

# Programming Language Implementation III

In this lecture we will look at **syntax analysis**. i.e. **parsing**.

- **Context free grammars.**
- **Recursive descent parsing.**
- **Removing left recursion and left factoring.**

The material is (loosely) based on Aho et al. Chapter 4.

## Reminder: Syntax Analysis

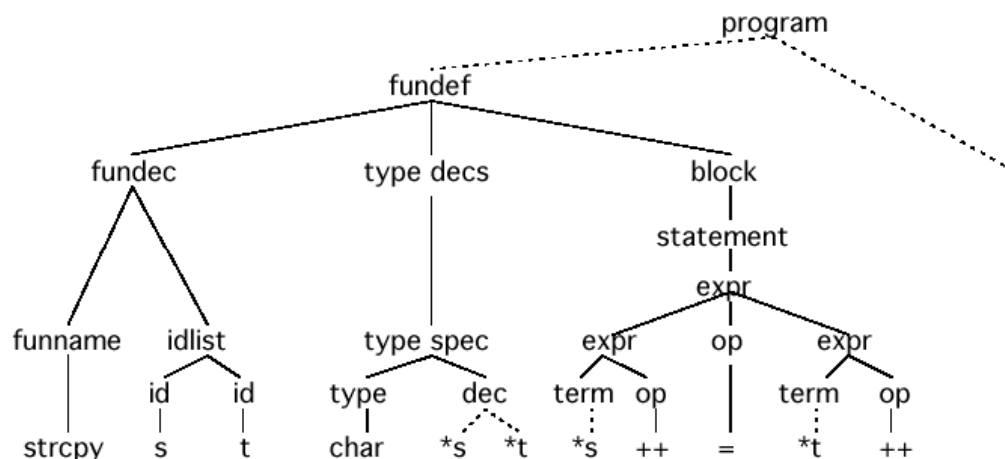
The task of the syntax analysis is to recover the hierarchical (recursive) structure from the flat token sequence.

### Example

```
[ strcpy, (, s, ,, t, ), char,  
*, s, ,, *, t, ;, {,  
while, (, *, s, +, +, =,  
*, t, +, +, ),;, } ]
```

```
[ res(strlen), (, id(1), ,, id(2), ),  
res(char), op(*), id(1), ,, op(*), id(2), ;,  
{, res(while), (, *, ... ]
```

The parse tree (syntax tree) reflects the hierarchical structure of the program.



## Syntactical Structures

Most programming language constructs have an inherently **recursive** structure.

For example, if  $S_1$  and  $S_2$  are statements and  $E$  is an expression then

**if  $E$  then  $S_1$  else  $S_2$**

is a statement.

Obviously, a regular grammar is not powerful enough to express this recursive structure of statements. A derivation according to regular grammar always has a linear structure (this is obvious from the form of its productions). Thus a more flexible form of grammar is needed.

## Context Free Grammars

A (**context-free**) **grammar**  $G$  consists of:

- A set of **terminal** symbols (also called **tokens**,  $T$ );
- A set of **non-terminal** symbols,  $N$ ;
- A **start** symbol,  $S \in N$ ;
- A set of **productions** each of form  $X \rightarrow \alpha$  where  $X \in N$  and  $\alpha$  is a sequence from  $(N \cup T)^*$ .

The following grammar defines a subset of arithmetic expressions. It has terminal symbols,

**int** + \* ( )

and non-terminal symbols  $exp$ ,  $term$  and  $factor$  where  $exp$  is the start symbol and the productions are:

$$\begin{aligned}exp &\rightarrow term \\exp &\rightarrow exp + term \\term &\rightarrow factor \\term &\rightarrow term * factor \\factor &\rightarrow \mathbf{int} \\factor &\rightarrow (exp)\end{aligned}$$

Usually we combine productions with the same LHS symbol  $A$  into alternatives for  $A$ , thus we write

$$exp \rightarrow term \mid exp + term$$

## Language and Parse Trees

We let  $\alpha, \beta, \gamma$  denote **sentences**, that is sequences from  $(N \cup T)^*$ .

$\alpha \Rightarrow \beta$  indicates that  $\beta$  can be **derived** from  $\alpha$  by using a single production rule.

$\alpha \Rightarrow^* \beta$  indicates that  $\beta$  can be derived from  $\alpha$  by using zero or more applications of a production rule.

The **language** generated by grammar  $G$  with start symbol  $S$  is

$$\{\alpha \mid \alpha \in T^* \text{ and } S \Rightarrow^* \alpha\}.$$

For instance,  $12 * (3 + 4)$  (more exactly  $int * (int + int)$ ) is in the language of our example.

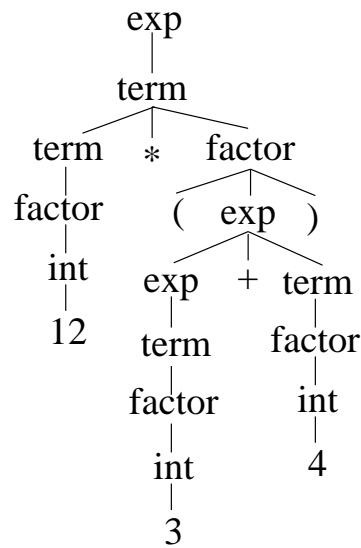
A **derivation** can be represented using a **parse** or **syntax** tree:

- each leaf is a terminal
- the leaves (in pre-order) are the input sequence
- each inner node is a non-terminal
- the root is the start symbol of the grammar

## Parse Trees

$$\{\alpha \mid \alpha \in T^* \text{ and } S \Rightarrow^* \alpha\}.$$

Consider the derivation for  $12 * 3 + 4$ :



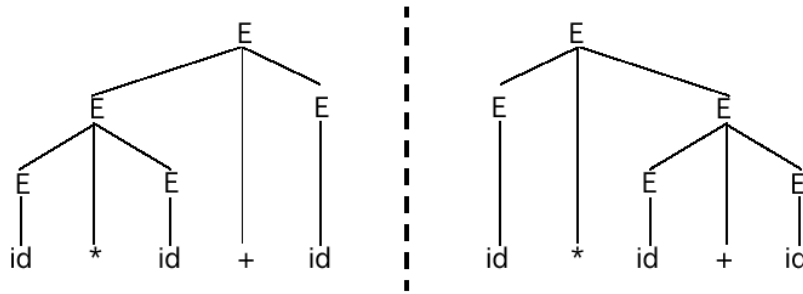
Notice that the parse tree can abstract away from the order in which production rules are used.

## Ambiguity

A grammar is **ambiguous** if some sentences have more than one parse tree.

For example, the following grammar is ambiguous:

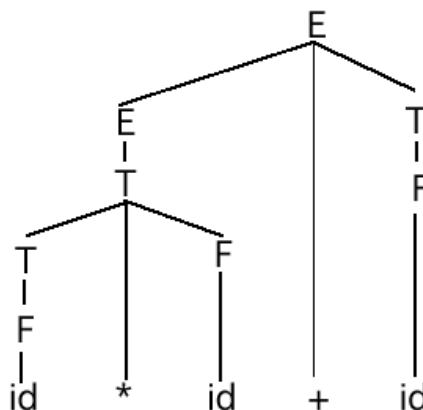
$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$



However, the modification

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

allows only a single parse tree.



## Disambiguation

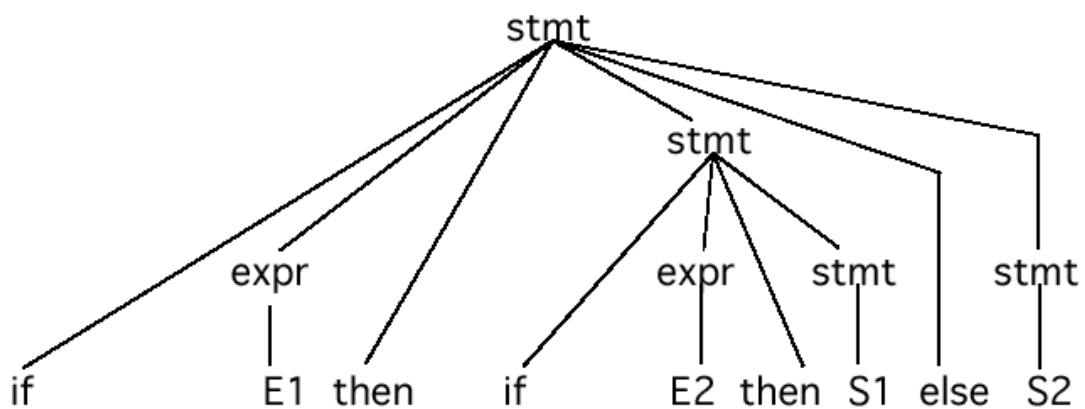
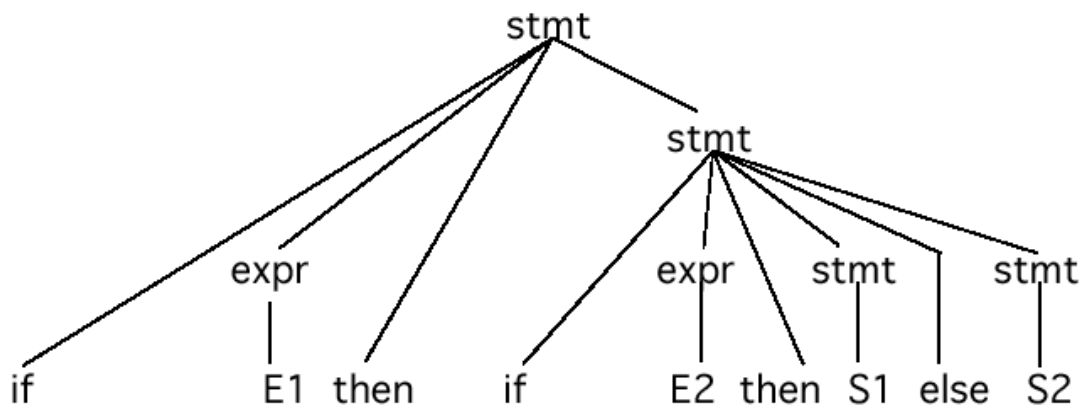
Some context-free languages are **inherently ambiguous** while in other cases it can be eliminated.

$$\begin{array}{l} stmt \rightarrow \mathbf{if\ exp\ then\ stmt\ else\ stmt} \\ \quad | \quad \mathbf{if\ exp\ then\ stmt} \\ \quad | \quad \mathbf{other} \end{array}$$

is ambiguous since

**if e1 then if e2 then s1 else s2**

has two parse trees.



## Disambiguation (cont.)

Disambiguation of course requires knowledge about the intended structure of the language.

If we choose to associate the **dangling else** with the closest previous unmatched **then**, we can use the grammar

$$\begin{array}{l} \textit{stmt} \quad \rightarrow \textit{matched} \\ \quad \quad \quad | \textit{unmatched} \\ \textit{matched} \quad \rightarrow \mathbf{if\ } \textit{exp} \mathbf{\ then\ } \textit{matched} \mathbf{\ else\ } \textit{matched} \\ \quad \quad \quad | \mathbf{\ other} \\ \textit{unmatched} \rightarrow \mathbf{if\ } \textit{exp} \mathbf{\ then\ } \textit{stmt} \\ \quad \quad \quad | \mathbf{if\ } \textit{exp} \mathbf{\ then\ } \textit{matched} \mathbf{\ else\ } \textit{unmatched} \end{array}$$

## Limitations of Context-free Grammars

Many programming languages constructs cannot be expressed or defined by context-free grammars. Consider the following examples:

- All identifiers have to be declared before they are used.  
Corresponding formal language:  $L = \{w c w \mid w \in (a \mid b)^*\}$
- Formal parameters in procedure declarations must agree with their usage in procedure calls.  
Corresponding formal language:  $L = \{a^n b^m c^n d^m \mid n, m \geq 1\}$

Such checks therefore have to be left to the semantic analysis phase.

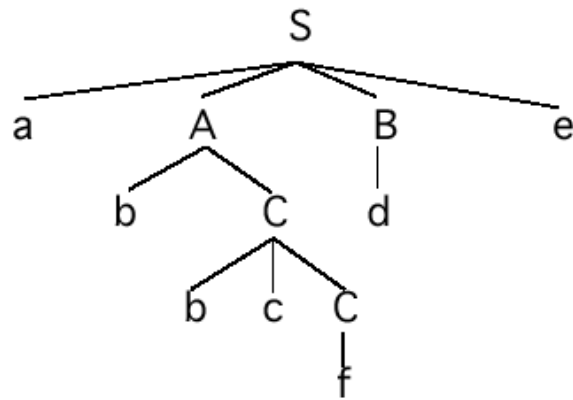
Using the (more complex) *Pumping Lemma* for context-free languages, it can be proven that the above languages are not context-free.

Beware:  $L = \{a^n b^m c^m d^n \mid n, m \geq 1\}$  is context-free! Why?

## Top-Down Parsing

In **top-down parsing** the parse tree is built from the root downwards interpreting productions from left to right. Example

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bC \\ B &\rightarrow d \\ C &\rightarrow bcC \mid f \end{aligned}$$



leftmost:

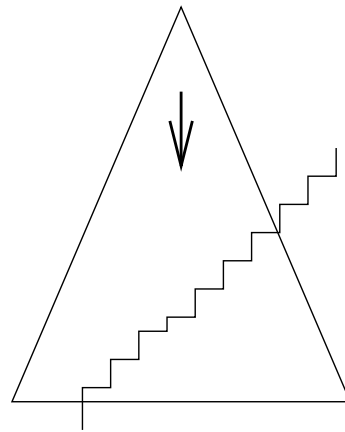
$$S \rightarrow aABe \rightarrow abCBe \rightarrow abbcCBe \rightarrow abbcfBe \rightarrow abbcfdde.$$

rightmost:

$$S \rightarrow aABe \rightarrow aAde \rightarrow abCde \rightarrow abbcCde \rightarrow abbcfdde.$$

leftmost (rightmost) derivations always replace the leftmost (rightmost) non-terminal in the current sentential form.

## Predictive Top-down Parsing



top-down  
parsing

The parser scans the input-left-to-right one token at a time.

If the parser is always able to guess which rule to use then it is *deterministic* otherwise it is *non-deterministic* and will need to employ backtracking.

If it is able to guess which rule to use it is said to be a **predictive** parser.

We will look at **recursive descent** and **table-driven** predictive parsers.

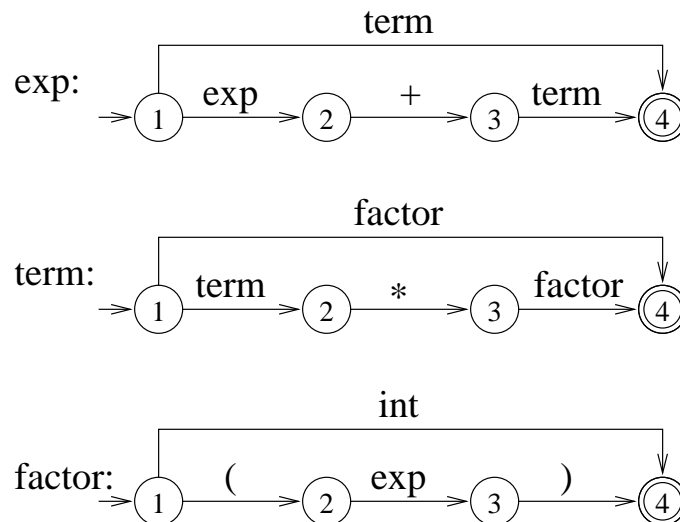
## Transition Diagrams

In practice computational languages are often described by **transition diagrams** instead of grammars. However, this is just a convenient graphical representation of the grammar.

Recall the grammar

$$\begin{aligned} \text{exp} &\rightarrow \text{term} \mid \text{exp} + \text{term} \\ \text{term} &\rightarrow \text{factor} \mid \text{term} * \text{factor} \\ \text{factor} &\rightarrow \mathbf{int} \mid (\text{exp}) \end{aligned}$$

The transition diagram for it is:



## Recursive-Descent Parsing

The idea behind **recursive-descent** parsing is to write mutually recursive functions, one for each non-terminal symbol, which mirror the grammar.

A recursive-descent parser works off the transition diagram as follows.

It begins from the start state for the start symbol.

Now if it is in state  $s$  with an edge labelled by symbol  $x$  going to state  $t$  it does the following:

- If  $x$  is the terminal symbol  $a$  and the next input symbol is  $a$ , then the parser moves the input cursor one symbol right (ie consumes  $a$ ) and goes to state  $t$ .
- If  $x$  is the non-terminal symbol  $A$ , the parser calls itself recursively, beginning from the starting state of  $A$ . If it ever reaches the final state for  $A$  it returns and immediately moves to state  $t$ , in effect having *read*  $A$  from the input stream.
- Finally, if  $x$  is  $\epsilon$  it simply moves to state  $t$  without reading any input.

Does the previous transition diagram lend itself to recursive-descent parsing?

## Elimination of Left Recursion

The preceding transition diagram is not suitable for recursive-descent parsing because it is **left-recursive** that is, for some non-terminal  $A$ ,

$$A \Rightarrow \dots \Rightarrow A\alpha$$

for some string  $\alpha$ .

Top-down parsing methods cannot handle grammars with left-recursion. Fortunately, we can always remove left-recursion.

In the simple case we have **direct recursion**, say with the rule

$$A \rightarrow A\alpha \mid \beta.$$

This can be replaced by the non-left-recursive productions

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

Using this transformation we can eliminate left-recursion from the grammar:

$$\begin{aligned} \mathit{exp} &\rightarrow \mathit{term} \mid \mathit{exp} + \mathit{term} \\ \mathit{term} &\rightarrow \mathit{factor} \mid \mathit{term} * \mathit{factor} \\ \mathit{factor} &\rightarrow \mathbf{int} \mid (\mathit{exp}) \end{aligned}$$

## Elimination of Left Recursion (Cont.)

More generally, direct recursion can be removed from the rule

$$A \rightarrow A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

by replacing it by the productions

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \cdots \mid \beta_m A' \\ A' &\rightarrow \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \epsilon \end{aligned}$$

We can also remove **indirect recursion** using the following algorithm.

Arrange the non-terminals in some order  $A_1, \dots, A_n$ .

**for**  $i = 1, \dots, n$  **do**

**for**  $j = 1, \dots, i - 1$  **do**

        replace each production of form  $A_i \rightarrow A_j \gamma$

        by the production

$$A_i \rightarrow \delta_1 \gamma \mid \cdots \mid \delta_k \gamma$$

        where  $A_j$  currently has the production

$$A_j \rightarrow \delta_1 \mid \cdots \mid \delta_k$$

**endfor**

        eliminate direct left recursion from the  $A_i$  productions

**endfor**

## Elimination of Cycles and $\epsilon$ -Productions

The above algorithm assumes that the input grammar has no **cycles** or  **$\epsilon$ -productions**. So do practically all parsing algorithms. Luckily, these can be eliminated automatically.

A context-free grammar can be made  $\epsilon$ -free (i.e. it does not contain any  $\epsilon$ -productions) if the language generated by  $G$  does not contain  $\epsilon$ .

- determine all non-terminals  $X$  for which  $X \Rightarrow^* \epsilon$ . We call these non-terminals *nullable*.
- replace each production  $p$  of the form  $A \rightarrow B_1, B_2, \dots, B_n$  by a set of productions that is composed of copies of  $p$  with each possible combination of nullable non-terminals removed on the LHS.

A grammar is not cycle-free if  $A \Rightarrow^+ A$  for some non-terminal  $A$ . Obviously, a cycle in a derivation performs no useful function. How can cycles be removed automatically?

## Left Factoring

**Left factoring** is another grammar transformation which is useful to produce a grammar suitable for predictive parsing.

The key idea is that when it is not clear which of two alternative productions to use to expand a non-terminal  $A$ , rewrite the productions so as to delay the commitment to one or the other.

For example, consider the grammar

$$\begin{aligned} stmt &\rightarrow \mathbf{if\ exp\ then\ stmt\ else\ stmt} \\ &\quad | \mathbf{if\ exp\ then\ stmt} \\ &\quad | \mathbf{other} \end{aligned}$$

if we encounter an **if** which production do we use?

In general if a nonterminal  $A$  has productions some of which share a non-trivial common prefix  $\alpha \neq \epsilon$  then

$$A \rightarrow \alpha\beta_1 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m$$

can be rewritten to

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m \\ A' &\rightarrow \beta_1 \mid \cdots \mid \beta_n \end{aligned}$$

Repeated application of this rewriting will ensure that no alternatives share a common prefix.

The above grammar will be rewritten to:

$$\begin{aligned} stmt &\rightarrow \mathbf{if\ exp\ then\ stmt\ else\ stmt} \\ &\quad | \mathbf{other} \\ else\ stmt &\rightarrow \mathbf{else\ stmt} \\ &\quad | \epsilon \end{aligned}$$

## Recursive-Descent Parsing (Example Grammar)

We now implement a recursive descent parser for our transformed expression grammar:

$$\begin{aligned} \textit{exp} &\rightarrow \textit{term exp1} \\ \textit{exp1} &\rightarrow + \textit{exp} \\ \textit{exp1} &\rightarrow - \textit{exp} \\ \textit{exp1} &\rightarrow \epsilon \\ \textit{term} &\rightarrow \textit{factor term1} \\ \textit{term1} &\rightarrow * \textit{term} \\ \textit{term1} &\rightarrow / \textit{term} \\ \textit{term1} &\rightarrow \epsilon \\ \textit{factor} &\rightarrow \textit{id} \\ \textit{factor} &\rightarrow (\textit{Exp}) \end{aligned}$$

## Recursive-Descent Parsing in ML

For the scanner stage of the following recursive descent parser we use the scanner that we have generated in the last lecture using ML-Lex.

First, we make the internal structure of the lexer (in particular the tokens) available and declare a type for the syntax tree. ML-Lex uses a datatype `CalcLex.Internal.result` for the token types. We also need to access `CalcLex.UserDeclarations` which contains the individual constructors for the tokens.

```
open CalcLex.Internal;
open CalcLex.UserDeclarations;

datatype opNode = OP of result;

datatype synTree =
  Empty
  | EXP of synTree * synTree
  | EXP1 of opNode * synTree
  | TERM of synTree * synTree
  | TERM1 of opNode * synTree
  | FACTOR of result;
```

## Recursive-Descent Parsing in ML (cont.)

Each production is implemented by a single function that passes back a tuple consisting of the syntaxtree for this production application and the unprocessed rest of the input.

```
fun exp tokenList =
  let val (termTree, rest1) = term tokenList in
    let val (exp1Tree, rest) = exp1 rest1 in
      (EXP(termTree, exp1Tree), rest)
    end
  end
and exp1 (PLUS::more) =
  let val (expTree, rest) = exp more in
    (EXP1(OP(PLUS), expTree), rest)
  end
| exp1 (SUB::more) =
  let val (expTree, rest) = exp more in
    (EXP1(OP(SUB), expTree), rest)
  end
| exp1 rest = (Empty, rest)
```

continued on the next page...

## Recursive-Descent Parsing in ML (cont.)

continued from previous page...

```
and term tokenlist =
  let val (factorTree, rest1) = factor tokenlist in
    let val (term1Tree, rest) = term1 rest1 in
      (TERM(factorTree, term1Tree), rest)
    end
  end
and term1 (DIV::more) =
  let val (termTree, rest) = term more in
    (TERM1(OP(DIV), termTree), rest)
  end
| term1 (TIMES::more) =
  let val (termTree, rest) = term more in
    (TERM1(OP(TIMES), termTree), rest)
  end
| term1 rest = (Empty, rest)
and factor (ID(s)::rest) = (FACTOR(ID(s)), rest)
| factor (NUM(n)::rest) = (FACTOR(NUM(n)), rest)
| factor (LPAREN::more) =
  let val (expTree, (RPAREN::rest)) = exp more in
    (expTree, rest)
  end;
```

Note how (in the general case) the functions have to be chained by **and** because the productions could be mutually recursive.

## Recursive-Descent Parsing in ML (cont.)

Finally, we need some help functions to couple the scanner and the recursive descent parser:

```
exception SYNTAX_ERROR of lexresult list;
```

```
fun allTokens (lexer) =  
    let val token = (lexer():lexresult) in  
        if token=EOF  
        then [EOF]  
        else token::allTokens(lexer)  
    end;
```

```
fun parse () = parse1 []  
and parse1 [] =  
    let val infile = (TextIO.openIn("testfile")) in  
        let val lexer =  
            CalcLex.makeLexer(  
                fn n => TextIO.inputLine infile ) in  
            let val (parseTree, rest) = exp (allTokens(lexer)) in  
                if rest=[ EOF ] then parseTree  
                else raise SYNTAX_ERROR(rest)  
            end  
        end  
    end;
```

## Summary

We have looked at syntax analysis:

- Context free grammars.
- Recursive descent parsing.
- Removing left recursion and left factoring.

## Homework

- Read Sections 4.2, 4.3, and 4.4 of Aho et al.
- Modify the grammar for arithmetic expressions to include function applications (like  $\sin(x)$ )
- Build a recursive descent parser for the extended grammar.
- Modify your parser so that it computes the value of the expression, as long as it is syntactically valid.
- Use the left-recursion algorithm to remove recursion from the grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \end{aligned}$$