316 10-01-09

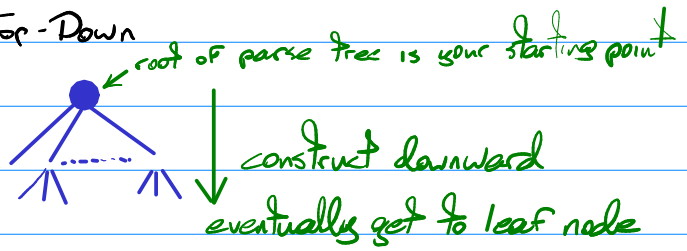
Midterm 1 - 10-20-09

- Up to & including top-down parsers with instruction emissions
  - class notes
  - up to and including course notes #6
  - exercise sets 1 and 2
  - Simple Java lex analyzers and parser

Project 1 Due 10-13-09

Two Main Parsing Methods

- Top-Down



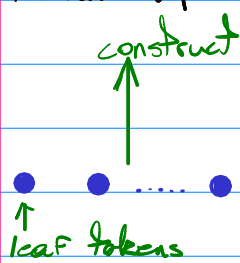
Done by a set of recursive functions
- create one function for each syntactic category
- body code is derived from the production rules

- suitable for hand-coding
- can't handle left-recursive production rules

$\langle X \rangle \rightarrow \ldots \mid \langle X \rangle \alpha \mid \ldots$

left-recursive rule

- Bottom-Up



- Done by a "shift-reduce" process and a stack
- Essentially implementations of push-down automata for context-free languages
- Suitable for automated parser generators (YACC Software)
- Can handle left-recursive production rules.

Example=

right-recursive

$\langle E \rangle \rightarrow \langle term \rangle [(+|-)\langle E \rangle]$

$\langle term \rangle \rightarrow \langle primary \rangle [(*|/)\langle term \rangle]$

unary minus

$\langle primary \rangle \rightarrow \langle id \rangle | \langle int \rangle | \langle float \rangle | "(" \langle E \rangle ")" | -\langle primary \rangle$

Operator	Procedure	A*B--C
higher ↑	- (unary minus)	(A*B)-(-C)
	*, /	A*B--C+D
	+, -	(A*B)-((-C)+D)

String var t holds the current token extracted by getToken()

- code so that the leading token is already in 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();
  }
}
```

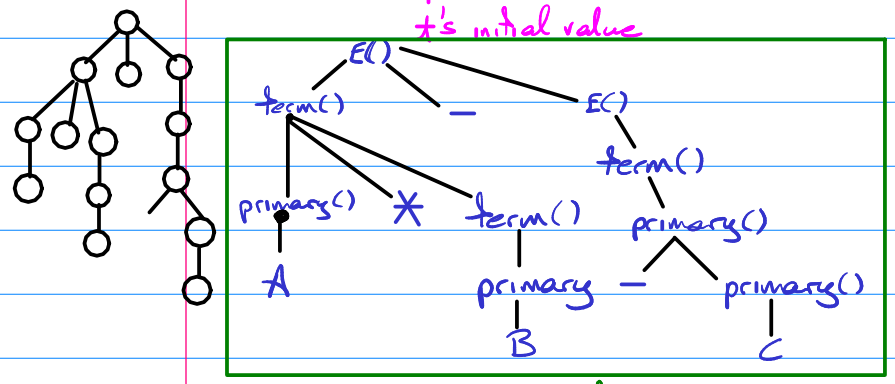
```
void term()
{
  primary();
  if (t is "*" || t is "/")
  {
    getToken();
    term();
  }
}
```

```
void primary()                    // can be decided by states of DFA
{
  if (t is <id>, <int>, <float>)
  {
    getToken();
  }
  else if (t is "(")
  {
    getToken();
    E();
    if (t is ")") getToken();
    else print("Error: ) expected");
  }
  else if (t is "-")
  {
    getToken();
    primary();
  }
  else
  {
    print("Error: <id>, <int>, <float>, (, or - expected");
  }
}
```

Top-down parsers construct parse trees implicitly in the form of trees of function calls.

Example: A*B--C

↑ t's initial value



This tree represents (A*B)-(-C)

```
main()
{
  getToken();
  E();
}
```

<E> → <term> { (+|-) <term> }

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

<primary> → <id> | <int> | <float> | "(" <E> ")" | -<primary>

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

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

General Idea

- Collect all production rules for <X> in one line

  <X> → $\alpha_1 | \alpha_2 | \ldots\ldots | \alpha_n$

- The leading tokens of α_i should be clear
- Code schema:

```
void X()
{
  if(t is leading token for α1 { getToken(); process α1;}
     :

  else if(t is leading token for αn) { getToken(); process αn; }
  else print("Error: ....");
}
```