evaluate ORs
on the same level after the ands)

Evaluation is out of scope for the presentation but for details see method:

`HEQLCondition.evaluate(HackforgeEventFromSite event)`

Back to the Listener – Constructing the Condition Tree – Create a Leaf Condition

```
@Override
public void enterCondComparison(CondComparisonContext ctx) {
   super.enterCondComparison(ctx);

   currentCondition = new HEQLCondition();
   currentConditionParent.addChild(currentCondition);
}
```

Listener – Populate a Leaf Condition with Data

```
@Override
public void enterLogicalAnd(LogicalAndContext ctx) {
   super.enterLogicalAnd(ctx);
   currentCondition.setType(HEQLCondition.TYPE_AND);
}

@Override
public void enterLogicalOr(LogicalOrContext ctx) {
   super.enterLogicalOr(ctx);
   currentCondition.setType(HEQLCondition.TYPE_OR);
}
```

```
@Override
public void enterCondField(CondFieldContext ctx) {
super.enterCondField(ctx);
String fieldText = ctx.getStart().getText();
currentCondition.setField(fieldText);
}
```

```
@Override
public void enterLiteral(LiteralContext ctx) {
   super.enterLiteral(ctx);

   currentCondition.setLiteral(ctx.getStart().getText());
}

@Override
public void enterOperator(OperatorContext ctx) {
   super.enterOperator(ctx);
   String opText = ctx.getStart().getText().toUpperCase();

   if(opText.equals("LIKE"))
      currentCondition.setOperator(HEQLCondition.OP_LIKE);
   else if(opText.equals("="))
      currentCondition.setOperator(HEQLCondition.OP_EQUALS);
}
```

Listener – Enter a Parenthesized Condition – Move Down the Tree

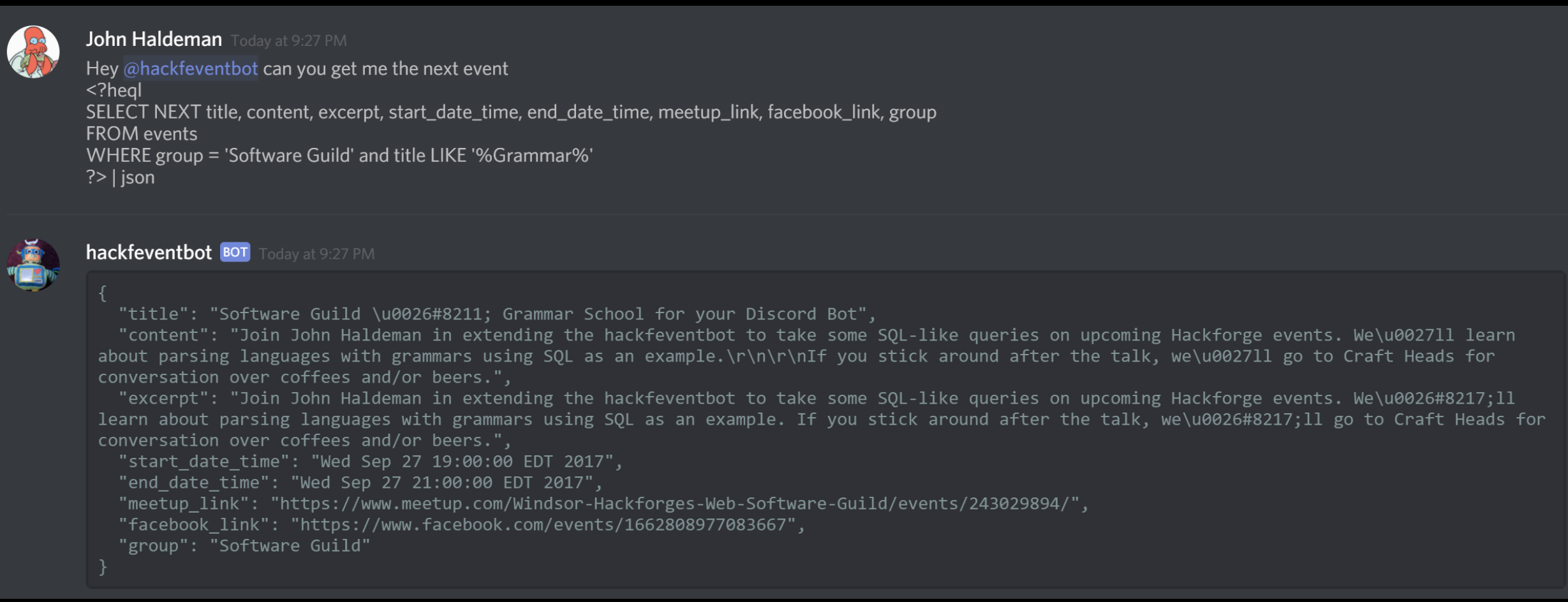
```
@Override
public void enterCondParen(CondParenContext ctx) {
    super.enterCondParen(ctx);

    HEQLCondition newParent = new HEQLCondition();
    currentConditionParent.addChild(newParent);
    currentConditionParent = newParent;
}
```

Listener – Exit a Parenthesized Condition – Move Back Up the Tree

```
@Override
public void exitCondParen(CondParenContext ctx) {
    super.exitCondParen(ctx);

    currentConditionParent = currentConditionParent.getParent();
}
```

For Every Last Detail – You can explore the bot's code

https://github.com/johnhaldeman/hackfeventbot



Summary

Parsing – Understanding input

Grammars – Provide a powerful way (more powerful than regular expressions) to express an input's format

ANTLR – A toolset to generate parsers and lexers from Grammars with multiple target languages (Java, javascript, C#, python, etc.)

Listeners – A great way to interact with parsing output

Expression Trees – A good way to compute logical or arithmetic expressions