EECS 440 System Design of a Search Engine

Winter 2021

Lecture 13: The query compiler

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

Agenda

1. Course details.
2. Exam answers.
3. TDRD parsing.
4. The query compiler.

Agenda

1. Course details.
2. Exam answers.
3. TDRD parsing.
4. The query compiler.

details

1. The hashing and optional Bloom filter homeworks are now live on the AG. Working on publishing the expression parser and LinuxTiny server assignments.
2. Exams looked pretty good though obviously someone was not happy. Hope to have everything graded this weekend.
3. Remaining important lecture topics you need to build your engine: Query compiler, ranking, web server. After that, a potpourri of less essential topics.
4. You'll also have a chance to meet as a team with my staff in another week or so for another check-in.

Agenda

1. Course details.
2. Exam answers.
3. TDRD parsing.
4. The query compiler.

Agenda

1. Course details.
2. Exam answers.
3. TDRD parsing.
4. The query compiler.

The compiler problem

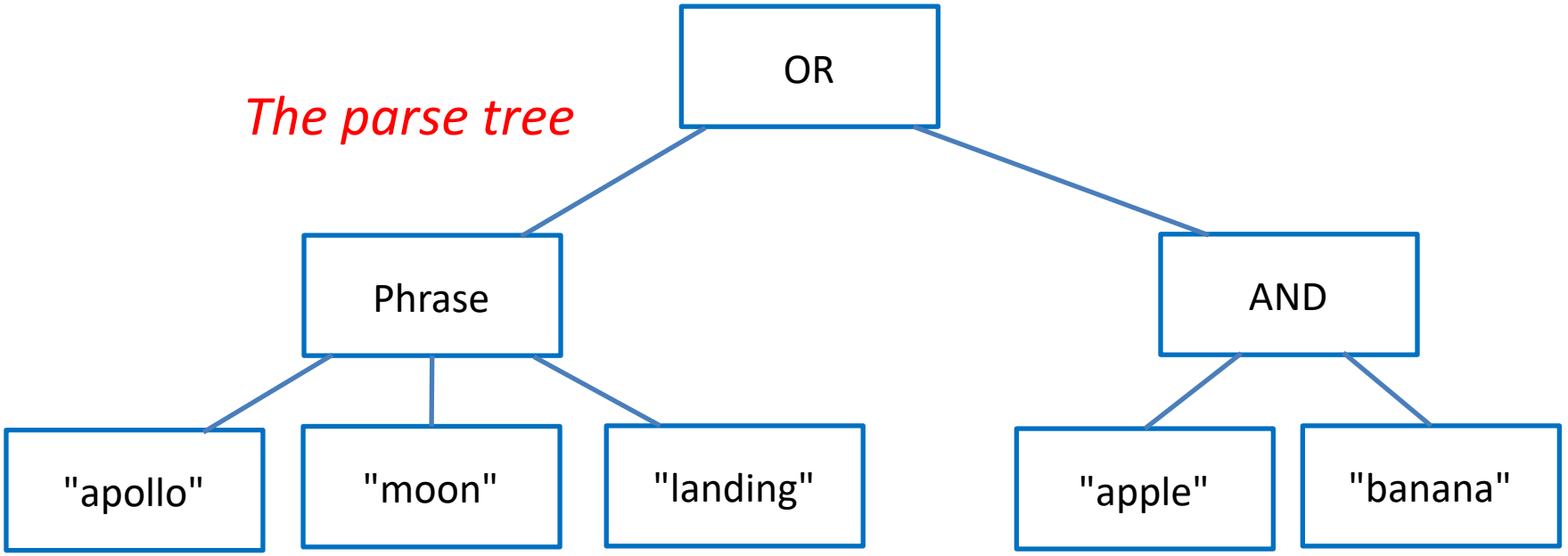
Need a way to compile a query into a structure of ISRs that can be passed to the constraint solver.

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

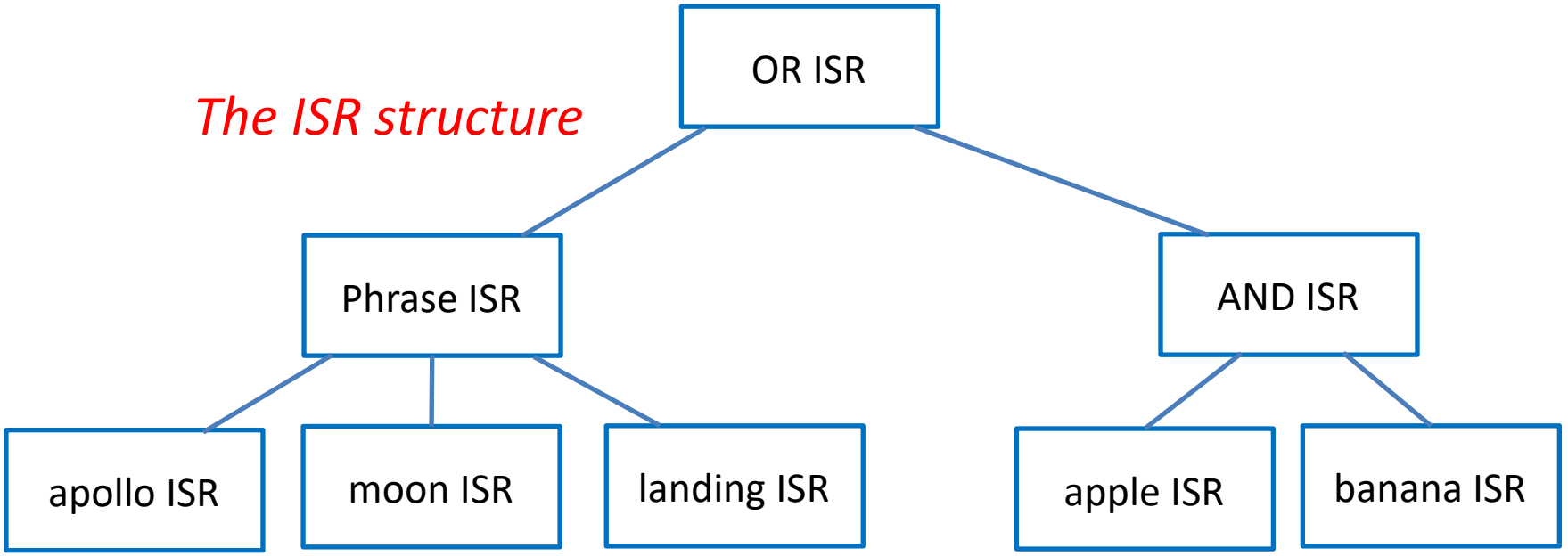
"apollo moon landing" | (apple banana)



The query language and the ISRs can be recursive

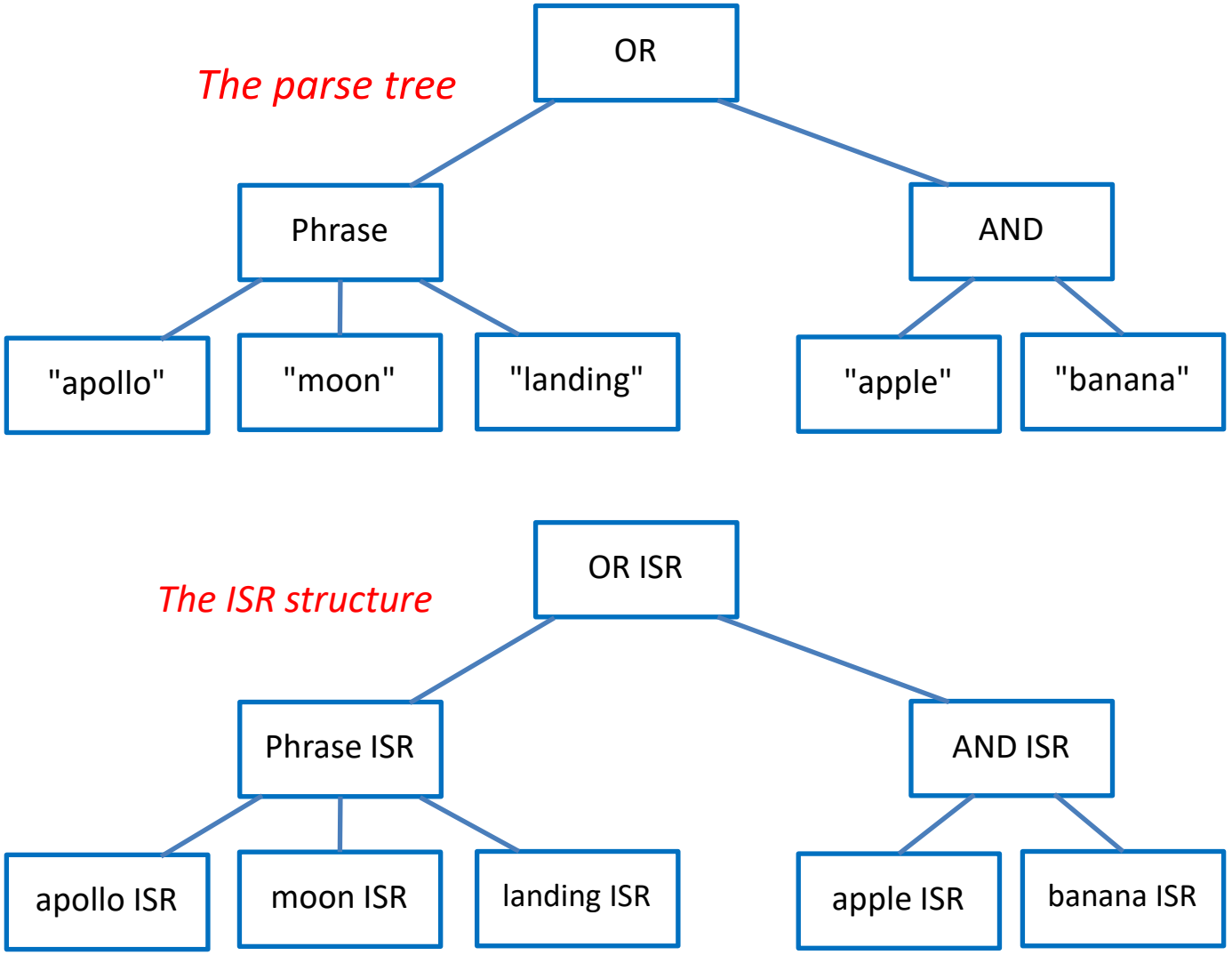
"apollo moon landing" | (apple banana)

The ISR structure



"apollo moon landing" | (apple banana)

The trees are the same.



Two parts

1. Tokenizing the input to recognize keywords, literals, operators and variable names.

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

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

Tokenizing

Usually quite simple, often ad hoc, e.g., breaking on white space or special characters.

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

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

```
// Reserved words and tokens

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,            "*",                 ),
```

```
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;
   };
```

```
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;
   };
```

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 push the token, do a reduction on the top items, or pop the stack. Typically written with a "compiler generator", e.g., YACC / LEX or BISON / Flex.
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 and another project.

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.

Notation for syntax

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

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

Usually written in Backus-Naur form (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"

Notation for syntax

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

Usually written in Backus-Naur form (BNF).

Legal sentences:

Cats meow.
Cats bark.
Dogs meow.
Dogs bark.

Not a sentence:
Birds tweet.

<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-lever 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

BNF allows for repletion and recursion.

{ … } means zero or more repetitions.

Notice how this creates precedence.

<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

Full notation:

`<x> ::= <y> <z>`	`<x>` is composed of a `<y>` followed by a `<z>`
`<x> ::= <y> | <z>`	`<x>` is composed of either a `<y>` or a `<z>`
`<x> ::= <y> [ <z> ]`	`<x>` is composed of a `<y>` followed by an optional `<z>`
`<x> ::= <y> { <z> }`	`<x>` is composed of a `<y>` followed by zero or more `<z>` elements
`<x> ::= "c"`	`<x>` is a literal string, as specified between the quotes

Operator precedence

Consider typical C/C++ operator precedence.

```
( )
*   /    %
+    -
<   <=   >=
>
==   !=
&&
||
```

Rules are written in reverse of the desired precedence

( )

* / %

+ -

< <= >= >

== !=

&&

||

```
<OrExpression> ::= <AndExpression> { '||' <AndExpression> }

<AndExpression> ::= <AndExpression> { '&&' <AndExpression> }

<EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }

<EquateOp> ::= '==' | '!='

<RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }

<RelateOp> ::=  '<' | '<=' | '>=' | '>'

<AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }

<AddOp>  ::= '+' | '-'

<MultiplyExpression> ::- <BaseExpression> { <MultiplyOp> <BaseExpression> }

<MultiplyOp> ::= '*' | '/' | '%'

<BaseExpression> ::= '(' <Expression> ')' | <Literal>

<Literal> ::= an integer string
```

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> ::= <AndExpression> { '&&' <AndExpression> }

<EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }

<EquateOp> ::= '==' | '!='

<RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }

<RelateOp> ::=  '<' | '<=' | '>=' | '>'

<AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }

<AddOp>  ::= '+' | '-'

<MultiplyExpression> ::- <BaseExpression> { <MultiplyOp> <BaseExpression> }

<MultiplyOp> ::= '*' | '/' | '%'

<BaseExpression> ::= '(' <Expression> ')' | <Literal>

<Literal> ::= an integer string
```

Rules are written in reverse of the desired precedence

The grammar is recursive but not left-recursive.

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

```
<OrExpression> ::= <AndExpression> { '||' <AndExpression> }

<AndExpression> ::= <AndExpression> { '&&' <AndExpression> }

<EquateExpression> ::= <RelateExpression> { <EquateOp> <RelateExpression> }

<EquateOp> ::= '==' | '!='

<RelateExpression> ::= <AddExpression> { <RelateOp> <AddExpression> }

<RelateOp> ::=  '<' | '<=' | '>=' | '>'

<AddExpression> ::= <MultiplyExpression> { <AddOp> <MultiplyExpression> }

<AddOp>  ::= '+' | '-'

<MultiplyExpression> ::- <BaseExpression> { <MultiplyOp> <BaseExpression> }

<MultiplyOp> ::= '*' | '/' | '%'

<BaseExpression> ::= '(' <Expression> ')' | <Literal>

<Literal> ::= an integer string
```

Let's see how we'd implement a parser for part of this.

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

```
class Tuple
   {
   public:
      Tuple    *Next;
      virtual int Eval( );
      Tuple( );
      virtual ~Tuple( );
   }

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

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

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

class AddExpression : TupleList
   {
   public:
      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;
         }

      TokenType Operation;
   }
```

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

TupleList *Parser::FindAddExpression( )
   {
   Tuple *e;
   if ( e = FindMultiplyExression( )
      {
      TokenType t;
      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;
   }
```

```
// <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;
   }
```

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

TupleList *Parser::FindMultiplyExpression( )
   {
   Tuple *e;
   if ( e = FindBaseExression( )
      {
      TokenType t;
      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;
   }
```

Agenda

1. Course 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 is about 2.4 words.

Simple search engine query language

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

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

<BaseConstraint>     ::= <SimpleConstaint> { [ <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( );
   }
```