turtle graphics
- STk has built in support for basic 2D graphics!
- Turtle sits on the canvas
- As the turtle "walks" around the canvas, it leaves a trail
- Images are drawn by issuing commands to the turtle

```
(define (triangle)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120))
```

```
(begin (forward 100)
    (right 120)
    (forward 100)
    (right 120)
    (forward 100)
    (right 120))
```

```
(begin <exp1> <exp2> ...
    <expn>)
```

programming languages
computers have software written in many different languages
machine languages: statements can be interpreted by hardware
- all data are represented as a sequence of bits
- all statements are primitive instructions
high-level languages: hide concerns about those details
- primitive data types beyond just bits
- statements/expressions, data can be non-primitive (e.g. calls)
- evaluation process is defined in software, not hardware

high-level languages are built on top of low-level languages
The Scheme-Syntax Calculator Language
A subset of Scheme that includes:
• Number primitives
• Built-in arithmetic operators: +, -, *, /
• Call expressions
  > (+ (* 3 5) (- 10 6))
  19
  > (+ (* 3
       (+ (* 2 4)
           (+ 3 5)))
    (+ (- 10 7)
        6))
  57

Syntax and Semantics of Calculator
Expression types:
• A call expression is a Scheme list
• A primitive expression is an operator symbol or number
Operators:
• The + operator returns the sum of its arguments
• The - operator returns either
  • the additive inverse of a single argument, or
  • the sum of subsequent arguments subtracted from the first
• The * operator returns the product of its arguments
• The / operator returns the real-valued quotient of a dividend and
divisor (i.e. a numerator and denominator)

Reading Scheme Lists
A Scheme list is written as elements in parentheses:
(\texttt{<element_0> <element_1> ... <element_n>})

Each <element> can be a combination or primitive
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
The task of parsing a language involves coercing a string representation of an expression to the expression itself
Parsers must validate that expressions are well-formed

Lexical analysis
• It's hard to directly determine the program structure from a string
  '(+ 1 (* 3 4) 2)'
  Tokenization!
  ['(', '+', '1', '(', '*', '3', ',', '4', ',', ')', ')', 2, ')']
• Split the string into a sequence of “tokens”
• From the token sequence, it's a lot easier to determine the program structure
Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested.

Each call to `read_exp` consumes the input tokens for exactly one expression.

```
'( ', '+', 2, '( ', '-', '23', ')', '(', '*', 4, 5.6, ')', ')
```

Base case: symbols and numbers

Recursive call: call `read_tail`, which uses `read_exp` for sub-expressions and combines them as pairs

Expression Trees

A basic interpreter has two parts: a parser and an evaluator

<table>
<thead>
<tr>
<th>lines</th>
<th>expression</th>
<th>calc_eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>'(+ 2 2)'</td>
<td>Pair('+', Pair(2, Pair(2, nil)))</td>
<td>4</td>
</tr>
<tr>
<td>'(* (+ 1 (- 23)) 2)'</td>
<td>Pair('*', Pair(Pair('+', ...)))</td>
<td>-46</td>
</tr>
</tbody>
</table>

String forming a Scheme expression

A number or a Pair with an operator as its first element

A number

Evaluation

Evaluation discovers the form of an expression and then executes a corresponding evaluation rule

Primitive expressions are evaluated directly

Call expressions are evaluated recursively:

- Evaluate each operand expression
- Collect their values as a list of arguments
- Apply the named operator to the argument list

Applying Operators

Calculator has a fixed set of operators that we can enumerate

```
def calc_apply(op, args):
    """Apply an operator to a list of args."""
    if op == '+':
        if len(args) == 0:
            raise TypeError('Not enough arguments')
    ...  
    if op == '-':
        if len(args) == 2:
            raise TypeError('Not enough arguments')
    ...  
```

Dispatch on operator name

Raising Application Errors

The `–` and `/` operators have restrictions on argument number

Raising exceptions in `apply` can identify such issues

```
def calc_apply(op, args):
    """Apply an operator to a list of args."""
    if op == '-':
        if len(args) == 0:
            raise TypeError('Not enough arguments')
    ...
    if op == '/':
        if len(args) == 2:
            raise TypeError('Not enough arguments')
    ...
```

Read-Eval-Print Loop

The user interface to many programming languages is an interactive loop, which

- Reads an expression from the user,
- Parses the input to build an expression tree,
- Evaluates the expression tree,
- Prints the resulting value of the expression

The REPL handles errors by printing informative messages for the user, rather than crashing

A well-designed REPL should not crash on any input!