Top-Down Parsers, Instruction Emission
The significance of parse trees is three-fold:

- Intermediate code is generated from structures of parse trees and semantic information encoded in them.
- Type checking (for statically typed languages) and semantic analysis rely on the syntactic structures represented by parse trees.
- Syntax errors are detected by failure to construct parse trees.

Two major parsing methods are used in programming language implementations:

- **Top-Down, Recursive-Descent Parsers**. Parse trees are constructed from the root downward by a collection of recursive functions. The recursive functions have direct structural correspondence to the BNF/EBNF grammar's production rules, and their actions are intuitively understandable. This method is suitable for hand coding of parsers. Parse trees are constructed not in actual data structures but implicitly in the trees of function calls. Top-down parsers cannot be constructed from left-recursive production rules.
- **Bottom-Up Parsers**. Parse trees are constructed from the leaf nodes upward by a "shift-reduce" process using an action table and a stack. Generally, bottom-up parsers can issue better syntax error messages than top-down parsers. This method is too complex for hand coding, but is highly amenable to automatic generation of parsers from BNF/EBNF grammars. A popular automatic parser generator YACC system, for example, uses this method. Bottom-up parsers can be constructed from left-recursive production rules.

In this course only top-down parsers will be reviewed. We begin with an example top-down parser for the following grammar for arithmetic expressions:

<E> → <term> [(+|−) <E>] // equivalent to <E> → <term> | <term> + <E> | <term> − <E>
<term> → <primary> [(*|/) <term>] // equivalent to <term> → <primary> | <primary> * <term> | <primary> / <term>
<primary> → <id> | <int> | <float> | − <primary> | "(" <E> ")"

The category <primary> includes the unary − operator, which has the highest precedence.

We assume that the `getToken()` function is given to extract the next token and assign it to the string variable *t*; the tokens are <id>, <int>, <float>, +, −, *, /, (,). We create a function for each non-token category, in this case `E()`, `term()`, and `primary()`, and code its body in a way that mirrors the right-hand side(s) of its production rule(s). Each function will parse strings in the corresponding category. To ease consistent calls to `getToken()`, we use the following convention:

- code so that the leading token is already in the variable *t* whenever a parsing function is called;
- each parsing function always extracts the next token just before returning.

```
void E()
{
    term();
    if ( t is "+" || t is "-" )
    {
        getToken();
        E();
    }
}
```

```
void term()
{
    primary();
    if ( t is "*" || t is "/" )
    {
        getToken();
        term();
    }
}

void primary()
{
    if ( t is <id>, <int>, <float> )
        getToken();
    else if ( t is "-" )
    {
        getToken();
        primary();
    }
    else if ( t is "(" )
    {
        getToken();
        E();
        if ( t is ")" )
            getToken();
        else
            print("Error: ) expected");
    }
    else
        print("Error: <id>, <int>, <float>, -, or ( expected");
}
```

The production rules for <E> and <term> can be given in iterative form:

<E> → <term> {(+|−) <term>}
<term> → <primary> {(*|/) <primary>}

The `E()` and `term()` functions are then coded with while-loops to mirror the iterative structure:

```
void E()
{
    term();
    while ( t is "+" || t is "-" )
    {
        getToken();
        term();
    }
}

void term()
{
    primary();
    while ( t is "*" || t is "/" )
    {
        getToken();
        primary();
    }
}
```

In order to derive top-down parsers, the production rules must be given in a form that clearly shows the functions to be called and the actions to be taken. In particular, left-recursive production rules should not be used since they may fail to show the functions to be called and the actions to be taken, or recursive functions derived from them may fall into infinite loops. (Bottom-up parsers can handle left-recursive rules without problems.) The production rules for each category <X> need to be combined into a single rule of the form:

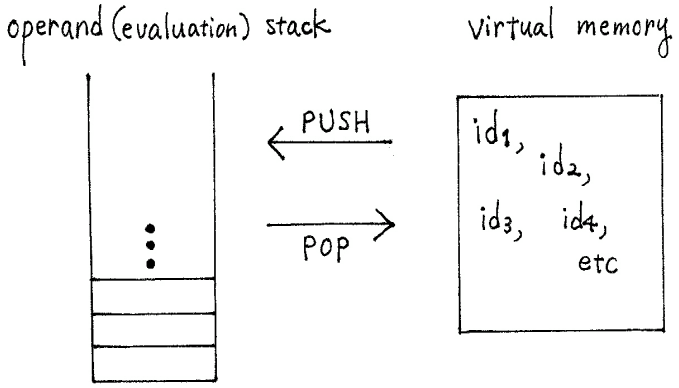
$$\langle X \rangle \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

If $n \geq 2$, the processing of multiple α_i needs to be coded by a chain of conditional statements where the testing for choices is done by inspection of the leading tokens of α_i. The iterative constructs $\{ \cdots \}$ and $\{ \cdots \}^+$ are coded by suitable while-loops.

Emission of Intermediate Code Instructions We illustrate emission of intermediate-code instructions using a simple, abstract instruction set for a stack-based virtual machine. Although simple, it captures the core idea of stack-based machines. The following table describes the instructions and their actions (the "stack" refers to the operand stack):

```
push  <id>    :  push the value of variable <id> onto the stack
push  <const> :  push the constant literal <const> onto the stack
pop   <id>    :  pop the top of the stack and store it in variable <id>
add           :  pop the top two values of the stack, compute
stack[top-1]+stack[top],
                 and push the value onto the stack
sub           :  pop the top two values of the stack, compute
stack[top-1]-stack[top],
                 and push the value onto the stack
mul           :  pop the top two values of the stack, compute
stack[top-1]*stack[top],
                 and push the value onto the stack
div           :  pop the top two values of the stack, compute
stack[top-1]/stack[top],
                 and push the value onto the stack
neg           :  pop the top value of the stack, compute -stack[top], and push the
value onto the stack
```

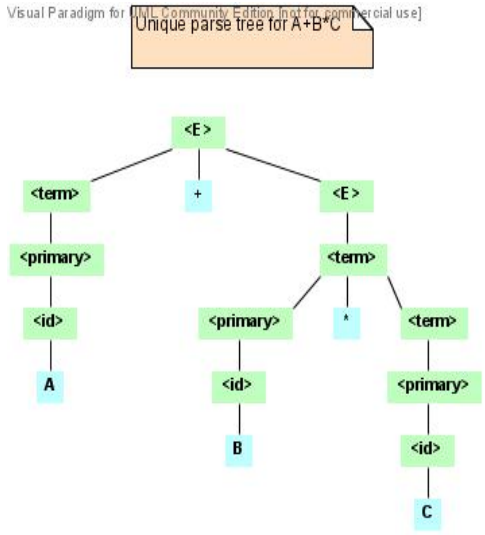
In this virtual machine, the virtual memory simply consists of a set of identifiers, and all expression evaluation occurs in the stack.



In stack-based machines (whether virtual or real hardware), each instruction implementing an operator *f* with *n* operands pops the top *n* values of the stack, compute f(stack[top−*n*+1], ..., stack[top]), and pushes the value onto the stack.

A merit of this type of stack-based instructions is that they can be easily generated by **postorder traversal** of parse trees' leaves. Postorder traversal of the subtree representing an expression $f(E_1, ..., E_n)$ traverses $E_1, ..., E_n$ recursively in postorder, then visits the operator *f*. Whenever an identifier or a constant literal *a* is visited, emit *push a*, and whenever an operator is visited, emit the corresponding instruction.

For example, the following parse tree for A+B*C



would generate:

```
push  A
push  B
push  C
mul
add
```

As you can see from tracing the evaluation process by writing down a transition sequence of the stack, the last stage of the stack will contain a single value which is the evaluation result.
The assignment statement schema *x* = *E* would generate the instructions:

```
code to evaluate E  // puts the value of E at top of the stack
pop x
```

The parsing functions, such as E(), term(), primary(), will implicitly construct parse trees in the form of trees of function calls. The crucial fact is that the temporal order of function calls corresponds directly to that of postorder traversal of (implicit) parse trees, and the correspondence is manifest in the functions' body code. Instruction emission of top-down parsers takes advantage of this fact. The following code incorporates instruction emission into the recursive version of `E()`, `term()`, and `primary()`. Note how operator instructions are emitted *after* instruction emissions of their arguments.

```
void E()
{
    term(); // generates instruction stream for <term>
    if ( t is "+" || t is "-" )
    {
        saved_t = t;
        getToken();
        E(); // generates instruction stream for <E>
        // now generate the operator instruction to compute <term>(+|-)<E>
        if ( saved_t is "+" )
            emit add;
        else
            emit sub;
    }
}

void term()
{
    primary(); // generates instruction stream for <primary>
    if ( t is "*" || t is "/" )
    {
        saved_t = t;
        getToken();
        term(); // generates instruction stream for <term>
        // now generate the operator instruction to compute <primary>(*|/)<term>
        if ( saved_t is "*" )
            emit mul;
        else
            emit div;
    }
}

void primary()
{
    if ( t is <id>, <int>, <float> )
    {
        emit push t;
        getToken();
    }
    else if ( t is "-" )
    {
        getToken();
        primary(); // generates instruction stream for <primary>
        // now generate neg to compute -<primary>
        emit neg;
    }
    else if ( t is "(" )
    {
        getToken();
        E(); // generates instruction stream for <E>
        if ( t is ")" )
            getToken();
        else
            print("Error: ) expected");
    }
    else
        print("Error: <id>, <int>, <float>, -, or ( expected");
}
```

In class we also discussed the following:

- instruction emission in the iterative version of `E()` and `term()`;
- emission of the *pop* instruction in `assignment()`.

In real-world stack-based virtual machines such as the Java Virtual Machine and the Common Intermediate Language, the *push* and *pop* instructions are respectively called *load* and *store*.