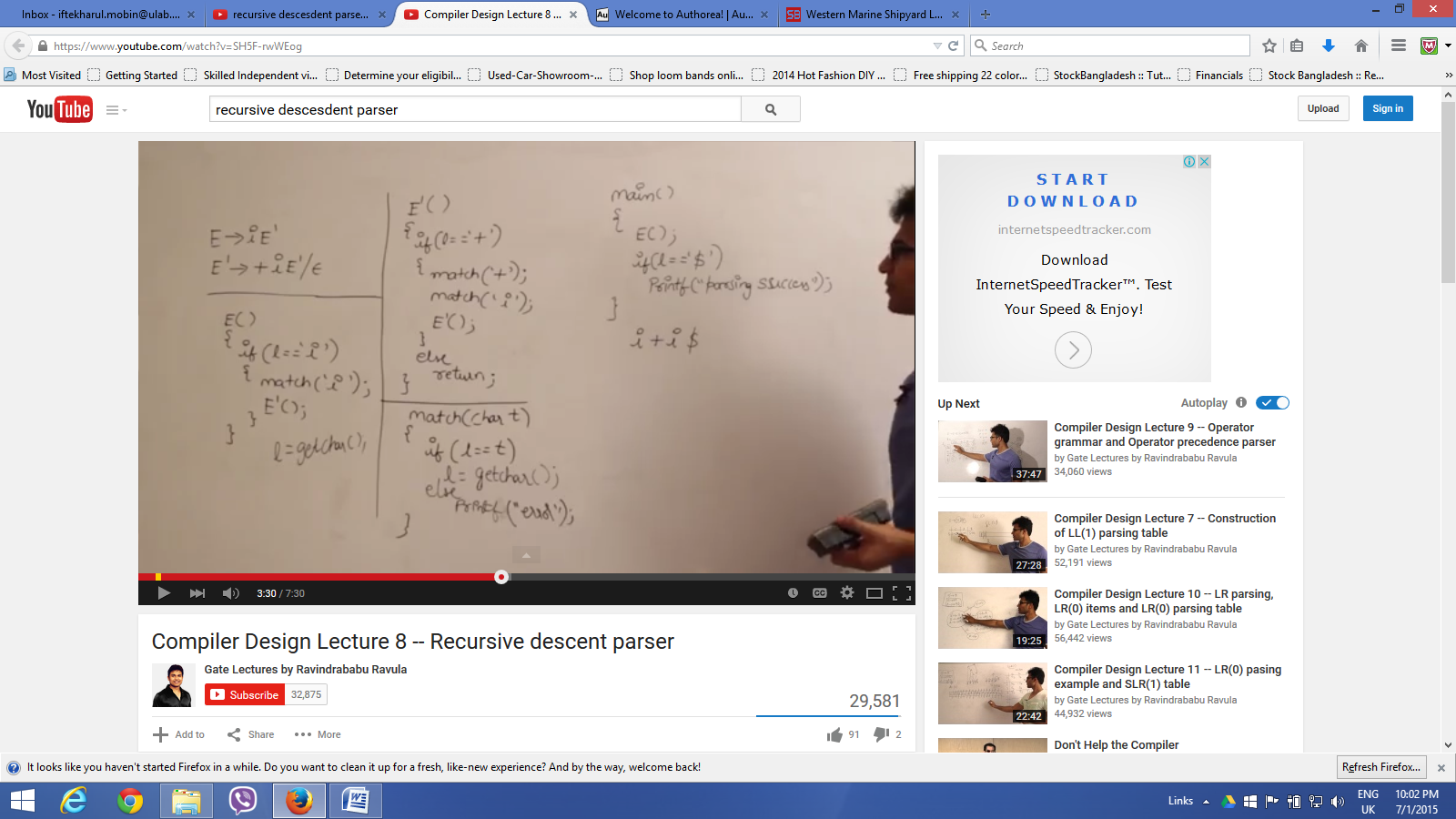
**http://www.cs.utsa.edu/~wagner/CS3723/rdparse/rdparser.html**

**http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html**

**A *recursive descent parser***

A *recursive descent parser* is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools. It is a simple and effective technique, but is not as powerful as some of the shift-reduce parsers -- not the one presented in class, but fancier similar ones called *LR parsers*.



This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

Perhaps the hardest part of a recursive descent parser is the scanning: repeatedly fetching the next token from the scanner. It is tricky to decide when to scan, and the parser doesn't work at all if there is an extra scan or a missing scan.

**Initial Example:**

* Consider the grammar used before for simple arithmetic expressions

|  |
| --- |
| **P ---> E**  **E ---> E + T | E - T | T**  **T ---> T \* S | T / S | S**  **S ---> F ^ S | F**  **F ---> ( E ) | char** |

* **The above grammar won't work for recursive descent because of the left recursion in the second and third rules. (The recursive function for E would immediately call E recursively, resulting in an indefinite recursive regression.)**

In order to eliminate left recursion, one simple method is to introduce new notation: curley brackets, where **{xx}** means "zero or more repetitions of **xx**", and parentheses **()** used for grouping, along with the or-symbol: **|**. Because of the many metasymbols, it is a good idea to enclose all terminals in single quotes. Also put a **'$'** at the end. The resulting grammar looks as follows:

|  |
| --- |
| **P ---> E '$'**  **E ---> T {('+'|'-') T}**  **T ---> S {('\*'|'/') S}**  **S ---> F '^' S | F**  **F ---> '(' E ')' | char** |

Now the grammar is suitable for creation of a recursive descent parser. Notice that this is a *different* grammar that describes the *same* language, that is the same sentences or strings of terminal symbols. A given sentence will have a similar parse tree to one given by the previous grammar, but not necessarily the same parse tree.

One could alter the first grammar in other ways to make it work for recursive descent. For example, one could write the rule for **E** as:

|  |
| --- |
| **E ---> T '+' E | T** |

This eliminates the left recursion, and leaves the language the same, but it changes the *semantics* of the language. With this change, the operator **'+'** would *associate from right to left*, instead of from left to right, so this method is not acceptable.

* *Recursive Descent parser* in C for the above grammar for arithmetic expressions:
  + C source file: [**arith0.c**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/arith0.c)
  + File showing a run: [**.txt**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/arith0.txt),  [**.html**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/arith0.html),  [**.ps**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/arith0.ps),  [**.pdf**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/arith0.pdf)
* Java version of the recursive descent parser for a grammar for arithmetic expressions:
  + Java source files: [**Arith0.java**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/Arith0.java),  [**GetChar.java**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/GetChar.java).
  + File showing a run: [**.txt**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/Arith0.txt), [**.html**](http://www.cs.utsa.edu/~wagner/CS3723/arith0/Arith0.html).
* Diagram showing parse trees for the examples above:
  + Parse trees: [**diagram.pdf**](http://www.cs.utsa.edu/~wagner/CS3723/rdparse/diagram.pdf)

/\* arith0.c: simple parser -- no output

\* grammar:

\* P ---> E '#'

\* E ---> T {('+'|'-') T}

\* T ---> S {('\*'|'/') S}

\* S ---> F '^' S | F

\* F ---> char | '(' E ')'

\*/

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

char next;

void E(void);

void T(void);

void S(void);

void F(void);

void error(int);

void scan(void);

void enter(char);

void leave(char);

void spaces(int);

int level = 0;

// P ---> E '#'

void main(void)

{

scan();

E();

if (next != '#') error(1);

else printf("Successful parse\n");

}

**// E ---> T {('+'|'-') T}**

void E(void)

{

enter('E');

T();

while (next == '+' || next == '-') {

scan();

T();

}

leave('E');

}

**// T ---> S {('\*'|'/') S}**

void T(void)

{

enter('T');

S();

while (next == '\*' || next == '/') {

scan();

S();

}

leave('T');

}

**// S ---> F '^' S | F**

void S(void)

{

enter('S');

F();

if (next == '^') {

scan();

S();

}

leave('S');

}

**// F ---> char | '(' E ')'**

void F(void)

{

enter('F');

if (isalpha(next)) {

scan();

}

else if (next == '(') {

scan();

E();

if (next == ')') scan();

else error(2);

}

else {

error(3);

}

leave('F');

}

void scan(void)

{

while (isspace(next = getchar()))

;

}

void error(int n)

{

printf("\n\*\*\* ERROR: %i\n", n);

exit(1);

}

void enter(char name)

{

spaces(level++);

printf("+-%c: Enter, \t", name);

printf("Next == %c\n", next);

}

void leave(char name)

{

spaces(--level);

printf("+-%c: Leave, \t", name);

printf("Next == %c\n", next);

}

void spaces(int local\_level)

{

while (local\_level-- > 0)

printf("| ");

}