```
/****
 *
 * This file contains model class defintions for the SeeDo language.  These
 * definitions define the four major constructs of the language:
 *
 *      packaging: program, focus, see/do rules
 *      road patterns: five-tile patterns, road indicators such as stop signs
 *                     and speed gauges
 *      program expressions: textual expressions for coding that cannot be done
 *                           with pattern rules
 *      memory: a collection of name/value/property bindings
 *
 * These model data types will be translated into at least two lower-level data
 * representations for the purposes of execution:
 *
 *      Clips facts
 *      YAIR trees
 *
 * YAIR stands for yet another internal representation.  It is a
 * language-indpenendent internal form that is executable using the interpreter
 * that Trang To built for her MS thesis.  This is the same interpreter that's
 * being used to execute C+- programs in the Enchilada environment.
 *
 * There are a number of general lists that need to be declared here.  For
 * these we'll use the standard Fisher lists, which require specialization of
 * type List and ListElem.  See galaxy:~gfisher/projects/support/design/list.h
 * for documentation of lists.
 *
 * Regarding style, all data members are declared public, since we're doing
 * experimental prototyping here, not production software engineering.  The
 * Fisher convention of all-pointer allocation of class types is followed.  See
 * galaxy:~gfisher/projects/support/documentation/conventions.{ps,html} for
 * further discussion of Fisher's C++ conventions, some of which are followed
 * here.
 *
 */


/**
 * Forward class defs so that things can be defined top down.  (This definitely
 * calls for yet another note that C++ sucks.)
 */
class FocusList;
class Memory;
class ProcDeclList;
class NameList;
class ProcBody;
class Rule;
class ProgramExpr;
class Pattern;
class Tile;
class Bug;
class RoadBlock;
class ZamboniStatus;
class PatternExpr;
class Binding;


/*
 * A Program consists of a list of focuses and a memory.
 */
class Program {
  public:
    FocusList* foci;
    Memory* mem;

    Program();
    /*
     * Construct an empty Program.
     */

    ~Program();
    /*
     * Deep (for now) destruct a Program.
     */

    Program* Build(Am_Object);
    /*
     * Build this from the given Am_Object, which is in the form of a
     * scrollGroup.  Call all the little-buddy Build functions in classes
     * defined below, q.q.v.
     */

    char* GenClipsFacts();
    /*
     * Generate the CLIPS rule base for the pattens represented in this.
     */

    ParseStruct* GenInterpStructs();
    /*
     * Generate an interpreter parse tree and symbol table for the program
     * represented in this.
     */
};

class FocusList : public List {};
class FocusListElem : public ListElem {};

/*
 * A focus has a name and list of proc decls.
 */
class Focus {
  public:
    String* name;
    ProcDeclList* procdecls;
};

class ProcDeclList : public List {};
class ProcDeclListElem : public ListElem {};

/*
 * A proc decl has a name, optional formal parm list and body.  Since SeeDo is
```

```
 * a dynamically typed language, formal parms are declared simply as a name
 * list, as opposed to the name/type lists of statically typed languages.
 */
class ProcDecl {
  public:
    String* name;
    NameList* parms;            // Null if no formal parms
    ProcBody* body;

    ProcDecl* Build(Am_Object ao);
    /*
     * Build one proc decl from the given Am_Object, which is of the form of an
     * Am_Object with four slots: SEEDO_NAME, SEEDO_PARMS, UPPER_GROUP, and
     * LOWER_GROUP.  The SEEDO_NAME slot is a Am_Text, which gets converted to
     * String* and stored in this->name.  The UPPER_ and LOWER_GROUPs are
     * Am_Scrolling_Groups, which contain List_Slot slot, which is an
     * Am_ObjectList.  These groups get converted to a Rule, which is stored in
     * this->body.  The PARMS slot is unused at present.
     */
};

typedef NameList StrList;

/*
 * A procedure body can be a single pattern-based rule, or a general program
 * expression.
 */
enum BodyType {
    RULE,
    EXPR
};

/*
 * ProcBody is the generic class for a procedure body, of which RuleBody and
 * ExprBody are subclasses.  ProcBody will presumably expand when member
 * functions are added.
 */
class ProcBody {
  public:
    BodyType type;
};

/*
 * A rule body has exactly one pattern-based rule.  This is in keeping with the
 * design decision to disallow multi-rule bodies, owing to the incresased
 * semantic complexity that would arise.  A major limiting constraint is that
 * we are using CLIPS, where representation of multi-rule proc bodies would be
 * difficult.
 */
class RuleBody : public ProcBody {
  public:
    Rule* rule;
};

/*
 * An expr body is a program expression, defined below after the pattern defs.
 */
class ExprBody : public ProcBody {
  public:
    ProgramExpr* expr;
};

/*
 * A rule is the basic see/do pattern.  (Here's a fun deal: try defining the
 * second data member of this class "do" instead of "dopat".  Who's idea was it
 * to make "do" a keyword in C anyway??)
 */
class Rule {
  public:
    Pattern* seepat;
    Pattern* dopat;

    Rule* Build(Am_Object upper, Am_Object lower);
    /*
     * Build one rule from the given Am_Objects, which are both in the form of
     * Am_ScrollingGroups.  The upper has the see pattern grid list and the
     * lower has the do pattern grid list, both stored as Am_ObjectLists in the
     * List_Slot slot of the scrolling groups.  Each elem of the lists is an
     * Am_Group, with slots for five Am_Rects, plus one slot for the prefix not
     * operator, with enum value NOT_OP or NULL_OP, and one slot for the infix
     * operator, with enum value OR_OP or AND_OP or NULL_OP.
     *
     * Each of the five rects are built into a RoadPattern.  Then the
     * RoadPatterns are built into a Pattern object, with is either a single
     * pattern or a pattern expr, per the Build ops in thes classes below.
     */
};

/*
 * A pattern comes in five possible types.
 */
enum PatternType {
    FIXED_ROAD_PATTERN,         // Five tiles in standard fixed positions
    GENERAL_ROAD_PATTERN,       // Indefinite tiles -- NOT CURRENTLY supported
    ROAD_INDICATOR,             // A constant or varialbe indicator, e.g., a
                                //   stop sign or speed gauge
    PATTERN_EXPR,               // Patterns and'd, or'd, or not'd
    PROGRAM_EXPR                // General program expr -- CURRENTLY TEXT ONLY
};

/*
 * Pattern is the generic pattern class, of which RoadPattern and PatternExpr
 * are subclasses.
 */
class Pattern {
  public:
    PatternType type;
};

/*
 * RoadPattern is the standard five-tile pattern used in see/do rules.  We're
 * not defining the more general road pattern (with arbitrary number and
```

```
 * placement of tiles) in version 1 of the language.
 */
class RoadPattern : public Pattern {
  public:
    Tile* center;
    Tile* left;
    Tile* top;
    Tile* right;
    Tile* bottom;

    RoadPattern* Build(Am_Object);
    /*
     * Build one five-part road pattern from the given Am_Object, which is in
     * the form of an Am_Group with five Am_Rects, slot names of which are
     * Grid_{Center,Left,Right,Up,Down}.  Each of the rects will in turn have
     * four slots named Tile_Type, Is_Bugged, Tile_Marker, and Tile_Blockage.
     *
     * Tile_Type is of type TileType enum below.  Is_Bugged is a bool.
     * Tile_Marker is an Am_Group of containing a StringEditor for the marker
     * name, a Label for the name/val separator colon, and another StringEd for
     * the value.  Tile_Blockage is of type RoadBlock* defined below (at least
     * for now).
     */
};

/*
 * TileType defines all the possible tile empty configurations.
 */
enum TileType {
    CROSS_TILE,
    STRAIGHT_VERTICAL_TILE,
    STRAIGHT_HORIZONTAL_TILE,
    TEE_LEFTENTRANCE_TILE,
    TEE_TOPENTRANCE_TILE,
    TEE_RIGHTENTRANCE_TILE,
    TEE_BOTTOMENTRANCE_TILE,
    TURN_TOP_LEFT_TILE,
    TURN_TOP_RIGHT_TILE,
    TURN_BOTTOM_LEFT_TILE,
    TURN_BOTTOM_RIGHT_TILE,
    TURN_DOWN_TILE,
    DONT_KNOW_TILE,
    DONT_CARE_TILE,
    BLANK_TILE
};

/*
 * A tile marker is the same as a memory binding (see below).
 */
typedef Binding Marker;

/*
 * Class Tile defines a tile as it appears in a pattern.  It contains a type
 * and the things that can be on it, i.e., the bug, a marker, and/or a road
 * block.
 */
class Tile {
  public:
    TileType type;
    Bug* bug;                    // Non-null if bug is on this tile
    Marker* marker;              // Non-null if this tile has a marker
    RoadBlock* block;            // Non-null if this tile has road block(s)
};

/*
 * Orientation enumerates the four possible bug orientations.
 */
enum Orientation {
    LEFT_ORIENTATION,
    RIGHT_ORIENTATION,
    UP_ORIENTATION,
    DOWN_ORIENTATION
};

/*
 * The Bug is defined by its orientation and status data.  Note that the
 * zamboni status structure is really too low level here, but it's all that's
 * fully defined for now.  We'll define a higher-level status structure soon.
 */
class Bug {
  public:
    Orientation orientation;
    ZamboniStatus* zambstat;
};

/*
 * ZamboniStatus is the lowest level status data for the bug.  As noted above,
 * we'll define a higher-level status info block shortly.
 */
class ZamboniStatus {
    int syncno;          // Number of the command being responding to
    int distl;           // Distance counter for left motor
    int distr;           // Distance counter for right motor
    int lrstate;         // Three-bit encoding of three center sensors
    int codestate;       // Two-bit encoding of the outer two sensors
    int lrbumpers;       // Two-bit encoding of the two front bumpers
};

/*
 * A road block is represented as a list of the exits that are blocked, based
 * on the placement of the physical block with respect to the tile type.  E.g.,
 * if there is a road block in on a vertical straight, then the top and bottom
 * exits are blocked.  If is there a block located on the upper exit of a
 * left-entry tee, then only upper exit is block, not the lower exit.
 */
class RoadBlock {
  public:
    bool leftblocked;
    bool rigntblocked;
    bool topblocked;
    bool bottomblocked;
};
```

```

/*
 * IndicatorType enumerates the kind of road indicators that can appear in a
 * pattern in addition to a road tile.
 */
enum IndicatorType {
    STOP_INDICATOR,             // Stop sign
    GO_BACK_INDICATOR,          // Go back (Greg's mean-looking pit bull)
    SPEED_INDICATOR,            // The bug speed gauge
    HEADING_INDICATOR,          // The bug heading indicator (N,E,S,W)
    VOLUME_INDICATOR,           // The bug sound volume indicator
    PLAY_SOUND_INDICATOR        // The bug sound pitch
};

/*
 * The RoadIndicator class has a type and generic integer value.
 */
class RoadIndicator {
  public:
    IndicatorType type;
    int value;                  // Type-specific value:
                                //   speed between 0 and 7
                                //   heading in degrees mod 90
                                //   volume between 0 and 8
                                //   sound as midi step
};

/*
 * PatternExprOp enumerates the three logical connectives that can be used to
 * define see/do pattern exprs.  Semantically, and'ing of see patterns will not
 * be defined in Version 1.
 */
enum PatternExprOp {
    AND_PATTERN_OP,
    OR_PATTERN_OP,
    NOT_PATTERN_OP
};

class PatternExpr : public Pattern {
  public:
    PatternExprOp op;
    PatternExpr* leftoprnd;
    PatternExpr* rightoprnd;    // Null in unary pattern expr

    PatternExpr* Build(Pattern* left, Pattern* right);
    /*
     * Build this out of the given left and right operands.
     */
};

/*
 * ProgramExprOp enumerates all of the general program expression operators.
 * In Version 1, program expressions will be textually represented.
 */
enum ProgramExprOp {
    NULL_OP,                    // Indicates atomic expr (i.e., leaf in expr
                                // tree)

    SEQ_OP,                     // Sequence two exprs; n-ary sequences are
                                //   hence represented as binary trees

    AND_OP,                     // Logical and
    OR__OP,                     // Logical or
    NOT_OP,                     // Logical not

    EQ_OP,                      // Boolean equals
    NOT_EQOP,                   // Boolean not equals
    LESS_OP,                    // Boolean less than
    GREATER_OP,                 // Boolean greater than
    LESSEQ_OP,                  // Boolean less than or equal
    GREATEREQ_OP,               // Boolean greater than or equal

    PLUS_OP,                    // Arithmetic plus
    MINUS_OP,                   // Arithmetic minus
    TIMES_OP,                   // Arithmetic times
    DIVIDE_OP,                  // Arithmetic divide
    EXPONENT_OP                 // Arithmetic exponentiation
};

/*
 * Program expr is the generic class of which atomic and binary expr are
 * subclasses.
 */
class ProgramExpr {
  public:
    ProgramExprOp op;
};

/*
 * An AtomicExpr is a value, as defined below after class Memory.
 */
class AtomicExpr : public ProgramExpr {
  public:
    Value* val;                 // Represent a SeeDo value as a YAIR value
                                //   descriptor.
};

/*
 * A BinExpr is a standard binary or unary expression.
 */
class BinExpr : public ProgramExpr {
    PatternExpr* leftoprnd;
    PatternExpr* rightoprnd;    // Null in unary pattern expr
};

class BindingList : public List {};
class BindingListElem : public ListElem {};

/*
 * Memory is simply a list of value bindings.
 */
class Memory {
```

```
  public:
    BindingList* bindings;
};

class PropertyList : public List {};
class PropertyListElem : public ListElem {};

/*
 * A binding is a standard Lisp-style triple of name, value, and property list.
 * The value is represented as a YAIR taged value.
 */
class Binding {
  public:
    String* name;
    Value* val;
    PropertyList* props;
};

/*
 * A property is a la Lisp -- a name/value pair.
 */
class Property {
  public:
    String* name;
    Value* val;
};
```