

EECS 498 System Design of a Search Engine

Winter 2026

Lecture 16: The query compiler

Nicole Hamilton

[https://web.eecs.umich.edu/~nham/
nham@umich.edu](https://web.eecs.umich.edu/~nham/nham@umich.edu)

agenda

1. details
2. name the octopus!
3. TDRD parsing
4. the query compiler

agenda

1. details
2. name the octopus!
3. TDRD parsing
4. the query compiler

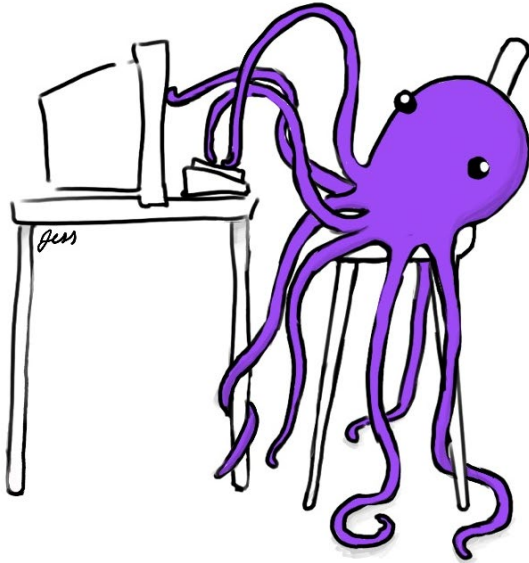
details

1. Midterm course eval reports posted to Canvas. Most of you were pretty happy and you liked the AG homework assignments.
 - a. Some thought we spent too much time on OS concepts they already knew while others reported that without 482, it's been difficult.
 - b. A ~~surprising~~ *refreshing* number of you liked creating your own STL.
2. Still need to finish the robots.txt assignment.
 - a. Adding sitemap:.
 - b. Clarification on priority: If two directives apply equally, choose the one from a rule that specifies a literal user-agent over one that specified *.
 - c. May release an early version of the starter files for you to work on before the AG is up.
3. Working on finalizing how to order a tee shirt.

agenda

1. details
2. name the octopus!
3. TDRD parsing
4. the query compiler

We have a design, thanks to Jess Morton.



EECS 498 WINTER 2026
SEARCH ENGINE DEV TEAM

Front



I SURVIVED NICOLE HAMILTON'S
SEARCH ENGINE DESIGN CLASS

Back

any we should add to the survey?

Octothorpe

Jeff

Zork

tf/idf

Purple

Sami

Carlos

Rahul

Jess

So Cute

Searchy

Rose

Gilbert

George

Octavious

Hamilton

Purelle

Nicky

Squishy

Seashell

Ultros

agenda

1. details
2. name the octopus!
3. TDRD parsing
4. the query compiler

the compiler problem

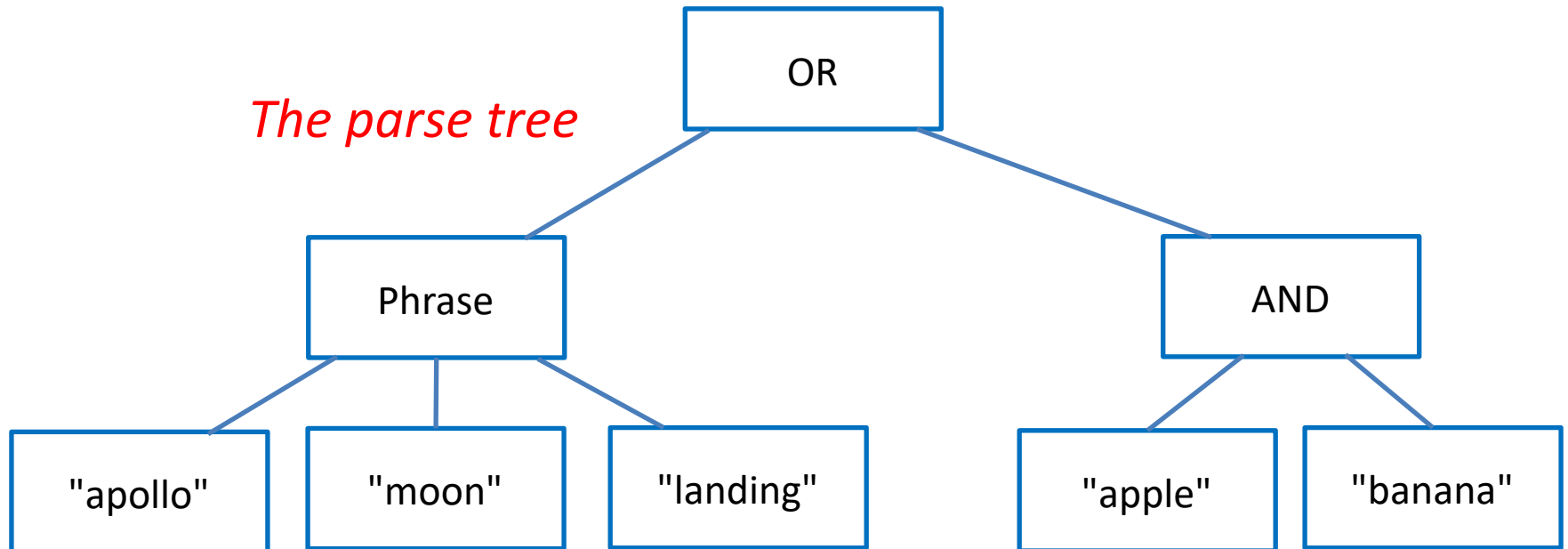
Need a way to compile a query into a structure of ISRs that can be passed to the constraint solver.

Both are recursive and they match one-to-one.

To solve this, we'll introduce some notation and a parsing technique called top-down recursive descent (TDRD).

The query language and the ISRs can be recursive

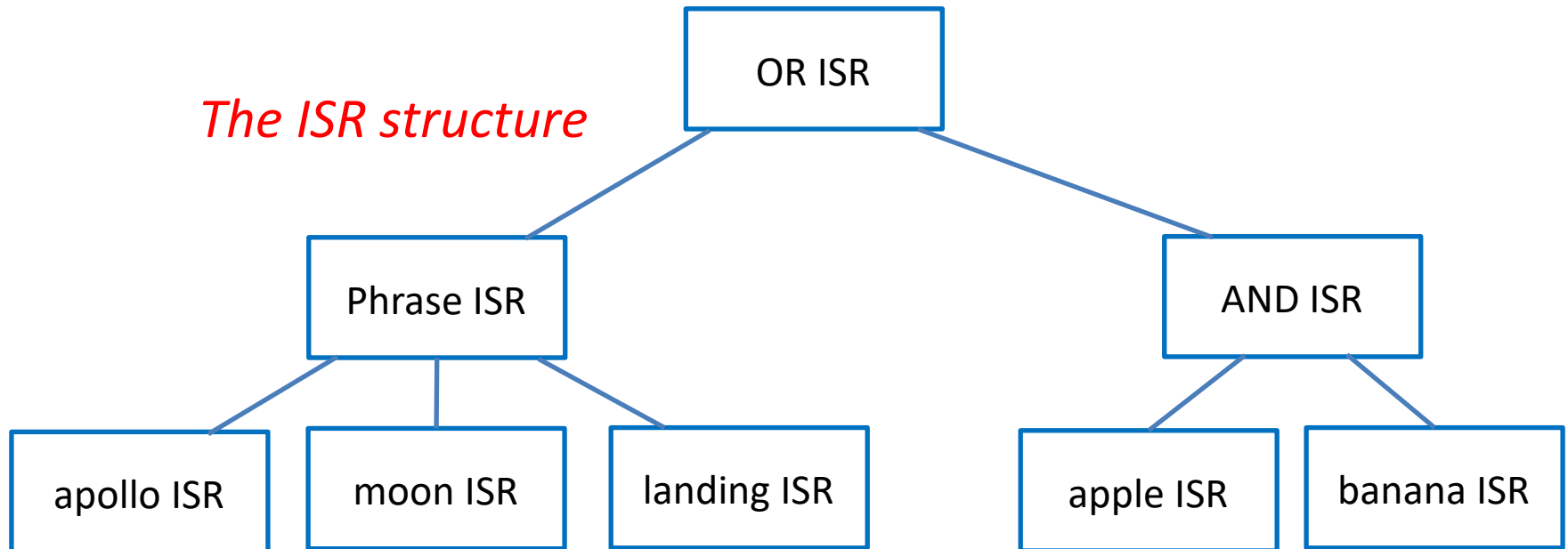
"apollo moon landing" | (apple banana)



The query language and the ISRs can be recursive

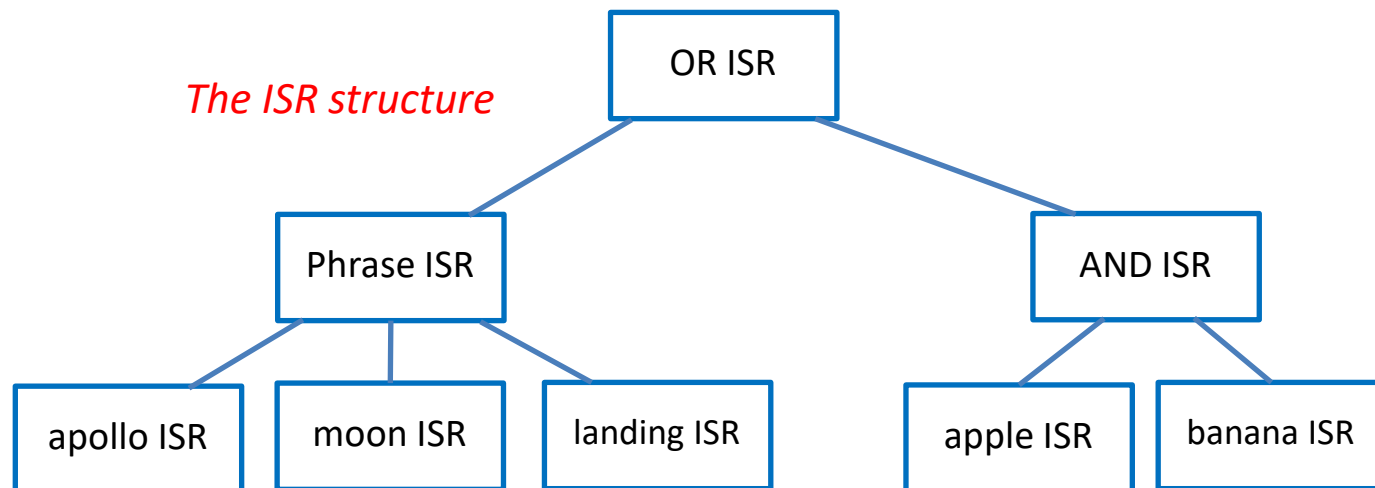
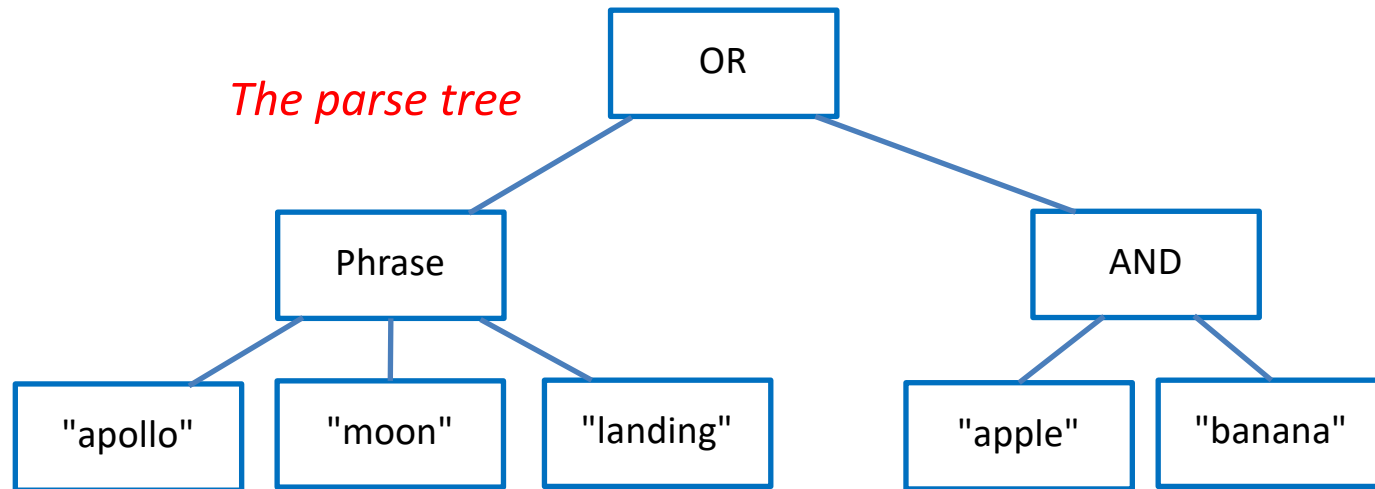
"apollo moon landing" | (apple banana)

The ISR structure



"apollo moon landing" | (apple banana)

The trees are the same.



two parts to compiling

1. Tokenizing the input to recognize keywords, literals, operators and variable names. For example, in C++:

`"a=b->c"` → { `"a"`, `"="`, `"b"`, `"->"`, `"c"` }

2. Parsing the stream of tokens to recognize the language constructs, compiling that into an executable or *evaluable* form.

Tokenizing

Can be quite simple, e.g.,
breaking on white space or
special characters.

```
#include <iostream>
using namespace std;

int main( )
{
    string token;
    while ( cin >> token )
    {
        // ...
    }
}
```

But most languages, like C++, have lots of operators of differing precedence and they can be run together.

The tokenizer has to separate and recognize them.

Precedence	Operator	Description
1	::	Scope resolution
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation
4	.* ->*	Pointer-to-member
5	a*b a/b a%b	Multiplication, division, and remainder
6	a+b a-b	Addition and subtraction
7	<< >>	Bitwise left shift and right shift
8	<=>	Three-way comparison operator (C++20)
9	< <= > >=	For relational operators < and ≤ and > and ≥ respectively
10	== !=	For equality operators = and ≠ respectively
11	&	Bitwise AND
12	^	Bitwise XOR (exclusive or)
13		Bitwise OR (inclusive or)
14	&&	Logical AND
15		Logical OR
16	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^= =	Ternary conditional throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR
17	,	Comma

The tokenizer produces a stream of tokens you can examine, consume (take), or consume only if it matches what you're looking for (match).

```
class TokenStream
{
public:
    Token *CurrentToken( );
    Token *TakeToken( );
    bool Match( TokenType t );
    TokenStream( );
    TokenStream( char *filename );
    TokenStream( ifstream &is );

private:
    ifstream input;
    string currentTokenString;
    Token *currentToken;
};
```

An example taken
(sort of) from my C
shell's parser

```
enum TokenType
{
    TokenInvalid = -1, TokenEOF = 0,
    TokenNotBuiltin,
    TokenPlus, TokenNOT, TokenMinus, TokenDoubleQuote, ...
};
```

```
class Token
{
public:
    virtual TokenType GetTokenType( ) const;
    virtual char *TokenString( ) const
    {
        return nullptr;
    }
};
```

```
class NotBuiltin : Token
{
public:
    TokenType GetTokenType( ) const
    {
        return TokenNotBuiltin;
    }
    char *TokenString( ) const
    {
        return string;
    }
private:
    char *string;
};
```

An example taken
(sort of) from my C
shell's parser

```
// Constructors for reserved words and tokens that define the symbolic
// name, the string representation, and the TokenType value.
// (Missing value mean add one to the previous value.)
```

```
Token( TokenInvalid,          NULL,          = -1 ),
Token( TokenEOF,              NULL,          = 0 ),

Token( TokenNotBuiltin,      NULL,          ),

Token( TokenPlus,            "+",          ),
Token( FirstToken,          NULL,          = TokenPlus ),
Token( FirstOperator,        NULL,          = FirstToken ),
Token( TokenNOT,             "NOT",        ),
Token( TokenMinus,           "-",          = TokenNOT ),
Token( TokenDoubleQuote,     "\"",          ),
Token( TokenLeftParenthesis, "(",          ),
Token( TokenRightParenthesis, ")",          ),
Token( TokenLeftBracket,     "[",          ),
Token( TokenRightBracket,    "]",          ),
Token( TokenAnd,             "&&",        ),
Token( TokenOr,              "||",        ),
Token( TokenOR,              "OR",        = TokenOr ),
Token( TokenVerticalBar,     "|",          = TokenOR ),
Token( TokenColon,           ":",          ),
Token( TokenEqual,           "=",          ),
Token( TokenMultiply,        "*",          ),
Token( TokenDivide,          "/*",        ),
```

two basic approaches to parsing

1. *Bottom-up*: Uses a stack and a machine generated comparison table that decides, based on the top of the stack and the next token, whether to
 - a. Push the token onto the stack or
 - b. Do a *reduction* on the top items, replacing them with something else and emitting some code to implement whatever language construct was just recognized.

Typically written with a "compiler compiler", e.g., YACC / LEX or BISON / Flex. LEX and Flex do tokenizing. YACC and BISON recognize the language.

2. *Top-down recursive descent (TDRD)*: Uses recursive functions that look for each of the language constructs. Instead of an explicit stack, it uses the call stack to look for matches. Usually hand-written.

top-down recursive descent

I happen to like this strategy and used it in my C shell and at Microsoft on both the search engine query compiler and for SQL parser for Windows Update.

1. It involves no dependencies on tools.
2. It's quite fast.
3. The code is pretty easy to read and naturally expresses precedence in the language.
4. Lots of opportunity to special-case the parsing if need be.

Only disadvantage is you have to write a procedure for every construct in the language.

BNF

Need a way of describing what constitute legal constructs in the language.

Usually written in Backus-Naur form (BNF).

```
<Sentence> ::= <Noun> <Verb> "."  
<Noun> ::= "Cats" | "Dogs"  
<Verb> ::= "meow" | "bark"
```

BNF

Backus-Naur Form or Backus-normal Form.

Notation for describing the *syntax* a recursive language with a set of possible tokens.

Consists of a set of rules that describe allowable sequences of tokens and how they can be combined into higher-level constructs.

Each rule is called a *production*.

<Sentence> ::= <Noun> <Verb> "."

<Noun> ::= "Cats" | "Dogs"

<Verb> ::= "meow" | "bark"

BNF

`::=` means the LHS is composed of the sequence elements on the right.

`a | b` means either a or b is a match.

`<Sentence> ::= <Noun> <Verb> "."`

`<Noun> ::= "Cats" | "Dogs"`

`<Verb> ::= "meow" | "bark"`

BNF

What sentences are allowed?

Are all of them sensible?

Are some sentences not allowed?

`<Sentence> ::= <Noun> <Verb> "."`

`<Noun> ::= "Cats" | "Dogs"`

`<Verb> ::= "meow" | "bark"`

BNF

Just because the syntax allows you to write something doesn't mean it's guaranteed to be sensible.

`<Sentence> ::= <Noun> <Verb> "."`

`<Noun> ::= "Cats" | "Dogs"`

`<Verb> ::= "meow" | "bark"`

BNF

Legal sentences:

Cats meow.

Cats bark.

Dogs meow.

Dogs bark.

but cats don't bark and dogs
don't meow.

Not a sentence:

Birds tweet.

$\langle \text{Sentence} \rangle ::= \langle \text{Noun} \rangle \langle \text{Verb} \rangle "."$

$\langle \text{Noun} \rangle ::= \text{"Cats"} \mid \text{"Dogs"}$

$\langle \text{Verb} \rangle ::= \text{"meow"} \mid \text{"bark"}$

BNF

BNF allows for repetition and recursion.

{ ... } means zero or more repetitions.

Notice how this example creates precedence.

Is multiplication done before or after addition?

```
<Add> ::= <Multiply> { "+" <Multiply> }  
<Multiply> ::= <Base> { "*" <Base> }  
<Base> ::= "(" <Add> ")" | <Literal>  
<Literal> :: an integer string
```

BNF

Notice how this creates precedence.

How would these be parsed?

5 + 3 * 4 + 6

5 +

* 4

hello

```
<Add> ::= <Multiply> { "+" <Multiply> }  
<Multiply> ::= <Base> { "*" <Base> }  
<Base> ::= "(" <Add> ")" | <Literal>  
<Literal> :: an integer string
```

BNF

Complete notation:

$\langle x \rangle ::= \langle y \rangle \langle z \rangle$

$\langle x \rangle$ is composed of a $\langle y \rangle$ followed by a $\langle z \rangle$.

$\langle x \rangle ::= \langle y \rangle \mid \langle z \rangle$

$\langle x \rangle$ is composed of either a $\langle y \rangle$ or a $\langle z \rangle$.

$\langle x \rangle ::= \langle y \rangle [\langle z \rangle]$

$\langle x \rangle$ is composed of a $\langle y \rangle$ followed by an optional $\langle z \rangle$.

$\langle x \rangle ::= \langle y \rangle \{ \langle z \rangle \}$

$\langle x \rangle$ is composed of a $\langle y \rangle$ followed by zero or more $\langle z \rangle$ elements.

$\langle x \rangle ::= \text{"c"}$

$\langle x \rangle$ is a literal string, specified between the quotes.

operator precedence

Consider typical C/C++
operator precedence.

()

* / %

+ -

< <= >=

>

== !=

&&

||

rules are written in reverse of the desired precedence

()	<code><OrExpression> ::= <AndExpression> { ' ' <AndExpression> }</code>
* / %	<code><AndExpression> ::= <EquateExpression> { '&&' <EquateExpression> }</code>
+ -	<code><EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }</code>
< <= >= >	<code><EquateOp> ::= '==' '!='</code>
== !=	<code><RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }</code>
&&	<code><RelateOp> ::= '<' '<=' '>=' '>'</code>
	<code><AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }</code>
	<code><AddOp> ::= '+' '-'</code>
	<code><MultiplyExpression> ::= <BaseExpression> { <MultiplyOp> <BaseExpression> }</code>
	<code><MultiplyOp> ::= '*' '/' '%'</code>
	<code><BaseExpression> ::= '(' <BaseExpression> ')' <Literal></code>
	<code><Literal> ::= an integer string</code>

rules are written in reverse of the desired precedence

Each rule gets turned into a procedure that tries to match that target on the input stream.

```
<OrExpression> ::= <AndExpression> { '||' <AndExpression> }
<AndExpression> ::= <EquateExpression> { '&&' <EquateExpression> }
<EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }
<EquateOp> ::= '==' | '!='
<RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }
<RelateOp> ::= '<' | '<=' | '>=' | '>'
<AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }
<AddOp> ::= '+' | '-'
<MultiplyExpression> ::= <BaseExpression> { <MultiplyOp> <BaseExpression> }
<MultiplyOp> ::= '*' | '/' | '%'
<BaseExpression> ::= '(' <BaseExpression> ')' | <Literal>
<Literal> ::= an integer string
```

Rules are written in reverse of the desired precedence

The grammar is recursive but only right-recursive.

No production has the target as the first component of itself.

```
<OrExpression> ::= <AndExpression> { '||' <AndExpression> }
<AndExpression> ::= <EquateExpression> { '&&' <EquateExpression> }
<EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }
<EquateOp> ::= '==' | '!='
<RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }
<RelateOp> ::= '<' | '<=' | '>=' | '>'
<AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }
<AddOp> ::= '+' | '-'
<MultiplyExpression> ::= <BaseExpression> { <MultiplyOp> <BaseExpression> }
<MultiplyOp> ::= '*' | '/' | '%'
<BaseExpression> ::= '(' <BaseExpression> ')' | <Literal>
<Literal> ::= an integer string
```

Consider how we'd implement a parser and interpreter for part of this.

Start with simple notions of a *tuple* and a list of tuples.

(A tuple is a finite, ordered sequence of elements.)

```
class Tuple
{
public:
    Tuple      *Next;
    TokenType Operation;

    virtual int Eval( );
    Tuple( );
    virtual ~Tuple( );
}

class TupleList : Tuple
{
public:
    Tuple *Top,
          *Bottom;

    void Append( Tuple *t );
    TupleList( );
    ~TupleList( );
}
```

```
// <AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }  
//  
// <AddOp> ::= '+' | '-'
```

```
TupleList *Parser::FindAddExpression( )  
{  
    Tuple *e;  
    if ( e = FindMultiplyExpression( ) )  
    {  
        TokenType t;  
        // Consume the next token if it's a + or a -.  
        while ( t = FindAddOp( ) )  
        {  
            AddExpression *a= new AddExpression( t );  
            a->Append( e );  
            if ( e = FindMultiplyExpression( ) )  
            {  
                a->Append( e );  
                e = a;  
            }  
            else  
            {  
                delete a;  
                return nullptr;  
            }  
        }  
    }  
    return e;  
}
```

```
// <AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }
//
// <AddOp> ::= '+' | '-'
```

```
class AddExpression : TupleList
```

```
{
```

```
public:
```

```
    TokenType Operation;
```

```
    int Eval( )
```

```
    {
```

```
        int sum = 0;
```

```
        switch ( Operation )
```

```
        {
```

```
            case TokenPlus:
```

```
                for ( Tuple *t = top; t; t = t->Next )
```

```
                    sum += t->Eval( );
```

```
                break;
```

```
            case TokenMinus:
```

```
                for ( sum = t->Eval( ), t = t->Next; t; t = t->next )
```

```
                    sum -= t->Eval( );
```

```
        }
```

```
        return sum;
```

```
    }
```

```
    TupleType GetType( ) const
```

```
    {
```

```
        return TT_AddExpression;
```

```
    }
```

```
}
```

```
// <MultiplyExpression> ::- <Expression> { <MultiplyOp> <Expression> }  
//  
// <MultiplyOp> ::= '*' | '/' | '%'
```

```
TupleList *Parser::FindMultiplyExpression( )  
{  
    Tuple *e;  
    if ( e = FindExpression( ) )  
    {  
        TokenType t;  
        // Consume the next token if it's an *, /, or %.  
        while ( t = FindMultiplyOp( ) )  
        {  
            MultiplyExpression *m = new MultiplyExpression( t );  
            m->Append( e );  
            if ( e = FindBaseExpression( ) )  
            {  
                m->Append( e );  
                e = a;  
            }  
            else  
            {  
                delete a;  
                return nullptr;  
            }  
        }  
    }  
    return e;  
}
```

```
// <MultiplyExpression> ::- <BaseExpression> { <MultiplyOp> <BaseExpression> }
//
// <MultiplyOp> ::= '*' | '/' | '%'
```

```
class MultiplyExpression : TupleList
```

```
{
public:
    int Eval( )
    {
        int result = 0;
        switch ( Operation )
        {
            case TokenMultiply:
                for ( Tuple *t = top; t; t = t->Next )
                    result *= t->Eval( );
                break;
            case TokenDivide:
                for ( result = t->Eval( ), t = t->Next; t; t = t->next )
                    result /= t->Eval( );
                break;
            case TokenModulo:
                for ( result = t->Eval( ), t = t->Next; t; t = t->next )
                    result %= t->Eval( );
        }

        return result;
    }
    TupleType GetType( ) const
    {
        return TT_MultiplyExpression;
    }
    TokenType Operation;
}
```

agenda

1. details
2. TDRD parsing
3. the query compiler

query language

Basic requirements are very simple.

1. Individual words.
2. Phrases in double quotes.
3. AND, OR and NOT.
4. Parenthesis.

Average query on Bing about 2.4 words in 2004.
With LLMs interpreting queries today, they're probably longer now.

simple search engine query language

`<Constraint> ::= <BaseConstraint> { <OrOp> <BaseConstraint> }`

`<OrOp> ::= 'OR' | '|' | '||'`

`<BaseConstraint> ::= <SimpleConstraint> { [ <AndOp> ] <SimpleConstraint> }`

`<AndOp> ::= 'AND' | '&' | '&&'`

`<SimpleConstraint> ::= <Phrase> | <NestedConstraint> |
 <UnaryOp> <SimpleConstraint> |
 <SearchWord>`

`<UnaryOp> ::= '+' | '-' | 'NOT'`

`<Phrase> ::= '"' { <SearchWord> } '"'`

`<NestedConstraint> ::= '(' <Constraint> ')'`

```

class QueryParser
{
public:
    Token *FindNextToken( );
    Tuple *FindConstraint( );
    bool    FindOrOp( );
    Tuple *FindBaseConstraint( );
    bool    FindAndOp( );
    Tuple *FindSimpleConstraint( );
    Tuple *FindPhrase( );
    Tuple *FindNestedConstraint( );
    Tuple *FindSearchWord( );

    QueryParser( char *query );
};

```

```

class Tuple
{
public:
    virtual ISR *Compile( );
};

```

```

class Constraint : Tuple
{
    ISR *Compile( );
}

```