COMP 304B – Object-Oriented Software Design
Assignment 2 – Class Diagrams, Collaboration Diagrams

Marc Provost

Due date: Monday 15 March 2004 before 23:55

Practical information

- Team size == 2 (pair design/programming) !
- Each team submits only *one full solution*. Use the `index.html` template provided on the assignments page. Use **exactly** this format to specify names and IDs of the team members. The other team member *must* submit a single `index.html` file containing only the coordinates of both team members. This will allow us to put in grades for both team members in WebCT. Beware: after the submission deadline there is no way of adding the other team member's `index.html` file and thus no way of entering a grade !
- Your submission must be in the form of a simple HTML file (`index.html`) with explicit references to *all submitted files* as well as inline inclusion of images. See the general assignments page for an `index.html` template.
- The submission medium is WebCT.

Goals

In this assignment, we look at the part of the spreadsheet design which is concerned with support for formulas such as `=2+3*4+MAX(A1:A10)-$B$2` which are entered in spreadsheet cells. At this point, the spreadsheet application does not yet have a graphical user interface component. The assignment covers the design using UML *Class Diagrams* and an implementation in Python of an Abstract Syntax Tree (AST). The AST is the data structure used to represent a formula in memory. You will also use the *visitor* Design Pattern to obtain a string representation of such a formula internal representation. You will present a *Collaboration Diagrams* to describe the interaction between various objects to produce the formula's string representation.

The front-end

First, an overview of the process followed to convert input strings into an appropriate internal structure will be given. In order to obtain this data-structure, several steps need to be taken. The first step for the spreadsheet to "understand" formulas entered as strings is to recognize the lexical entities ("words" or "tokens") that can be found in the formula *language*. A module which performs the task of converting a string into a series of tokens is called a *scanner*. A scanner can be generated automatically from a specification consisting of *regular expressions* specifying the structure of legal tokens. The syntax of a regular expression adheres to the following:

a	an ordinary character stands for itself
ε	empty string
	another way to write empty string
$M\|N$	Alternation, M or N
$M \cdot N$	Concatenation, M followed by N
MN	Another way to write concatenation
M^*	Repetition, zero or more times
M^+	Repetition, one or more times
$M?$	Optional, zero or one occurrence
$[a-zA-z]$	Alternation in a set of characters
.	A period stands for any single character except newline
"$a.+*$"	String in quote stands for itself literally

For example, valid expressions could be

"while"	Represent the string "while"
$[a-z][a-z0-9]*$	Represent legal variable names in a programming language
$[0-9]+$	Represent an integer number
([0-9]+"."[0-9]*)|(([0-9]*+"."[0-9]+)	Represent a real number

Once a regular expression is specified for each legal token that can be found in a formula, a scanner can be generated, which will take a potential formula string as input and will output tokens. For instance, consider the formula

```
=2+3+MAX(A1:A10)
```

The scanner will read the string from left to right and output the token stream

```
int(2) plus int(3) plus name("MAX") leftparen ref("A1") colon ref("A10") rightparen
```

This token stream will be fed to a *parser* module which verifies the order of the tokens and produces a *parse tree* representing the formula in memory. A *grammar* is used to specify the parser. A grammar, G, is a structure $< N, T, P, S >$ where N is a set of *non-terminals*, T is a set of *terminals*, P is a set of *productions*, and S is a special non-terminal called the start symbol of the grammar. The set of non-terminals recursively describes the structures of a language in terms of other non-terminals and terminals. As expected, the terminals are simply the tokens output by the scanner. For example, consider the following grammar:

$$\begin{aligned} S &\rightarrow EXP \\ EXP &\rightarrow EXP\ plus\ EXP \mid int \end{aligned}$$

This grammar recognizes simple addition expressions (e.g., `4+5+4+832;`). It specifies the structure the stream of the tokens produced by the scanner should have.

For this assignment, the formula language has been defined for you (in the form of regular expressions and a grammar). The specification can be found in the file `parser.g`. The module `Parser.py` was generated for you from `parser.g` using the Yapps compiler compiler (`http://theory.stanford.edu/~amitp/Yapps/`). The utility file `yappsrt.py` is required for the parser to function.

Requirements

Your goal is to design and implement a tree structure, called an *Abstract Syntax Tree* (AST). An AST represents information from the parse tree constructed by the parser in an "optimal" way. It does away with redundant information and is optimally suited for further processing.

First, the structure of the formulas needs to be discussed. A formula extends basic arithmetic expressions with functions and references to other formulas. Below are the potential nodes the AST can have:

- Atomic structures (future leaf nodes of the AST):
  - Numbers (e.g., 5, 3.55, 10)
  - References (e.g., A4, XBZ16, E123). Note how only upper-case column references are allowed.
  - Range-references (e.g., A1:B10)
- Composite structures (nodes having children)
  - Addition, subtraction
  - Multiplication, division
  - Exponentiation
  - Function invocations

Note that references and range-references do not have children. A reference in a formula F simply holds the (column,row) information of a cell containing another formula F' to be evaluated (later). A first draft for a design could be to create a class for each of the structures described above. However, we soon realize that the composite structures are all very similar. Indeed, all of them have operands (or arguments in the case of a function). They all apply an operation to the operands/arguments and produce an output. Addition, subtraction, multiplication, division and exponentiation can all be replaced by built-in functions. For instance, the expression 4+5 would be represented internally as:

```
Function(name="sum", [Number(4), Number(5)])
```

So, instead of creating distinct node types for each arithmetic operator, we reduce them to functions thus simplifying the design. The requirements for each class are described in the following.

A few notes:

- the parser will convert the parse tree to an AST for you. Have a look at `parser.g` (or the generated `Parser.py` to see how this is done.
- the AST will no longer be a binary tree, but will take into account associativity and commutativity of $+$ and $\times$ and turn these into their n-ary form. Note that this requires the introduction of `invSum` and `invMul` to encode $-$ and $/$ respectively.

ASTVisitor

- `ASTVisitor` is an interface.
- Contains the visit operations for each of the nodes of an AST.
- Interface:
  - `visitNumber(Number)`
  - `visitCellRef(CellRef)`
  - `visitRangeRef(RangeRef)`
  - `visitFunction(Function)`

StrReprVisitor

- A concrete implementation of the `ASTVisitor` interface.
- ProducesOperations to be performed on the nodes of an AST.
- Interface (in addition to the methods in `ASTVisitor`):
  - `setStrRepr(String strRepr="")`
  - `getStrRepr():String`

ASTNode

- The ancestor for all concrete AST nodes. Can *not* be instantiated.
- Interface:
  - `accept(ASTVisitor)`
  - `__str__()`
- Comments: concrete `ASTNode` sub-classes will specify how they are visited and represented as strings. `__str__()` will instantiate a `StrReprVisitor` and use it to produce the string representation for a formula AST rooted in the current node. This will follow the *visitor pattern* (see the collaboration diagram section for a more detailed description).

Number

- Extends `ASTNode`
- Attributes:
  - `float value` (private)
- Interface:
  - `__init__(int | float)`
  - `getValue():float`
- Comments: Take `int` or `float` as input of the constructor, but store the value as a `float` internally.

CellRef

- Extends `ASTNode`
- Attributes:
  - `int col` (private)
  - `int row` (private)
  - `bool isAbsCol` (private)
  - `bool isAbsRow` (private)
- Interface:
  - `__init__(Int,Int,Bool,Bool)`
  - `getColumn():int`
  - `getRow():int`
  - `isAbsColumn():bool`
  - `isAbsRow():bool`
- Comments: `isAbsRow`, `isAbsColumn` are True if the reference is absolute with respect to row and column, respectively. Note that `bool` may be implemented as an `int`. Recent versions of Python have a builtin `bool` type (enumeration of `True` and `False` with boolean operators defined) however.
  The column value must be smaller than 18278 (due to the fact that "ZZZ" converts to 18278, and that we require the maximal lenght of the string denoting the column value to be 3). The row value must be smaller than 10000.
  `ValueError` (Python built-in exception) should be raised if `col` or `row` < 0. The same exception should be raised when `col` or `row` exceed the above upper bounds.
  Thanks to the notation used, it is sufficient to check that row and column references are within the allowed range, irrespective of whether the reference is relative or absolute. When later, using the spreadsheet copy operation, relative references will be copied into a different context, it will be necessary to check that they do not point outside the valid range.

RangeRef

- Extends `ASTNode`
- Attributes:
  - `CellRef cornerA` (private)
  - `CellRef cornerB` (private)
- Interface:
  - `__init__(CellRef,CellRef)`
  - `getFirstCorner():CellRef`
  - `getSecondCorner():CellRef`
  - `getCellRefSet():ListOfCellRef`
- Comments: `getCellRefSet` returns a list of `CellRef`s. The content of the list is determined by the range `cornerA` to `cornerB`. The order in which `CellRef`s appear is row by row, from top to bottom and within a row, from left to right.
  When later, a formula will be copied, `cornerA` and `cornerB` may be changed in case they contain relative references. It is assumed that `getCellRefSet` is only called when we want to *evaluate* a formula. At that time, relative and absolute references refer to that same cells. This is why we choose (rather arbitrarily) to return a list of *absolute* references. For instance, `getCellRefSet` called on a `RangeRef B3:A1` returns `[$A$1,$B$1,$A$2,$B$2,$A$3,$B$3]`. Called on a `RangeRef $B3:$A$1`, `getCellRefSet` still returns `[$A$1,$B$1,$A$2,$B$2,$A$3,$B$3]`.
  The above detailed specification seems unnecessary as whether absolute or relative references are returned seems irrelevant. This may be true during this stage of the design, but it may no longer during a later stage of the design (for example, when the internal representations of `cornerA` and `cornerB` are changed. With the current specification, a caller of `getCellRefSet` knows *exactly* what to expect.

Function

- Extends `ASTNode`
- Attributes:
  - `String name` (private)
  - `ListOfASTNode args` (private)
- Interface:
  - `__init__(String,ListOfASTNode)`
  - `getName():String`
  - `getArgs():ListOfASTNode`
- Comments: `getArgs` returns a *copy* of the private list `args`. You should use the shallow `copy` method from the Python `copy` module (`http://www.python.org/doc/current/lib/module-copy.html`).

The Design

Class Diagram

You must provide a UML *class diagram*. Both attributes and associations (with corresponding role name) must be present.

Collaboration Diagram

Assume that we have an AST named `ast` in memory representing the formula `5+min(A1:A2)*10` and `str(ast)` is called. To obtain the string representation, your design must use the *visitor pattern* described in

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

In particular, `str(ast)` (implemented in the `__str__` method of `ASTNode`s will instantiate a `StrReprVisitor` and use it to visit the tree **with `ast` as root**. The visiting will be initiated by passing the `StrReprVisitor` instance **as an argument to `ast`'s `accept` method**. When this visiting stops, `__str__()` will access the full string representation by calling the `StrReprVisitor`'s method `getStrRepr`.

The formatting of the string representation must be such that it reproduces results given below (under "Interacting with the parser").

You must provide a complete UML *collaboration diagram* representing all the interactions between the instance of the `StrReprVisitor` class and the `ASTNode` instances corresponding to the above formula.

Implementation

An implementation of the design must be provided. In particular, the files `AST.py`, `ASTVisitor.py`, and `StrReprVisitor.py` must be provided.

Requirements common to all classes

- The implementation must follow the requirements *exactly* (i.e., same class names, method names etc.). The corrector will run his unit tests on your implementation. Two utility functions are provided (in `AST.py`) to convert an integer to its column string representation and vice versa.
- Every argument to a function call (including those passed to a class' constructor) must be type-checked. A `TypeError` (built-in Python) exception must be raised if an argument with the wrong type is passed.

Interacting with the parser

Once you have finished implementing, you experiment with the parser interactively. Simply run `main.py`. It will parse string inputs and print the AST string representation. You should get something like this:

```
Welcome to the formula parser!
Enter a formula (e.g. =4+5)
>>> =3+4-5
parse result: sum(3.0,4.0,invSum(5.0))
>>> =2*3+4^3+min($A$1:A2)
parse result: sum(mul(2.0,3.0),exp(4.0,3.0),min($A$1:A2))
>>> =3+4*2+min(2,3)
parse result: sum(3.0,mul(4.0,2.0),min(2.0,3.0))
>>> quit
```

Files

Required, but need not be modified:

- `parser.g` (the specification of the formula language)
- `Parser.py` (the lexer/parser generated by Yapps from `parser.g`)
- `ParseNode.py` (generated by Yapps, used by `Parser.py`)
- `yappsrt.py` (required by `Parser.py`)
- `main.py` (a small interactive driver for the formula parser)

Required, must be modified/created:

- `AST.py` (the AST classes)
- `ASTVisitor.py` (the ASTVisitor interface)
- `StrReprVisitor.py` (the StrReprVisitor class)

Tools

The produce the diagrams, you may for example use the diagram editor `dia` (`http://www.lysator.liu.se/~alla/dia/`) which is installed on the UNIX machines in the labs.