## The current topic: Syntax and semantics

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
✓ Logic programming: Prolog
  ✓ Introduction
  ✓ Rules, unification, resolution, backtracking, lists.
  ✓ More lists, math, structures.
  ✓ More structures, trees, cut.
  ✓ Negation.
- Syntax and semantics
- Exceptions



## Announcements

- Reminder: The deadline for submitting a re-mark request for Term Test 2 is ***the end of class* on Friday.** Make sure you understand the posted solutions before submitting a re-mark request.

- Reminder: Lab 3 is due on Monday (December 1st) at 10:30 am.



## Syntax and semantics

- Goals and definitions
- Parsing
- Translation

- Reference: Sebesta, chapters 3 and 4.



## What's a language?

- A *language* is a subset of the set of all strings over some alphabet.
  - *string*: a sequence of symbols
  - *alphabet*: a set of symbols

Example:
- alphabet: `{ a, b, c }`

- language 1: all two-character strings from the alphabet
  - `{ aa, ab, ac, ba, bb, bc, ca, cb, cc }`

- language 2: all three-character strings that start and end with `c`
  - `{ cac, cbc, ccc }`

## What do you need to know about a language?

Two things:

- What can you say?
- What does it mean?

- "What can you say?" is *syntax*.
  - e.g. In Python, a for loop must be written as...
- "What does it mean?" is *semantics*.
  - e.g. In Python, a for loop means that the following will happen...



## Programming-language semantics

- It's hard to specify the meaning of a statement in a programming language.
- Choices:
  - Operational semantics
    - defines effect of program in terms of program execution on a lower-level machine.
    - i.e. the meaning of a statement is the sequence of assembler statements it translates to, or the value of the expression it calculates.
    - similar to our usual explanations of meaning.
    - we've been using (something like) this definition.
  - Axiomatic semantics
    - used in program verification (in proofs of correctness).
    - defines effect of program in terms of preconditions and postconditions of individual statements.
  - Denotational semantics
    - gives meaning in terms of mappings (functions) from statements to changes of system state, where changes of system state are represented mathematically (using recursion).
- References: Sebesta, sections 3.4 and 3.5.



## Programming-language syntax

- Recall that there are standard ways to specify the form of a statement in a programming language.
  - Backus-Naur Form (BNF)
  - Extended BNF (EBNF)
    - adds alternatives and repetition to BNF
- Basic idea: A program consists of parts, each of which consists of subparts, and so on.
- The parts are of various kinds (functions, expressions, literal values, …).
- The structure expands into a tree -- the *parse tree*.
- At the leaves of the tree are the actual statements and expressions of a particular program.
  - That is: every program has its own parse tree.



## Backus-Naur form (BNF)

- On the left, the form being described.
- Then, the ::= (or →) symbol.
- Then, the components allowed in the form, with pipes used to separate multiple allowed definitions of the same form.
- Components that must be exactly as shown are put in quotation marks: e.g. '='.
- The names of the form and of components that are forms themselves are put in angle brackets <…> or distinguished by a special typeface.
- In other words, BNF is like this:

  <BNF description> ::= <form> '::=' <components>

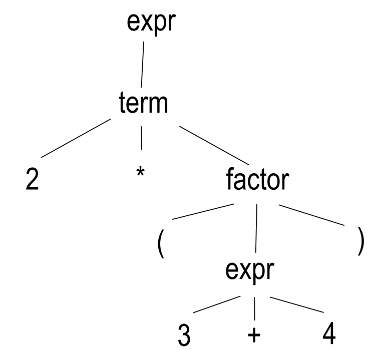
## Extended BNF

- Repetition: use curly brackets, which mean "zero or more" of their contents.
  <comma expr> ::= <expr> { ',' <expr> }
  - A comma expression consists of one or more expressions, separated by commas.
- Optional parts: use square brackets, which mean "zero or one" of their contents.
  <if stmt> ::= 'if' '(' <expr> ')' <stmt> [ 'else' <stmt> ]
  - An "if" statement has at most one "else" clause.
- Alternatives: use brackets and pipes, which mean "choose exactly one"
  <expr> ::= <vbl> ('+' | '-' | '*' | '\') <vbl>
  - An expression consists of a variable followed by exactly one symbol followed by another variable.
- BNF and Extended BNF are equivalent in what they can describe, but Extended BNF improves readability.



## A standard example, and its parse tree

- 2 * (3 + 4)
- Parse tree:






## EBNF for the grammar of the standard example

- An expression is the addition and/or subtraction of a sequence of terms:
  <expr> ::= <term> { ( '+' | '-' ) <term> }
- A term is the multiplication and/or division of a sequence of factors:
  <term> ::= <factor> { ( '*' | '/' ) <factor> }
- A factor is an explicit constant, or a parenthesized expression:
  <factor> ::= <literal> | '(' <expr> ')'
  - Note the quotation marks around the parentheses: the parentheses are part of the language being described.
- A literal is a sequence of digits:
  <literal> ::= '0' .. '9' { '0' .. '9' }



## Productions

We now continue describing how to specify syntax.

- A rule such as this is a <u>*production*</u>:
  <expr> ::= <term> { ( '+' | '-' ) <term> }
- The left-hand side of a production is a *nonterminal*.
- The right-hand side consists of a mixture of *terminals* and nonterminals.
- The production says that, when you're trying to see how a program can be understood in terms of the language's syntax, it's legal to replace the left-hand nonterminal by the string on the right.
  - That is, you can expand the LHS by replacing it with the RHS.

## Terminals and nonterminals

- *Terminals* are things that can be actually present in the program text:
  - e.g. '(', '3'
- *Nonterminals* are categories that have to be detailed further before you reach terminals.
  - internal nodes in the syntax tree (a syntax tree is a more detailed version of a parse tree)
  - e.g <expr>, <factor>, <if statement>



## Grammars

- A *grammar* consists of:
  - a set of terminals -- the *alphabet*
  - a set of nonterminals
  - a particular nonterminal called the *start symbol*
    - In our example, the start symbol is <expr>.
  - a set of productions
- A grammar defines what you can say in a language -- that is, it specifies the syntax.



## An example grammar

- alphabet: `{ a }`
- nonterminals: { <S> }
- start symbol: <S>
- productions: <S> ::= $\varepsilon$ | a<S>
  - ($\varepsilon$ is the empty string)
- The language is $\{ a^n \mid n \geq 0 \}$
  - that is: $\varepsilon$, a, aa, aaa, aaaa, ...



## Another example grammar

- Productions:
  - <sentence> :: = <subject> <verb> <object>'.'
  - <subject> ::= <article> <noun>
  - <verb> ::= 'walks' | 'bites'
  - <object> ::= <article> <noun>
  - <article> ::= 'a' | 'the'
  - <noun> ::= 'man' | 'dog'
- Some legal statements in this language:
  - the man walks the dog.
  - a dog walks the man.
  - the dog walks a dog.
  - the dog bites a man.

# Parsing

- To *parse* a statement is to show how it can be derived from the language's grammar.
- The steps in parsing a statement are *derivations*. At each derivation, one production is applied to advance the parsing process.
- The complete set of derivations is a *derivation sequence*.
- This material is covered in much more detail in CSC467 (Compilers and interpreters).



# Our example again

- The "statement" to be parsed:
  2*(3+4)
- The grammar we'll use:

  <expr> ::= <term> { ( '+' | '-' ) <term> }

  <term> ::= <factor> { ( '*' | '/' ) <factor> }

  <factor> ::= <literal> | '(' <expr> ')'

  <literal> ::= '0' .. '9' { '0' .. '9' }



# A derivation sequence for the example

- 2 * (3 + 4)

expr → term
term → factor * factor
factor → literal
literal → 2
factor → ( expr )
expr → term + term
term → factor
factor → literal
literal → 3
factor → literal
literal → 4

- the state so far:

expr
term
factor * factor
literal * factor
2 * factor
2 * ( expr )
2 * ( term + term )
2 * ( factor + factor )
2 * ( literal + factor )
2 * ( 3 + factor )
2 * ( 3 + literal )
2 * ( 3 + 4 )

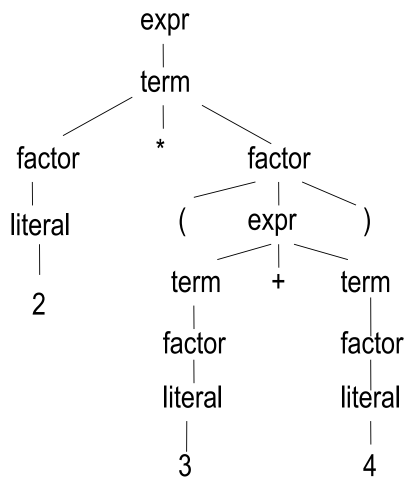


# A derivation sequence for the example

- The derivation we carried out was a *leftmost* derivation.
- At each step, the leftmost nonterminal was replaced by using an appropriate production.
- At each step, we could tell from the input (from the next "unexplained" terminal(s)) which production to choose.
  - What about the expr → term and term → factor choices?

## The parse tree for the example

- With all intermediate steps included, the parse tree may be called the "syntax tree":






## Semantics from the syntax

- The operator precedence (multiplication/division performed before addition/subtraction) is implied by the syntax:
  - <expr> ::= <term> { ( '+' | '-' ) <term> }
  - <term> ::= <factor> { ( '*' | '/' ) <factor> }

- This makes it easier to generate code that corresponds to the semantics, without having to provide rules that state the semantics explicitly.



## An ambiguous (and an unusable?) grammar

- E ::= E '+' E | E '*' E | '(' E ')' | number

- Why is it ambiguous?
  - Because a given expression may have more than one parse tree.
    - See next slide.

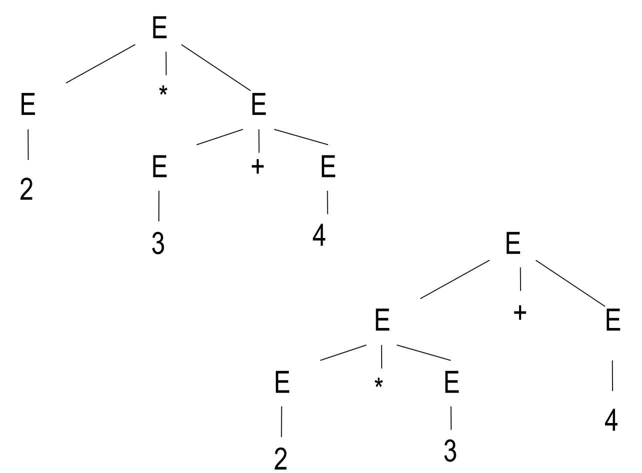
- F ::= F { ( '+' | '-' ) F }

- Why might it be unusable?
  - Because, for example, if we represent productions by functions, then the first step in the function F( ) will be to call F( ).

- We avoided both problems because of the way we wrote the grammar in our previous example.
  - We used different nonterminals for operations of different precedence.



## Ambiguity: two parse trees for 2 * 3 + 4



## Operational semantics

- We're using "operational semantics": the meaning of a statement is what it does when you run it.
  - That is, if we translate the statement to machine or assembly language, then we have given its meaning.

- Given the parse tree of a statement, we can do a tree-traversal to get the corresponding code.

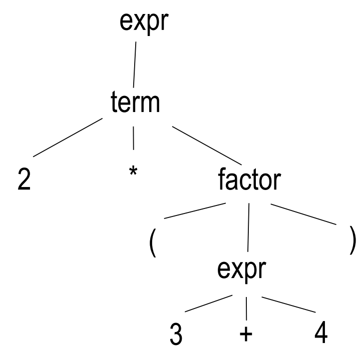


## Code for 2*(3+4) in terms of stack operations

- A postorder traversal of the parse tree:
  - "Postorder": Visit a node's children before visiting the node itself.
  - At operand leaves, emit code to push the operand.
  - At operator leaves, do nothing.
  - At internal nodes that have an operator as a child, emit code to call the operator.
  - At other internal nodes and at parentheses, do nothing.

- Code:

```
push 2
push 3
push 4
+
*
```



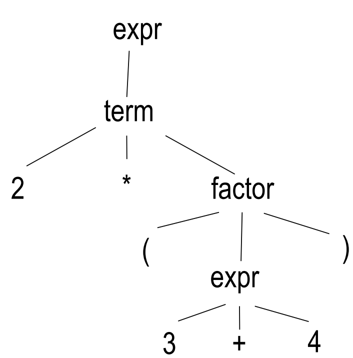



## Code for 2*(3+4) in assembly language

- A postorder traversal of the parse tree:
  - At operand leaves, emit code to load the operand into an available register.
  - At operator leaves, do nothing.
  - An internal nodes that have an operator as a child, emit code to call the operator on the appropriate registers.
  - At other internal nodes and at parentheses, do nothing.

- Code:

```
ldi #2, r1
ldi #3, r2
ldi #4, r3
add r3, r2
mul r2, r1
```






## Points glossed over

- Register management: the set of registers is smaller than the number of values and subexpressions in a program, and is different in different computers.
  - How do we decide which values to move to main memory while making best use of the registers in the CPU?

- We assumed we had an explicit parse tree. More likely, if we analyze the expression recursively, with a function for every nonterminal, we can generate code as the parser runs instead of building the entire tree data structure.

- We'll ignore register management, but spend some time on recursive-descent parsing.

## Recursive-descent parsing

- The productions of the grammar:
  <expr> ::= <term> { ( '+' | '-' ) <term> }
  <term> ::= <factor> { ( '*' | '/' ) <factor> }
  <factor> ::= <literal> | '(' <expr> ')'
  <literal> ::= '0' .. '9' { '0' .. '9' }
  - Every nonterminal is represented by a function: `expr()`, `term()`, and so on.
- The start symbol: <expr>
  - So the parsing process begins with a call of `expr()`.
- The productions for <expr> are <term> { ( '+' | '-' ) <term> }, so `expr()` begins something like this:

```
term();
while (true) {
  token = getNextToken();
  if (token == '+') { emit("add ..."); term(); }
  else if (token == '-') { emit ("sub ..."); term(); }
  else return;
}
```



## What term( ) and factor( ) do

- A <term> starts with a <factor>, so `term()` looks like `expr()`.
  - calls `factor()` to do most of its work.
- A <factor> can begin in several ways:
  <factor> ::= <literal> | '(' <expr> ')'
- What should `factor()` begin by doing?
  - It depends on what's next in the expression being parsed.
  - That is, again you have to read the next input token.
- If the next token is '(', call `expr()` and then look for ')'.
- If that fails, try calling `literal()`, which if it succeeds will take the appropriate action:
  - For example, emit "`ldi <literal>, rn`" to the object file.
- Reference: Sebesta section 4.4



## "Recursive-descent"

- This process is recursive: the program structure works by function calls related in the same way as the parse tree relates the parts of the expression.
- It is also literally top-down: it proceeds downward from the root of the parse tree to find all the leaves.