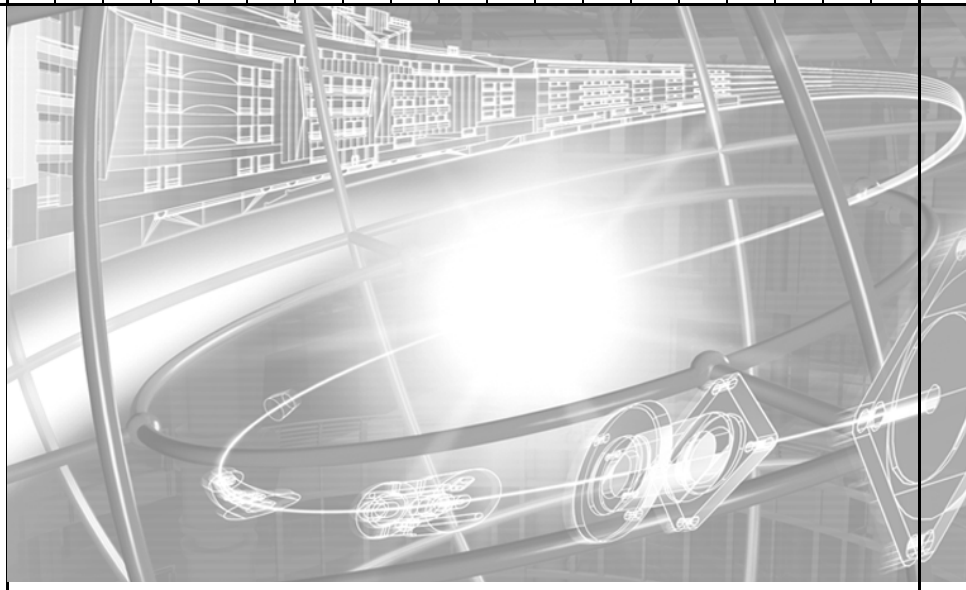
autodesk®

AutoCAD® 2000i



DIDACTICIEL VISUAL LISP™

00120-110000-5081 Juillet 2000

1 2 3 4 5 6 7 8 9 10

Publié par : Autodesk Development S. à r. l.
Rue du Puits-Godet 6
Case postale 35
2005 Neuchâtel
Suisse

Table des matières

Introduction

Ce didacticiel est destiné à présenter quelques-unes des puissantes fonctionnalités de l'environnement de programmation Visual LISP™ pour AutoCAD® ainsi que les fonctions du langage AutoLISP® qui vous étaient peut-être inconnues jusqu'alors.

L'objectif de ce didacticiel est de dessiner un sentier de jardin à l'aide d'un outil de dessin automatisé qui minimise le temps de création et démontre la puissance de la programmation paramétrique. Vous apprendrez à créer un sous-programme de dessin automatisant la génération d'une forme complexe (c'est le type de modèle que vous ne voudriez pas dessiner s'il s'agissait de toujours recommencer du début).

Dans ce chapitre

Une approche nouvelle du sentier de jardin : Travailler dans Visual LISP

Ce didacticiel est destiné aux utilisateurs d'AutoCAD expérimentés et se base sur leur connaissance du langage LISP ou AutoLISP. Il tient également compte du fait que vous maîtrisez les tâches de base relatives à la gestion de fichiers sous Windows®, par exemple la création de répertoires, la copie de fichiers et la navigation au sein du système de fichiers de votre disque dur ou du réseau.

Si vous maîtrisez AutoLISP et que vous avez utilisé les versions antérieures de ce didacticiel, vous remarquerez plusieurs différences :

- L'environnement Visual LISP (VLISP™) est incorporé au logiciel. Il vous propose des outils d'édition, de débogage, ainsi que d'autres outils spécifiques à la création d'applications AutoLISP. Les versions antérieures du didacticiel abordaient les concepts du langage AutoLISP, et non les outils de développement VLISP.
- Les nouvelles fonctions ActiveX™ et Réacteur d'AutoLISP vous sont présentées, ainsi que plusieurs autres extensions du langage AutoLISP proposées par VLISP.
- Le didacticiel a été entièrement remodelé. Même si vous connaissez bien la version antérieure, vous allez découvrir un code source totalement différent et parcourir un didacticiel beaucoup plus complet.
- Il existe deux contextes d'exécution correspondant au didacticiel de sentier de jardin. L'application peut être exécutée en tant qu'interpréteur LISP dans des petits bouts de fichiers et/ou des fonctions chargées dans un seul document. D'autre part, le code du programme peut être compilé dans une application VLX, repérée par un fichier ***.vlx*** exécutable. Une application VLX fonctionne à partir d'un espace mémoire indépendant qui peut être en relation avec le document de chargement de l'application.

Présentation du didacticiel

Tout au long de ce didacticiel, votre objectif sera de développer une nouvelle commande AutoCAD qui dessine un sentier dans un jardin et le pave de dalles circulaires. Le didacticiel est divisé en sept leçons. Au fur et à mesure de votre progression, vous recevrez de moins en moins d'informations détaillées sur l'exécution des tâches individuelles. Si vous avez des questions, utilisez l'aide se trouvant dans la documentation VLISP.

Les leçons 4 et 5 se situent à un niveau intermédiaire et dépassent le cadre des notions de base AutoLISP. Les leçons 6 et 7 comportent des tâches de programmation avancées et relativement complexes, destinées à des développeurs AutoLISP expérimentés.

Le code source correspondant au dessin du sentier de jardin à chaque étape du développement se trouve sur le CD-ROM d'installation d'AutoCAD ; les fichiers du didacticiel ne seront inclus à votre installation que si vous choisissez l'installation Complète, ou l'installation Personnalisée en sélectionnant l'option Exemples. Si vous avez déjà installé AutoCAD sans inclure les exemples, exécutez à nouveau l'installation, choisissez Personnalisée, puis sélectionnez uniquement l'option Exemples.

La structure de répertoires correspondant aux fichiers de code source suit le plan du didacticiel :

<Répertoire AutoCAD>\Tutorial\VisualLISP\Lesson1

<Répertoire AutoCAD>\Tutorial\VisualLISP\Lesson2

etc.

Nous vous recommandons de ne pas modifier les fichiers de code source fournis en exemple avec AutoCAD. Si votre programme ne fonctionne pas correctement, il est possible de copier le code source fourni dans votre propre répertoire de travail. Dans ce didacticiel, le répertoire de travail est défini comme suit :

<Répertoire AutoCAD>\Tutorial\VisualLISP\MyPath

Si vous choisissez un chemin d'accès différent pour votre répertoire de travail, changez le nom de votre répertoire au moment approprié.

Pour terminer, lisez la section Getting Started du *Visual LISP Developer's Guide*. Elle comporte une présentation succincte des nombreux concepts que vous aurez à utiliser pour terminer ce didacticiel.

1

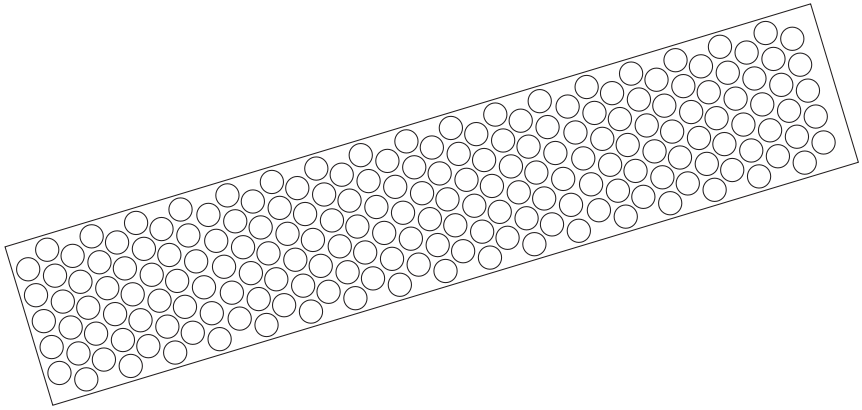
Conception et démarrage du programme

Cette première leçon vous permettra de définir ce que l'application sera appelée à faire. A l'aide de l'environnement de développement (VLISP), vous créerez un fichier LISP et commencerez à écrire le code AutoLISP de votre application. Au cours de ce travail, vous découvrirez progressivement comment VLISP facilite le développement d'une application.

Dans ce chapitre

Définition des objectifs généraux du programme

Le développement d'un programme AutoLISP implique tout d'abord l'automatisation de certaines fonctions d'AutoCAD. Il peut s'avérer nécessaire d'accélérer une fonction de dessin répétitive ou de simplifier une série complexe d'opérations. Dans le cadre de ce didacticiel, le sentier de jardin que vous chargez votre programme de dessiner est une forme complexe comportant un nombre variable de composants selon la saisie initiale de l'utilisateur. Voici ses caractéristiques :



Pour dessiner le sentier de jardin, votre programme doit effectuer les opérations suivantes :

- Dessiner un contour rectiligne à partir d'un point de départ, d'une extrémité et d'une largeur. Le contour peut être inséré dans n'importe quelle orientation 2D. Sa taille ne doit pas être limitée.
- Demander à l'utilisateur de sélectionner la taille et l'espacement des dalles. Ces dalles sont des cercles simples qui rempliront le contour mais qui ne doivent pas le chevaucher ou le traverser.
- Positionner les dalles en rangées alternées.

Pour vérifier le fonctionnement du programme, vous pouvez exécuter une version finalisée de l'application livrée avec AutoCAD.

Pour exécuter l'application fournie en exemple

1. Dans le menu Outils d'AutoCAD, choisissez Charger une application.
2. Sélectionnez *gardenpath.vlx* à partir du répertoire *Tutorial\VisualLISP*, puis cliquez sur Charger.
3. Cliquez sur Fermer.
4. Sur la ligne de commande, entrez **gpath**.
5. Répondez aux deux premières invites en sélectionnant un point de départ et une extrémité au sein de la fenêtre graphique d'AutoCAD.
6. Entrez **2** lorsque vous êtes invité à entrer une valeur de demi-largeur du sentier.
7. Dans la boîte de dialogue Garden Path Tile Specifications, cliquez sur OK.

Mise en route de Visual LISP

Maintenant que vous avez vu comment l'application est censée fonctionner, vous pouvez commencer à la développer avec VLISP. Tout d'abord, il est utile d'expliquer ce qui peut se passer lorsque VLISP attend qu'AutoCAD lui rende le contrôle. Vous avez peut-être déjà été confronté à ce type de situation.

Pour voir Visual LISP en attente qu'AutoCAD lui rende le contrôle

1. Sur la ligne de commande AutoCAD, entrez `vlisp` pour lancer Visual LISP.
2. Revenez à la fenêtre AutoCAD (soit en cliquant sur AutoCAD dans la barre des tâches, soit en appuyant sur ALT+TAB puis en sélectionnant AutoCAD), puis entrez `gpath` sur la ligne de commande AutoCAD.
3. Avant de répondre aux invites succédant **gpath**, revenez à la fenêtre VLISP.

Dans la fenêtre VLISP, le pointeur de la souris apparaît sous la forme d'un symbole VLISP ; il est impossible de choisir une commande ou d'entrer du texte dans la fenêtre VLISP. Le symbole correspondant au pointeur vous rappelle que vous devez terminer une opération dans AutoCAD avant de reprendre votre travail dans VLISP. Rappelez-vous en lorsque vous voyez apparaître le pointeur VLISP.

4. Revenez à la fenêtre AutoCAD et répondez à toutes les invites de **gpath**.

A présent, vous êtes prêt à créer l'application du sentier de jardin.

Pour commencer le développement de l'application avec Visual LISP

1 Dans le menu Fichier de VLISP, choisissez l'option Nouveau fichier.

2 Entrez le code suivant dans la fenêtre de l'éditeur de texte (intitulée "<SansNom-0>") ; vous pouvez omettre les commentaires si vous le souhaitez :

```
;;; Function C:GPath is the main program function and defines the
;;; AutoCAD GPATH command.
(defun C:GPath ()
  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters. Continue only if you have
  ;; valid input.
  (if (gp:getPointInput)      ;
    (if (gp:getDialogInput)
      (progn
        ;; At this point, you have valid input from the user.
        ;; Draw the outline, storing the resulting polyline
        ;; "pointer" in the variable called PolylineName.
        (setq PolylineName (gp:drawOutline))
        (princ "\nThe gp:drawOutline function returned <")
        (princ PolylineName)
        (princ ">")
        (Alert "Congratulations - your program is complete!")
      )
     (princ "\nFunction cancelled.")
    )
    (princ "\nIncomplete information to draw a boundary.")
  )
  (princ)  ; exit quietly
)
;;; Display a message to let the user know the command name.
(princ "\nType gpath to draw a garden path.")
(princ)
```

3 Choisissez Fichier ➤ Enregistrer sous à partir du menu, puis enregistrez le code dans le nouveau fichier sous *<Répertoire AutoCAD>\Tutorial\VisualLISP\MyPath\gpmain.lsp*

4 Vérifiez votre travail.

A propos du formatage de code Visual LISP

VLISP reconnaît les différents types de caractères et de mots constituant un fichier de programme AutoLISP et met les caractères en évidence en utilisant différentes couleurs. Vous pouvez ainsi repérer les erreurs plus rapidement. Par exemple, si vous oubliez de fermer les guillemets à la fin d'une chaîne de texte, ce qui vient à la suite reste affiché en magenta, la couleur correspondant aux chaînes. Lorsque vous fermez les guillemets, VLISP affiche le texte suivant la chaîne dans la couleur appropriée, en fonction de l'élément du langage qu'il représente.

VLISP formate aussi le texte tout au long de la saisie en ajoutant des espacements et des retraits. Pour que VLISP formate le code que vous copiez dans son éditeur de texte depuis un autre fichier, choisissez Outils ➤ Formater le code dans la barre de menus VLISP.

Analyse du code

L'instruction **defun** définit la nouvelle fonction. Vous remarquerez que la fonction principale est nommée **C:GPath**. Le préfixe **C:** signifie que vous pouvez appeler cette fonction depuis la ligne de commande d'AutoCAD. **GPath** représente le nom que les utilisateurs entrent pour lancer l'application à partir de la ligne de commande d'AutoCAD. Les fonctions obtenant des entrées de la part des utilisateurs s'intitulent **gp:getPointInput** et **gp:getDialogInput**. La fonction qui dessine le contour du sentier de jardin s'intitule **gp:drawOutline**. Ces noms comportent le préfixe gp pour indiquer qu'ils sont spécifiques à l'application Garden Path. Ce n'est pas une obligation, mais c'est une bonne convention de nomination, qui permet de distinguer les fonctions spécifiques aux applications des fonctions à usage général que vous utilisez fréquemment.

Dans la fonction principale, les expressions **princ** affichent les résultats du programme si ce dernier fonctionne correctement, ou un message d'avertissement s'il est confronté à une erreur inattendue. Par exemple, comme nous le verrons dans la leçon 2, si l'utilisateur appuie sur ENTREE au lieu de sélectionner un point à l'écran, l'appel **gp:getPointInput** s'interrompt prématurément, renvoyant une valeur nil à la fonction principale. Le programme affiche ensuite un message **princ** : "Incomplete information to draw a boundary".

L'appel **princ** situé vers la fin du programme fait office d'invite. Lors du chargement de l'application, l'invite informe les utilisateurs de la commande qu'ils doivent saisir pour commencer le dessin d'un sentier de jardin. L'appel **princ** final sans argument de chaîne contraint l'utilisateur à quitter le programme proprement, ce qui signifie que la valeur correspondant à l'expression finale de la fonction principale n'est pas renvoyée. Si l'expression finale supprimant la fonction **princ** était omise, l'invite s'afficherait à deux reprises.

Parachèvement du programme

Pour que le code fonctionne correctement dans ce nouveau fichier, vous devez écrire trois définitions de fonction supplémentaires. Le code principal du sentier de jardin contient les appels de trois fonctions personnalisées :

- **gp:getPointInput**
- **gp:getUserInput**
- **gp:drawOutline**

Pour l'instant, vous devez juste écrire des définitions de fonctions de test par tronçons. Une fonction de test par tronçons sert de marque de réservation pour l'intégralité de la fonction à suivre. Elle vous permet de tester des portions de votre code avant d'ajouter tous les détails nécessaires au lancement de l'application.

Pour définir des fonctions de test par tronçons pour l'application

1. Placez votre curseur en haut du code du programme dans la fenêtre de l'éditeur de texte, puis appuyez sur ENTREE à plusieurs reprises pour ajouter des lignes vides.
2. Entrez le code suivant en commençant à l'endroit où vous avez inséré les lignes vides :

```
;;; Function gp:getPointInput will get path location and size
(defun gp:getPointInput()
  (alert
    "Function gp:getPointInput will get user drawing input"
  )
  ;; For now, return T, as if the function worked correctly.
  T
)

;;; Function gp:getDialogInput will get path parameters
(defun gp:getDialogInput ()
  (alert
    "Function gp:getDialogInput will get user choices via a dialog"
  )
  ;;For now, return T, as if the function worked correctly.
  T
)
```

```
;;; Function gp:drawOutline will draw the path boundary
(defun gp:drawOutline ()
  (alert
    (strcat "This function will draw the outline of the polyline"
                "\nand return a polyline entity name/pointer."
    )
  )
  ;; For now, simply return a quoted symbol. Eventually, this
  ;; function will return an entity name or pointer.
  'SomeEname
)
```

Une ligne de code contenant uniquement un `T` est située juste avant la fin de chaque fonction d'entrée. Elle est utilisée comme valeur renvoyée à la fonction appelante. Toutes les fonctions AutoLISP renvoient une valeur à la fonction qui les a appelées. La lettre `T` signifie "true" ("vraie") dans AutoLISP ; si vous l'ajoutez, la fonction renvoie une valeur vraie. Etant donné la manière dont *gpmain.lsp* est structuré, chaque fonction d'entrée qu'il appelle doit renvoyer une valeur autre que `nil` (qui signifie "aucune valeur") pour que le programme passe à l'étape suivante.

Par défaut, une fonction AutoLISP renverra la valeur de la dernière expression évaluée. Dans les fonctions de test par tronçons, la seule expression est un appel de la fonction **`alert`**. Toutefois, cette fonction **`alert`** renvoie toujours la valeur `nil`. Si elle correspond à la dernière expression dans **`gp:getPointInput`**, elle renverra toujours la valeur `nil` et vous ne pourrez jamais passer de la fonction **`if`** à la fonction **`gp:getDialogInput`**.

De même, la fin de la fonction **`gp:DrawOutline`** renvoie un symbole entre guillemets (`'SomeEname`) correspondant à une marque de réservation. Un symbole entre guillemets représente une structure LISP non évaluée. (Si vous voulez connaître le fonctionnement du langage LISP, de nombreux livres traitant de ce sujet sont mentionnés à la fin de ce didacticiel.)

Vérification du code par Visual LISP

VLISP dispose d'une fonctionnalité performante de vérification de votre code permettant de repérer les erreurs de syntaxe. Utilisez cet outil avant d'exécuter le programme. Il est possible de repérer les erreurs de frappe communes, par exemple les oublis de parenthèses ou de guillemets, ainsi que d'autres problèmes d'ordre syntaxique.

Pour vérifier la syntaxe de votre code

1. Assurez-vous que la fenêtre de l'éditeur de texte contenant *gpmain.lsp* est active. (Cliquez dans la barre de titre de la fenêtre pour l'activer.)

2. A partir de la barre de menus VLISP, choisissez Outils ➤ Vérifier le texte dans l'éditeur.
3. La fenêtre Générer sortie s'affiche avec les résultats de la vérification de la syntaxe. Si VLISP ne détecte aucune erreur, la fenêtre affiche un texte semblable à celui qui suit :

```
[CHECKING TEXT GPMAIN.LSP loading...]
......
; Check done.
```

Si vous rencontrez des problèmes et si vous avez besoin d'aide, reportez-vous au chapitre "Developing Programs with Visual LISP" du *Visual LISP Developer's Guide*. Voyez si vous êtes en mesure de déterminer l'origine du problème. Si vous passez trop de temps à rechercher la source du problème, utilisez le fichier d'exemple *gpmain.lsp* se trouvant dans le répertoire *lesson1*, puis poursuivez avec le didacticiel.

Pour utiliser (si nécessaire) le programme *gpmain.lsp* fourni

1. Fermez la fenêtre de l'éditeur de texte contenant le code *gpmain.lsp* que vous avez entré.
2. Dans la barre de menus VLISP, choisissez Fichier ➤ Ouvrir un fichier, puis ouvrez le fichier *gpmain.lsp* se trouvant dans le répertoire *\Tutorial\VisualLISP\lesson1*.
3. Choisissez Fichier ➤ Enregistrer sous, puis enregistrez le fichier dans votre répertoire *\Tutorial\VisualLISP\MyPath* sous le nom *gpmain.lsp* en écrasant la copie que vous avez créée.

Exécution du programme avec Visual LISP

L'exécution de programmes AutoLISP dans VLISP vous permet d'utiliser les nombreuses fonctions de débogage de VLISP pour détecter les problèmes susceptibles de survenir dans votre application.

Pour charger et exécuter le programme

1. En activant la fenêtre de l'éditeur de texte, choisissez Outils ➤ Charger le texte dans la barre de menus VLISP.
2. A l'invite _$ dans la fenêtre de la Console VLISP, entrez **`(C:GPath)`** et appuyez sur ENTREE.

   La fenêtre de la console attend que vous entriez des commandes en utilisant la syntaxe AutoLISP ; tous les noms de fonction doivent donc figurer entre parenthèses.
3. Appuyez sur ENTREE ou cliquez sur OK pour répondre aux boîtes de message. Le dernier message doit être le suivant : "Congratulations – your program is complete!".

REMARQUE Si la fenêtre AutoCAD est réduite lorsque vous exécutez **`gpath`**, vous ne verrez pas les invites s'afficher à l'écran. Pour y remédier, restaurez la fenêtre AutoCAD (soit par l'intermédiaire de la barre des tâches, soit en appuyant sur ALT+TAB).

Résumé de la leçon 1

Au cours de cette leçon, vous avez

- Défini les objectifs du programme.
- Appris la valeur des fonctions de test par tronçons.
- Découvert les conventions de nomination pour les identifier comme fonctions spécifiques à votre application ou comme fonctions générales à utiliser régulièrement.
- Appris à utiliser VLISP pour vérifier votre code.
- Appris à charger et à exécuter un programme dans VLISP.

Cette leçon est terminée. Enregistrez à nouveau votre fichier de programme afin de vous assurer d'avoir la dernière version.

Utilisation des outils de débogage Visual LISP

2

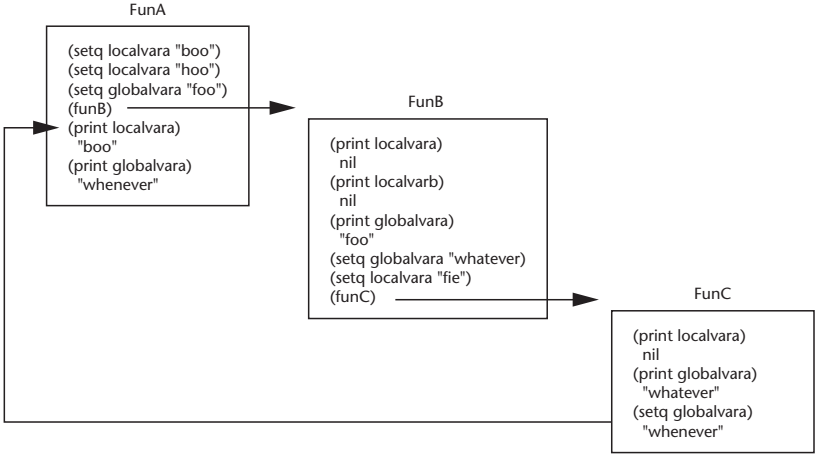
Cette leçon vous apprendra à utiliser plusieurs outils de débogage Visual LISP permettant d'accélérer le développement de programmes AutoLISP. Vous aborderez également la différence entre les variables locales et les variables globales ainsi que leur champ d'utilisation. Votre programme sera plus actif (il demandera aux utilisateurs d'entrer des informations). Les informations seront enregistrées dans une liste et vous commencerez à comprendre l'utilité des listes dans vos programmes AutoLISP.

Dans ce chapitre

Différenciation entre les variables locales et les variables globales

Cette leçon traite de l'utilisation des variables locales par rapport aux variables de document globales. Les variables globales sont accessibles par l'intermédiaire de toutes les fonctions chargées dans le document (un dessin AutoCAD). Elles peuvent conserver leur valeur une fois que le programme qui les a définies s'achève. C'est parfois ce que vous souhaitez. Un exemple vous sera présenté plus avant dans ce didacticiel.

Les variables locales ne conservent leur valeur que lorsque la fonction qui les a définies est en cours d'exécution. Une fois que l'exécution de la fonction est terminée, les valeurs de la variable locale sont automatiquement supprimées et le système récupère l'espace mémoire utilisé par la variable. C'est ce que l'on appelle une purge mémoire automatique ; on retrouve cette fonctionnalité dans la plupart des environnements de développement LISP, tels que VLISP. Les variables locales utilisent mieux la mémoire que les variables globales.



En outre, et c'est l'un des atouts majeurs, les variables locales permettent de mieux déboguer et mettre à jour vos applications. Avec les variables globales, vous n'êtes jamais sûr de la période à laquelle la valeur de la variable est susceptible d'être modifiée, ni de la fonction dans laquelle sera effectué le changement ; à l'inverse, les variables locales ne nécessitent pas une telle analyse. En général, les effets secondaires sont mineurs (une partie de programme peut, par exemple, avoir une incidence sur une variable d'une autre partie du programme).

Etant donnés les avantages mentionnés ci-dessus, le didacticiel utilisera les variables locales de manière quasi systématique.

REMARQUE Si vous travaillez avec AutoLISP depuis quelques temps, vous avez peut-être pris l'habitude d'utiliser les variables globales lors du développement afin d'examiner votre programme pendant sa création. Etant donné que VLISP dispose d'outils de débogage performants, vous pouvez abandonner cette méthode.

Utilisation de variables locales dans le programme

Reportez-vous à la fonction **gp:getPointInput** que vous avez créée dans la leçon 1 :

```
(defun gp:getPointInput()

  (alert
    "Function gp:getPointInput will get user drawing input"
  )
  ;; For now, return T, as if the function worked correctly.
  T
)
```

Jusqu'à présent, le rôle de cette fonction n'a pas été très important. Vous allez maintenant commencer à l'exploiter en ajoutant des fonctions pour obtenir des entrées de l'utilisateur, qui définiront le point de départ, l'extrémité et la largeur du sentier.

Lorsque vous créez des programmes AutoLISP, il est recommandé d'émuler le comportement d'AutoCAD. Pour cela, au lieu de demander à l'utilisateur d'indiquer la largeur en sélectionnant un point dans le dessin par rapport à l'axe d'une forme linéaire, votre programme devrait vous demander de sélectionner la demi-largeur.

Une fois la fonction **gp:getPointInput** complète, les variables, tout comme les valeurs qui leur sont affectées, n'existent plus. Par conséquent, vous enregistrerez les valeurs entrées par l'utilisateur dans les variables locales. Voici un aperçu de ce à quoi la fonction pourrait ressembler :

```
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
          T
      )
    )
  )
)
```

Les variables locales sont déclarées à la suite de la barre oblique dans l'instruction **defun** marquant le début de la fonction. Le premier appel de **getpoint** demande à l'utilisateur de sélectionner un point de départ. Ensuite, l'extrémité est définie par rapport au point de départ choisi. Lorsqu'il sélectionnera l'extrémité, l'utilisateur remarquera une ligne élastique s'étendant depuis le point de départ. De la même manière, lorsqu'il définira la valeur de la demi-largeur, l'utilisateur verra s'afficher une autre ligne élastique partant de l'extrémité et représentant la distance.

Pour voir comment fonctionne gp:getPointInput

1. Saisissez le code **gp:getPointInput** dans la fenêtre de la Console Visual LISP.
2. En plaçant le curseur de la fenêtre de la console derrière la dernière parenthèse du bloc de code (ou sur la ligne qui suit), appuyez sur ENTREE ; vous remplacerez ainsi toute version de la fonction **gp:getPointInput** chargée antérieurement.
3. Exécutez la fonction à partir de la fenêtre de la console en entrant **(gp:getPointInput)** à l'invite de la console.
4. Sélectionnez des points lorsque le programme vous y invite, puis entrez une valeur de demi-largeur.

Examen de la fonction gp:getPointInput

Lorsque vous avez exécuté la fonction **gp:getPointInput**, les commandes ont été automatiquement transférées de VLISP à AutoCAD. Vous avez répondu à trois messages, puis les commandes sont repassées d'AutoCAD à VLISP et un symbole `T` s'est affiché dans la fenêtre de la console.

Voici comment le programme interprète cette opération :

1. VLISP attend que vous sélectionniez le premier point.
2. Lorsque vous sélectionnez le premier point, le programme enregistre la valeur de votre sélection (une liste contenant trois coordonnées – une valeur *X, Y,* et *Z*) dans la variable `StartPt`.
3. La première fonction **if** examine le résultat afin de déterminer si une valeur correcte a été entrée ou si aucune valeur n'a été entrée. Lorsque vous sélectionnez un point de départ, la commande est transmise à la fonction **getpoint** suivante.
4. Lorsque vous sélectionnez une extrémité, la valeur du point est enregistrée dans la variable `Endpt`.
5. Le résultat de cette instruction est examiné par l'instruction **if** suivante et la commande est transmise à la fonction **getdist**.

6 La fonction **getdist** agit de manière identique selon que vous sélectionnez un point à l'écran ou entrez une valeur numérique. Le résultat de la fonction **getdist** est enregistré dans la variable HalfWidth.

7 Le déroulement du programme atteint une valeur T fortement imbriquée dans la fonction. Aucune autre fonction ne suit ; cela marque donc la fin de la fonction et la valeur T est renvoyée. C'est cette valeur T qui s'affiche dans la fenêtre de la console.

Dans tous les cas, vous devez renvoyer des valeurs d'une fonction à une autre. L'une des méthodes consiste à créer une liste de valeurs extraites de la fonction **gp:getPointInput**, comme indiqué dans le code suivant :

```
(defun gp:getPointInput ( / StartPt EndPt HalfWidth )
 (if (setq StartPt (getpoint "\nStart point of path: "))
    (if (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
     (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
            (list StartPt EndPt HalfWidth)
            )
    )
 )
)
```

Copiez cette version de la fonction **gp:getPointInput** dans la fenêtre de la console, puis appuyez sur ENTREE. Cette démarche vous permet de tester une autre fonction de la fenêtre de la console.

Pour utiliser la fonction historique de la fenêtre de la console dans le but d'exécuter gp:getPointInput

1 Appuyez sur TAB. La commande historique de la console est appelée ; elle vous permet de parcourir toutes les commandes entrées dans la fenêtre de la console. Si vous êtes trop avancé, appuyez sur SHIFT + TAB pour parcourir l'historique dans le sens inverse.

2 Lorsque **(gp:getPointInput)** s'affiche à l'invite de la Console, appuyez sur ENTREE pour exécuter à nouveau la fonction.

3 Répondez aux messages comme précédemment.

La fonction renvoie une liste contenant deux listes imbriquées ainsi qu'une valeur réelle (à virgule flottante). Le format des valeurs renvoyées est le suivant :

```
((4.46207 4.62318 0.0) (7.66688 4.62318 0.0) 0.509124)
```

Elles correspondent aux variables StartPt, EndPt et HalfWidth.

Utilisation de listes associatives pour grouper des données

Bien que l'exemple précédent permette d'atteindre le résultat souhaité, il existe une méthode plus efficace. Au cours du prochain exercice, vous devrez créer une *liste associative*, dont la forme abrégée en anglais est assoc list (d'après le nom de la fonction LISP relative aux listes associatives). Dans une liste associative, les valeurs auxquelles vous vous intéressez sont associées à des valeurs clés. En voici un exemple :

```
((10 4.46207 4.62318 0.0) (11 7.66688 4.62318 0.0) (40 . 1.018248))
```

Dans cet exemple, les valeurs clés sont 10, 11 et 40. Ces valeurs constituent l'unique index au sein de la liste. AutoCAD utilise cette méthode pour renvoyer les données d'entité à AutoLISP dans le cas où vous accédez à une entité depuis votre programme. La valeur clé 10 indique un point de départ, alors que la valeur 11 indique généralement une extrémité.

Quels sont les avantages d'une liste associative ? Tout d'abord, à la différence de la liste habituelle, l'ordre des valeurs renvoyées n'est pas significatif. Observez à nouveau la première liste :

```
((4.46207 4.62318 0.0) (7.66688 4.62318 0.0) 0.509124)
```

Au vu des valeurs renvoyées, il n'est pas évident d'identifier la sous-liste correspondant au point de départ, ni celle correspondant à l'extrémité. En outre, si vous modifiez la fonction à l'avenir, toute autre fonction reposant sur des données renvoyées dans un ordre spécifique risque d'être affectée.

Lorsque vous utilisez une liste associative, l'ordre des valeurs n'a aucune importance. Si l'ordre d'une liste associative change, vous pouvez toujours déterminer ce à quoi chaque valeur correspond. Par exemple, la valeur 11 correspond toujours à une extrémité, quel que soit son rang dans la liste globale :

```
((11 7.66688 4.62318 0.0)      ; order of list
 (40 . 1.018248)               ; has been
 (10 4.46207 4.62318 0.0))     ; modified
```

Utilisation de listes associatives

Lorsque vous utilisez des listes associatives, vous devez déterminer ce que vos valeurs clés représentent. Dans le cas du sentier de jardin, les valeurs clés 10, 11, 40, 41 et 50 se définiront comme suit :

- 10 représente les coordonnées 3D du point de départ du sentier de jardin.

- 11 représente les coordonnées 3D de l'extrémité du sentier de jardin.
- 40 représente la largeur (et non la demi-largeur) du sentier.
- 41 représente la longueur du sentier, du début à la fin.
- 50 représente le vecteur principal (ou l'angle) du sentier.

Le paragraphe suivant correspond à une version mise à jour de la fonction **gp:getPointInput**. Dans cette fonction, une fonction AutoLISP intitulée **cons** (abréviation de "constituer une liste") crée les sous-listes saisies appartenant à la liste associative. Copiez cette version dans la fenêtre de la console, appuyez sur ENTREE, puis exécutez à nouveau **(gp:getPointInput)** :

```
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if     (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
           ;; if you've made it this far, build the association list
            ;; as documented above.  This will be the return value
            ;; from the function.
            (list
              (cons 10 StartPt)
              (cons 11 EndPt)
              (cons 40 (* HalfWidth 2.0))
              (cons 50 (angle StartPt EndPt))
              (cons 41 (distance StartPt EndPt))
                    )
      )
    )
  )
)
```

Vous remarquerez que, lors de la création de la liste, le programme convertit la demi-largeur spécifiée par l'utilisateur en largeur globale en multipliant sa valeur par 2.

La fenêtre de la console affiche un résultat semblable à celui présenté ci-dessous :

```
_$ (gp:getPointInput)
((10 2.16098 1.60116 0.0) (11 12.7126 7.11963 0.0) (40 . 0.592604)
(50 . 0.481876) (41 . 11.9076))
_$
```

Enregistrement de la valeur renvoyée de la fonction gp:getPointInput dans une variable

Essayons maintenant une autre méthode. Rappelez la fonction en enregistrant la valeur renvoyée dans une variable appelée `gp_PathData`. Pour ce faire, entrez le texte suivant à l'invite de la fenêtre de la console :

```
(setq gp_PathData (gp:getPointInput))
```

Pour afficher la valeur de la variable que vous venez de définir, entrez son nom à l'invite de la fenêtre de la console :

```
_$ gp_PathData
```

VLISP renvoie les données comme suit :

```
((10 2.17742 1.15771 0.0) (11 13.2057 7.00466 0.0) (40 . 1.12747)
(50 . 0.487498) (41 . 12.4824))
```

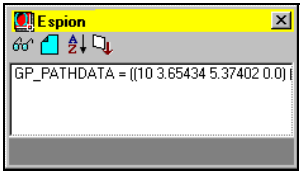
Examen des variables du programme

VLISP dispose d'un ensemble complet d'outils de programmation et de débogage. Un des outils les plus précieux est l'*Espion*. Il vous permet d'examiner les variables de façon plus détaillée que dans la Console VLISP. Vous pouvez également examiner les variables locales dans les fonctions, lors de leur exécution.

Pour examiner la valeur d'une variable

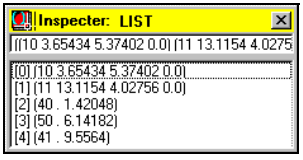
1 Dans la barre de menus VLISP, choisissez Débogage ➤ Ajouter un espion. Une boîte de dialogue intitulée "Ajouter un espion" s'affiche dans VLISP.

Entrez le nom de la variable que vous souhaitez examiner. Pour cet exemple, spécifiez `gp_PathData`,la variable que vous venez de définir à partir de la fenêtre de la console. Une fenêtre Espion s'affiche dans VLISP.



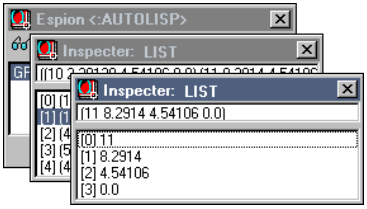
VLISP affiche la valeur de la variable sur une seule ligne dans la fenêtre Espion (fenêtre de base présentée dans l'illustration). Dans ce cas, la valeur de la variable représente une longue liste que vous ne pouvez pas voir en entier. Vous pouvez redimensionner la fenêtre Espion en faisant glisser son contour, mais il existe une meilleure solution.

2 Cliquez deux fois sur le nom de la variable dans la fenêtre Espion. Une fenêtre Inspecter s'ouvre :



Elle indique le type de données de la variable que vous vérifiez (dans le cas présent, une liste) ainsi que la valeur de la variable. En ce qui concerne les listes, Inspecter permet d'afficher chaque élément de la liste sur sa propre ligne.

3 Cliquez deux fois sur la ligne comportant la clé 11 de la liste associative. VLISP ouvre une autre fenêtre Inspecter :



4 Une fois la vérification des variables terminée, fermez toutes les fenêtres Inspecter, mais laissez la fenêtre Espion ouverte.

Réexamen du code de programme

Maintenant que vous savez utiliser les listes associatives dans le code AutoLISP, vous pouvez faire appel à cette méthode pour écrire la version finalisée de la fonction **`gp:getPointInput`**. En utilisant le code suivant, remplacez ou modifiez la version de **`gp:getPointInput`** que vous aviez précédemment enregistrée dans *gpmain.lsp*.

REMARQUE Si vous devez ou souhaitez saisir le code dans *gpmain.lsp* plutôt que le copier à partir d'un autre fichier, vous pouvez gagner du temps en ignorant les commentaires (toutes les lignes commençant par des points-virgules). Mais ne prenez pas l'habitude d'écrire du code sans commentaires !

```
;;;-------------------------------------------------------------;
;;;     Function: gp:getPointInput                              ;
;;;-------------------------------------------------------------;
;;;  Description: This function asks the user to select three    ;
;;;               points in a drawing, which will determine the  ;
;;;               path location, direction, and size.            ;
;;;-------------------------------------------------------------;
;;;  If the user responds to the get functions with valid data,  ;
;;;  use startPt and endPt to determine the position, length,    ;
;;;  and angle at which the path is drawn.                       ;
;;;-------------------------------------------------------------;
;;;  The return value of this function is a list consisting of:  ;
;;;   (10 . Starting Point) ;; List of 3 reals (a point) denoting ;
;;;                         ;;  starting point of garden path.   ;
```

```
;;;   (11 . Ending Point)   ;; List of 3 reals (a point) denoting ;
;;;                         ;;  ending point of garden path.     ;
;;;   (40 . Width)          ;; Real number denoting boundary     ;
;;;                         ;;  width.                           ;
;;;   (41 . Length)         ;; Real number denoting boundary     ;
;;;                         ;;  length.                          ;
;;;   (50 . Path Angle)     ;; Real number denoting the angle    ;
;;;                         ;;  of the path, in radians.         ;
;;;-------------------------------------------------------------;
(defun gp:getPointInput(/ StartPt EndPt HalfWidth)
  (if (setq StartPt (getpoint "\nStart point of path: "))
    (if      (setq EndPt (getpoint StartPt "\nEndpoint of path: "))
      (if (setq HalfWidth (getdist EndPt "\nHalf width of path: "))
         ;; if you've made it this far, build the association list
         ;; as documented above.  This will be the return value
         ;; from the function.
         (list
           (cons 10 StartPt)
           (cons 11 EndPt)
           (cons 40 (* HalfWidth 2.0))
           (cons 50 (angle StartPt EndPt))
           (cons 41 (distance StartPt EndPt))
) ) ) ) )
```

Vous devez ensuite mettre à jour la fonction principale **`C:GPath`** dans *gpmain.lsp*. Modifiez-la afin qu'elle corresponde au code suivant :

```
(defun C:GPath (/ gp_PathData)
  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters.  Continue only if you
  ;; have valid input.  Store the data in gp_PathData.
  (if (setq gp_PathData (gp:getPointInput))
    (if      (gp:getDialogInput)
      (progn
            ;; At this point, you have valid input from the user.
            ;; Draw the outline, storing the resulting polyline
            ;; pointer in the variable called PolylineName.
            (setq PolylineName (gp:drawOutline))
            (princ "\nThe gp:drawOutline function returned <")
            (princ PolylineName)
            (princ ">")
            (Alert "Congratulations - your program is complete!")
       ) ;_ end of progn
       (princ "\nFunction cancelled.")
    ) ;_ end of if
    (princ "\nIncomplete information to draw a boundary.")
  ) ;_ end of if
 (princ)            ; exit quietly
);_ end of defun
```

Si vous copiez et collez le code, ajoutez les commentaires suivants servant d'en-tête précédant `C:Gpath` :

```
;;;*************************************************************;
;;;     Function: C:GPath        The Main Garden Path Function   ;
;;;-------------------------------------------------------------;
;;;  Description: This is the main garden path function. It is a ;
;;;               C: function, meaning that it is turned into an ;
;;;               AutoCAD command called GPATH. This function    ;
;;;               determines the overall flow of the garden path ;
;;;               program.                                       ;
;;;*************************************************************;
;;; The gp_PathData variable is an association list of the form: ;
;;;  (10 . Starting Point) - List of 3 reals (a point) denoting  ;
;;;                           starting point of the garden path. ;
;;;  (11 . Ending Point)   - List of 3 reals (a point) denoting  ;
;;;                           endpoint of the garden path.       ;
;;;  (40 . Width)          - Real number denoting boundary       ;
;;;                           width.                             ;
;;;  (41 . Length)         - Real number denoting boundary       ;
;;;                           length.                            ;
;;;  (50 . Path Angle)     - Real number denoting the angle of   ;
;;;                           the path, in radians.              ;
;;;  (42 . Tile Size)      - Real number denoting the size       ;
;;;                           (radius) of the garden path tiles. ;
;;;  (43 . Tile Offset)    - Spacing of tiles, border to border. ;
;;;  ( 3 . Object Creation Style)                                ;
;;;                        - Object creation style indicates how ;
;;;                          the tiles are to be drawn.  The     ;
;;;                          expected value is a string and one  ;
;;;                          one of three values (string case    ;
;;;                          is unimportant):                    ;
;;;                              "ActiveX"                       ;
;;;                              "Entmake"                       ;
;;;                              "Command"                       ;
;;;  ( 4 . Polyline Border Style)                                ;
;;;                        - Polyline border style determines    ;
;;;                          the polyline type to be used for    ;
;;;                          path boundary.  The expected value  ;
;;;                          one of the following (string case is;
;;;                          unimportant):                       ;
;;;                              "Pline"                         ;
;;;                              "Light"                         ;
;;;*************************************************************;
```

Pour tester les révisions du code

1. Enregistrez le fichier mis à jour.
2. Utilisez la fonction de vérification pour rechercher les éventuelles erreurs de syntaxe.
3. Formatez le code afin qu'il soit plus lisible.
4. Chargez le code ; VLISP redéfinit ainsi les versions antérieures des fonctions.
5. Pour exécuter le programme, entrez **`(c:gpath)`** à l'invite de la console.

   Si le programme ne fonctionne pas correctement, essayez de remédier au problème puis de réexécuter le programme. Répétez cette procédure autant de fois que nécessaire. Si toutes les tentatives échouent, vous pouvez copier le code correct à partir du répertoire *Tutorial\VisualLISP\Lesson2*.

Commentaire du code de programme

VLISP traite une instruction AutoLISP commençant par un point-virgule comme un commentaire. Les deux derniers exemples de code contenaient un grand nombre de commentaires. Dans un programme AutoLISP, un commentaire correspond à quelque chose que vous écrivez pour vous-même, et non pour le programme. Commenter le code est une habitude de programmation que vous pouvez définir pour vous-même. A quoi servent les commentaires ?

- A vous fournir une explication sur le code lorsque vous éditez un programme que vous n'avez pas utilisé depuis neuf mois et auquel vous devez ajouter des fonctions que les utilisateurs vous ont demandées. Avec le temps, la séquence de fonctions la plus évidente peut facilement se transformer en une succession de parenthèses non identifiable.
- A expliquer le code à d'autres personnes chargées de mettre à jour le programme. Il est souvent très pénible de lire un code écrit par quelqu'un d'autre, en particulier lorsque ce code contient très peu de commentaires.

VLISP comporte des outils pour vous aider lorsque vous ajoutez des commentaires à votre code. Vous remarquerez que, dans les exemples, certains commentaires commencent par trois points-virgules (;;;), parfois deux (;;), voire un seul (;). Reportez-vous à la section “Applying Visual LISP Comment Styles” du *Visual LISP Developer's Guide* pour savoir comment VLISP traite les différents commentaires.

Afin d'économiser de l'espace et d'éviter un trop grand nombre d'arborescences, les exemples de code qu'il nous reste à aborder dans ce didacticiel n'incluent pas tous les commentaires figurant dans les exemples de fichiers source. Nous supposons que vous avez déjà mesuré l'intérêt des commentaires explicites et que vous en ferez l'usage sans que nous n'ayons besoin de vous le rappeler.

Définition d'un point d'arrêt et utilisation d'autres espions

Un point d'arrêt est un symbole (point) que vous placez dans le code source pour indiquer l'emplacement où vous souhaitez que l'exécution du programme s'interrompe. Lorsque vous exécutez votre code, VLISP opère normalement jusqu'à ce qu'il rencontre un point d'arrêt. Au niveau de ce point, VLISP interrompt l'exécution et attend vos instructions. Le programme n'est pas définitivement arrêté ; le mode Animation est simplement en pause.

Pendant la suspension de votre programme, vous pouvez :

- Parcourir votre code fonction par fonction ou expression par expression.
- Reprendre l'exécution normale de votre programme à partir de n'importe quel point.
- Modifier la valeur des variables de façon dynamique, puis changer les résultats du programme en cours d'exécution.
- Ajouter des variables à la fenêtre Espion.

Utilisation de la barre d'outils Débogage

La barre d'outils Débogage comporte plusieurs outils que vous utiliserez tout au long de cette section. Par défaut, cette barre d'outils est associée aux barres d'outils Vue et Outils, et apparaît sous la forme d'une simple barre d'outils VLISP :

La barre d'outils Débogage est matérialisée par le groupe d'icônes situé à l'extrême gauche de la fenêtre. La plupart des options de cette barre d'outils restent inactives jusqu'à ce que vous exécutiez votre programme en mode débogage (c'est-à-dire avec un ou plusieurs points d'arrêt définis).

Si vous ne l'avez pas encore fait, déplacez la barre d'outils Débogage de son emplacement en haut de l'écran. Pour ce faire, saisissez la barre d'outils, puis faites-la glisser à l'aide des deux poignées verticales situées sur sa gauche. Vous pouvez déplacer toutes les barres d'outils VLISP et les positionner à l'emplacement le plus approprié à votre méthode de travail.

La barre d'outils Débogage se compose de trois groupes de boutons principaux comportant chacun trois boutons. Lorsque vous exécutez un programme en mode débogage, la barre d'outils prend l'aspect suivant :

- Les trois premiers boutons vous permettent de parcourir votre code de programme.
- Le groupe de boutons suivant définit comment VLISP doit procéder lorsqu'il s'arrête au niveau d'un point d'arrêt ou d'une erreur.
- Le troisième groupe de trois boutons permet de définir ou de supprimer un point d'arrêt, d'ajouter un espion et de vous positionner dans votre code source à l'emplacement du dernier point d'arrêt.

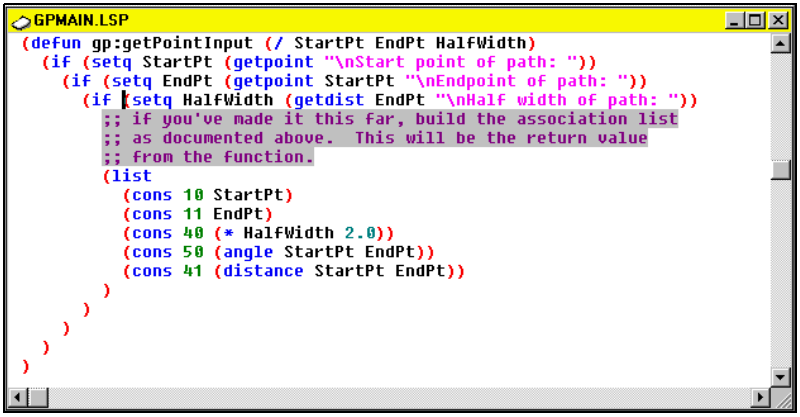
Enfin, le dernier bouton de la barre d'outils Débogage permet d'activer un indicateur d'étapes. Il n'exécute aucune fonction, mais donne une indication visuelle de l'emplacement de votre curseur lorsque vous parcourez le code. Lorsque le mode débogage n'est pas activé, ce bouton est estompé.

Pour définir un point d'arrêt

1 Dans la fenêtre de l'éditeur VLISP contenant *gpmain.lsp*, placez votre curseur devant la prenthèse ouvrante de la fonction **`setq`** se trouvant sur la ligne de code suivante, au sein de la fonction **`gp:getPointInput`** :

```
(setq HalfWidth (getdist EndPt "\nHalf width of path: "))
```

2 Cliquez une fois. La position est indiquée sur la capture d'écran suivante :



3 Une fois le point d'insertion du texte défini, cliquez sur le bouton Basculer le point d'arrêt de la barre d'outils Débogage.

Le bouton Basculer le point d'arrêt agit comme une bascule, que vous pouvez activer ou désactiver. Si aucun point d'arrêt ne se trouve à l'emplacement du curseur, il en définit un ; si le point d'arrêt existe déjà à cet emplacement, il le supprime.

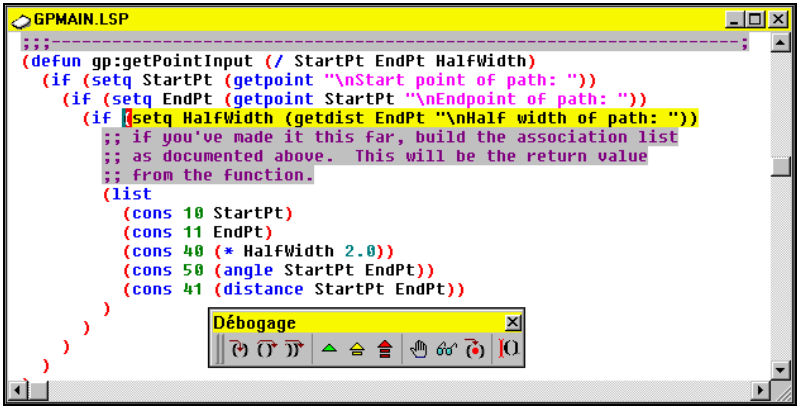
4 Pour charger le fichier, cliquez sur le bouton Charger la fenêtre d'édition active de la barre d'outils Outils.

5 Exécutez la fonction **(C:Gpath)** à l'invite de la Console VLISP.

VLISP exécute le programme normalement jusqu'au point d'arrêt. Dans ce cas, vous devez sélectionner les deux premiers points, à savoir le point de départ et l'extrémité du sentier.

6 Indiquez le point de départ et l'extrémité lorsque le programme vous le demande.

Une fois ces points définis, VLISP suspend l'exécution du programme et réactive la fenêtre de l'éditeur de texte, en mettant en surbrillance la ligne de code correspondant à l'emplacement du point d'arrêt :



Deux remarques s'imposent :

- Le curseur est placé sur le point d'arrêt. Il peut être difficile de le repérer, c'est pourquoi VLISP propose un autre indice.
- Dans la barre d'outils Débogage, l'icône de l'indicateur d'étapes affiche un curseur en I rouge devant une paire de parenthèses. Il indique que le débogueur VLISP est arrêté avant l'expression.

Parcourir le code

Vous pouvez parcourir votre code à l'aide de trois boutons. Ces boutons correspondent aux trois icônes situées à l'extrême gauche de la barre d'outils Débogage. Par ordre d'apparition, ils permettent d'accomplir les tâches suivantes :

- Entrer dans l'expression mise en surbrillance
- Accéder à la fin de l'expression mise en surbrillance
- Accéder à la fin de la fonction où vous êtes actuellement arrêté

Avant de sélectionner un élément, vérifiez à nouveau l'état du code mis en surbrillance, la position du curseur et le bouton d'indicateur d'étapes. En résumé, une expression représentant une fonction **`getdist`** imbriquée dans une fonction **`setq`** est mise en surbrillance et le curseur est placé au début du bloc sélectionné.

Pour parcourir le code à partir du point d'arrêt

1 Cliquez sur le bouton Pas à pas principal.

Lorsque vous cliquez sur le bouton Pas à pas principal, le contrôle est rendu à AutoCAD et vous êtes invité à indiquer la largeur du sentier.

2 Répondez au message.

Une fois la largeur définie, le contrôle revient de nouveau à VLISP. Repérez la position de votre curseur et observez le bouton d'indicateur d'étapes.

VLISP procède à une évaluation de la totalité de l'expression mise en surbrillance puis s'arrête à la fin de ladite expression.

3 Cliquez à nouveau sur le bouton Pas à pas principal. VLISP passe au début du bloc de code suivant et le met entièrement en surbrillance

4 Cliquez sur le bouton Pas à pas détaillé (et non Pas à pas principal).

REMARQUE Si, au cours de cet exercice, vous sélectionnez un élément incorrect et sautez une ou deux étapes, vous pouvez le recommencer très facilement. Tout d'abord, cliquez sur le bouton Réinitialiser de la barre d'outils Débogage. L'exécution de tous les codes VLISP se termine, et le système VLISP se réinitialise et vous renvoie en haut de la console. Ensuite, recommencez à partir de l'étape 1.

La première fonction **`cons`** est mise en surbrillance et VLISP s'arrête juste devant (observez le bouton d'indicateur d'étapes).

Examen des variables tout au long de l'inspection d'un programme

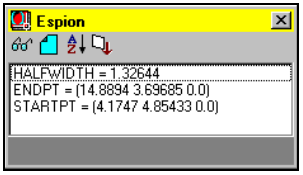
Tout en parcourant votre programme, vous pouvez ajouter des variables à la fenêtre Espion et modifier leurs valeurs.

Si la fenêtre Espion n'apparaît pas à l'écran, cliquez sur le bouton correspondant dans la barre d'outils Vue afin de l'activer.

Si votre fenêtre Espion contient toujours la variable `gp_PathData`, cliquez sur le bouton Effacer fenêtre situé en haut de la fenêtre.

Pour ajouter des variables à la fenêtre Espion

1. Cliquez deux fois sur n'importe quelle occurrence de `StartPt` dans la fenêtre de l'éditeur de texte VLISP. Il s'agit du nom de la variable dont vous recherchez la valeur.
2. Cliquez sur le bouton Ajouter un espion dans la fenêtre Espion, ou cliquez à l'aide du bouton droit de la souris, puis choisissez l'option Ajouter un espion.
3. Répétez cette procédure pour les variables `EndPt` et `HalfWidth`. Votre fenêtre Espion doit se présenter comme suit :



Si vous déboguez un programme qui ne fonctionne pas correctement, utilisez les points d'arrêt associés à des espions pour vous assurer que vos variables contiennent les valeurs souhaitées.

Si une variable ne comporte pas la valeur souhaitée, vous pouvez changer cette valeur et voir l'incidence qu'elle a sur le programme. Par exemple, indiquez que la valeur `halfwidth` doit être un nombre entier. Etant donné que vous ne prêtiez pas attention à la sélection de points lors de la définition d'entrées, vous avez obtenu une valeur approximative de type 1.94818.

Pour changer la valeur d'une variable lorsque le programme est en cours d'exécution

1 Entrez le code suivant à l'invite de la console :

```
(setq halfwidth 2.0)
```

Vous remarquerez que la valeur dans la fenêtre Espion change. Mais comment pouvez-vous vous assurer que la valeur sera utilisée lorsque la sous-liste de largeur (40 . width) sera créée dans la liste associative ? Ajoutez une ou plusieurs expressions à la fenêtre Espion pour tester cette fonction.

2 Dans la barre de menus VLISP, choisissez Débogage ➤ Consulter dernière interprétation.

Une variable appelée **`*LastValue*`** est ajoutée à votre fenêtre Espion. **`*Last-Value*`** est une variable globale dans laquelle VLISP enregistre automatiquement la valeur de la dernière expression évaluée.

3 Parcourez le programme (en cliquant sur le bouton Pas à pas détaillé ou sur le bouton Pas à pas principal) jusqu'à ce que l'expression chargée de la création de la sous-liste `width` soit évaluée. Le code correspondant à cette action est le suivant :

```
(cons 40 (* HalfWidth 2.0))
```

Si vous avez remplacé la valeur **`HalfWidth`** comme indiqué, l'évaluation de cette expression doit renvoyer (40 . 4.0) dans la fenêtre Espion.

Sortir de la fonction gp:getPointInput et entrer dans la fonction C:Gpmain

Il nous reste un point à aborder : qu'advient-il de la valeur des variables locales dans la fonction **`gp:getPointInput`** une fois que vous l'avez quittée ?

Pour sortir de gp:getPointInput et rendre le contrôle à c:gpath

1 Cliquez sur le bouton Pas à pas sortant.

VLISP accède alors à la fin de la fonction **`gp:getPointInput`** et s'arrête avant d'en sortir.

2 Cliquez sur le bouton Pas à pas détaillé.

Le contrôle est alors rendu à **`c:gpmain`**, la fonction qui a appelé **`gp:getPointInput`**.

Examinez les valeurs des variables dans la fenêtre Espion. Etant donné que ce sont des variables locales par rapport à la fonction **`gp:getPointInput`**, les variables `endpt` et `StartPt` sont égales à `nil`. VLISP récupère automatiquement l'espace mémoire occupé par ces variables. Normalement, la troisième variable de fonction locale **`HalfWidth`** contient également une valeur `nil`, mais en raison de l'opération de débogage, cette variable a été entièrement remplacée dans la fenêtre de la console et possède toujours la valeur 2.0 dans la fenêtre Espion. La variable globale `*Last-Value*` s'affiche également dans la liste associative créée par la fonction **`gp:getPointInput`**.

Votre première session de débogage est terminée. Toutefois, n'oubliez pas que le mode Animation de votre programme est toujours en pause.

Pour terminer cette leçon

1. Cliquez sur le bouton Continuer de la barre d'outils Débogage. Répondez aux messages. Le programme se termine.
2. Dans la barre de menus VLISP, choisissez Débogage ➤ Effacer tous les points d'arrêt. Répondez au message par oui. Tous les points d'arrêt insérés dans votre code sont ainsi supprimés.

   N'oubliez pas : vous pouvez supprimer les points d'arrêt un par un en plaçant le curseur au niveau du point d'arrêt et en cliquant sur le bouton Basculer le point d'arrêt.

Résumé de la leçon 2

Au cours de cette leçon, vous avez

- Découvert les variables locales et les variables globales.
- Défini et supprimé des points d'arrêt dans un programme.
- Parcouru un programme en cours d'exécution.
- Examiné et changé de façon dynamique la valeur des variables d'un programme en cours d'exécution.
- Vu comment les variables locales sont rétablies à la valeur `nil` une fois que l'exécution de la fonction qui les définissait se termine.

Les outils que vous avez appris à utiliser au cours de cette leçon s'intégreront à votre environnement de travail quotidien si vous envisagez de développer des applications AutoLISP à l'aide de VLISP.

Dessin du contour du sentier

3

Au cours de cette leçon, vous allez développer les capacités de votre programme afin qu'il dessine un élément dans AutoCAD : le contour de la polyligne du sentier de jardin. Pour dessiner le contour du sentier, vous devez créer certaines fonctions utilitaires qui ne sont pas spécifiques à une application mais bien de nature générale, et qui peuvent être recyclées pour la suite. Vous apprendrez également à écrire des fonctions acceptant des arguments (données transmises à la fonction depuis l'extérieur) et saurez pourquoi l'utilisation d'arguments constitue un concept de programmation puissant. A la fin de cette leçon, vous saurez dessiner une forme AutoCAD paramétrique, c'est-à-dire dessiner de façon dynamique une forme en fonction des paramètres de données spécifiques fournis par l'utilisateur.

Dans ce chapitre

Planification des fonctions utilitaires réutilisables

Les fonctions utilitaires effectuent des tâches communes aux nombreuses applications que vous allez concevoir. Ces fonctions constituent une boîte à outils dont vous pourrez vous servir très souvent.

Lorsque vous créez une fonction faisant partie d'une boîte à outils, prenez le temps de la documenter précisément. Dans vos commentaires, si vous en avez le temps, notez également les caractéristiques que vous souhaiteriez ajouter à votre fonction à l'avenir.

Conversion des degrés en radians

Vous allez à présent créer une fonction qui vous permettra d'éviter de resaisir une équation. Elle se présente comme suit :

```
(defun Degrees->Radians (numberOfDegrees)
(* pi (/ numberOfDegrees 180.0)))
```

Cette fonction est appelée **`Degrees->Radians`**. Le nom indique son champ d'utilisation.

Pourquoi est-il nécessaire de définir une fonction pour convertir des mesures angulaires ? En arrière-plan, AutoCAD utilise les radians pour assurer le suivi des mesures angulaires alors que la plupart des utilisateurs raisonnent en degrés. Cette fonction, que vous trouverez dans votre boîte à outils, vous permet d'exprimer les angles en degrés et laisse AutoLISP convertir les mesures en radians.

Pour tester la fonction utilitaire

1 Entrez le code suivant à l'invite de la Console VLISP :

```
(defun Degrees->Radians (numberOfDegrees)
(* pi (/ numberOfDegrees 180.0)))
```

2 Entrez le code suivant à l'invite de la Console VLISP :

```
(degrees->radians 180)
```

La fonction renvoie le nombre 3.14159. Selon le fonctionnement de cette fonction, 180 degrés équivalent à 3.14159 radians. (Ce nombre évoque-t-il quelque chose ? Si oui, traitez en nombre de pi.)

Pour utiliser cette fonction au sein de votre programme, il vous suffit de copier sa définition à partir de la fenêtre de la console dans votre fichier *gpmain.lsp*. Vous pouvez la coller à n'importe quel endroit de votre fichier, sauf dans une fonction existante.

Pour mettre au propre votre travail, sélectionnez le texte que vous venez d'insérer, puis cliquez sur le bouton Formater la sélection ; VLISP indentera et formatera correctement le code.

Ensuite, ajoutez des commentaires décrivant la fonction. Une fois que vous aurez entièrement documenté la fonction, votre code devrait se présenter comme suit :

```
;;;-------------------------------------------------------------;
;;;     Function: Degrees->Radians                              ;
;;;-------------------------------------------------------------;
;;; Description: This function converts a number representing an ;
;;;              angular measurement in degrees, into its radian ;
;;;              equivalent. There is no error checking on the   ;
;;;              numberOfDegrees parameter -- it is always       ;
;;;              expected to be a valid number.                  ;
;;;-------------------------------------------------------------;
(defun Degrees->Radians (numberOfDegrees)
  (* pi (/ numberOfDegrees 180.0))
)
```

Conversion de points 3D en points 2D

Une des fonctions utiles figurant dans le programme du sentier de jardin permet de convertir des points 3D en points 2D. AutoCAD fonctionne généralement avec des coordonnées 3D, mais certaines entités comme les polylignes fines sont toujours en mode 2D. Les points que la fonction **`getpoint`** renvoie sont des points 3D ; vous devez donc créer une fonction pour les convertir.

Pour convertir un point 3D en un point 2D

1. Entrez le code suivant à l'invite de la fenêtre de la console :

   ```
   (defun 3dPoint->2dPoint (3dpt)(list (car 3dpt) (cadr 3dpt)))
   ```

2. Testez la fonction en entrant le code suivant à l'invite de la console :

   ```
   (3dpoint->2dpoint (list 10 20 0))
   ```

   Cette méthode fonctionne, mais il convient d'ajouter une autre remarque concernant l'application du sentier de jardin. Bien que la qualité de nombre entier ou de nombre réel importe peu dans les fonctions LISP, il n'en est pas de même pour les fonctions ActiveX que vous utiliserez plus loin dans cette leçon. Les fonctions ActiveX requièrent des nombres réels. Il est facile de modifier la fonction afin qu'elle renvoie des nombres réels plutôt que des nombres entiers.

3. Entrez le code suivant à l'invite de la console :

   ```
   (defun 3dPoint->2dPoint (3dpt)(list (float(car 3dpt))
   (float(cadr 3dpt))))
   ```

4 Exécutez à nouveau la fonction :

```
(3dpoint->2dpoint (list 10 20 0))
```

Vous remarquerez que les valeurs renvoyées sont désormais des nombres réels (identifiables par les valeurs décimales).

5 Testez à nouveau la fonction en utilisant la fonction **getpoint**. Entrez le code suivant à l'invite de la console :

```
(setq myPoint(getpoint))
```

6 Sélectionnez un point dans la fenêtre graphique d'AutoCAD.

La fonction **getpoint** renvoie un point 3D.

7 Entrez le code suivant à l'invite de la console :

```
(3dPoint->2Dpoint myPoint)
```

Notez le point 2D renvoyé.

Ajoutez à présent la fonction au fichier *gpmain.lsp*, en procédant de la même manière qu'avec les **degrés->radians**. Le nouveau code devrait se présenter comme suit :

```
;;;-------------------------------------------------------------;
;;; Function: 3dPoint->2dPoint                                  ;
;;;-------------------------------------------------------------;
;;; Description: This function takes one parameter representing a;
;;;              3D point (list of three integers or reals), and ;
;;;              converts it into a 2D point (list of two reals).;
;;;              There is no error checking on the 3D point     ;
;;;              parameter -- it is assumed to be a valid point. ;
;;;-------------------------------------------------------------;
;;; To do: Add some kind of parameter checking so that this     ;
;;;        function won't crash a program if it is passed a     ;
;;;        null value, or some other kind of data type than a   ;
;;;        3D point.                                            ;
;;;-------------------------------------------------------------;
(defun 3dPoint->2dPoint (3dpt)
  (list (float(car 3dpt)) (float(cadr 3dpt)))
)
```

Notez que l'en-tête de la fonction comprend un commentaire concernant certaines tâches que vous devrez entreprendre à l'avenir sur cette fonction. Si vous souhaitez augmenter les performances de votre système, réfléchissez à la manière de fiabiliser cette fonction afin d'éviter que des données incorrectes provoquent des pannes.

Conseil : fonctions **numberp** et **listp**...

```
(listp '(1 1 0)) => T
(numberp 3.4) => T
```

Dessin d'entités avec AutoCAD

La plupart des programmes AutoLISP dessinent des entités en suivant une des méthodes suivantes :

- Fonctions ActiveX
- fonction **entmake**
- fonction **command**

Cette leçon est consacrée à la création d'une entité à l'aide d'ActiveX. Au cours de la leçon 5, vous mettrez en pratique les fonctions alternatives **entmake** et command d'AutoCAD.

Création d'entités à l'aide des fonctions ActiveX

Le mode de création d'entités le plus récent est l'utilisation des fonctions ActiveX dans VLISP. ActiveX présente plusieurs avantages par rapport à **entmake** et à **command**.

- Les fonctions ActiveX sont plus rapides.
- Les noms de fonctions ActiveX indiquent l'action qu'elles exécutent, ce qui améliore leur lisibilité, leur maintenance et leur débogage.

Vous trouverez plus loin dans cette leçon un exemple de fonction ActiveX.

Utilisation d'entmake pour la construction d'entités

La fonction **entmake** permet de construire une entité en rassemblant des valeurs pour des éléments tels que des emplacements et des orientations de coordonnées, des calques et des couleurs à l'intérieur d'une liste associative, pour ensuite demander à AutoCAD de construire l'entité à votre place. La liste associative que vous créez pour la fonction **entmake** ressemble beaucoup à celle que vous obtenez lorsque vous appelez la fonction **entget**. La différence est que **entget** renvoie des informations concernant une entité, alors que **entmake** construit une nouvelle entité à partir de nouvelles données brutes.

Utilisation de la ligne de commande d'AutoCAD

Lorsqu'AutoLISP est apparu pour la première fois dans AutoCAD, le seul moyen de créer une entité était d'utiliser la fonction **command**. Elle permet à un programmeur AutoLISP de coder à peu près n'importe quelle commande exécutable à partir de la ligne de commande d'AutoCAD. Cette méthode est fiable, mais plus lente que les méthodes ActiveX, et elle ne possède pas la souplesse d'**entmake**.

Activation de la fonction permettant de dessiner les contours du dessin

A l'issue de la dernière leçon, la fonction **gp:drawOutline** avait l'aspect suivant :

```
;;;-------------------------------------------------------------;
;;;     Function: gp:drawOutline                                ;
;;;-------------------------------------------------------------;
;;; Description: This function draws the outline of the         ;
;;;              garden path.                                   ;
;;;-------------------------------------------------------------;
(defun gp:drawOutline ()
  (alert
    (strcat "This function will draw the outline of the polyline "
      "\nand return a polyline entity name/pointer."
    )
  )
  ;; For now, simply return a quoted symbol.  Eventually, this
  ;; function will return an entity name or pointer.
  'SomeEname
)
```

Tel qu'il existe, ce code n'est pas très utile. Toutefois, la liste associative stockée dans la variable `gp_PathData`, vous donne assez d'informations pour calculer les points de contour du chemin. Vous devez à présent définir comment transmettre les informations de cette variable à **gp:drawOutline**.

Rappelons que `gp_PathData` est une variable locale définie dans la fonction **C:GPath**. Dans AutoLISP, les variables locales déclarées dans une fonction sont visibles par toutes les fonctions appelées à partir de celle-ci (reportez-vous à "Différenciation entre les variables locales et les variables globales" à la page 16 pour plus d'informations à ce sujet).[Actually, that section does not clarify this; need to clarify!] La fonction **gp:drawOutline** est appelée à partir de **C:GPath**. Vous pouvez faire référence à la variable `gp-PathData` de **gp:drawOutline**, mais ce n'est pas une méthode de programmation recommandée.

Pourquoi ? Lorsque les deux fonctions utilisant la même variable sont définies dans le même fichier, comme dans les exemples repris jusqu'ici, il n'est pas très difficile de savoir où la variable est définie et de connaître son champ d'utilisation. Toutefois, si les fonctions sont définies dans des fichiers différents, comme c'est souvent le cas, il faut effectuer une recherche dans les deux fichiers pour savoir ce que `gp_PathData` représente.

Transmission de paramètres aux fonctions

Une méthode plus efficiente de transmission d'informations d'une fonction à une autre consiste à transmettre des paramètres à la fonction appelée. La fonction doit être conçue de manière à attendre la transmission d'un certain nombre de valeurs. Vous souvenez-vous de la fonction **`Degrees->Radians`** ? Cette fonction reçoit un paramètre nommé `numberOfDegrees`:

```
(defun Degrees->Radians (numberOfDegrees)
(* pi (/ numberOfDegrees 180.0))
```

Lorsque vous appelez la fonction, vous êtes supposé lui transmettre un nombre. Le nombre compris dans **`Degrees->Radians`** est déclaré comme le paramètre nommé `numberOfDegrees`. Par exemple :

```
_$ (degrees->radians 90)
1.5708
```

Dans ce cas, le nombre 90 est attribué au paramètre `numberOfDegrees`.

Vous pouvez aussi transmettre une variable à une fonction. Par exemple, vous pourriez avoir une variable appelée `aDegreeValue` qui contient le nombre 90. Les commandes suivantes définissent `aDegreeValue` et transmettent la variable à **`Degrees->Radians`**:

```
_$ (setq aDegreeValue 90)
90
_$ (degrees->radians aDegreeValue)
1.5708
```

Utilisation d'une liste associative

Vous pouvez transmettre la liste associative de la variable `gp_PathData` à la fonction **`gp:drawOutline`** en appelant cette fonction de la manière suivante :

```
(gp:drawOutline gp_PathData)
```

C'est simple, mais il vous faut encore trouver le moyen de traiter les informations stockées dans la liste associative. La fonction Inspecter de VLISP peut vous aider à déterminer la marche à suivre.

Pour utiliser la fonction Inspecter de VLISP dans le but d'analyser la liste associative

1. Chargez le texte contenu dans la fenêtre de l'éditeur de texte.
2. Entrez l'expression suivante à l'invite de la console :

   ```
   (setq BoundaryData (gp:getPointInput))
   ```

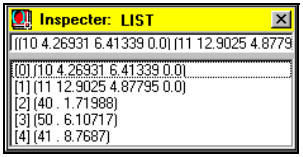
   VLISP stocke les informations fournies dans une variable nommée `BoundaryData`.

3 En réponse aux invites, entrez le point de départ, l'extrémité et la demi-largeur.

4 Sélectionnez le nom de la variable `BoundaryData` dans la fenêtre de la console en cliquant deux fois dessus.

5 Dans la barre de menus VLISP, choisissez Vue ➤ Inspecter.

VLISP affiche une fenêtre de ce type :



La fenêtre Inspecter affiche toutes les sous-listes contenues dans la variable `BoundaryData`.

6 Entrez le code suivant à l'invite de la Console VLISP :

```
(assoc 50 BoundaryData)
```

La fonction **`assoc`** renvoie l'entrée de la liste associative identifiée par la clé spécifiée. Dans cet exemple, la clé spécifiée est 50 ; elle est associée à l'angle du sentier de jardin (voir "Utilisation de listes associatives" à la page 20 pour une liste de paires de valeurs clés définies pour cette application).

7 Entrez le code suivant à l'invite de la Console VLISP :

```
(cdr(assoc 50 BoundaryData))
```

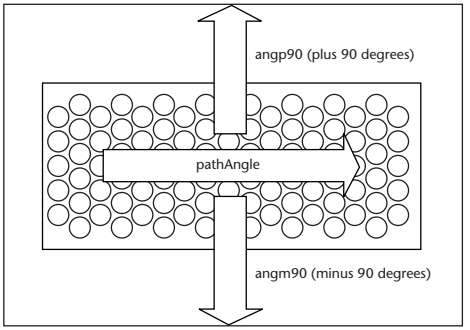
La fonction **`cdr`** renvoie le second élément et tous les éléments restants d'une liste. Dans notre exemple, **`cdr`** extrait la valeur angulaire, qui représente le second et dernier élément de l'entrée renvoyé par la fonction **`assoc`**.

A ce stade, vous ne devriez pas éprouver de difficulté à comprendre le fragment de code suivant :

```
(setq PathAngle  (cdr (assoc 50 BoundaryData))
      Width      (cdr (assoc 40 BoundaryData))
      HalfWidth  (/ Width 2.00)
      StartPt    (cdr (assoc 10 BoundaryData))
      PathLength (cdr (assoc 41 BoundaryData))
```

Utilisation des angles et définition des points

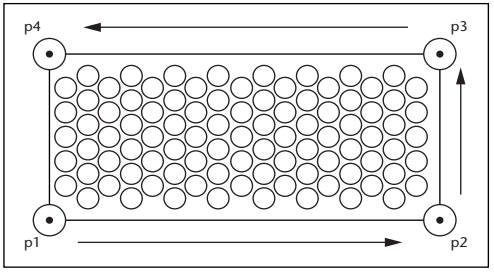
Quelques problèmes subsistent. En premier lieu, vous devez trouver la façon de dessiner le sentier sous l'angle spécifié par l'utilisateur. A partir de la fonction **`gp:getPointInput`**, vous pouvez aisément déterminer l'angle principal du sentier. Pour le dessiner, il vous faut deux vecteurs supplémentaires orientés perpendiculairement à l'angle principal.



C'est ici que la fonction **`Degrees->Radians`** s'avère utile. Le fragment de code suivant indique la procédure de définition des deux vecteurs perpendiculaires à l'aide de la variable `PathAngle` transmise comme argument à la fonction **`Degrees->Radians`** :

```
(setq angp90 (+ PathAngle (Degrees->Radians 90))
      angm90 (- PathAngle (Degrees->Radians 90)))
```

Les données dont vous disposez vous permettent de définir les quatre coins du sentier à l'aide de la fonction **`polar`** :




```
(setq p1 (polar StartPt angm90 HalfWidth)
      p2 (polar p1 PathAngle PathLength)
      p3 (polar p2 angp90 Width)
      p4 (polar p3 (+ PathAngle (Degrees->Radians 180))
```

La fonction **`polar`** renvoie un point 3D à une certaine distance et sous un certain angle par rapport à un point donné. Par exemple, **`polar`** situe le point `p2` en projetant `p1`, une distance de `PathLength` sur un vecteur orienté selon un angle de `PathAngle`, calculé dans le sens trigonométrique à partir de l'axe des X.

Compréhension du code ActiveX dans gp:drawOutline

La fonction **gp:drawOutline** émet des appels ActiveX pour afficher le contour du sentier (matérialisé par une polyligne) dans AutoCAD. Le fragment de code suivant utilise ActiveX pour dessiner le contour.

```
;; Add polyline to the model space using ActiveX automation.
(setq pline   (vla-addLightweightPolyline
    *ModelSpace*; Global Definition for Model Space
    VLADataPts; vertices of path boundary
    ) ;_ end of vla-addLightweightPolyline

  ) ;_ end of setq

(vla-put-closed pline T)
```

Comment interprétez-vous ce code ? Vous trouverez des informations essentielles dans la section *ActiveX and VBA Reference,* où sont décrites les méthodes et les propriétés auxquelles ont accès les clients ActiveX, telle que cette application "sentier de jardin". Le chapitre “Working with ActiveX” du *Visual LISP Developer’s Guide* explique comment traduire la syntaxe VBA™ de la section *ActiveX and VBA Reference* en appels ActiveX en syntaxe AutoLISP.

Pour le moment, toutefois, vous pouvez comprendre le code dans ses grandes lignes en examinant le modèle des deux appels **vla-** de l'exemple précédent. Les noms de toutes les fonctions AutoLISP ActiveX qui agissent avec les objets AutoCAD comportent le préfixe **vla-**. Par exemple, addLightweightPolyline est le nom d'une méthode ActiveX, tandis que **vla-addLightweightPolyline** est la fonction AutoLISP qui appelle cette méthode. L'appel **vla-put-closed** met à jour la propriété closed de l'objet `pline`, la polyligne dessinée par **vla-addLightweightPolyline**.

Les objets Automation qui régissent les appels AutoLISP ActiveX sont soumis à quelques règles standard :

- Le premier argument de l'appel d'une méthode **vla-put**, **vla-get,** ou **vla-** est l'objet modifié ou recherché, soit par exemple `*ModelSpace*` dans le premier appel de fonction et `pline` dans le second appel.
- La valeur renvoyée par l'appel d'une méthode **vla-** est un *objet VLA* qui peut être utilisé dans les appels suivants. Par exemple, **vla-addLightweightPolyline** génère en retour un objet, `pline`, qui est modifié dans l'appel ActiveX suivant.
- Le modèle d'objet ActiveX est structuré de façon hiérarchique. Les objets évoluent depuis l'objet d'application au niveau le plus élevé jusqu'au primitives de dessins individuelles telles que les objets polyligne ou cercle. Dès lors, la fonction **gp:drawOutline** n'est pas encore complète parce que l'accès à l'objet automation `*ModelSpace*` doit se faire en premier lieu par l'objet de l'application racine.

Vérification du chargement d'ActiveX

La fonctionnalité ActiveX n'est pas automatiquement activée lorsque vous démarrez AutoCAD ou VLISP ; vos programmes doivent donc s'assurer du chargement d'ActiveX. C'est ce que réalise l'appel de fonction suivant :

```
(vl-load-com)
```

Si le support d'ActiveX n'est pas encore disponible, l'exécution de **`vl-load-com`** initialise l'environnement AutoLISP ActiveX. Si ActiveX est déjà chargé, **`vl-load-com`** est sans effet.

Obtention d'un pointeur vers l'espace objet

Lorsque vous ajoutez des entités par l'intermédiaire des fonctions ActiveX, vous devez identifier l'espace objet ou l'espace papier dans lequel l'entité doit être insérée. (Dans la terminologie ActiveX, les entités sont des *objets*, mais nous continuerons d'utiliser le terme entité dans ce didacticiel.) Pour indiquer à AutoCAD l'espace que les nouvelles entités doivent occuper, vous devez obtenir un pointeur vers l'espace concerné. Malheureusement, l'obtention d'un pointeur vers l'espace objet n'est pas une fonction simple et directe. Le fragment de code suivant indique comment l'opération doit être configurée :

```
(vla-get-ModelSpace (vla-get-ActiveDocument
  (vlax-get-Acad-Object)))
```

Agissant de l'intérieur vers l'extérieur, la fonction **`vlax-get-Acad-Object`** extrait un pointeur vers AutoCAD. Ce pointeur est transmis à la fonction **`vla-get-ActiveDocument`**, qui extrait un pointeur vers le dessin actif (document) dans AutoCAD. Le pointeur du document actif est ensuite transmis à la fonction **`vla-get-ModelSpace`**, qui extrait un pointeur vers l'espace objet du dessin courant.

Ce n'est pas le genre d'expression que vous êtes prêt à resaisir sans cesse. Voyez par exemple combien le code servant à ajouter une polyligne à l'aide d'ActiveX semble compliqué lorsque l'intégralité de l'expression de l'espace objet est utilisée :

```
(setq pline (vla-addLightweightPolyline
               (vla-get-ModelSpace
                 (vla-get-ActiveDocument
                    (vlax-get-Acad-Object)
                 )
               )
               VLADataPts)
)
(vla-put-closed pline T)
```

La fonction est nettement moins compréhensible. En outre, vous devez répéter le même jeu de fonctions imbriquées dans toutes les expressions dans lesquelles une entité est créée. Ceci est un bon exemple des quelques utilisations intéressantes des variables globales. L'application du sentier de jardin peut ajouter un grand nombre d'entités à l'espace objet (pensez à toutes les dalles du sentier). Par conséquent, vous devez définir une variable globale pour stocker le pointeur dans l'espace objet, comme dans le code suivant :

```
(setq *ModelSpace* (vla-get-ModelSpace (vla-get-ActiveDocument
                  (vlax-get-Acad-Object))))
```

Vous pouvez utiliser la variable `*ModelSpace*` chaque fois que vous appelez une fonction de création d'entités ActiveX. Le seul inconvénient de ce schéma est que la variable `*ModelSpace*` doit être prête à fonctionner avant que vous ne commenciez à dessiner. Pour cette raison, la fonction **`setq`** établissant la variable sera appelée lors du chargement de l'application, immédiatement après l'appel de **`vl-load-com`**. Ces appels seront lancés avant le placement d'une instruction **`defun`** dans le fichier de programme. Par conséquent, ils seront exécutés dès le chargement du fichier.

Construction d'un réseau de points de polyligne

Le dernier problème à régler est celui de la conversion des variables de points individuelles - `p1`, `p2`, `p3` et `p4` au format requis pour la fonction **`vla-addLightweightpolyline`**. En premier lieu, obtenez de l'aide sur la rubrique.

Obtention d'informations concernant une fonction

1. Cliquez sur le bouton Aide de la barre d'outils VLISP.
2. Entrez **`vla-addLightweightpolyline`** dans la boîte de dialogue Entrez le nom de l'élément, puis cliquez sur OK. (Le système d'aide ne tient pas compte de la casse. Il est inutile de la respecter dans le nom de la fonction.)

   L'aide en ligne indique que **`AddLightWeightPolyline`** vous demande de définir, sous forme d'un variant, les sommets de la polyligne comme un réseau de doubles. Voici la description de ce paramètre telle qu'elle est fournie par l'aide.

   ```
   The array of 2D WCS coordinates specifying the vertices of the
   polyline. At least two points (four elements) are required for
   constructing a lightweight polyline. The array size must be a
   multiple of 2.
   ```

Un variant est une construction ActiveX qui sert de conteneur pour divers types de données. Les chaînes, les entiers et les réseaux peuvent tous être représentés par des variants. Le variant stocke les données avec les informations qui les identifient.

A ce stade, vous disposez de quatre points, chacun d'entre eux défini dans un format (x, y, z). La difficulté consiste à transformer ces quatre points en une liste prenant la forme suivante :

```
(x1 y1 x2 y2 x3 y3 x4 y4)
```

La fonction **append** prend des listes composites et les combine. Pour créer une liste composée des quatre points dans le format requis par la fonction ActiveX, vous pouvez utiliser l'expression suivante :

```
(setq polypoints (append (3dPoint->2dPoint p1)
                         (3dPoint->2dPoint p2)
                         (3dPoint->2dPoint p3)
                         (3dPoint->2dPoint p4)))
```

Réécrire quatre fois la fonction **3dPoint->2dPoint** est quelque peu fastidieux. Vous pouvez réduire encore le code en utilisant les fonctions **mapcar** et **apply**. Lorsqu'elle est sélectionnée, **mapcar** exécute une fonction sur des éléments individuels d'une ou plusieurs listes, et **apply** transmet une liste d'arguments à la fonction spécifiée. Le code qui en résulte prend l'aspect suivant :

```
(setq polypoints (apply 'append (mapcar '3dPoint->2dPoint
(liste p1 p2 p3 p4))))
```

Avant l'appel de **mapcar**, la liste des points prend la forme suivante :

```
((x1 y1 z1) (x2 y2 z2) (x3 y3 z3) (x4 y4 z4))
```

Après **mapcar**, la liste prend la forme suivante :

```
((x1 y1) (x2 y2) (x3 y3) (x4 y4))
```

Et finalement, après l'application de la fonction **append** à la liste renvoyée par **mapcar**, vous obtenez ce qui suit :

```
(x1 y1 x2 y2 x3 y3 x4 y4)
```

Construction d'un variant à partir d'une liste de points

A présent, les données contenues dans la variable polypoints sont dans un format de liste adapté à un grand nombre d'appels AutoLISP. Toutefois, les données doivent être fournies comme un paramètre à un appel ActiveX qui attend un réseau variant de coordonnées doubles. Vous pouvez utiliser une autre fonction utilitaire pour effectuer la conversion requise de la liste au variant :

```
(defun gp:list->variantArray (ptsList / arraySpace sArray)
  ; allocate space for an array of 2d points stored as doubles
  (setq arraySpace (vlax-make-safearray
                   vlax-vbdouble ; element type
                   (cons 0
                         (- (length ptsList) 1)
                         ) ; array dimension
                   )
    )
  (setq sArray (vlax-safearray-fill arraySpace ptsList))
  ; return array variant
  (vlax-make-variant sArray)
  )
```

Les actions suivantes se produisent dans **`gp:list->variantArray`**:

- La fonction **`vlax-make-safearray`** est appelée pour allouer un réseau de doubles (`vlax-vbdouble`). La fonction **`vlax-make-safearray`** vous oblige aussi à spécifier les contours d'index inférieur et supérieur du réseau. Dans **`gp:list->variantArray`**, l'appel de **`vlax-make-safearray`** spécifie un index de départ de 0 et fixe la limite supérieure une unité en dessous du nombre d'éléments qui lui est transmis (`ptsList`).
- La fonction **`vlax-safearray-fill`** est appelée pour renseigner le réseau avec les éléments contenus dans la liste de points.
- La fonction **`vlax-make-variant`** est appelée pour convertir le réseau safearray en un variant. En tant que dernier appel de fonction dans **`gp:list->variantArray`**, la valeur renvoyée est transmise à la fonction appelante.

Voici un exemple d'appel de fonction qui active **`gp:list->variantArray`** pour convertir une liste en un réseau variant de doubles :

```
; data conversion from list to variant
(setq VLADataPts (gp:list->variantArray polypoints))
```

Assemblage

Vous disposez à présent de tout le code nécessaire pour dessiner le contour du sentier.

Pour mettre votre code à jour

1 Remplacez votre ancien code de la fonction **gp:drawOutline** par ce qui suit :

```
;;;-------------------------------------------------------------
;;;     Function: gp:drawOutline
;;;-------------------------------------------------------------
;;;  Description: This function will draw the outline of the garden
;;;               path.
;;;-------------------------------------------------------------
;;;  Note: No error checking or validation is performed on the
;;;  BoundaryData parameter.  The sequence of items within this
;;;  parameter does not matter, but it is assumed that all sublists
;;;  are present and contain valid data.
;;; -------------------------------------------------------------
(defun gp:drawOutline (BoundaryData / VLADataPts PathAngle
           Width  HalfWidth  StartPt PathLength
           angm90  angp90     p1 p2
           p3  p4     polypoints pline
          )
  ;; extract the values from the list BoundaryData
  (setq   PathAngle  (cdr (assoc 50 BoundaryData))
    Width   (cdr (assoc 40 BoundaryData))
    HalfWidth  (/ Width 2.00)
    StartPt   (cdr (assoc 10 BoundaryData))
    PathLength (cdr (assoc 41 BoundaryData))
    angp90   (+ PathAngle (Degrees->Radians 90))
    angm90   (- PathAngle (Degrees->Radians 90))
    p1   (polar StartPt angm90 HalfWidth)
    p2   (polar p1 PathAngle PathLength)
    p3   (polar p2 angp90 Width)
    p4   (polar p3 (+ PathAngle (Degrees->Radians 180)) PathLength)
    polypoints (apply 'append
      (mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
       )

  )
  ;; ***** data conversion *****
  ;; Notice, polypoints is in AutoLISP format, consisting of a list
  ;; of the 4 corner points for the garden path.
  ;; The variable needs to be converted to a form of input parameter
  ;; acceptable to ActiveX calls.
  (setq VLADataPts (gp:list->variantArray polypoints))

  ;; Add polyline to the model space using ActiveX automation.
  (setq     pline (vla-addLightweightPolyline
    *ModelSpace*; Global Definition for Model Space
    VLADataPts
     ) ;_ end of vla-addLightweightPolyline

  ) ;_ end of setq
```

```
  (vla-put-closed pline T)
  ;; Return the ActiveX object name for the outline polyline
  ;; The return value should look something like this:
  ;; #<VLA-OBJECT IAcadLWPolyline 02351a34>
  pline
) ;_ end of defun
```

Notez que **gp:drawOutline** renvoie à présent la variable `pline`, et non plus le symbole entre guillemets `'SomeEname` utilisé dans la version de test par tronçons de la fonction.

2 Mettez en forme le code que vous venez d'entrer en le sélectionnant et en cliquant sur le bouton Formater la sélection de la barre d'outils VLISP.

3 Activez ActiveX et ajoutez l'affectation de la variable globale au pointeur de l'espace objet comme indiqué auparavant. Remontez jusqu'au sommet de la fenêtre de l'éditeur de texte et ajoutez le code suivant avant la première instruction **defun** :

```
;;;---------------------------------------------------------------
;;; First step is to load ActiveX functionality. If ActiveX support
;;; already exists in document (can occur when Bonus tools have been
;;; loaded into AutoCAD), nothing happens. Otherwise, ActiveX
;;; support is loaded.
;;;---------------------------------------------------------------

(vl-load-com)

;;; In Lesson 4, the following comment and code is moved to utils.lsp
;;;---------------------------------------------------------------
;;; For ActiveX functions, we need to define a global variable that
;;; "points" to the Model Space portion of the active drawing. This
;;; variable, named *ModelSpace* will be created at load time.
;;;---------------------------------------------------------------
(setq *ModelSpace*
    (vla-get-ModelSpace
         (vla-get-ActiveDocument (vlax-get-acad-object))
    ) ;_ end of vla-get-ModelSpace
) ;_ end of setq
```

Remarquez que le code ci-dessus existe en dehors de toute instruction **defun**. C'est pour cette raison que VLISP exécute automatiquement le code au moment de charger le fichier.

4 Recherchez la ligne suivante dans la fonction **C:GPath** :

```
(setq PolylineName (gp:drawOutline))
```

Remplacez-la par ceci :

```
(setq PolylineName (gp:drawOutline gp_PathData))
```

La fonction **gp:drawOutline** attend à présent un paramètre, la liste contenant les données de contour de polyligne et le changement suivant répond à cette attente.

5 Ajoutez la fonction **`gp:list->variantArray`** reprise dans la section "Construction d'un variant à partir d'une liste de points" à la page 48 à la fin de *gpmain.lsp*.

Faites une tentative de chargement et d'exécution du programme révisé. VLISP retire le contrôle à AutoCAD avant que vous ne puissiez voir le résultat final. Par conséquent, retournez dans la fenêtre AutoCAD une fois que le contrôle est rendu à VLISP. Si le programme s'est exécuté correctement, vous devriez voir apparaître une bordure autour du sentier. Si vous trouvez des erreurs, déboguez le code et réessayez.

Résumé de la leçon 3

Au cours de cette leçon, vous avez

- Ecrit des fonctions utilitaires qui pourront être réutilisées dans d'autres applications.
- Ajouté une logique de création d'entités à votre programme.
- Appris à utiliser les fonctions ActiveX.
- Appris à travailler avec des listes associatives.
- Configuré votre programme de manière à dessiner une bordure de sentier.

Si un détail de cette leçon vous a échappé, nous vous recommandons de l'étudier à nouveau avant de passer à la leçon 4. (Dans ce cas, copiez le code complet de votre répertoire *Lesson2* afin de reprendre la leçon au bon endroit.) Et, si tout le reste échoue, vous pouvez toujours copier le code à partir du répertoire *Tutorial\VisualLISP\Lesson3*.

Création d'un projet et ajout de l'interface

4

Au cours de cette leçon, vous effectuerez deux tâches principales : vous créerez un projet VLISP et ajouterez à votre application une interface à base de boîtes de dialogue. Au cours de l'opération, vous séparerez le fichier AutoLISP unique sur lequel vous avez travaillé jusqu'ici (*gpmain.lsp*) en plusieurs fichiers plus petits, de manière à mieux assimiler le concept de modularité du code.

A partir de cette leçon, le didacticiel fournit des descriptions plus générales des tâches à accomplir, sauf lorsque de nouveaux sujets sont abordés. De même, les fragments de code seront commentés de manière succincte pour économiser de la place. Ceci ne vous libère évidemment pas de l'obligation de commenter votre code de manière complète !

Dans ce chapitre

Décomposition du code en éléments modulaires

A la suite du travail réalisé au cous de la leçon 3, votre fichier *gpmain.lsp* s'est considérablement agrandi. Pour VLISP, cela ne constitue pas un problème. Mais il est plus facile d'assurer la maintenance d'un code si vous le divisez en fichiers contenant des fonctions liées entre elles de manière logique. Votre code est également plus facile à déboguer. Par exemple, dans un fichier unique comportant 150 fonctions, une seule parenthèse manquante peut être difficile à trouver.

Les fichiers du didacticiel seront organisés comme suit :

Organisation de fichiers du didacticiel

Nom de fichier	Contenu
GP-IO.LSP	Toutes les fonctions d'entrée et de sortie (fonctions I/O) telles que les entrées utilisateur. Ce fichier contient aussi le code AutoLISP requis par l'interface de la boîte de dialogue que vous ajouterez.
UTILS.LSP	Reprend toutes les fonctions génériques susceptibles d'être réutilisées dans d'autres projets. Contient aussi les codes d'initialisation au moment du chargement.
GPDRAW.LSP	Toutes les routines de dessin : le code qui crée effectivement les entités AutoCAD.
GPMAIN.LSP	La fonction de base C:Gpath.

Séparation de gpmain.lsp en quatre fichiers

1 Créez un nouveau fichier, puis coupez et collez les fonctions suivantes depuis *gpmain.lsp* dans le nouveau fichier :

- **gp:getPointInput**
- **gp:getDialogInput**

Enregistrez le nouveau fichier dans votre répertoire de travail sous le nom *gp-io.lsp*.

2 Créez un nouveau fichier, puis coupez et collez les fonctions suivantes depuis *gpmain.lsp* dans le nouveau fichier :

- **Degrees->Radians**
- **3Dpoint->2Dpoint**
- **gp:list->variantArray**

De même, insérez au début du fichier les lignes de code destinées à activer la fonctionnalité ActiveX (**`vl-load-com`**) et à affecter la variable globale (`*ModelSpace*`).

Enregistrez le fichier sous le nom *utils.lsp*.

3 Créez un nouveau fichier, puis coupez et collez les fonctions suivantes de *gpmain.lsp* dans le nouveau fichier :

- **`gp:drawOutline`**

Enregistrez ce fichier sous le nom *gpdraw.lsp*.

4 Après avoir extrait le code de *gpmain.lsp*, enregistrez-le et vérifiez-le. Seule la fonction d'origine, **`C:GPath`**, doit rester dans le fichier.

Votre bureau VLISP commence à être encombré. Vous pouvez réduire les fenêtres dans VLISP sans qu'elles cessent d'être accessibles. Dans la barre d'outils Vue, cliquez sur le bouton Sélectionner fenêtre pour choisir une fenêtre de la liste, ou sélectionnez Fenêtre dans la barre de menus VLISP et choisissez une fenêtre à afficher.

Utilisation de projets Visual LISP

La fonction Projet de VLISP met à votre disposition un moyen pratique de gérer les fichiers qui constituent votre application. En outre, elle vous permet d'ouvrir un fichier de projet unique au lieu d'ouvrir individuellement chaque fichier LISP de l'application. Dès que le projet est ouvert, il vous suffit d'un double clic pour accéder aux fichiers qui le composent.

Création d'un projet VLISP

1 Dans la barre de menus VLISP, choisissez Projet ➤ Nouveau projet.

2 Enregistrez le fichier dans le répertoire *Lesson4*, en lui attribuant le nom *gpath.prj*.

Une fois l'enregistrement du fichier effectué, VLISP affiche la boîte de dialogue Propriétés du projet.

3 Cliquez sur le bouton (Dé)sélectionner tout All situé sur le côté gauche de la boîte de dialogue Propriétés du projet.

4 Cliquez sur le bouton portant une flèche pointée vers la droite. Cette opération ajoute les fichiers sélectionnés à votre projet.

Dans la boîte de dialogue Propriétés du projet, la zone de liste située à gauche affiche tous les fichiers LISP qui résident dans le même répertoire que votre fichier de projet et qui *ne sont pas* compris dans ce projet. La zone de liste

située à droite reprend tous les fichiers qui appartiennent au projet. Lorsque les fichiers sélectionnés sont ajoutés au projet, leur nom passe de la zone de liste de gauche à celle de droite.

5 Dans la zone de liste située à droite de la boîte de dialogue, sélectionnez *gpmain*, puis cliquez sur le bouton Bas. Cette opération place le fichier au bas de la liste.

VLISP charge les fichiers dans l'ordre où ils sont répertoriés. Etant donné que l'invite indiquant aux utilisateurs le nom de la commande se situe à la fin du fichier *gpmain.lsp*, vous devez placer ce fichier au bas de la liste. Le chargement de ce fichier en dernier lieu a pour résultat d'afficher l'invite à l'intention des utilisateurs. Le fichier *utils.lsp* doit être chargé en premier lieu parce qu'il contient le code d'initialisation de l'application. Par conséquent, sélectionnez *utils* dans la zone de liste de la boîte de dialogue et cliquez sur le bouton Haut.

6 Cliquez sur OK

VLISP ajoute une petite fenêtre de projet à votre bureau VLISP. Cette fenêtre répertorie les fichiers contenus dans votre projet. Cliquez deux fois sur un fichier pour l'ouvrir dans l'éditeur de texte VLISP (s'il n'est pas déjà ouvert) et pour en faire la fenêtre active de l'éditeur.

Ajout de l'interface de la boîte de dialogue

Le chapitre suivant de la leçon porte l'ajout d'une interface de boîte de dialogue à l'application du sentier de jardin. Pour y parvenir, vous devrez utiliser un autre langage appelé *dialog control language (DCL)*.

Actuellement, votre fonction **`gpath`** accepte les entrées uniquement au niveau de la ligne de commande. Vous avez introduit une fonction de test par tronçons **`(gp:getDialogInput)`** avec l'intention d'ajouter une interface de boîte de dialogue. Le moment est venu d'ajouter l'interface.

La création d'une interface de boîte de dialogue fonctionnelle comporte deux étapes :

- Définir l'aspect et le contenu des boîtes de dialogue.
- Ajouter un code de programme permettant de contrôler le comportement de la boîte de dialogue.

La description et le format d'une boîte de dialogue sont définis dans un fichier *.dcl*. Le langage DCL est décrit dans les chapitre 11, “Designing Dialog Boxes,” chapitre 12, “Managing Dialog Boxes,” et chapitre 13, “Programmable Dialog Box Reference”du *Visual LISP Developer's Guide*.

Le code de programme qui initialise les paramètres par défaut et répond à l'interaction de l'utilisateur sera ajouté à **`gp:getDialogInput`**.

Création de la boîte de dialogue à l'aide du langage DCL

Commencez par examiner la boîte de dialogue que vous devez créer.

La boîte de dialogue comporte les éléments suivants :

- Deux jeux de boutons de radio.
  Un jeu de boutons détermine le style de polyligne du contour et l'autre spécifie la méthode de création des entités dalles (ActiveX, **`entmake`** ou **`command`**). Un seul bouton de radio contenu dans un jeu peut être sélectionné à la fois.
- Des zones d'édition permettant de spécifier le rayon des dalles et l'espace qui les sépare.
- Un jeu standard de boutons OK et Annuler (Cancel).

Les éléments des boîtes de dialogue sont appelés *composants* en langage DCL. Ecrire le contenu complet d'un fichier de boîte de dialogue DCL peut paraître une tâche insurmontable. L'astuce consiste à faire un croquis de ce que vous voulez, de le diviser en sections, puis de décrire chaque section séparément.

Pour créer la boîte de dialogue

1. Ouvrez un nouveau fichier dans l'éditeur de texte VLISP.
2. Entrez l'instruction suivante dans le nouveau fichier :

   ```
   label = "Garden Path Tile Specifications";
   ```

   Cette instruction DCL définit le titre de la fenêtre de la boîte de dialogue.

3 Définissez les boutons de radio permettant de spécifier le type de polyligne en ajoutant le code suivant :

```
: boxed_radio_column {    // defines the radio button areas
  label = "Outline Polyline Type";
  : radio_button {        // defines the lightweight radio button
    label = "&Lightweight";
    key = "gp_lw";
    value = "1";
   }
: radio_button {      // defines the old-style polyline radio button
  label = "&Old-style";
  key = "gp_hw";
 }
}
```

La directive DCL `boxed_radio_column` définit un contour de zone et vous permet de spécifier un libellé pour le jeu de boutons. A l'intérieur du contour, vous spécifiez les boutons de radio dont vous avez besoin en ajoutant des directives `radio_button`. Chaque bouton de radio exige un libellé et une *clé*. La clé est le nom par lequel votre code AutoLISP peut faire référence au bouton.

Notez qu'une valeur de 1 est attribuée au bouton de radio libellé "lightweight". Une valeur de 1 (une chaîne et non un entier) attribuée à un bouton en fait l'option par défaut d'une rangée de boutons. En d'autres termes, lorsque vous affichez la boîte de dialogue pour la première fois, le bouton est sélectionné. Notez également que, dans le fichiers DCL, les commentaires sont indiqués par des doubles barres obliques, et non par des points-virgules comme dans AutoLISP.

4 Définissez la colonne radio pour la sélection du style de création d'entités en ajoutant le code suivant :

```
: boxed_radio_column {     // defines the radio button areas
  label = "Tile Creation Method";
  : radio_button {         // defines the ActiveX radio button
    label = "&ActiveX Automation";
    key = "gp_actx";
    value = "1";
   }
: radio_button {          // defines the (entmake) radio button
  label = "&Entmake";
  key = "gp_emake";
 }
: radio_button {          // defines the (command) radio button
  label = "&Command";
  key = "gp_cmd";
 }
}
```

5 Ajoutez le code suivant pour définir les composants de la zone d'édition qui permettent aux utilisateurs d'entrer les valeurs correspondant à la taille et à l'espacement des dalles :

```
: edit_box {      // defines the Radius of Tile edit box
  label = "&Radius of tile";
  key = "gp_trad";
  edit_width = 6;
}
: edit_box {      // defines the Spacing Between Tiles edit box
  label = "S&pacing between tiles";
  key = "gp_spac";
  edit_width = 6;
}
```

Notez que cette définition n'attribue aucune valeur initiale aux zones d'édition. Vous définirez les valeurs par défaut de chaque zone d'édition dans votre programme AutoLISP.

6 Ajoutez le code suivant pour les boutons OK et Annuler (Cancel) :

```
: row {          // defines the OK/Cancel button row
  : spacer { width = 1; }
  : button {    // defines the OK button
    label = "OK";
    is_default = true;
    key = "accept";
    width = 8;
    fixed_width = true;
  }
  : button {    // defines the Cancel button
    label = "Cancel";
    is_cancel = true;
    key = "cancel";
    width = 8;
    fixed_width = true;
  }
  : spacer { width = 1;}
}
```

Les deux boutons sont définis à l'intérieur d'une rangée, de sorte qu'ils s'alignent horizontalement.

7 Placez-vous au début de la fenêtre de l'éditeur de texte et insérez l'instruction suivante à la première ligne de votre DCL :

```
gp_mainDialog : dialog {
```

8 La directive `dialog` requiert une accolade fermante. Faites défiler le fichier jusqu'à la fin et ajoutez l'accolade à la dernière ligne du code DCL :

```
}
```

Enregistrement d'un fichier DCL

Avant d'enregistrer le fichier contenant votre DCL, n'oubliez pas qu'AutoCAD doit être en mesure de localiser votre fichier DCL en cours d'exécution. Pour cette raison, le fichier doit être placé dans un des emplacements du chemin de recherche des fichiers de support. (Si vous avez des doutes concernant ces emplacements, choisissez Outils ➤ Options dans le menu AutoCAD et, sur l'onglet Fichiers, vérifiez les emplacements du chemin de recherche des fichiers de support.)

Pour l'instant, vous pouvez enregistrer le fichier dans le répertoire *Support* d'AutoCAD.

Pour enregistrer votre fichier DCL

1. Dans la barre de menus VLISP, choisissez Fichier ➤ Enregistrer sous.
2. Dans le champ Enregistrer sous de la boîte de dialogue, sélectionnez Fichiers source DCL.
3. Remplacez le chemin d'enregistrement par le chemin de support du *<répertoire AutoCAD>*\.
4. Entrez le nom de fichier *gpdialog.dcl*.
5. Cliquez sur Enregistrer.

   Notez que VLISP remplace l'ensemble des couleurs de la syntaxe une fois le fichier enregistré. VLISP est conçu pour reconnaître les fichiers DCL et mettre en surbrillance les divers types d'éléments syntaxiques.

Affichage d'un aperçu de la boîte de dialogue

VLISP est doté d'une fonction d'aperçu permettant de vérifier les résultats de votre code DCL.

Pour afficher l'aperçu d'une boîte de dialogue créée par l'intermédiaire du langage DCL

1. Dans la barre de menus VLISP, choisissez Outils ➤ Outils d'interface ➤ Aperçu DCL dans l'éditeur.
2. Cliquez sur OK lorsque vous êtes invité à spécifier un nom de boîte de dialogue.

   Dans ce cas, votre fichier DCL définit une seule boîte de dialogue, de sorte qu'aucun choix ne soit à faire. Toutefois, si vous créez des applications plus étendues, il se peut que vos fichiers DCL contiennent plusieurs boîtes de dialogue. C'est ici que vous pouvez sélectionner la boîte que vous souhaitez afficher.

3 Si la boîte de dialogue s'affiche correctement, cliquez sur n'importe quel bouton pour la refermer.

VLISP transmet le contrôle à AutoCAD afin d'afficher la boîte de dialogue. Si AutoCAD trouve des erreurs syntaxiques, il affiche une ou plusieurs fenêtres de message identifiant les erreurs.

Si AutoCAD détecte des erreurs DCL que vous n'êtes pas en mesure de corriger, copiez le fichier *Gpdialog.dcl* dans votre répertoire *Tutorial\VisualLISP\Lesson4* et enregistrez-le dans le répertoire *Support*.

Interaction avec la boîte de dialogue à partir du code AutoLISP

Vous devez à présent programmer la fonction AutoLISP pour interagir avec la boîte de dialogue. Cette activité se réalisera au niveau de la fonction de test par tronçons **`gp:getDialogInput`**. Cette fonction se trouve actuellement dans le fichier *gp-io.lsp* que vous avez extrait précédemment de *gpmain.lsp*.

Le développement d'une interface de boîte de dialogue peut paraître déroutant lors des premières tentatives. Cette opération vous oblige à planifier et à vous poser des questions telles que :

- Cette boîte de dialogue doit-elle être configurée avec des valeurs par défaut ?
- Que se passe-t-il lorsque l'utilisateur clique sur un bouton ou entre une valeur ?
- Que se passe-t-il si l'utilisateur clique sur Annuler ?
- Si le fichier de boîte de dialogue (*.dcl*) est manquant, que doit-il se passer ?

Configuration des valeurs de boîte de dialogue

Lorsque vous exécutez l'application du sentier de jardin dans son intégralité, vous noterez que la boîte de dialogue s'ouvre toujours avec ActiveX comme méthode de création d'objet par défaut et Lightweight comme style de polyligne. La taille des dalles par défaut présente un phénomène plus intéressant : les valeurs changent en fonction de la largeur du chemin. Le fragment de code suivant indique comment définir les valeurs par défaut à afficher dans la boîte de dialogue :

```
(setq   objectCreateMethod  "ACTIVEX"
        plineStyle  "LIGHT"
        tilerad   (/ pathWidth 15.0)
        tilespace (/ tilerad 5.0)
        dialogLoaded T
        dialogShow T
  ) ;_ end of setq
```

Pour le moment, ne vous préoccupez pas du rôle des variables `dialogLoaded` et `dialogShow`. Il deviendra évident dans les deux prochaines sections.

Chargement du fichier de boîte de dialogue

En premier lieu, votre programme doit charger le fichier DCL à l'aide de la fonction **`load_dialog`**. Cette fonction recherche les fichiers de boîte de dialogue en fonction du chemin de recherche des fichiers de support AutoCAD, à moins que vous ne spécifiez un nom de chemin complet.

A chaque fonction **`load_dialog`** doit correspondre une fonction **`unload_dialog`** à un stade postérieur du code. Vous le verrez dans un moment. Pour l'instant, intéressez-vous à la méthode de chargement de votre boîte de dialogue :

```
  ;; Load the dialog box.  Set up error checking to make sure
  ;; the dialog file is loaded before continuing
 (if (= -1 (setq dcl_id (load_dialog "gpdialog.dcl")))
    (progn
      ;; There's a problem - display a message and set the
      ;; dialogLoaded flag to nil
      (princ "\nCannot load gpdialog.dcl")
      (setq dialogLoaded nil)
    ) ;_ end of progn
  ) ;_ end of if
```

La variable `dialogLoaded` indique si le chargement de la boîte de dialogue s'est effectué correctement. Lorsque vous définissez les valeurs initiales de la boîte de dialogue, vous attribuez à `dialogLoaded` une valeur initiale `T`. Comme vous pouvez le constater dans le fragment de code ci-dessus, une valeur `nil` est attribuée à `dialogLoaded` si un problème intervient lors du chargement.

Chargement d'une boîte de dialogue spécifique en mémoire

Nous avons signalé précédemment qu'un seul fichier DCL peut contenir plusieurs définitions de boîte de dialogue. La prochaine étape dans l'utilisation d'une boîte de dialogue consiste à spécifier la définition de la boîte de dialogue à afficher. Les lignes de code suivantes en donnent un exemple :

```
(if (and dialogLoaded
        (not (new_dialog "gp_mainDialog" dcl_id))
     ) ;_ end of and
    (progn
      ;; There's a problem...
      (princ "\nCannot show dialog gp_mainDialog")
      (setq dialogShow nil)
    ) ;_ end of progn
) ;_ end of if
```

Observez l'utilisation de la fonction **and** dans le but de confirmer si la boîte de dialogue a bien été chargée et si l'appel de **new_dialog** a abouti. Si plusieurs expressions sont évaluées à l'intérieur d'un appel de fonction **et**, l'évaluation des expressions suivantes se termine par la première expression dont la valeur est `nil`. Dans ce cas, si le drapeau de `dialogLoaded` est `nil` (ce qui signifie que la fonction de chargement de la section précédente a échoué), VLISP renonce à exécuter la fonction **new_dialog**.

Remarquez que le code tient compte de la possibilité d'un mauvais fonctionnement du fichier DCL et attribue dans ce cas la valeur `nil` à la variable `dialogShow`.

La fonction **new_dialog** charge simplement la boîte de dialogue en mémoire, sans l'afficher. La fonction **start_dialog** affiche la boîte de dialogue. Toute opération d'initialisation de la boîte de dialogue (comme la définition des valeurs des composants, la création d'images ou de listes pour les zones de liste et les associations à des composants spécifiques) doit avoir lieu après l'appel de la fonction **new_dialog** et avant celui de **start_dialog**.

Initialisation des valeurs par défaut de la boîte de dialogue

Si le chargement de la boîte de dialogue s'est effectué correctement, vous êtes en mesure de définir les valeurs qui s'afficheront pour les utilisateurs. Le chargement est réussi si les variables de drapeau `dialogLoaded` et `dialogShow` prennent toutes deux la valeur `T` (true).

Définissez à présent les valeurs initiales du rayon et de l'espacement des dalles. La fonction **set_tile** attribue une valeur à une dalle. Une zone d'édition traite les chaînes plutôt que les nombres ; vous donc devez utiliser la fonction **rtos** (nombre réel en chaîne) pour convertir la variable de taille des composants en chaînes de format décimal avec une précision de deux chiffres. Les lignes suivantes traitent cette conversion :

```
(if (and dialogLoaded dialogShow)
    (progn
      ;; Set the initial state of the tiles
      (set_tile "gp_trad" (rtos tileRad 2 2))
      (set_tile "gp_spac" (rtos tileSpace 2 2))
```

Assignation d'actions aux composants

Une définition par le langage DCL ne fait que créer une boîte de dialogue inerte. Vous devez connecter cette boîte de dialogue inerte à votre code dynamique AutoLISP à l'aide de la fonction **action_tile**, comme le montre l'exemple ci-dessous :

```
;; Assign actions (the functions to be invoked) to dialog buttons
(action_tile
  "gp_lw"
  "(setq plineStyle \"Light\")"
)
(action_tile
  "gp_hw"
  "(setq plineStyle \"Pline\")"
)
(action_tile
  "gp_actx"
  "(setq objectCreateMethod \"ActiveX\")"
)
(action_tile
  "gp_emake"
  "(setq objectCreateMethod \"Entmake\")"
)
(action_tile
  "gp_cmd"
  "(setq objectCreateMethod \"Command\")"
)
(action_tile "cancel" "(done_dialog) (setq UserClick nil)")
(action_tile
  "accept"
  (strcat "(progn (setq tileRad (atof (get_tile \"gp_trad\")))"
       "(setq tileSpace (atof (get_tile \"gp_spac\")))"
       "(done_dialog) (setq UserClick T))"
  )
)
```

Remarquez tous les guillemets qui entourent le code AutoLISP. Lorsque vous rédigez une fonction AutoLISP **action_tile**, tout se passe comme si votre code indiquait au composant : "mémorise cette chaîne, puis retransmets-la moi lorsque l'utilisateur te sélectionnera". La chaîne (tout ce qui se trouve entre doubles guillemets) reste inactive jusqu'à ce que l'utilisateur sélectionne le composant en question. A ce moment, le composant transmet la chaîne à AutoCAD qui la convertit en code fonctionnel AutoLISP et exécute ce code.

Prenons l'exemple de l'expression **action_tile** suivante, qui est connectée au bouton de radio polyligne lightweight :

```
(action_tile
   "gp_lw"
   "(setq plineStyle \"Light\")"
)
```

Le code attribue la chaîne `"(setq plineStyle \"Light\")"` au bouton de radio. Lorsqu'un utilisateur sélectionne le bouton, la chaîne est retransmise à AutoCAD et transformée immédiatement en expression AutoLISP :

```
(setq plineStyle "Light")
```

Examinez cet autre fragment de code. Il s'agit de l'expression **`action_tile`** assignée au bouton OK :

```
(action_tile
    "accept"
    (strcat "(progn (setq tileRad (atof (get_tile \"gp_trad\")))"
      "(setq tileSpace (atof (get_tile \"gp_spac\")))"
      "(done_dialog) (setq UserClick T))"
)
```

Lorsqu'un utilisateur clique sur le bouton OK, la longue chaîne attribuée au bouton est transmise à AutoCAD et transformée en code AutoLISP :

```
(progn
   (setq tileRad (atof (get_tile "gp_trad")))
   (setq tileSpace (atof (get_tile "gp_spac")))
   (done_dialog)
   (setq UserClick T)
)
```

Ce code effectue plusieurs opérations : Il extrait les valeurs courantes des composants dont les valeurs clés sont `gp_trad` (rayon des dalles) et `gp_spac` (espacement des dalles). Ensuite, **`atof`** convertit la chaîne de nombre en un nombre réel. La boîte de dialogue se termine par la fonction **`done_dialog`** et une valeur `T`, ou true, est attribuée à la variable `UserClick`.

Vous avez fini d'assigner des actions aux boutons. Il ne vous reste plus qu'à tout mettre en marche.

Affichage de la boîte de dialogue

La fonction **`start_dialog`** affiche une boîte de dialogue et accepte l'entrée de l'utilisateur. En outre, cette fonction n'exige aucun argument.

```
(start_dialog)
```

Le contrôle est rendu aux utilisateurs lorsque vous entrez **`start_dialog`**. Les utilisateurs peuvent faire des choix à l'intérieur de la boîte de dialogue, jusqu'au moment où ils cliquent sur le bouton OK ou sur le bouton Cancel (Annuler).

Déchargement de la boîte de dialogue

Lorsqu'un utilisateur clique sur le bouton OK ou sur le bouton Annuler (Cancel), vous devez décharger la boîte de dialogue. Tout comme **`start_dialog`**, **`unload_dialog`** est une fonction simple.

```
(unload_dialog dcl_id)
```

Choix de l'étape suivante

Si l'utilisateur a cliqué sur OK, vous devez créer une liste contenant les valeurs définies par l'interaction de l'utilisateur et de la boîte de dialogue. Cette liste sera renvoyée par **`gp:getDialogInput`** à sa fonction appelante. Si l'utilisateur a cliqué sur Annuler (Cancel), la fonction renvoie `nil` :

```
(if UserClick          ; User clicked Ok
  ;; Build the resulting data
  (progn
    (setq Result (list
        (cons 42 tileRad)
        (cons 43 TileSpace)
        (cons 3 objectCreateMethod)
        (cons 4 plineStyle)
        )
     )
  )
)
```

Assemblage du code

Avec les exemples ci-dessus et quelques lignes supplémentaires, vous disposez du code nécessaire pour achever la fonction **`gp:getDialogInput`**.

Pour assembler gp:getDialogInput

1. Ouvrez votre copiede *gp-io.lsp* dans une fenêtre d'éditeur de texte VLISP.
2. Supprimez le code contenu dans **`gp:getDialogInput`** (l'instruction `defun gp:getDialogInput` et tout ce qui la suit)
3. Entrez l'instruction `defun` suivante comme première ligne de code de la fonction **`gp:getDialogInput`** :

   ```
   (defun gp:getDialogInput (pathWidth / dcl_id objectCreateMethod
                  plineStyle tilerad tilespace result UserClick
                  dialogLoaded dialogShow)
   ```

   La fonction attend un seul argument (`pathwidth`) et fixe le nombre de variables locales.
4. A la suite du code ajouté à l'étape 3, entrez l'exemple de code de chacune des sections suivantes de ce chapitre.

   - "Configuration des valeurs de boîte de dialogue"
   - "Chargement du fichier de boîte de dialogue"
   - "Chargement d'une boîte de dialogue spécifique en mémoire"
   - "Initialisation des valeurs par défaut de la boîte de dialogue"
   - "Assignation d'actions aux composants"

REMARQUE Entrez seulement le premier exemple de code extrait de "Assignation d'actions aux composants,", et non les fragments des explications qui suivent. Ces fragments ne font que répéter des éléments de l'exemple.

- "Affichage de la boîte de dialogue"
- "Déchargement de la boîte de dialogue"
- "Choix de l'étape suivante"

5 Après la dernière ligne de code, ajoutez ce qui suit :

```
    )
  )
   Result;
) ;_ end of defun
```

6 Formatez le code que vous avez entré en choisissant, dans la barre de menus VLISP, Outils ➤ Formater le code dans l'éditeur.

Mise à jour d'une fonction de test par tronçons

Vous avez révisé la fonction **`gp:getDialogInput`**. A chaque modification d'une fonction de test par tronçons, vous devez vérifier un certain nombre de points.

- L'instruction `defun` a-t-elle changé ? En d'autres termes, la fonction prend-elle toujours le même nombre d'arguments ?
- La fonction renvoie-t-elle quelque chose de différent ?

Dans le cas de **`gp:getDialogInput`**, la réponse aux deux questions est oui. La fonction accepte les paramètres de largeur du sentier (pour définir la taille et l'espacement par défaut des dalles). Et, au lieu de renvoyer `T`, qui est la valeur de la version de test par tronçons de la fonction renvoyée, **`gp:getDialogInput`** renvoie à présent une liste associative contenant quatre nouvelles fonctions.

Les deux modifications ont une incidence sur le code qui appelle la fonction et le code qui traite les valeurs renvoyées par les fonctions. Remplacez votre ancienne version de la fonction **`C:Gpath`** dans *gpmain.lsp* par le code suivant :

```
(defun C:GPath (/ gp_PathData gp_dialogResults)
  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters.  Continue only if you
  ;; have valid input.  Store the data in gp_PathData.
  (if (setq gp_PathData (gp:getPointInput))
    (if (setq gp_dialogResults (gp:getDialogInput (cdr(assoc 40
                               gp_PathData))))
      (progn
       ;; Now take the results of gp:getPointInput and append this
       ;; to the added information supplied by gp:getDialogInput.

       (setq gp_PathData (append gp_PathData gp_DialogResults))

       ;; At this point, you have all the input from the user.
       ;; Draw the outline, storing the resulting polyline
       ;; "pointer" in the variable called PolylineName.
       (setq PolylineName (gp:drawOutline gp_PathData))
      ) ;_ end of progn
     (princ "\nFunction cancelled.")
     ) ;_ end of if
    (princ "\nIncomplete information to draw a boundary.")
  ) ;_ end of if
  (princ)  ; exit quietly
) ;_ end of defun
```

Examinez les lignes en gras de la révision de la fonction principale **`C:GPath`**. Deux modifications essentielles doivent être apportées au programme pour lui permettre de fonctionner correctement :

- Lorsque la fonction **`gp:getDialogInput`** est appelée, la largeur du sentier lui est transmise. A cette fin, la valeur associée à l'index de clé 40 de la liste associative `gp_PathData` est extraite.
- La liste associative renvoyée par **`gp:getPointInput`** est assignée à une variable appelée `gp_dialogResults`. Si cette variable est affectée d'une valeur, son contenu doit être ajouté aux valeurs de la liste associative déjà stockées dans `gp_PathData`.

Des modifications supplémentaires interviennent dans le code à la suite du remplacement de marques de réservation dans la fonction de test par tronçons. Le procédé le plus aisé consiste à copier ce code à partir du didacticiel en ligne et de le coller dans votre fichier.

Mise à disposition d'un choix de types de lignes de contour

Une exigence de l'application Sentier de jardin était de permettre aux utilisateurs de dessiner le contour en polyligne fine ou ancien style. La première version de **gp:drawOutline** que vous avez dessinée utilisait toujours une polyligne fine pour le dessin du contour. Maintenant que l'interface de la boîte de dialogue est prête à fonctionner, vous pouvez inclure l'option de dessin d'une polyligne ancien style. Pour y parvenir, **gp:drawOutline** doit déterminer le type de polyligne, puis la dessiner.

Les changements à apporter à **gp:drawOutline** sont inclus dans le fragment de code suivant. Procédez à la modification à partir du fichier *gpdraw.lsp* indiqué en gras :

```
(setq PathAngle  (cdr (assoc 50 BoundaryData))
    Width        (cdr (assoc 40 BoundaryData))
    HalfWidth    (/ Width 2.00)
    StartPt      (cdr (assoc 10 BoundaryData))
    PathLength   (cdr (assoc 41 BoundaryData))
    angp90       (+ PathAngle (Degrees->Radians 90))
    angm90       (- PathAngle (Degrees->Radians 90))
    p1       (polar StartPt angm90 HalfWidth)
    p2       (polar p1 PathAngle PathLength)
    p3       (polar p2 angp90 Width)
    p4       (polar p3 (+ PathAngle (Degrees->Radians 180))
              PathLength)
    poly2Dpoints (apply 'append
                 (mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
                )
    poly3Dpoints (mapcar 'float (append p1 p2 p3 p4))
    ;; get the polyline style
    plineStyle   (strcase (cdr (assoc 4 BoundaryData)))
) ;_ end of setq
;; Add polyline to the model space using ActiveX automation
(setq pline (if (= plineStyle "LIGHT")
     ;; create a lightweight polyline
     (vla-addLightweightPolyline
          *ModelSpace*      ; Global Definition for Model Space
          (gp:list->variantArray poly2Dpoints) ;data conversion
      ) ;_ end of vla-addLightweightPolyline
     ;; or create an old-style polyline
     (vla-addPolyline
          *ModelSpace*
          (gp:list->variantArray poly3Dpoints) ;data conversion
     ) ;_ end of vla-addPolyline
    ) ;_ end of if
) ;_ end of setq
```

Entrer les changements au clavier peut se révéler difficile, car vous devez non seulement ajouter du code, mais supprimer également certains fragments et en réorganiser d'autres. Nous vous recommandons de copier toute l'instruction **setq** à partir du didacticiel en ligne et de la coller dans votre propre code.

Nettoyage

Si vous ne l'avez pas encore fait, effacez le fragment de code suivant de la fonction **C:GPath** dans *gpmain.lsp* :

```
(princ "\nThe gp:drawOutline function returned <")
(princ PolylineName)
(princ ">")
(Alert "Congratulations - your program is complete!")
```

Vous avez utilisé ce code comme une marque de réservation mais, depuis que **gp:drawOutline** fonctionne, vous n'en avez plus besoin.

Exécution de l'application

Avant d'exécuter votre programme, enregistrez tous les fichiers modifiés si vous ne l'avez pas encore fait. Choisissez Fichier ➤ Enregistrer tout dans la barre de menus VLISP ou utilisez le ALT+SHIFT+S raccourci clavier permettant d'enregistrer tous vos fichiers ouverts.

La deuxième tâche que vous devez entreprendre consiste à recharger tous les fichiers dans VLISP.

Pour charger et exécuter tous les fichiers dans votre application

1. Si le fichier de projet que vous avez créé précédemment dans cette leçon n'est pas encore ouvert, choisissez Projet ➤ Ouvrir un projet dans la barre de menus VLISP, puis entrez le nom du fichier de projet *gpath* en omettant l'extension *.prj*. Si VLISP ne trouve pas le fichier de projet, cliquez sur le bouton Parcourir et sélectionnez le fichier dans la boîte de dialogue Ouvrir un projet. Cliquez sur Ouvrir.

2. Cliquez sur le bouton Charger les fichiers source dans la fenêtre de projet.
3. Entrez la commande **(C:GPath)** lorsque la Console VLISP vous invite à exécuter le programme. Si vous devez déboguer votre programme, essayez les outils que vous avez appris à utiliser au cours des leçons 2 et 3. Et n'oubliez pas que, si tout le reste échoue, vous pouvez toujours copier le code à partir du répertoire *Vlisp\Tutorial\Lesson4*.

Essayez également de dessiner le sentier à l'aide des polylignes lighweight et old-style. Après avoir dessiné les sentiers, utilisez la commande **`list`** d'AutoCAD pour déterminer si votre programme dessine les entités correctes.

Résumé de la leçon 4

Au cours de cette leçon, vous avez :

- Décomposé votre code en éléments modulaires en le divisant en quatre fichiers.
- Organisé vos modules de code en un projet VLISP.
- Appris à créer une boîte de dialogue à l'aide du langage DCL (Dialog Control Language).
- Ajouté le code AutoLISP permettant de définir et de traiter les entrées dans la boîte de dialogue.
- Modifié votre code pour proposer aux utilisateurs un choix de types de lignes de contour.

A présent, vous disposez d'un programme qui dessine le contour d'un sentier de jardin. Au cours de la leçon suivante, vous ajouterez les dalles au sentier. D'autres outils de développement VLISP vous seront présentés pendant cette opération.

Dessin des dalles

5

A la fin de cette leçon, votre application répondra aux principales exigences de la leçon 1. Vous ajouterez les fonctions permettant de dessiner des dalles dans le contour du sentier, et ce en utilisant différentes méthodes de création d'entités. Vous apprendrez aussi à connaître certains raccourcis clavier et de nouveaux outils d'édition.

Dans ce chapitre

Présentation des nouveaux outils d'édition Visual LISP

Ouvrez votre copie de *gpdraw.lsp* dans une fenêtre de l'éditeur de texte VLISP si le fichier n'est pas encore ouvert. Ce code comporte quelques aspects qui se retrouvent très fréquemment dans les codes que vous développerez dans VLISP. Tout d'abord, vous y trouvez de nombreuses parenthèses, y compris des parenthèses à l'intérieur de parenthèses. Ensuite, vous rencontrerez de nombreux appels de fonctions, et certaines de ces fonctions ont des noms très longs (**`vla-addLightweightPolyline`**, par exemple). VLISP a prévu des outils d'édition pour vous aider à traiter ces caractéristiques communes du code AutoLISP.

Appariement des parenthèses

VLISP vous propose une fonction d'appariement des parenthèses pour vous aider à trouver la parenthèse fermante qui correspond à une parenthèse ouvrante.

Pour apparier une parenthèse ouverte et une parenthèse fermée

1. Placez votre curseur devant la parenthèse qui précède l'appel de fonction **`setq`**.
2. Appuyez sur CTRL + SHIFT +). (Cela fonctionne aussi si vous cliquez deux fois.)

   VLISP recherche la parenthèse fermante qui correspond à celle que vous avez choisie et sélectionne tout le code compris entre ces deux parenthèses. Cette fonction vous permet non seulement de vérifier que vous avez tapé le nombre correct de parenthèses, mais aussi de copier ou de couper le texte sélectionné. Vous auriez pu en apprécier l'utilité lors de la mise à jour d'un appel de fonction à la fin de la leçon 4.

   Quelles sont les autres utilisations possibles ? Vous pouvez copier-coller un fragment de code dans la fenêtre de la Console VLISP et le tester. Ou peut-être avez-vous trouvé le moyen de remplacer 50 lignes de code par un code en trois lignes bien meilleur. Vous pouvez rapidement sélectionner l'ancien code à l'aide de l'outil d'appariement des parenthèses, puis l'éliminer en appuyant sur une seule touche. Il est beaucoup plus rapide de laisser à VLISP le soin de trouver un bloc entier de code que de rechercher vous-même chacune des parenthèses fermantes.

Une touche de commande correspondante permet d'apparier et de sélectionner vers l'arrière. Pour l'essayer, placez votre curseur à la suite d'une parenthèse fermante et cliquez deux fois ou appuyez sur CTRL + SHIFT + (. VLISP recherche la parenthèse ouvrante correspondante et la sélectionne en même temps que le code situé entre les deux.

Les deux commandes sont également accessibles en choisissant, dans la barre de menus VLISP, les options Edition ➤ Apparier les parenthèses.

Achèvement automatique des mots

Supposons que vous ajoutiez une nouvelle fonction à votre programme en utilisant le code suivant :

```
ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
(if (equal ObjectCreationStyle  "COMMAND")
 (progn
  (setq firstCenterPt(polar rowCenterPt (Degrees->Radians 45)
distanceOnPath))
    (gp:Create_activeX_Circle)
  )
)
```

(Ne vous préoccupez pas de ce que fait éventuellement le code. Il s'agit seulement d'un exemple comportant plusieurs variables et noms de fonctions longs.)

VLISP peut vous faire gagner du temps en complétant les mots à votre place.

Utilisation de la fonction d'achèvement des mots recherchés dans Visual LISP

1 Placez-vous au bas du fichier *gpdraw.lsp* et entrez le code suivant :

```
ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
   (if (equal Ob
```

2 Appuyez sur CTRL + SPACEBAR.

VLISP vous a épargné dix-sept frappes en parcourant le fichier ouvert à la recherche des correspondances les plus proches avec les deux dernières lettres que vous avez tapées.

3 Complétez la ligne de code pour la faire ressembler à ce qui suit :

```
(if (equal ObjectCreationStyle  "COMMAND")
```

4 Ajoutez les lignes suivantes :

```
 (progn
  (setq firstCenterPt(p
```

5 Appuyez sur CTRL + SPACEBAR.

VLISP recherche le mot le plus récent commençant par "p", soit `progn`. Néanmoins, le mot que vous recherchez est `polar`. Si vous maintenez enfoncées les touches CTRL + SPACEBAR, VLISP vérifie toutes les correspondances possibles dans votre code. Au cours de cette recherche, il atteint finalement `polar`.

6 Effacez le code que vous venez d'entrer et qui ne servait qu'à des fins de démonstration.

La fonction Mot entier est également accessible à partir du menu Rechercher de VLISP.

Achèvement d'un mot selon A propos

Si vous avez déjà travaillé avec AutoLISP, vous avez peut-être déjà eu à taper une expression similaire à celle-ci :

```
(setq myEnt (ssname mySelectionSet ssIndex))
```

Souvent, conserver la trace de toutes les fonctions du jeu de sélection peut prêter à confusion : **`ssname`**, **`ssget`**, **`sslength`**, etc. VLISP peut être utile grâce à la fonction Mot selon A propos.

Utilisation de la fonction Mot selon A propos de Visual LISP

1 Placez-vous au bas du fichier *gpdraw.lsp* et entrez le code suivant sur une ligne vide :

```
(setq myEnt (ent
```

2 Appuyez sur CTRL + SHIFT + SPACEBAR.

VLISP affiche une liste de tous les symboles AutoLISP commençant par les lettres *ent*.

Utilisez les touches du curseur (touches fléchées vers le haut et vers le bas) pour vous déplacer dans la liste. Sélectionnez `ENTGET`, puis appuyez sur ENTREE.

VLISP remplace le `ent` que vous avez tapé par `ENTGET`.

3 Effacez le code.

Obtenir de l'aide sur une fonction

Le code qui ajoute une polyligne fine au dessin appelle une fonction nommée **`vla-addLightweightPolyline`**. Non seulement le terme est long à écrire, mais vous utiliserez pour créer des entités plusieurs fonctions dont le nom commence par **`vla-add`**. Plutôt que de consulter un manuel à la recherche des noms de fonctions chaque fois que vous créez un programme, utilisez VLISP.

Pour obtenir de l'aide sur l'utilisation d'une fonction

1. Entrez le code suivant sur une ligne vide :

   ```
   (vla-add
   ```

2. Appuyez sur CTRL + SHIFT + SPACEBAR.
3. Faites défiler la liste à la recherche de `vla-addLightweightPolyline`.
4. Cliquez deux fois sur `vla-addLightweightPolyline`.

   VLISP affiche la boîte de dialogue Service des symboles pour la fonction sélectionnée.

5. Cliquez sur le bouton Aide de la boîte de dialogue Service des symboles. (Pour les fonctions ActiveX, vous serez orientés vers *ActiveX and VBA Reference*.)
6. Effacez les modifications introduites dans *gpdraw.lsp*, car elles n'ont qu'une fonction illustrative. Fermez également les fenêtres Service de symboles et A propos.

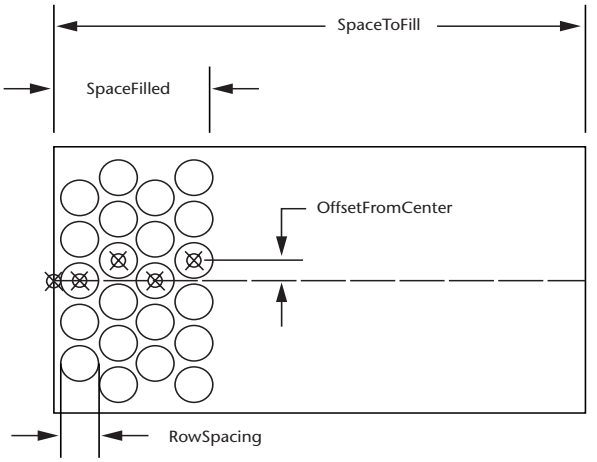
Ajout de dalles au sentier de jardin

Vous disposez à présent du contour du sentier et vous êtes prêt à y placer des dalles. Il vous faudra user d'un peu de logique et de géométrie.

Un peu de logique

Vous devez trouver le moyen d'espacer les dalles et de les dessiner. S'il s'agissait d'un simple quadrillage de dalles rectilignes, vous pourriez utiliser la commande AutoCAD RESEAU pour compléter les dalles. Cependant, dans le cas de ce sentier, vous devez décaler chaque rangée de dalles par rapport à la précédente.

Le décalage des rangées obéit à un modèle répétitif. Pensez à votre manière de procéder si vous posiez réellement les dalles du sentier. Vous choisiriez probablement de commencer à une extrémité et de poser des rangées de dalles jusqu'à ce qu'il n'y ait plus de place libre.



Voici cette logique traduite en pseudo-code :

```
At the starting point of the path
Figure out the initial row offset from center (either centered on
the path or offset by one "tile space").
While the space of the boundary filled is less than the space to
fill,
  Draw a row of tiles.
  Reset the next start point (incremented by one "tile space").
  Add the distance filled by the new row to the amount of space
  rempli.
  Toggle the offset (if it is centered, set it up off-center, or
  vice versa).
  Go back to the start of the loop.
```

Un peu de géométrie

Il vous suffit de connaître quelques cotes pour dessiner le sentier de jardin. La demi-largeur est facile à trouver : c'est tout simplement la moitié de la largeur du sentier. Vous avez déjà défini le code qui vous permet d'obtenir cette largeur auprès des utilisateurs et de l'enregistrer dans une liste associative.

L'espacement des dalles est également facile à obtenir : il équivaut au double du rayon (soit le diamètre) augmenté de l'espace entre les dalles. Les cotes sont également fournies par les utilisateurs.

L'espacement des rangées est un peu plus difficile à trouver, à moins que vous ne maîtrisiez la trigonométrie. La formule est la suivante :

```
Row Spacing = (Tile Diameter + Space between Tiles) * (the sine
                of 60 degrees)
```

Dessin des rangées

Essayez d'analyser la fonction suivante. Comparez-la au pseudo-code et essayez de saisir les calculs géométriques qui viennent d'être décrits. Certaines fonctions AutoLISP peuvent vous être inconnues. Si vous souhaitez obtenir de l'aide sur ces fonctions, reportez-vous à *AutoLISP Reference*. Pour l'instant, contentez-vous de lire le code, sans rien écrire.

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData / PathLength
          TileSpace TileRadius SpaceFilled SpaceToFill
          RowSpacing offsetFromCenter
          rowStartPoint pathWidth pathAngle
          ObjectCreationStyle   TileList)
  (setq PathLength    (cdr (assoc 41 BoundaryData))
        TileSpace     (cdr (assoc 43 BoundaryData))
        TileRadius    (cdr (assoc 42 BoundaryData))
        SpaceToFill   (- PathLength TileRadius)
        RowSpacing    (* (+ TileSpace (* TileRadius 2.0))
                         (sin (Degrees->Radians 60))
                      ) ;_ end of *
        SpaceFilled   RowSpacing
        offsetFromCenter 0.0
        offsetDistance (/ (+ (* TileRadius 2.0) TileSpace) 2.0)
        rowStartPoint  (cdr (assoc 10 BoundaryData))
        pathWidth      (cdr (assoc 40 BoundaryData))
        pathAngle      (cdr (assoc 50 BoundaryData))
        ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData)))
  ) ;_ end of setq

  ;; Compensate for the first call to gp:calculate-Draw-tile Row
  ;; in the loop below.
  (setq rowStartPoint
       (polar rowStartPoint
            (+ pathAngle pi)
            (/ TileRadius 2.0)
       ) ;_ end of polar
  ) ;_ end of setq
  ;; Draw each row of tiles.
  (while (<= SpaceFilled SpaceToFill)
    ;; Get the list of tiles created, adding them to our list.
    (setq tileList   (append tileList
              (gp:calculate-Draw-TileRow
                (setq rowStartPoint
                     (polar rowStartPoint
                            pathAngle
                            RowSpacing
                     ) ;_ end of polar
                ) ;_ end of setq
                TileRadius
                TileSpace
                pathWidth
                pathAngle
                offsetFromCenter
                ObjectCreationStyle
              ) ;_ end of gp:calculate-Draw-TileRow
```

```
            ) ;_ end of append
      ;; Calculate the distance along the path for the next row.
      SpaceFilled   (+ SpaceFilled RowSpacing)
      ;; Alternate between a zero and a positive offset
      ;; (causes alternate rows to be indented).
      offsetFromCenter
            (if (= offsetFromCenter 0.0)
                  offsetDistance
                  0.0
            ) ;_ end of if
    ) ;_ end of setq
  ) ;_ end of while
  ;; Return the list of tiles created.
  tileList
) ;_ end of defun
```

Certaines sections du code peuvent exiger un complément d'explication.

Le fragment de code suivant intervient juste avant le début de la boucle **`while`** :

```
;; Compensate for the very first start point!!
(setq rowStartPoint(polar rowStartPoint
   (+ pathAngle pi)(/ TileRadius 2.0)))
```

La logique de l'algorithme se présente comme un puzzle de trois pièces :

- La variable `rowStartPoint` trouve son origine à l'intérieur des fonctions **`gp:Calculate-et-Draw-Tiles`** lorsque le point de départ du sentier sélectionné par l'utilisateur lui est assigné.
- Le tout premier argument transmis à la fonction **`gp:calculate-Draw-TileRow`** effectue l'opération suivante :
  `(setq rowStartPoint(polar rowStartPoint pathAngle RowSpacing))`

  En d'autres termes : au moment où la fonction **`gp:calculate-Draw-TileRow`** est appelée, la variable `rowStartPoint` prend la valeur d'une unité de distance de `RowSpacing` au-delà du point de départ `rowStartPoint` courant.
- L'argument `rowStartPoint` est utilisé dans **`gp:calculate-Draw-TileRow`** comme point de départ du centre de chaque cercle de la rangée.

Afin de compenser le déplacement en avant initial du point `rowStartPoint` pendant le dessin de la première rangée (soit le premier cycle de la boucle **`while`**), vous devez déplacer légèrement le point `rowStartPoint` dans le sens opposé. Le but est d'éviter l'apparition d'un espace vide entre le contour du sentier et la première rangée. Un déplacement du point de la moitié du rayon des dalles `TileRadius` sera suffisant. Vous pouvez y parvenir en utilisant **`polar`** pour projeter `rowStartPoint` selon un vecteur orienté à 180 degrés de `PathAngle`. Vous remarquerez que ceci situe temporairement le point à l'extérieur du contour du sentier.

Le fragment suivant (modifié pour le rendre plus lisible), peut surprendre un peu :

```
(setq tileList (append tileList
                  (gp:calculate-Draw-TileRow
                    (setq rowStartPoint
                      (polar rowStartPoint pathAngle RowSpacing)
                    ) ;_ end of setq
                    TileRadius TileSpace pathWidth pathAngle
                    offsetFromCenter ObjectCreationStyle
                  )))
```

Nous avons essentiellement **`setq`** qui entoure **`append`** qui entoure l'appel de **`gp:calculate-Draw-TileRow`**.

La fonction **`gp:calculate-Draw-TileRow`** renverra les ID d'objet de chaque dalle dessinée. (L'ID de l'objet pointe vers l'objet dalle dans le dessin.) Vous dessinez les dalles rangée par rangée, de sorte que la fonction renvoie les ID d'objets d'une rangée à la fois. La fonction **`append`** ajoute les nouvelles ID d'objets à toutes les ID d'objets existantes stockées dans `tileList`.

A proximité de la fin de la fonction, vous pouvez trouver le fragment suivant :

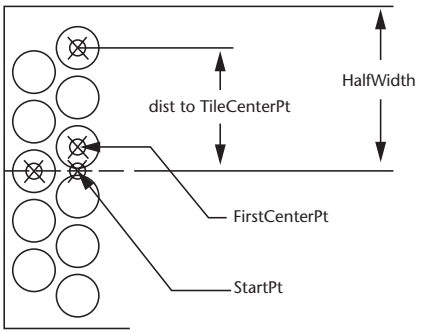
```
(setq offsetFromCenter
  (if (= offsetFromCenter 0.0)
       offsetDistance
       0.0
  )
)
```

Il s'agit de la bascule de décalage qui détermine si la rangée doit commencer par un cercle centré sur le sentier ou décalé par rapport à celui-ci. Voici le pseudo-code de cet algorithme :

```
Set the offset amount to the following:
   If the offset is currently zero, set it to the offset distance;
      Otherwise, set it back to zero.
```

Dessin des dalles d'une rangée

A présent que vous possédez la logique du dessin du sentier, il vous reste à trouver comment dessiner les dalles à l'intérieur de chaque rangée. Le schéma suivant expose deux cas : une rangée pour laquelle le décalage par rapport au centre du sentier est égal à 0.0, et un cas où le décalage est différent de zéro. Examinez le schéma, puis lisez le pseudo-code qui suit.




```
Set up variables for StartPoint, angp90, angm90, and so on.
Set the variable FirstCenterPoint to the StartPoint + offset amount
    (which may be 0.0).
Set the initial value of TileCenterPt to FirstCenterPoint.

(Comment: Begin by drawing the circles in the angp90 direction.)
While the distance from the StartPoint to the TileCenterPt is less
than the HalfWidth:
    Draw a circle (adding to the accumulating list of circles).
    Set TileCenterPt to the next tile space increment in the angp90
    direction.
End While

Reset the TileCenterPoint to the FirstCenterPoint + the tile space
increment at angm90.
While the distance from the StartPoint to the TileCenterPt is less
than the HalfWidth:
    Draw a circle (adding to the accumulating list of circles).
    Set TileCenterPt to the next tile space increment in the angm90
    direction.
End While

Return the list of circles.
```

Examen du code

A présent, examinez le code de la fonction **`gp:calculate-Draw-TileRow`** :

```
(defun gp:calculate-Draw-TileRow (startPoint TileRadius
          TileSpace  pathWidth pathAngle offsetFromCenter
          ObjectCreationStyle / HalfWidth TileDiameter
                        ObjectCreationFunction angp90 angm90
                        firstCenterPt TileCenterPt TileList)
  (setq HalfWidth (- (/ pathWidth 2.00) TileRadius)
     Tilespacing (+ (* TileRadius 2.0) TileSpace)
     TileDiameter (* TileRadius 2.0)
     angp90 (+ PathAngle (Degrees->Radians 90))
     angm90 (- PathAngle (Degrees->Radians 90))
     firstCenterPt (polar startPoint angp90 offsetFromCenter)
     tileCenterPt firstCenterPt
     ObjectCreationStyle(strcase ObjectCreationStyle)
```

```
      ObjectCreationFunction
      (cond
           ((equal ObjectCreationStyle "ACTIVEX")
             gp:Create_activeX_Circle
           )
           ((equal ObjectCreationStyle "ENTMAKE")
             gp:Create_entmake_Circle
           )
           ((equal ObjectCreationStyle "COMMAND")
             gp:Create_command_Circle
           )
           (T
             (alert (strcat "ObjectCreationStyle in function
                    gp:calculate-Draw-TileRow"
                  "\nis invalid. Contact developer for assistance."
                   "\n        ObjectCreationStyle set to ACTIVEX"
                    )
             )
              setq ObjectCreationStyle "ACTIVEX")
           )
       )
      )
      ;; Draw the circles to the left of the center.
      (while (< (distance startPoint tileCenterPt) HalfWidth)
        ;; Add each tile to the list to return.
       (setq tileList
            (cons
            (ObjectCreationFunction tileCenterPt TileRadius)
                  tileList
            )
       )

       ;; Calculate the center point for the next tile.
       (setq tileCenterPt
         (polar tileCenterPt angp90 TileSpacing)
       )
      );_ end of while

      ;; Draw the circles to the right of the center.
      (setq tileCenterPt
        (polar firstCenterPt angm90 TileSpacing))
      (while (< (distance startPoint tileCenterPt) HalfWidth)
         ;; Add each tile to the list to return.
         (setq tileList
              (cons
                 (ObjectCreationFunction tileCenterPt TileRadius)
                   tileList
              )
         )

        ;; Calculate the center point for the next tile.
       (setq tileCenterPt (polar tileCenterPt angm90 TileSpacing))
      );_ end of while

      ;; Return the list of tiles.
      tileList
) ;_ end of defun
```

La logique du code AutoLISP suit le pseudo-code, avec le complément suivant :

```
(setq ObjectCreationFunction
  (cond
     ((equal ObjectCreationStyle "ACTIVEX")
        gp:Create_activeX_Circle
     )
     ((equal ObjectCreationStyle "ENTMAKE")
       gp:Create_entmake_Circle
     )
     ((equal ObjectCreationStyle "COMMAND")
       gp:Create_command_Circle
     )
     (T
       (alert
         (strcat
        "ObjectCreationStyle in function gp:calculate-Draw-TileRow"
           "\nis invalid.  Contact the developer for assistance."
           "\n        ObjectCreationStyle set to ACTIVEX"
         ) ;_ end of strcat
       ) ;_ end of alert
      (setq ObjectCreationStyle "ACTIVEX")
     )
  ) ;_ end of cond

) ;_ end of setq
```

Vous souvenez-vous de la spécification destinée à permettre aux utilisateurs de dessiner les dalles (cercles) aussi bien à l'aide d'ActiveX que de la fonction **`entmake`** ou de la fonction **`command`** ? Une des trois fonctions est assignée à la variable `ObjectCreationFunction`, en fonction du paramètre `ObjectCreationStyle` (transmis à partir de **`C:GPath`** en passant par **`gp:Calculate-and-Draw-Tiles`**). Voici les trois fonctions telles qu'elles seront définies dans *gpdraw.lsp* :

```
(defun gp:Create_activeX_Circle (center radius)
  (vla-addCircle *ModelSpace*
    (vlax-3d-point center) ; convert to ActiveX-compatible 3D point
    radius
  )
) ;_ end of defun

(defun gp:Create_entmake_Circle(center radius)
  (entmake
    (list (cons 0 "CIRCLE") (cons 10 center) (cons 40 radius))
  )
  (vlax-ename->vla-object (entlast))
)

(defun gp:Create_command_Circle(center radius)
  (command "_CIRCLE" center radius)
  (vlax-ename->vla-object (entlast))
)
```

La première fonction dessine un cercle à l'aide d'une fonction ActiveX et renvoie un objet ActiveX.

La seconde fonction dessine un cercle à l'aide d'**`entmake`**. Elle renvoie un nom d'entité converti en objet ActiveX.

La troisième fonction dessine un cercle à l'aide de **`command`**. Elle renvoie aussi un nom d'entité converti en objet ActiveX.

Test du code

Si vous êtes parvenu jusqu'ici, vous avez la possibilité d'utiliser un raccourci.

Pour tester le code

1. Fermez les fenêtres actives dans AutoLISP, y compris les fenêtres de projet qui pourraient être ouvertes.
2. Copiez tout le contenu du répertoire *Tutorial\VisualLISP\Lesson5* dans votre répertoire didacticiel *MyPath*.
3. Ouvrez le fichier de projet *gpath5.prj* en choisissant, dans la barre de menus VLISP, les options Sélectionner un projet ➤ Ouvrir un projet.
4. Chargez les fichiers source du projet.
5. Activez (basculez dans) la fenêtre AutoCAD et lancez la commande **`gpath`** pour exécuter le programme.
6. Exécutez **`gpath`** pour dessiner le sentier à trois reprises, en utilisant à chaque fois une méthode de création d'entité différente. Remarquez-vous une différence de vitesse lors du dessin réalisé avec chaque méthode ?

Résumé de la leçon 5

Vous avez entamé cette leçon en apprenant les fonctions d'édition VLISP qui vous aident à :

- Rechercher les parenthèses dans votre code.
- Rechercher et compléter un nom de fonction.
- Obtenir de l'aide en ligne sur une fonction.

Vous avez terminé la leçon par la construction du code qui dessine les dalles du sentier de jardin. Vous disposez à présent d'un programme qui répond aux exigences énoncées au début de ce didacticiel.

A ce stade, vous avez probablement acquis une expérience suffisante de VLISP pour vous lancer seul dans la programmation. Mais si vous souhaitez vous perfectionner, ce didacticiel comprend encore deux leçons consacrées à l'utilisation des fonctions réacteur et d'autres fonctions avancées de l'environnement VLISP.

Utilisation des réacteurs

6

Dans cette leçon, vous allez vous familiariser avec les réacteurs et apprendre à les associer aux événements et aux entités de dessin. Les réacteurs permettent à votre application d'être avertie par AutoCAD lorsque des événements particuliers surviennent. Si, par exemple, un utilisateur déplace une entité à laquelle votre application a associé un réacteur, votre application en sera avisée. Vous pouvez prévoir le déclenchement d'autres opérations, telles que le déplacement d'autres entités associées à celles que l'utilisateur a déplacées, voire peut-être la mise à jour d'une étiquette de texte qui enregistre des informations de révision concernant la fonction de dessin modifiée. Ceci équivaut, en fait, à munir votre application d'un récepteur de poche et à indiquer à AutoCAD d'appeler l'application quand quelque chose se passe.

Dans ce chapitre

Principes de base des réacteurs

Un réacteur est un objet que vous associez à l'éditeur de dessin lui-même ou à des entités spécifiques à l'intérieur d'un dessin. Poursuivant la métaphore du récepteur de poche, l'objet réacteur est un combiné automatique qui sait comment appeler votre récepteur de poche lorsqu'un événement important survient. Dans votre application, le récepteur de poche est une fonction AutoLISP appelée par le réacteur. Cette fonction est appelée *fonction de rétro-appel*.

REMARQUE La complexité du code d'application et le niveau de compétences requis pour ces deux leçons finales est beaucoup plus élevé que pour les leçons 1 à 5. De nombreuses informations sont présentées, mais elles ne sont pas développées de façon aussi détaillée que dans les leçons précédentes. Si vous êtes débutant, ne vous inquiétez pas si vous échouez à la première tentative. Considérez ceci comme un avant-goût des fonctions très puissantes mais techniquement plus complexes de VLISP.

Types de réacteur

Les types de réacteur AutoCAD sont nombreux. Chaque type de réacteur répond à un ou plusieurs événements d'AutoCAD. Ils sont regroupés dans les catégories suivantes :

Réacteurs d'éditeur	Notifient votre application à chaque fois qu'une commande AutoCAD est appelée.
Réacteurs d'éditeur de liens	Notifient votre application chaque fois qu'une application ObjectARX est chargée ou déchargée.
Réacteurs de base de données	Correspondent à des entités ou à des objets spécifiques à l'intérieur d'une base de données.
Réacteurs de documents	En mode MDI, notifient votre application d'un changement apporté au document de dessin en cours, tel que l'ouverture d'un nouveau document de dessin, l'activation d'une autre fenêtre de document ou la modification de l'état de verrouillage d'un document.
Réacteurs d'objet	Vous informent chaque fois qu'un objet est modifié, copié ou supprimé.

A l'exception des réacteurs d'éditeur, il existe un type de réacteur par catégorie de réacteur. Les réacteurs d'éditeur regroupent une classe étendue de réacteurs : par exemple, les réacteurs DXF™ qui informent une application de l'importation ou de l'exportation d'un fichier DXF, et les réacteurs de souris, qui notifient les événements de la souris, tels que les doubles clics.

Parmi les catégories de réacteurs, il existe de nombreux événements spécifiques auxquels vous pouvez associer un réacteur. AutoCAD permet aux utilisateurs d'effectuer différents types d'actions, et c'est à vous de déterminer les actions qui vous intéressent. Ensuite, vous pouvez associer votre réacteur "combiné automatique" à l'événement, puis écrire la fonction de rétro-appel qui est déclenchée lorsque l'événement survient.

Conception de réacteurs pour le sentier de jardin

Pour mettre en œuvre les fonctions des réacteurs dans l'application Sentier de jardin, commencez par traiter un petit nombre d'événements, sans chercher d'emblée à couvrir toutes les actions accessibles à l'utilisateur.

Sélection d'événements de réacteurs pour le sentier de jardin

Dans le cadre de ce didacticiel, fixez-vous les objectifs suivants :

- Lorsqu'un coin (sommet) du contour du sentier du jardin est déplacé, redessinez le sentier de manière que le contour reste rectiligne. Redessinez également les dalles en fonction de leurs nouvelles taille et forme.
- Lorsque le contour du sentier est effacé, effacez aussi les dalles.

Planification des fonctions de rétro-appel

Pour chaque événement de réacteur, vous devez planifier la fonction qui sera appelée lorsque l'événement aura lieu. Le pseudo-code suivant ébauche la séquence logique des événements qui doivent se produire lorsque les utilisateurs déplacent un des sommets de la polyligne vers une nouvelle position.

```
Defun gp:outline-changed
       Erase the tiles.
       Determine how the boundary changed.
       Straighten up the boundary.
       Redraw new tiles.
End function
```

Nous allons cependant rencontrer une difficulté. Lorsque l'utilisateur commence à déplacer le contour d'un sommet de polyligne, AutoCAD informe votre application en lançant un événement **`:vlr-modified`**. A ce stade, toutefois, l'utilisateur entame seulement le déplacement de l'un des sommets de la polyligne. Si vous appelez immédiatement la fonction **`gp:outline-changed`**, vous interrompez l'action amorcée par l'utilisateur. Vous n'êtes pas en mesure de déterminer l'emplacement du nouveau sommet, parce que l'utilisateur ne l'a pas encore sélectionné. Et finalement, AutoCAD n'autorisera pas votre fonction à modifier l'objet de polyligne tant que l'utilisateur sera occupé à le faire glisser. L'objet de polyligne est d'ailleurs ouvert pour modification dans AutoCAD et le restera jusqu'à ce que l'utilisateur ait terminé de repositionner l'objet.

Vous devez modifier votre approche. Voici la logique mise à jour :

```
When the user begins repositioning a polyline vertex,
  Invoke the gp:outline-changed function
  Defun gp:outline-changed.
    Set a global variable that stores a pointer to the polyline
    being modified by the user.
  End function
When the command completes,
  Invoke the gp:command-ended function
  Defun gp:command-ended.
           Erase the tiles.
         Get information on the previous polyline vertex locations.
           Get information on the new polyline vertex locations.
           Redefine the polyline (straighten it up).
           Redraw the tiles.
  End function
```

Lorsqu'un utilisateur complète la modification d'un contour de sentier, AutoCAD en notifie votre application en lançant un événement **`:vlr-commandEnded`**, pour autant que vous ayez installé un réacteur d'éditeur.

L'utilisation d'une variable globale pour identifier la polyligne modifiée par l'utilisateur s'avère nécessaire, car il n'y a pas de continuité entre les fonctions **`gp:outline-changed`** et **`gp:command-ended`**. Dans votre application, ces deux fonctions sont appelées et exécutées indépendamment l'une de l'autre. La variable globale stocke des informations importantes définies dans une fonction et accessibles dans l'autre.

Voyons maintenant la démarche à suivre si l'utilisateur efface le contour du sentier. L'objectif final est d'effacer toutes les dalles. La logique est ébauchée dans le pseudo-code suivant :

```
When the user begins to erase the boundary,
  Invoke the gp:outline-erased function
  Defun gp:outline-erased.
     Set a global variable that stores a pointer to the reactor
     attached to the polyline currently being erased.
  End function
When the erase is completed,
  Invoke the gp:command-ended function
  Defun gp:command-ended.
     Erase the tiles that belonged to the now-deleted polyline.
  End function
```

Planification de réacteurs multiples

Les utilisateurs peuvent afficher plusieurs sentiers à l'écran et en effacer plusieurs. Vous devez prévoir cette possibilité.

Le réacteur associé à une entité est un réacteur d'objet. Si le dessin contient plusieurs entités, il peut y avoir plusieurs réacteurs d'objet, un par entité. Un événement d'édition spécifique tel que la commande `erase`, peut déclencher de nombreux rétro-appels, en fonction du nombre d'entités sélectionnées auxquelles des réacteurs sont associés. Les réacteurs d'éditeur, en revanche, sont uniques par nature. Votre application doit associer un seul réacteur d'événement **`:vlr-commandEnded`**.

La séquence des événements pour les deux modifications, le déplacement d'un sommet et l'effacement d'une polyligne, débouchent sur des actions qui doivent être effectuées à l'intérieur de la fonction **`gp:command-ended`**. Déterminez le jeu d'actions qui doit être effectué pour chaque condition. La logique est ébauchée dans le pseudo-code suivant :

```
Defun gp:command-ended (2nd version)
  Retrieve the pointer to the polyline (from a global variable).
      Conditional:
        If the polyline has been modified then:
           Erase the tiles.
         Get information on the previous polyline vertex locations.
           Get information on the new polyline vertex locations.
           Redefine the polyline (straighten it up).
           Redraw the tiles.
        End conditional expression.
        If the polyline has been erased then:
           Erase the tiles.
        End conditional expression.
      End Conditional
End function
```

Association des réacteurs

La prochaine étape de la planification d'une application à base de réacteurs consiste à déterminer comment et quand associer les réacteurs. Il vous faut associer deux réacteurs d'objet pour le contour de polyligne des sentiers (un pour répondre à un événement de modification, l'autre pour répondre à l'effacement), et un réacteur d'éditeur destiné à alerter votre application lorsque les utilisateurs achèvent leur modification de la polyligne. Les réacteurs d'objet sont associés à des entités, alors que les réacteurs d'éditeur sont enregistrés dans AutoCAD.

Un autre point doit être pris en considération. Pour recalculer le contour de la polyligne, c'est-à-dire le faire revenir à une forme rectiligne après une modification effectuée par l'utilisateur, vous devez connaître la configuration du sommet avant la modification. Ces informations ne peuvent être établies après la modification de la polyligne. A ce stade, vous pouvez seulement récupérer des informations sur la nouvelle configuration. Comment résoudre ce problème ? Vous pourriez conserver ces informations dans une variable globale, mais cette solution pose un problème majeur. Comme les utilisateurs ont la possibilité de dessiner autant de sentiers qu'ils le souhaitent, chaque sentier exigerait une nouvelle variable globale. Le résultat serait très confus.

Stockage de données dans un réacteur

Vous pouvez résoudre le problème de l'enregistrement de la configuration d'origine en utilisant une autre caractéristique des réacteurs VLISP : la faculté d'enregistrer des données dans un réacteur. La première fois que l'utilisateur dessine un contour de sentier, vous attachez un réacteur au contour, de même que les informations que vous devez enregistrer. Ceci entraîne la modification de la fonction de votre programme principal, **`C:GPath`**, de la manière suivante :

```
Defun C:GPath
   Do everything that is already done in the garden path
   (and don't break anything).

   Attach an object reactor to the polyline using these parameters:
      A pointer to the polyline just drawn,
      A list of data that you want the reactor to record,
      A list of the specific polyline object events to be tracked,
      along with the LISP callback functions to be invoked.
   End of the object reactor setup.

   Attach editor reactor to the drawing editor using the
   following parameters:
      Any data you want attached to the reactor (in this case,none)
      A list of the specific editor reactor events to be tracked,
      along with the LISP callback functions to be invoked.
   End of the editor reactor setup.
End function
```

Mise à jour de la fonction C:GPath

Mettez à jour la fonction **`C:GPath`** en y ajoutant une logique de création de réacteur.

Ajout d'une logique de création de réacteur dans C:GPath

1 Remplacez votre version de *gpmain.lsp* par la version mise à jour reprise ci-dessous. Copiez ce code à partir du *<répertoire AutoCAD>\Tutorial\VisualLISP\lesson6* :

```
(defun C:GPath (/
        gp_PathData
        gp_dialogResults
        PolylineName
        tileList
           )
  (setvar "OSMODE" 0)              ;; Turn off object snaps.
  ;|
  ;; Lesson 6 adds a stubbed-out command reactor to AutoCAD.
  ;; However, it would be undesirable to react to every
  ;; drawing of a circle should the COMMAND tile creation
  ;; method be chosen by the user.  So, disable the
  ;; *commandReactor* in case it exists.
  |;
  (if *commandReactor*
    (progn
      (setq *commandReactor* nil)
      (vlr-remove-all :VLR-Command-Reactor)
    )
  )

  ;; Ask the user for input: first for path location and
  ;; direction, then for path parameters.  Continue only if you
  ;; have valid input.  Store the data in gp_PathData.
  (if (setq gp_PathData (gp:getPointInput))
    (if (setq gp_dialogResults
           (gp:getDialogInput
         (cdr (assoc 40 gp_PathData))
           ) ;_ end of gp:getDialogInput
    ) ;_ end of setq

      (progn
    ;; Now take the results of gp:getPointInput and append this to
    ;; the added information supplied by gp:getDialogInput.

    (setq gp_PathData (append gp_PathData gp_DialogResults))

    ;; At this point, you have all the input from the user.
    ;; Draw the outline, storing the resulting polyline "pointer"
    ;; in the variable called PolylineName.

    (setq PolylineName (gp:drawOutline gp_PathData))

    ;; Next, it is time to draw the tiles within the boundary.
    ;; The gp_tileList contains a list of the object pointers for
```

```
;; the tiles.  By counting up the number of points (using the
;; length function), we can print out the results of how many
;; tiles were drawn.
(princ "\nThe path required ")
(princ
  (length
    (setq tileList (gp:Calculate-and-Draw-Tiles gp_PathData))
  ) ;_ end of length
) ;_ end of princ
(princ " tiles.")

;; Add the list of pointers to the tiles (returned by
;; gp:Calculate-and-Draw-Tiles) to gp_PathData. This will
;; be stored in the reactor data for the reactor attached
;; to the boundary polyline.  With this data, the polyline
;; "knows" what tiles (circles) belong to it.

(setq gp_PathData
       (append (list (cons 100 tileList))
                ; all the tiles
           gp_PathData
       ) ;_ end of append
) ;_ end of setq

;; Before we attach reactor data to an object, let's look at
;; the function vlr-object-reactor.
;; vlr-object-reactor has the following arguments:
;;  (vlr-object-reactor owner's data callbacks)
;;      The callbacks Argument is a list comprised
;;      '(event_name . callback_function).
;;
;; For this exercise we will use all arguments
;; associated with vlr-object-reactor.
;; These reactor functions will execute only if
;; the polyline in PolylineName is modified or erased.

(vlr-object-reactor

  ;; The first argument for vlr-object-reactor is
  ;; the "Owner's List" argument.  This is where to
  ;; place the object to be associated with the
  ;; reactor.  In this case, it is the vlaObject
  ;; stored in PolylineName.

  (list PolylineName)

  ;; The second argument contains the data for the path

  gp_PathData
```

```
      ;; The third argument is the list of specific reactor
      ;; types that we are interested in using.
      '
       (
        ;; reactor that is called upon modification of the object.
        (:vlr-modified . gp:outline-changed)
        ;; reactor that is called upon erasure of the object.
        (:vlr-erased . gp:outline-erased)
       )
    ) ;_ end of vlr-object-reactor

    ;; Next, register a command reactor to adjust the polyline
    ;; when the changing command is finished.
    (if (not *commandReactor*)
      (setq *commandReactor*
         (VLR-Command-Reactor
         nil        ; No data is associated with the command reactor
            '(
              (:vlr-commandWillStart . gp:command-will-start)
              (:vlr-commandEnded . gp:command-ended)
             )
         ) ;_ end of vlr-command-reactor
      )
    )

  ;; The following code removes all reactors when the drawing is
  ;; closed. This is extremely important!!!!!!!!!
  ;; Without this notification, AutoCAD may crash upon exiting!
  (if (not *DrawingReactor*)
       (setq *DrawingReactor*
          (VLR-DWG-Reactor
          nil        ; No data is associated with the drawing reactor
            '((:vlr-beginClose . gp:clean-all-reactors)
             )
          ) ;_ end of vlr-DWG-reactor
      )
  )
       ) ;_ end of progn
       (princ "\nFunction cancelled.")
     ) ;_ end of if
     (princ "\nIncomplete information to draw a boundary.")
   ) ;_ end of if
   (princ)               ; exit quietly
  ) ;_ end of defun

  ;;; Display a message to let the user know the command name.
  (princ "\nType GPATH to draw a garden path.")
  (princ)
```

2 Révisez les modifications du code et les commentaires décrivant le rôle de chaque nouvelle instruction Le didacticiel affiche tout le code modifié en gras.

Ajout de fonctions de rétro-appel de réacteurs

La fonction de rétro-appel des réacteurs augmente considérablement la quantité de code de votre application. Ce code se trouve dans le répertoire *Lesson6*.

Ajout à votre programme des fonctions de rétro-appel des réacteurs

1. Copiez le fichier *gpreact.lsp* à partir du répertoire *Tutorial\VisualLISP\Lesson6* dans votre répertoire de travail *MyPath*.

2. Ouvrez le projet *GPath* (si ce n'est déjà fait) et cliquez sur le bouton Propriétés du projet de la fenêtre du projet gpath.
3. Ajoutez le fichier *gpreact.lsp* à votre projet.
4. Le fichier *gpreact.lsp* peut se situer n'importe où dans l'ordre des fichiers entre *utils.lsp*, qui doit rester en première place, et *gpmain.lsp*, qui doit se trouver à la fin. Au besoin, déplacez certains fichiers, puis cliquez sur OK.
5. Ouvrez le fichier *gpreact.lsp* en cliquant deux fois sur son nom dans la fenêtre du projet gpath.

   Lisez les commentaires du fichier pour mieux comprendre les opérations qu'il effectue. Remarquez que les fonctions de rétro-appel sont tronquées. Leur seule fonction est d'afficher des messages d'alerte lorsqu'elles sont lancées.

   La dernière fonction du fichier est d'une importance telle qu'elle mérite un titre à elle seule.

Nettoyage des réacteurs

Les réacteurs sont extrêmement actifs. Si vous concevez une application ayant recours à eux, vous risquez de passer beaucoup de temps à réparer les pannes de votre programme et même d'AutoCAD. Il peut être utile de disposer d'un outil permettant de supprimer, en cas de nécessité, tous les réacteurs que vous avez ajoutés.

Le fichier *gpreact.lsp* contient une fonction **`gp:clean-all-reactors`** qui n'a pas de réelle action intrinsèque. En revanche, elle appelle la fonction **`CleanReactors`**. Ajoutez cette fonction à votre fichier *utils.lsp* en copiant le code suivant à la fin du fichier :

```
;;;-------------------------------------------------------------;
;;;     Function: CleanReactors                                 ;
;;;-------------------------------------------------------------;
;;;  Description: General utility function used for cleaning up  ;
;;;               reactors. It can be used during debugging, as  ;
;;;               well as cleaning up any open reactors before   ;
;;;               a drawing is closed.                           ;
;;;-------------------------------------------------------------;
(defun CleanReactors ()
  (mapcar 'vlr-remove-all
          '(:VLR-AcDb-reactor
            :VLR-Editor-reactor
            :VLR-Linker-reactor
            :VLR-Object-reactor
           )
  )
)
```

Test de vos réacteurs

Vous disposez à présent de tous les éléments pour utiliser quelques réacteurs en situation réelle.

Pour tester le code du réacteur

1. Chargez tout le code source de votre projet. (Cliquez sur le bouton Charger les fichiers source dans la fenêtre du projet gpath.)
2. Lancez la fonction `C:GPath` et essayez-la.

   Le programme dessinera un sentier, comme dans la leçon 5. Vous ne remarquerez rien d'intéressant à première vue.
3. Essayez d'effectuer les actions suivantes après avoir dessiné le sentier :

   - Déplacez un sommet de polyligne. Sélectionnez la polyligne et activez-en les poignées, puis faites glisser le sommet vers un nouvel emplacement.
   - Allongez la polyligne.
   - Déplacez la polyligne.
   - Effacez la polyligne.

   Examinez les messages qui s'affichent. Vous observez les activités d'arrière-plan d'une fonction puissante.

   (Si votre application ne fonctionne pas correctement et vous ne voulez pas prendre le temps de la dépanner immédiatement, vous pouvez exécuter l'exemple de code fourni dans le répertoire *Tutorial\VisualLISP\Lesson6*. Utilisez le projet *Gpath6* dans ce répertoire.)

REMARQUE Compte tenu du comportement des réacteurs, vous constaterez peut-être qu'après avoir testé une séquence de réacteurs dans AutoCAD, vous ne pouvez pas revenir dans VLISP en appuyant sur ALT+TAB, ou en cliquant pour activer la fenêtre VLISP. En pareil cas, entrez simplement **`vlisp`** sur la ligne de commande d'AutoCAD pour retourner dans VLISP.

Examen détaillé du comportement des réacteurs

Sur un bloc de papier, faites le relevé des événements des réacteurs à l'intérieur de l'application. Voici un exemple de ce que vous devez rechercher :

Command: erase Object: Polyline border
Reactor sequence
1. Type in erase command 2. Callback: GP:COMMAND-WILL-START 3. Select objects: pick a polyline 4. Hit Enter (object selection complete) 5. Callback: GP:OUTLINE-ERASED 6. Callback: GP:OUTLINE-CHANGED 7. Callback: GP:COMMAND-ENDED

Dessinez dix sentiers, puis relevez les combinaisons Option/Objet suivantes en sélectionnant tour à tour les polylignes :

- Effacer/contour de la polyligne (sentier 1).
- Effacer/cercle dans une polyligne (sentier 2).
- Effacer/deux polylignes (sentiers 3 et 4).
- Déplacer/contour de la polyligne (sentier 5).
- Déplacer/cercle dans une polyligne (sentier 6).
- Déplacer/deux polylignes et plusieurs cercles (sentiers 7 et 8).
- Déplacer sommet (à l'aide des poignées)/contour de la polyligne (sentier 9).
- Allonger/contour de la polyligne (sentier 10).

Cet exercice vous permettra de bien comprendre ce qui se passe en arrière-plan. Au cours de la leçon 7, lorsque la fonction des réacteurs deviendra difficile à comprendre, vous pourrez consulter vos "feuilles de repérage des réacteurs".

Résumé de la leçon 6

Au cours de cette leçon, vous vous êtes familiarisé avec les réacteurs AutoCAD et vous avez appris à les mettre en œuvre à l'aide de VLISP. Vous avez conçu un plan pour ajouter des réacteurs à votre application de sentier de jardin et vous avez ajouté une partie du code nécessaire à votre programme pour mettre ce plan en œuvre.

Vous venez d'aborder des sujets nouveaux et particulièrement intéressants du point de vue d'AutoLISP. Les réacteurs sont en mesure d'ajouter de nombreuses fonctions à une application, mais n'oubliez pas ceci : plus un programme est puissant, et plus ses pannes sont importantes.

D'autre part, compte tenu de la manière dont votre application est conçue, les fonctions des réacteurs ne se conserveront pas d'une session de dessin à l'autre. Si vous enregistrez un dessin qui contient un sentier de jardin associé à des réacteurs, ceux-ci seront absents la prochaine fois que vous ouvrirez le dessin. Vous pouvez obtenir des informations sur l'ajout de réacteurs permanents en vous reportant à la rubrique “Transient versus Persistent Reactors” du *Visual LISP Developer's Guide*, puis en lisant les commentaires concernant les références aux fonctions dans *AutoLISP Reference*.

Assemblage

7

Dans la leçon 6, vous avez été initié aux mécanismes fondamentaux qui sous-tendent les applications comportant des réacteurs. La leçon 7 vous permettra d'étendre ces connaissances à d'autres fonctions et de créer un sentier de jardin qui sait quand et comment se modifier lui-même. Après avoir testé votre application et constaté qu'elle fonctionne correctement, vous allez créer une application VLISP à partir de votre projet VLISP.

Considérez cette partie comme la section avancée du didacticiel. Si vous êtes débutant, vous n'êtes probablement pas en mesure de comprendre tout le code AutoLISP présenté ici. Vous trouverez à la fin de la leçon les références de plusieurs livres AutoLISP qui vous fourniront des informations plus détaillées concernant les concepts avancés d'AutoLISP qui sont présentés ici.

Dans ce chapitre

Planification de toute la procédure des réacteurs

Vous devrez définir plusieurs fonctions dans cette leçon. Plutôt que de vous donner tous les détails du nouveau code, cette leçon présente une vue d'ensemble et indique les concepts sur lesquels repose le code. A la fin de la leçon, vous disposerez du code source nécessaire à la création d'une application sentier de jardin identique à l'exemple de programme que vous avez exécuté dans la leçon 1.

REMARQUE Lorsque vous développez ou déboguez des applications à base de réacteurs, vous courez toujours le risque de rendre AutoCAD instable. Ceci peut être lié à plusieurs causes, comme par exemple, oublier de retirer un réacteur des entités supprimées. Pour cette raison, nous vous recommandons, avant d'aborder la leçon 7, de fermer VLISP, d'enregistrer par la même occasion tous les fichiers ouverts, de quitter AutoCAD, puis de redémarrer les deux applications.

Commencez par charger le projet tel qu'il se présentait à la fin de la leçon 6.

Deux tâches évidentes doivent encore être effectuées dans l'application Sentier de jardin.

- La rédaction des rétro-appels des réacteurs d'objet
- La rédaction des rétro-appels des réacteurs d'éditeur

Vous devez aussi envisager la manière de traiter les variables globales de votre programme. Il est souvent souhaitable de conserver une valeur dans une variable globale au cours d'une session de dessin AutoCAD. Dans le cas des réacteurs, ce n'est toutefois pas le cas. A titre d'exemple, supposons qu'un utilisateur de votre application de sentier de jardin ait dessiné plusieurs sentiers dans un seul dessin. Après quoi, l'utilisateur les efface, tout d'abord un par un, puis deux à la fois, etc., jusqu'à ce que tous les sentiers soient effacés à l'exception d'un seul.

La leçon 5 introduit la variable globale `*reactorsToRemove*`, chargée de stocker les pointeurs des réacteurs des polylignes à effacer. Lorsque `*reactorsToRemove*` est déclarée dans **`gp:outline-erased`**, l'événement vous permet de déterminer que la polyligne est sur le pont d'être supprimée. La polyligne n'est réellement éliminée qu'au moment où l'événement **`gp:command-ended`** survient.

La première fois que l'utilisateur efface une polyligne, vous obtenez le résultat escompté. Dans **gp:outline-erased**, vous stockez un pointeur désignant le réacteur. En lançant **gp:command-ended**, vous supprimez les dalles associées à la polyligne à laquelle le réacteur se rattache. Ensuite, l'utilisateur décide d'effacer deux sentiers. Votre application recevra donc deux appels de **gp:outline-erased**, un pour chacune des polylignes sur le point d'être supprimées. Deux problèmes risquent de se poser :

- Lorsque vous définissez la variable `*reactorsToRemove*` à l'aide de **setq**, vous devez ajouter un pointeur de réacteur à la variable globale, en veillant à ne pas écraser les valeurs qui y sont stockées. Ceci signifie que `*reactorsToRemove*` doit être une structure de liste pour que vous puissiez lui ajouter des pointeurs de réacteurs. Vous pouvez alors accumuler plusieurs pointeurs de réacteurs correspondant au nombre de sentiers effacés par l'utilisateur dans une seule commande d'effacement.
- A chaque déclenchement de **gp:command-will-start**, indiquant le début d'une nouvelle séquence de commande, vous devez réinitialiser la variable `*reactorsToRemove*` sur `nil`. Ceci est nécessaire pour éviter que la variable globale n'enregistre les pointeurs de réacteurs de la commande d'effacement qui précède.

  Si vous ne réinitialisez pas la variable globale ou si vous n'utilisez pas la structure de données correcte (en l'occurrence, une liste), vous obtiendrez un comportement imprévu. Dans le cas des réacteurs, un comportement imprévu peut être fatal à votre session AutoCAD.

Voici la suite des événements qui doivent s'enchaîner pour que les utilisateurs effacent deux sentiers à l'aide d'une seule commande d'effacement. Notez le traitement des variables globales :

- Lancez la commande **erase**. Ceci déclenche la fonction **gp:command-will-start**. Définissez `*reactorsToRemove*` sur `zéro`.
- Sélectionnez deux polylignes. Votre application n'est pas encore notifiée.
- Appuyez sur ENTREE pour effacer les deux polylignes sélectionnées. Votre application reçoit un rétro-appel de **gp:outline-erased** pour une des polylignes. Ajoutez le pointeur du réacteur à la variable globale nulle, `*reactorsToRemove*`.
  Votre application reçoit un rétro-appel de **gp:outline-erased** pour la deuxième polyligne. Ajoutez son pointeur de réacteur à la variable globale `*reactorsToRemove*` qui contient déjà le premier pointeur de réacteur.
- AutoCAD supprime les polylignes.
- Votre fonction de rétro-appel **gp:command-ended** est lancée. Eliminez toutes les dalles associées aux pointeurs enregistrés dans `*reactorsToRemove*`.

Outre la variable globale `*reactorsToRemove*`, votre application contient aussi la variable globale `*polyToChange*`, qui enregistre un pointeur vers la polyligne à modifier. Deux variables globales supplémentaires destinées à cette application seront présentées plus tard au cours de cette leçon.

Réaction à d'autres commandes appelées par l'utilisateur

Lors de la rédaction d'une application comportant des réacteurs, vous devez traiter toutes les commandes qui ont une incidence sur vos objets d'une manière significative. Une de vos tâches, au cours de la conception d'un programme, consiste à passer en revue toutes les commandes d'édition d'AutoCAD et à déterminer de quelle manière votre application doit répondre à chacune d'elles. Le format des feuilles de repérage des réacteurs, tel que nous l'avons indiqué à la fin de la leçon 6, est très utile à ce propos. Appelez les commandes que l'utilisateur est susceptible d'employer et indiquez par écrit le type de réponse que votre application doit apporter. Voici les autres actions à planifier :

- Déterminer l'action à exécuter lorsque les utilisateurs lancent des commandes ANNULER et RETABLIR.
- Déterminer l'action à exécuter lorsque les utilisateurs lancent la commande REPRISE après avoir supprimé des entités liées aux réacteurs.

Pour éviter que ce sujet déjà complexe ne devienne vraiment *très* compliqué, le didacticiel ne prétend pas couvrir toutes les possibilités qui devraient être abordées, et les aspects fonctionnels de cette leçon sont réduits au strict minimum.

Même sans entreprendre la construction de tout l'édifice fonctionnel de ces nouvelles commandes, examinez ce que quelques fonctions d'édition supplémentaires vous obligeraient à faire :

- Si les utilisateurs étirent un contour de polyligne (à l'aide de la commande ETIRER), plusieurs cas sont envisageables. Le contour peut être étiré dans n'importe quelle direction, et non seulement en suivant le grand ou le petit axe, ce qui peut lui conférer une forme inhabituelle. Vous devez considérer en outre le nombre de sommets qui ont été modifiés. Une situation dans laquelle un sommet est étiré donnerait une polyligne bien différente de celle dans laquelle deux sommets seraient déplacés. Dans tous les cas, les dalles doivent être supprimées et les nouvelles positions devront être recalculées dès que les ajustements du contour auront été définis.

- Si les utilisateurs déplacent un contour de polyligne, toutes les dalles doivent être supprimées, puis redessinées à leur nouvel emplacement. C'est une opération assez simple, parce que le contour de la polyligne n'a changé ni de taille ni de forme.
- Si les utilisateurs modifient l'échelle d'un contour de polyligne, vous devez faire un choix. Les dalles seront-elles également mises à l'échelle afin que le sentier comporte le même nombre de dalles ? Ou les dalles conserveront-elles la même taille, ce qui amènera l'application à ajouter ou à supprimer des dalles selon que la polyligne s'agrandit ou se réduit ?
- Si les utilisateurs font pivoter un contour de polyligne, toutes les dalles doivent être supprimées, puis redessinées selon leur nouvelle orientation.

Toutefois, pour commencer, contentez-vous de planifier ce qui suit :

- Avertir l'utilisateur au départ que la commande sélectionnée (`étirement`, `déplacement`, ou `rotation`) aura des effets négatifs sur le sentier de jardin.
- Si l'utilisateur confirme l'exécution de la commande, supprimer les dalles et ne pas les redessiner.
- Supprimer les réacteurs du contour du sentier.

REMARQUE Outre les commandes AutoCAD appelées par l'utilisateur, les applications AutoLISP et ObjectARX peuvent également modifier ou supprimer des entités. L'exemple fourni dans le didacticiel Sentier de jardin n'aborde pas les manipulations programmées du contour de polyligne telles que (**`entdel`** <polyline entity>). Dans ce cas, les événements des réacteurs d'éditeur `:vlr-commandWillStart` et `:vlr-commandEnded` ne seront pas déclenchés.

Enregistrement des informations dans les réacteurs

Le type d'informations à associer à l'objet réacteur créé pour chaque entité de polyligne est un autre aspect important de cette application. Dans la leçon 6, vous avez ajouté le code qui associe le contenu de **`gp_PathData`** (la liste associative) au réacteur. Vous avez étendu les données contenues dans **`gp_PathData`** en ajoutant un nouveau champ de saisie (100) à la liste associative. Cette nouvelle sous-liste est une liste de pointeurs vers toutes les entités cercle attribuées à un contour de polyligne.

En raison du travail nécessaire pour recalculer le contour de la polyligne, quatre valeurs clés doivent être ajoutées à **`gp_pathData`** :

```
;;; StartingPoint                                                ;
;;; (12 . BottomStartingPoint)     15------------------------14   ;
;;; (15 . TopStartingPoint)         |                          |   ;
;;; EndingPoint                     10    ----pathAngle--->   11   ;
;;; (13 .  BottomEndingPoint)       |                          |   ;
;;; (14 . TopEndingPoint)          12------------------------13   ;
;;;                                                              ;
```

Ces points sont nécessaires pour le calcul du nouveau contour de polyligne chaque fois que l'utilisateur fait glisser une poignée de coin vers un nouvel emplacement. Ces informations existent déjà dans la fonction **`gp:drawOutline`** contenue dans *gpdraw.lsp*. Examinez la valeur renvoyée par la fonction. Actuellement, seul le pointeur de l'objet polyligne est renvoyé. Vous devez donc procéder en trois étapes.

- Assembler les points du périmètre dans le format requis.
- Modifier la fonction de façon qu'elle renvoie les listes de points du périmètre et le pointeur vers la polyligne.
- Modifier la fonction **`C:GPath`** afin qu'elle gère correctement le nouveau format des valeurs renvoyées par **`gp:drawOutline`**.

L'assemblage des listes de points du périmètre est simple. Examinez le code de **`gp:drawOutline`**. La variable locale `p1` correspond à la valeur clé 12, `p2` à la valeur clé 13, `p3` à la valeur clé 14, et `p4` à la valeur clé 15. Vous pouvez ajouter l'appel de fonction suivant pour assembler cette information :

```
(setq polyPoints(list
  (cons 12 p1)
  (cons 13 p2)
  (cons 14 p3)
  (cons 15 p4)
))
```

Modifier la fonction afin qu'elle renvoie les points du périmètre de la polyligne et le pointeur de polyligne est aussi une opération facile. Comme dernière expression de **`gp:drawOutline`**, assemblez une liste des deux éléments d'information que vous souhaitez renvoyer.

```
(list pline polyPoints)
```

Ajout d'une logique de programmation pour enregistrer les points du périmètre de polyligne

1 Modifiez **`gp:drawOutline`** en effectuant les changements indiqués en gras dans le code suivant (n'oubliez pas d'ajouter la variable locale `polyPoints` à l'instruction **`defun`**) :

```
(defun gp:drawOutline (BoundaryData        /       PathAngle
               Width       HalfWidth   StartPt     PathLength
               angm90      angp90      p1      p2
               p3      p4          poly2Dpoints
               poly3Dpoints        plineStyle  pline
               polyPoints
               )
  ;; extract the values from the list BoundaryData.
  (setq PathAngle    (cdr (assoc 50 BoundaryData))
    Width        (cdr (assoc 40 BoundaryData))
    HalfWidth    (/ Width 2.00)
    StartPt      (cdr (assoc 10 BoundaryData))
    PathLength   (cdr (assoc 41 BoundaryData))
    angp90       (+ PathAngle (Degrees->Radians 90))
    angm90       (- PathAngle (Degrees->Radians 90))
    p1       (polar StartPt angm90 HalfWidth)
    p2       (polar p1 PathAngle PathLength)
    p3       (polar p2 angp90 Width)
    p4      (polar p3 (+ PathAngle

                (Degrees->Radians 180)) PathLength)
    poly2Dpoints (apply 'append
                (mapcar '3dPoint->2dPoint (list p1 p2 p3 p4))
                )
    poly3Dpoints (mapcar 'float (append p1 p2 p3 p4))

    ;; get the polyline style.

    plineStyle   (strcase (cdr (assoc 4 BoundaryData)))

    ;; Add polyline to the model space using ActiveX automation.
    pline        (if (= plineStyle "LIGHT")
                ;; create a lightweight polyline.
                (vla-addLightweightPolyline
              *ModelSpace*   ; Global Definition for Model Space
              (gp:list->variantArray poly2Dpoints)
                    ;data conversion
              ) ;_ end of vla-addLightweightPolyline
               ;; or create a regular polyline.
                (vla-addPolyline
              *ModelSpace*
              (gp:list->variantArray poly3Dpoints)
                    ;data conversion
              ) ;_ end of vla-addPolyline
                ) ;_ end of if
    polyPoints   (list
                (cons 12 p1)
                (cons 13 p2)
                (cons 14 p3)
                (cons 15 p4)
                )
    ) ;_ end of setq
  (vla-put-closed pline T)

  (list pline polyPoints)
) ;_ end of defun
```

2 Modifiez la fonction **C:GPath** (dans *gpmain.lsp*). Recherchez la ligne de code se présentant de la manière suivante :

```
(setq PolylineName (gp:drawOutline gp_PathData))
```

Modifiez-la pour lui donner l'aspect suivant :

```
(setq PolylineList     (gp:drawOutline gp_PathData)
      PolylineName     (car PolylineList)
      gp_pathData      (append gp_pathData (cadr PolylineList))
 ) ;_ end of setq
```

La variable `gp_PathData` contient à présent toutes les informations requises par la fonction du réacteur.

3 Ajoutez `PolylineList` à la section variables locales de la définition de la fonction **C:GPath**.

Ajout d'une nouvelle fonctionnalité de réacteur

Dans la leçon 6, vous avez raccordé la fonction de rétro-appel **gp:command-will-start** à l'événement de réacteur `:vlr-commandWillStart`. Telle qu'elle est actuellement, la fonction affiche certains messages et donne à deux variables globales, `*polyToChange*` et `*reactorsToRemove*`, la valeur initiale `nil`.

Ajout de fonctionnalité à la fonction de rétro-appel gp:command-will-start

1 Ouvrez votre fichier *gpreact.lsp*.

2 Dans la fonction **gp:command-will-start**, ajoutez deux variables à l'appel de fonction **setq** en le modifiant comme suit :

```
;; Reset all four reactor globals to nil.
(setq   *lostAssociativity* nil
        *polyToChange* nil
        *reactorsToChange* nil
        *reactorsToRemove* nil)
```

3 Remplacez le code restant dans **gp:command-will-start**, jusqu'au dernier appel de fonction **princ**, par le code suivant :

```
(if (member (setq currentCommandName (car command-list))
       '( "U"    "UNDO"     "STRETCH"   "MOVE"
          "ROTATE"  "SCALE"   "BREAK"  "GRIP_MOVE"
          "GRIP_ROTATE"   "GRIP_SCALE"  "GRIP_MIRROR")
    ) ;_ end of member
    (progn
      (setq *lostAssociativity* T)
      (princ "\nNOTE: The ")
      (princ currentCommandName)
      (princ " command will break a path's associativity .")
    ) ;_ end of progn
) ;_ end of if
```

Ce code vérifie si l'utilisateur a lancé une commande rompant l'associativité entre les dalles et le sentier. Si l'utilisateur a lancé une telle commande, le programme définit la variable globale `*lostAssociativity*` et avertit l'utilisateur.

A mesure que vous vous entraînez sur l'application sentier de jardin, vous découvrirez des commandes d'édition supplémentaires capables de modifier le sentier et de provoquer la perte de l'associativité. Ajoutez ces commandes à la liste entre guillemets pour que l'utilisateur soit informé de ce qui se produira. Lorsque cette fonction se déclenche, l'utilisateur a lancé une commande, mais n'a pas sélectionné des entités à modifier. L'utilisateur peut encore annuler la commande, n'entraînant ainsi aucun changement.

Ajout d'activité aux fonctions de rétro-appel des réacteurs d'objets

Dans la leçon 6, vous avez enregistré deux fonctions de rétro-appel contenant des événements de réacteurs d'objets. La fonction **`gp:outline-erased`** était associée à l'événement de réacteur `:vlr-erased`, et **`gp:outline-changed`** était associée à l'événement `:vlr-modified`. Vous devez faire en sorte que ces fonctions remplissent leur rôle.

Obtenir que les fonctions de rétro-appel des réacteurs d'objets remplissent leur rôle.

1 Dans *gpreact.lsp*, modifiez **`gp:outline-erased`** pour lui donner l'aspect suivant :

```
(defun gp:outline-erased (outlinePoly reactor parameterList)
  (setq *reactorsToRemove*
       (cons reactor *reactorsToRemove*))
  (princ)
) ;_ end of defun
```

Dans ce cas, une seule opération est exécutée. Le réacteur associé à la polyligne est enregistré dans la liste de tous les réacteurs à supprimer. (Rappel : bien que les réacteurs soient associés à des entités, ils constituent des objets entièrement séparés, et leurs relations avec des entités doivent être gérées avec le même soin que les entités AutoCAD habituelles.)

2 Modifiez **gp:outline-changed** pour faire apparaître le code suivant :

```
(defun gp:outline-changed (outlinePoly reactor parameterList)
  (if *lostAssociativity*
    (setq *reactorsToRemove*
          (cons reactor *reactorsToRemove*))
    (setq *polytochange*     outlinePoly
          *reactorsToChange* (cons reactor *reactorsToChange*))
    )
  (princ)
)
```

Deux catégories de fonctions peuvent modifier le contour de la polyligne. La première catégorie contient les commandes qui rompront l'associativité du sentier et des dalles. Vous avez recherché cette condition dans **gp:command-will-start** et définit la variable globale `*lostAssociativity*` en conséquence. Dans ce cas, les dalles doivent être supprimées et le sentier se trouve aux mains de l'utilisateur. L'autre catégorie est le mode Poignées de la commande ETIRER, où l'associativité est conservée et où vous devez redresser le contour dès que l'utilisateur a fait glisser un sommet vers un nouvel emplacement.

La variable `*polyToChange*` enregistre un pointeur d'objet VLA vers la polyligne elle-même. Ceci servira dans la fonction **gp:command-ended**, au moment de recalculer la bordure de la polyligne.

Conception de la fonction de rétro-appel gp:command-ended

C'est au niveau de la fonction de rétro-appel du réacteur d'éditeur **gp:command-ended** que la plupart des phénomènes interviennent. Jusqu'à l'appel de cette fonction, les polylignes du contour du sentier de jardin sont *"ouvert en écriture"*, ce qui signifie que les utilisateurs sont toujours en mesure de manipuler les bordures dans AutoCAD. Dans la séquence des réacteurs, vous devez attendre qu'AutoCAD ait accompli sa part du travail avant de pouvoir agir.

Le pseudo-code suivant illustre la logique de la fonction **gp:command-ended** :

```
Determine the condition of the polyline.
  CONDITION 1 - POLYLINE ERASED (Erase command)
    Erase the tiles.
  CONDITION 2 - LOST ASSOCIATIVITY (Move, Rotate, etc.)
    Erase the tiles.
  CONDITION 3 - GRIP_STRETCH - REDRAW AND RE-TILE
    Erase the tiles.
    Get the current boundary data from the polyline.
    If it is a lightweight polyline,
       Process boundary data as 2D
    Else
       Process boundary data as 3D
    End if
  Redefine the polyline border (pass in parameters of the current
         boundary configuration, as well as the old).
  Get the new boundary information and put it into the format
         required for setting back into the polyline entity.
  Regenerate the polyline.
  Redraw the tiles (force ActiveX drawing).
  Put the revised boundary information back into the reactor
         named in *reactorsToChange*.
End function
```

Ce pseudo-code est assez simple, mais il renferme plusieurs détails importants, dont certains que vous n'êtes pas censé connaître à ce stade.

Gestion de types d'entités multiples

Premièrement, votre application peut dessiner deux types de polyligne : old-style et fine. Ces différents types de polyligne renvoient leurs données d'entité sous des formats différents. La polyligne old-style renvoie une liste de douze nombres réels : quatre jeux de points *X, Y,* et *Z*. La polyline fine, en revanche, renvoie une liste de huit nombres réels : quatre jeux de points *X* et *Y*.

Vous devez procéder à un certain nombre de calculs pour définir le contour de polyligne révisé après le déplacement d'un des sommets par un utilisateur. Ces calculs seront considérablement plus faciles à réaliser si les données de polyligne ont le même format.

La version donnée par la leçon 7 du fichier *utils.lsp* contient des fonctions permettant d'assurer la conversion de format : **`xyzList->ListOfPoints`** extrait et formate les listes de points 3D en une liste de listes, tandis que **`xyList->ListOfPoints`** extrait et formate les listes de points 2D en une liste de listes.

Ajout de code pour la conversion des données de polyligne en un même format

1. Si vous avez une copie de *utils.lsp* ouverte dans une fenêtre de l'éditeur de texte VLISP, fermez-la.
2. Copiez dans votre répertoire de travail la version de *utils.lsp* contenue dans le répertoire *Tutorial\VisualLISP\Lesson7*.

   Outre les deux fonctions qui reformatent les données des polylignes, *utils.lsp* contient d'autres fonctions utilitaires nécessaires à la gestion des modifications apportées par l'utilisateur au sentier de jardin.
3. Ouvrez *utils.lsp* dans une fenêtre de l'éditeur de texte VLISP et revoyez le nouveau code.

Utilisation des méthodes ActiveX dans les fonctions de rétro-appel de réacteurs

Le second détail intéressant du pseudo-code apparaît près de la fin, au stade de la régénération des dalles. L'instruction du pseudo-code est la suivante :

```
Redraw the tiles (force ActiveX drawing)
```

La phrase entre parenthèses est explicite : forcer le dessin par ActiveX. Pourquoi ceci est-il nécessaire ? Pourquoi l'application ne peut-elle utiliser la préférence de création d'objet enregistrée dans la sous-liste associative ?

Parce que vous ne pouvez utiliser la fonction **`command`** pour la création d'entités à l'intérieur d'une fonction de rétro-appel de réacteur. Ceci est lié à certaines opérations internes d'AutoCAD. Vous devez forcer la routine de dessin des dalles pour utiliser ActiveX. Ce problème sera à nouveau abordé ultérieurement dans cette leçon.

Gestion des séquences de réacteurs non linéaires

Le dernier détail important se rapporte à une particularité de la séquence commande/réacteur dans AutoCAD, lorsque les utilisateurs modifient une polyligne en se servant des commandes spécialisées GRIP. Ces commandes, telles que GRIP_MOVE et GRIP_ROTATE, sont accessibles par un menu contextuel au moyen d'un clic droit après la sélection des poignées d'un objet. La séquence du réacteur n'est pas aussi linéaire qu'une simple commande DEPLACER ou EFFACER. En effet, l'utilisateur passe à une commande différente alors qu'une commande est en cours. Pour illustrer cette situation, vous pouvez charger le code de la leçon 6 qui retrace la séquence des événements de réacteur. Ou simplement examiner le résultat de cette fenêtre de console VLISP pour comprendre le processus :

```
;; To start, select the polyline and some of the circles by using a
;; crossing selection box. The items in the selection set--
;; the chosen circles and the polyline--are now shown with grips on.
;; To initiate the sequence, click on one of the polyline grips:

(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_STRETCH))

;; Now change the command to a move by right-clicking and choosing
;; MOVE from the pop-up menu.  Notice that the command-ended
;; reactor fires in order to close out the GRIP_STRETCH command
;; without having fired an object reactor event:

(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_STRETCH))
(GP:COMMAND-WILL-START #<VLR-Command-reactor> (GRIP_MOVE))

;; Now drag the outline (and the selected circles) to a new location.

(GP:OUTLINE-CHANGED #<VLA-OBJECT IAcadLWPolyline 028f3188>
                    #<VLR-Object-reactor> nil)
(GP:COMMAND-ENDED #<VLR-Command-reactor> (GRIP_MOVE))
```

Ceci montre que les rétro-appels de vos réacteurs d'objets ne seront pas nécessairement exécutés dans tous les cas.

Cette séquence comprend une autre particularité liée à la première. Même au cours du rétro-appel final de command-ended, les cercles qui font encore partie du jeu de sélection des poignées ne peuvent être effacés. Ces cercles sont toujours ouverts dans AutoCAD. Si vous essayez de les effacer lors du rétro-appel de command-ended, AutoCAD risque de s'interrompre brusquement. Pour éviter ce problème, utilisez une autre variable globale pour enregistrer une liste de pointeurs vers les objets dalles en attendant qu'ils puissent être effacés.

Traitement des séquences de réacteur non linéaires

1 Ajoutez la fonction suivante au fichier *gpreact.lsp* :

```
(defun gp:erase-tiles (reactor / reactorData tiles tile)
  (if (setq reactorData (vlr-data reactor))
    (progn
      ;; Tiles in the path are stored as data in the reactor.
      (setq tiles (cdr (assoc 100 reactorData)))
      ;; Erase all the existing tiles in the path.
      (foreach tile tiles
          (if (and (null (member tile *Safe-to-Delete*))
                   (not (vlax-erased-p tile))
                   )
              (progn
                 (vla-put-visible tile 0)
                (setq *Safe-to-Delete* (cons tile *Safe-to-Delete*))
                )
          )
      )
      (vlr-data-set reactor nil)
      )
    )
  )
```

Cette nouvelle fonction sera utilisée au cours de la première phase de l'effacement des dalles. Remarquez que les dalles ne sont pas réellement effacées : elles sont rendues invisibles et ajoutées à une variable globale nommée `*Safe-to-Delete*`.

2 Ajoutez la fonction suivante au fichier *gpreact.lsp* :

```
(defun Gp:Safe-Delete (activeCommand)
  (if (not (equal
      (strcase (substr activeCommand 1 5))
        "GRIP_"
      )
  )
  (progn
     (if *Safe-to-Delete*
        (foreach Item *Safe-to-Delete*
          (if (not (vlax-erased-p Item))
            (vla-erase item)
          )
        )
      )
      (setq *Safe-to-Delete* nil)
      )
    )
  )
```

Cette fonction peut être appelée lorsque les commandes GRIP_MOVE ou GRIP_STRETCH ne sont pas en cours d'exécution.

Codage de la fonction command-ended

A présent que vous avez vu le pseudo-code et traité certains détails importants, remplacez le code test du rétro-appel de réacteur **`gp:command-ended`** par le suivant :

```
(defun gp:command-ended (reactor      command-list
             /        objReactor
             reactorToChange  reactorData
             coordinateValues currentPoints
             newReactorData   newPts
             tileList
             )
  (cond

    ;; CONDITION 1 - POLYLINE ERASED (Erase command)
    ;; If one or more polyline borders are being erased (indicated
    ;; by the presence of *reactorsToRemove*), erase the tiles
    ;; within the border, then remove the reactor.
    (*reactorsToRemove*
     (foreach objReactor *reactorsToRemove*
       (gp:erase-tiles objReactor)
       )
     (setq *reactorsToRemove* nil)
     )
```

```
;; CONDITION 2 - LOST ASSOCIATIVITY (Move, Rotate, etc.)
;; If associativity has been lost (undo, move, etc.), then
;; erase the tiles within each border
;;
((and *lostassociativity* *reactorsToChange*)
 (foreach reactorToChange *reactorsToChange*
   (gp:erase-tiles reactorToChange)
   )
 (setq *reactorsToChange* nil)
 )

;; CONDITION 3 - GRIP_STRETCH
;; In this case, the associativity of the tiles to the path is
;; kept, but the path and the tiles will need to be
;; recalculated and redrawn. A GRIP_STRETCH can only be
;; performed on a single POLYLINE at a time.
((and (not *lostassociativity*)
  *polytochange*
  *reactorsToChange*
  (member "GRIP_STRETCH" command-list)
  ;; for a GRIP_STRETCH, there will be only one reactor in
  ;; the global *reactorsToChange*.
  (setq reactorData
     (vlr-data (setq reactorToChange
                (car *reactorsToChange*)
               )
             )
    )
  )

 ;; First, erase the tiles within the polyline border.
 (gp:erase-tiles reactorToChange)
 ;; Next, get the current coordinate values of the polyline
 ;; vertices.
 (setq coordinateValues
    (vlax-safearray->list
      (vlax-variant-value
    (vla-get-coordinates *polyToChange*)
    )
      )
   )

 ;; If the outline is a lightweight polyline, you have
 ;; 2d points, so use utility function xyList->ListOfPoints
 ;; to convert the coordinate data into lists of
 ;; ((x y) (x y) ...) points. Otherwise, use the
 ;; xyzList->ListOfPoints function that deals
 ;; with 3d points, and converts the coordinate data into
 ;; lists of ((x y z) (x y z) ... ) points.
 (setq CurrentPoints
    (if (= (vla-get-ObjectName *polytochange*) "AcDbPolyline")
      (xyList->ListOfPoints coordinateValues)
      (xyzList->ListOfPoints coordinateValues)
      )
   )
```

```
;; Send this new information to RedefinePolyBorder -- this
;; will return the new Polyline Border
(setq NewReactorData
   (gp:RedefinePolyBorder CurrentPoints reactorData)
  )
;; Get all the border Points and ...
(setq newpts (list (cdr (assoc 12 NewReactorData))
       (cdr (assoc 13 NewReactorData))
       (cdr (assoc 14 NewReactorData))
       (cdr (assoc 15 NewReactorData))
       )
  )

;; ...update the outline of the polyline with the new points
;; calculated above.  If dealing with a lightweight polyline,
;; convert these points to 2D (since all the points in
;; newpts are 3D), otherwise leave them alone.
(if (= (cdr (assoc 4 NewReactorData)) "LIGHT")
  (setq newpts (mapcar '(lambda (point)
              (3dPoint->2dPoint Point)
              )
            newpts
            )
    )
  )

;; Now update the polyline with the correct points.
(vla-put-coordinates
 *polytochange*
;; For description of the list->variantArray see utils.lsp.
 (gp:list->variantArray (apply 'append newpts))
)

;; Now use the current definition of the NewReactorData,
;; which is really the same as the garden path data
;; structure. The only exception is that the field (100)
;; containing the list of tiles is nil.  This is OK since
;; gp:Calculate-and-Draw-Tiles does not require this field
;; to draw the tiles. In fact this function creates the tiles
;; and returns a list of drawn tiles.
(setq tileList (gp:Calculate-and-Draw-Tiles
         ;; path data list without correct tile list
         NewReactorData
         ;; Object creation function
         ;; Within a reactor this *MUST* be ActiveX
         "ActiveX"
         )
  )
```

```
    ;; Now that you have received all the tiles drawn, rebuild
    ;; the data structure with the correct tileList value and
    ;; reset the data property in the reactor.
    ;; Update the tiles associated with the polyline border.
    (setq NewReactorData
       (subst (cons 100 tileList)
          (assoc 100 NewReactorData)
          NewReactorData
          )
      )

    ;; By now you have the new data associated with the polyline.
    ;; All there is left to do is associate it with the reactor
    ;; using vlr-data-set.
    (vlr-data-set (car *reactorsToChange*) NewReactorData)

    ;; Remove all references to the temporary
    ;; variables *polytochange* and *reactorsToChange*.
    (setq *polytochange*     nil
      *reactorsToChange* nil
      )
    )
       )
  ;; Delete any items in the *Safe-to-Delete* global if you can!!!
  (Gp:Safe-Delete (car command-list))
  (princ)
  )
```

Mise à jour de gp:Calculate-and-Draw-Tiles

Comme nous l'avons vu précédemment dans cette leçon, vous devez forcer **`gp:Calculate-and-Draw-Tiles`** à utiliser ActiveX pour la création d'objets désignés à partir d'un rétro-appel de réacteur. Ceci peut entraîner, le cas échéant, le remplacement du style de création d'objets (ActiveX, **`entmake`**, ou **`command`**) choisi par l'utilisateur. Le code que vous venez de mettre à jour dans la fonction **`gp:command-ended`** contient l'appel de la routine de dessin des dalles suivant :

```
(setq tileList (gp:Calculate-and-Draw-Tiles
    ;; path data list without correct tile list.
    NewReactorData
    ;; Object creation function.
    ;; Within a reactor this *MUST* be ActiveX.
    "ActiveX"
   )
)
```

Deux paramètres sont transmis à **`gp:Calculate-and-Draw-Tiles`**: `NewReactorData` (une liste ayant le format de la liste associative `gp_PathData` d'origine), et la chaîne `"ActiveX"` (qui sera chargée de définir le style de création d'objets). Mais examinez la définition actuelle de **`gp:Calculate-and-Draw-Tiles`**. (Rappelons que cette fonction est définie dans *gpdraw.lsp*.) Voici la partie de la fonction qui spécifie les paramètres et les variables locales :

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData /
     PathLength          TileSpace
     TileRadius          SpaceFilled
     SpaceToFill         RowSpacing
     offsetFromCenter    rowStartPoint
     pathWidth           pathAngle
     ObjectCreationStyle TileList)
```

Remarquez qu'un seul paramètre est actuellement spécifié et que `ObjectCreationStyle` est identifié comme une variable locale. Vérifiez, un peu plus loin dans la fonction, comment la variable `ObjectCreationStyle` est définie :

```
(setq ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData))))
```

La variable `ObjectCreationStyle` est actuellement définie à titre interne à l'intérieur de la fonction par l'extraction de la valeur conservée dans la variable `BoundaryData` (la liste associative). A présent, il vous faut être en mesure de remplacer cette valeur.

Modification de gp:Calculate-and-Draw-Tiles afin d'accepter un argument de style de création d'objets

1. Ajoutez la variable `ObjectCreationStyle` à l'argument de la fonction.
2. Supprimez `ObjectCreationStyle` des variables locales.

   L'instruction **`defun`** de la fonction doit prendre l'aspect suivant :

```
(defun gp:Calculate-and-Draw-Tiles (BoundaryData
                    ObjectCreationStyle
    / PathLength     TileSpace
      TileRadius     SpaceFilled
      SpaceToFile    RowSpacing
      offsetFromCenter   rowStartPoint
      pathWidth   pathAngle
      TileList)         ; remove ObjectCreationStyle from locals
```

   Notez que, si vous déclarez une variable à la fois comme un paramètre (avant la barre oblique) et comme une variable locale (après la barre oblique), VLISP vous le signale. Par exemple : si après avoir déclaré `ObjectCreationStyle` à la fois comme paramètre et comme variable, vous appliquez l'outil de vérification syntaxique de VLISP à la fonction **`gp:Calculate-and-Draw-Tiles`**, le message suivant s'affichera dans la fenêtre Générer sortie :

```
; *** WARNING: same symbol before and after / in arguments list:
OBJECTCREATIONSTYLE
```

3 Modifiez la première expression **setq** de **gp:Calculate-and-Draw-Tiles** de manière à lui donner l'aspect suivant (modifications indiquées en gras) :

```
(setq
   PathLength  (cdr (assoc 41 BoundaryData))
   TileSpace   (cdr (assoc 43 BoundaryData))
   TileRadius  (cdr (assoc 42 BoundaryData))
   SpaceToFill (- PathLength TileRadius)
   RowSpacing (* (+ TileSpace (* TileRadius 2.0))
      (sin (Degrees->Radians 60))
   )
   SpaceFilled RowSpacing
   offsetFromCenter 0.0
   offsetDistance  /(+(* TileRadius 2.0)TileSpace)2.0)
   rowStartPoint  cdr (assoc 10 BoundaryData))
   pathWidth      cdr (assoc 40 BoundaryData))
   pathAngle      cdr (assoc 50 BoundaryData))
) ;_ end of setq
 (if (not ObjectCreationStyle)
  (setq ObjectCreationStyle (strcase (cdr (assoc 3 BoundaryData))))
)
```

L'instruction d'affectation d'origine de `ObjectCreationStyle` a été supprimée. Le code vérifie à présent si une valeur a été fournie pour `ObjectCreationStyle`. Si la variable `ObjectCreationStyle` n'est pas définie (en d'autres termes, si la valeur est `nil`), la fonction lui attribue une valeur de la variable `BoundaryData`.

Vous devez encore effectuer une série de modifications dans **gp:Calculate-and-Draw-Tiles**.

Modification des autres appels de gp:Calculate- and-Draw-Tiles

Dans le rétro-appel du réacteur, une chaîne non modifiable `"ActiveX"` est transmise à **gp:Calculate-and-Draw-Tiles** comme argument de `ObjectCreationStyle`. Mais que se passe-t-il lors des autres appels de la fonction **gp:Calculate-and-Draw-Tiles** ?

Rappelons qu'à la leçon 4, il est précisé qu'à chaque fois que vous modifiez une fonction stubbed-out, vous devez poser les questions suivantes :

- L'appel de fonction a-t-il changé ? En d'autres termes, la fonction prend-elle toujours le même nombre d'arguments ?
- La fonction renvoie-t-elle quelque chose de différent ?

Posez ces questions à chaque modification importante que vous apportez à une fonction opérationnelle à mesure que vous créez, perfectionnez et mettez à jour vos applications. Dans ce cas, il vous faut trouver toutes les fonctions de votre projet qui appellent **`gp:Calculate-and-Draw-Tiles`**. VLISP dispose d'une fonction qui vous aide à y parvenir.

Recherche de tous les appels de gp:Calculate-and-Draw-Tiles dans le projet

1 Dans la fenêtre de l'éditeur de texte de VLISP, cliquez deux fois sur le mot **`gp:Calculate-and-Draw-Tiles`** dans le fichier *gpdraw.lsp*.

2 Choisissez Rechercher ➤ Rechercher dans la barre de menus VLISP.

Puisque vous avez présélectionné le nom de la fonction, il est déjà répertorié comme étant la chaîne à rechercher.

3 Dans la boîte de dialogue Rechercher, sélectionnez le bouton Projet dans la zone de liste Rechercher.

Lorsque vous sélectionnez cette option, le bas de la boîte de dialogue Rechercher s'agrandit pour vous permettre de sélectionner le projet à rechercher.

4 Spécifiez le nom de votre projet, puis cliquez sur le bouton Rechercher.

VLISP affiche les résultats dans la fenêtre Rechercher la sortie :

5 Examinez les résultats de la fenêtre Rechercher la sortie pour connaître les autres emplacements où votre code appelle la fonction **`gp:Calculate-and-Draw-Tiles`**. Il doit y en avoir un seul : un emplacement dans *gpmain.lsp*.

6 Dans la fenêtre Rechercher la sortie, cliquez deux fois sur la ligne de code qui appelle **`gp:Calculate-and-Draw-Tiles`**.

VLISP active une fenêtre d'éditeur de texte et permet d'atteindre directement cette ligne de code dans *gpmain.lsp*. Le code s'affiche comme suit :

```
(setq tilelist (gp:Calculate-and-Draw-Tiles gp_PathData))
```

7 Remplacez cette ligne de code par ce qui suit :

```
(setq tilelist (gp:Calculate-and-Draw-Tiles gp_PathData nil))
```

Pourquoi `nil` ? Examinez à nouveau le pseudo-code :

```
If ObjectCreationStyle is nil, assign it from the BoundaryData.
```

La transmission de `nil` comme paramètre à **`gp:Calculate-and-Draw-Tiles`** a pour effet de vérifier le choix de dessin des dalles déterminé par l'utilisateur (défini dans la boîte de dialogue et enregistré dans **`gp_PathData`**). Les appels suivants en provenance du rétro-appel du réacteur command-ended passeront outre cette opération en forçant l'utilisation d'ActiveX.

Bravo ! La fonctionnalité de réacteur principale est à présent en place. Si vous préférez, copiez les fichiers *gpmain.lsp* et *gpdraw.lsp* à partir de *Tutorial\VisualLISP\Lesson7* dans votre répertoire de travail et examinez le code complété et débogué.

Ne vous réjouissez pas trop vite. Vous avez encore fort à faire, et ce en raison de ce simple fragment de code de la fonction **`gp:Command-ended`** :

```
(setq NewReactorData
    (gp:RedefinePolyBorder CurrentPoints reactorData)
) ;_ end of setq
```

Redéfinition du contour de la polyligne

Vous êtes parvenu à ce point non sans peine, et vous avez probablement votre compte, pour le moment, de nouveaux concepts, termes, commandes et autres informations. Nous vous recommandons, par conséquent, de copier l'exemple de code fourni par le didacticiel, plutôt que de l'entrer vous-même.

Copie du code permettant de redéfinir le contour de la polyligne

1. Copiez dans votre répertoire de travail le fichier de *gppoly.lsp* contenu dans le répertoire *Tutorial\VisualLISP\Lesson7*.
2. Dans la fenêtre de projet, cliquez sur le bouton Propriétés du projet.
3. Ajoutez le fichier *gppoly.lsp* au projet.
4. Cliquez sur OK pour accepter le projet avec le fichier supplémentaire.
5. Dans la fenêtre de projet, cliquez deux fois sur le fichier *gppoly.lsp* afin de l'ouvrir.

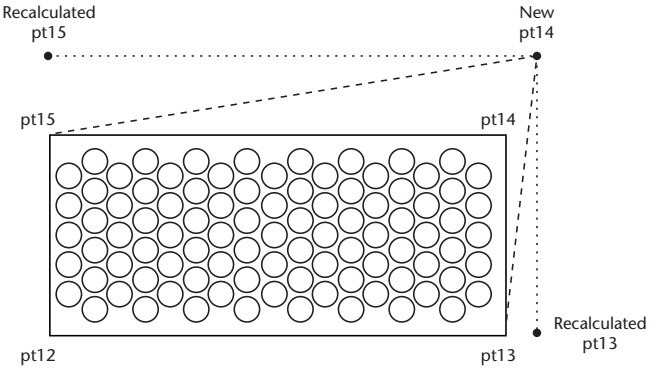
Examen des fonctions de *gppoly.lsp*

Le fichier *gppoly.lsp* contient un certain nombre de fonctions requises pour le redressement d'une polyligne après l'étirement d'une seule poignée. Seules quelques-unes de ces fonctions seront expliquées dans le cadre de ce didacticiel.

REMARQUE Cette section du didacticiel Sentier de jardin contient certains fragments de code et certains concepts qui comptent parmi les plus complexes de toute la leçon. Si vus êtes débutant, il est préférable de passer d'emblée à la section "Création d'une application" plus avant dans le chapitre.

Les fonctions contenues dans le fichier *gppoly.lsp* sont organisées selon une méthode que vous avez peut-être déjà remarquée dans d'autres fichiers de code source AutoLISP. La fonction de niveau le plus élevé, en général la fonction principale ou la fonction **`C:`** (dans ce cas, **`gp:Redefine-PolyBorder`**) est située au bas du fichier. Les fonctions appelées dans la fonction principale sont définies au-dessus d'elle dans le fichier source. Cette convention remonte aux origines de la programmation, lorsque certains environnements de développement exigeaient que les fichiers soient organisés de cette façon. Dans le cas de VLISP, c'est une question de choix personnel, car rien n'impose d'organiser les fonctions dans un ordre spécifique.

Avant de vous plonger dans les détails, prenez un peu de recul et examinez la marche à suivre pour recalculer et dessiner le contour du sentier de jardin. L'illustration suivante montre un exemple de sentier de jardin de même que les points clés de la liste associative enregistrés parmi les données de réacteur :



Dans cet exemple, le point clé 12 correspond au coin inférieur gauche, le point clé 13 correspond au coin inférieur droit et ainsi de suite. Si l'utilisateur déplace le coin supérieur droit (le point clé 14), le programme devra recalculer deux des points existants : les points inférieur droit (13) et supérieur gauche (15).

Compréhension de la fonction gp:RedefinePolyBorder

Le pseudo-code suivant illustre la logique qui sous-tend la fonction principale,**`gp:RedefinePolyBorder`** :

```
Function gp:RedefinePolyBorder
   Extract the previous polyline corner points (12, 13, 14, and 15
     key values).
   Find the moved corner point by comparing the previous
     polyline corner points with the current corner points.
```

```
  (The one "misfit" point will be the point that moved.)
Set the new corner points by recalculating the two points
  adjacent to the moved point.
Update the new corner points in the reactor data (that will
  be stored back in the reactor for the modified polyline).
Update other information in the reactor data. (Start point,
   endpoint, width, and length of path need to be recalculated.)
```

Compréhension de la fonction gp:FindMovedPoint

La fonction **`gp:FindMovedPoint`** contient certaines expressions LISP très puissantes qui concernent la manipulation des listes. Au fond, cette fonction compare la liste des points de la polyligne en cours (dont l'un a été déplacé par l'utilisateur) aux points précédents, et renvoie la liste codée (les coordonnées <xvalue> <yvalue>du point 13) du point déplacé.

La meilleure façon de comprendre le fonctionnement de cette fonction est de parcourir le code étape par étape et de vérifier les valeurs qu'il manipule. Définissez un point d'arrêt dès la première expression `(setq result . . .)` et surveillez les variables suivantes en avançant pas à pas dans la fonction :

- `KeyListToLookFor`
- `PresentPoints`
- `KeyedList`
- `Résultat`
- `KeyListStatus`
- `MissingKey`
- `MovedPoint`

Les fonctions **`mapcar`** et **`lambda`** seront examinées dans la fonction suivante. Pour l'instant, toutefois, suivez les commentaires du code pour vérifier si vous pouvez comprendre le fonctionnement des fonctions.

Compréhension de la fonction gp:FindPointInList

L'en-tête de la fonction dans le code source explique comment **`gp:FindPointInList`** transforme les informations avec lesquelles elle travaille. A l'instar de la fonction précédente, **`Gp:FindMovedPoint`**, cette fonction exploite les possibilités de manipulation des listes de LISP pour s'acquitter de sa tâche. Lorsque vous travaillez avec des listes, vous verrez souvent les fonctions **`mapcar`** et **`lambda`** utilisées ensemble comme dans le cas présent. Dans un premier temps, ces fonctions peuvent vous sembler déroutantes et difficiles à comprendre, car leurs noms n'indiquent pas ce qu'elles font réellement. Quand vous aurez appris à vous en servir, vous découvrirez toutefois que ces deux fonctions comptent parmi les plus puissantes du répertoire AutoLISP. Voici une rapide présentation de **`mapcar`** et de **`lambda`**.

La fonction **mapcar** applique (associe) une expression à chaque élément d'une liste. Par exemple, dans le cas d'une liste de nombre entiers 1, 2, 3 et 4, **mapcar** permet d'appliquer la fonction **1+** qui ajoute 1 à chaque nombre de la liste :

```
_$ (mapcar '1+ '(1 2 3 4))
(2 3 4 5)
```

La finalité première de **mapcar** est d'associer la fonction donnée dans le premier paramètre aux éléments successifs du second paramètre (la liste). La valeur résultant d'une opération **mapcar** est la liste transformée par la fonction ou l'expression qui lui a été appliquée. (En réalité, **mapcar** offre d'autres possibilités mais, pour l'instant, cette définition suffira.)

Dans l'exemple indiqué, chaque valeur de la liste `'(1 2 3 4)` a été transmise à la fonction **1+**. Pour l'essentiel, **mapcar** a effectué les opérations suivantes et rassemblé les valeurs résultantes dans une liste :

```
(1+ 1)  -> 2
(1+ 2)  -> 3
(1+ 3)  -> 4
(1+ 4)  -> 5
```

Voici un autre exemple de **mapcar**, qui utilise cette fois la fonction **null** pour tester si les valeurs d'une liste sont des valeurs nulles (non vrai) :

```
_$ (mapcar 'null (list 1 (= 3 "3") nil "Steve"))
(nil T T nil)
```

Pour l'essentiel, voici ce qui s'est produit dans ce code :

```
(null 1)    -> nil
(null (= 3 "3")   -> T
(null nil)   -> T
(null "Steve")   -> nil
```

Vous pouvez utiliser de nombreuses fonctions AutoLISP existantes dans **mapcar**. Vous pouvez même utiliser vos propres fonctions. Supposons, par exemple, que vous ayez créé une fonction très puissante appelée **equals2** :

```
_$ (defun equals2(num)(= num 2))
EQUALS2
_$ (mapcar 'equals2  '(1 2 3 4))
(nil T nil nil)
```

Admettons, **equals2** n'est pas si puissante que cela. Mais c'est en pareils cas que la fonction **lambda** est utile. Vous pouvez utiliser **lambda** lorsque vous n'envisagez pas (ou lorsque vous n'avez pas besoin) de définir une fonction. Vous verrez parfois **lambda** définie comme une fonction anonyme. Par exemple, au lieu de définir une fonction appelée **equals2**, vous pourriez écrire une expression **lambda** pour effectuer la même opération sans qu'il soit nécessaire de définir une fonction :

```
_$ (mapcar '(lambda (num) (= num 2)) '(1 2 3 4))
(nil T nil nil)
```

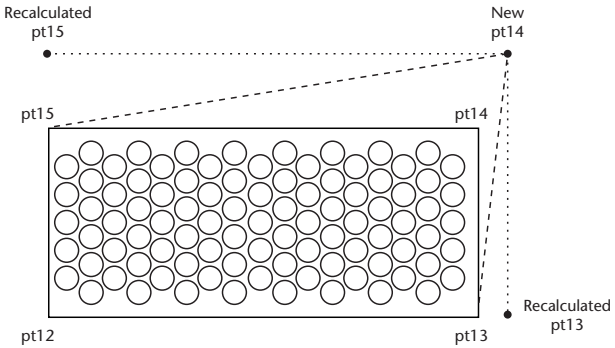
Voici ce qui se produit dans le code :

```
(= 1 2)      -> nil
(= 2 2)      -> T
(= 3 2)      -> nil
(= 4 2)      -> nil
```

Sachant cela, voyez si vous comprenez la fonction **`gp:FindPointInList`**. Une fois encore, reportez-vous aux commentaires du code source.

Compréhension de la fonction gp:recalcPolyCorners

Pour comprendre le fonctionnement de **`gp:recalcPolyCorners`** le mieux est de revoir le schéma qui indique la signification des valeurs clés 12 à 15 :



Dans le schéma, l'utilisateur a déplacé le coin associé à la valeur 14. Ceci signifie que les coins associés au 13 et au 15 doivent être recalculés.

Le point 15 doit être déplacé le long du vecteur défini par les points 12 et 15, de sorte qu'il se trouve dans l'alignement du nouveau point 14. Les vecteurs définis par les points 12 et 15 et par les points 14 et 15 doivent être perpendiculaires entre eux. La même opération doit être réalisée pour recalculer le nouvel emplacement du point 13.

A présent, réexaminez le code pour vérifier que vous le comprenez.

Compréhension des fonctions gp:pointEqual, gp:rtos2, et gp:zeroSmallNum

Ces trois fonctions sont requises pour éviter une des particularités de la programmation dans un système AutoCAD, lequel, comme vous le savez, permet d'atteindre une grande précision. Occasionnellement toutefois, les nombres ne sont pas assez précis en raison des arrondis de décimales définissant des positions géométriques. Puisqu'il vous faut comparer un jeu de points avec d'autres points, vous devez être en mesure de traiter ce genre de cas.

Avez-vous remarqué que, au moment de répertorier les informations associées à une entité AutoCAD, vous voyez apparaître quelquefois une valeur du type `1.0e-017` ? Ce nombre est *proche de* zéro mais, lorsque vous le comparez à zéro dans un programme à l'intérieur d'un programme LISP, *proche de* n'est pas pris en compte.

Dans le cadre du sentier de jardin, vous devez avoir la possibilité de comparer des nombres sans vous préoccuper du fait que `1.0e-017` n'est pas tout à fait zéro. Les fonctions **`gp:pointEqual`**, **`gp:rtos2`**, et **`gp:zeroSmallNum`** gèrent les éventuelles différences d'arrondi dans les comparaisons de listes de points.

Ceci complète le passage en revue des fonctions de *gppoly.lsp*.

Clôture du code

A ce stade de la leçon, vous avez appris à :

- Modifier la fonction **`gp:drawOutline`** pour qu'elle renvoie les points du périmètre de la polyligne en plus du pointeur vers la polyligne. Ajouter ces informations à la variable `gp_PathData`. Cette variable est enregistrée avec les données de réacteur du réacteur d'objets associé à chaque sentier de jardin.
- Mettre à jour les fonctions de réacteur dans *gpreact.lsp*.
- Ajouter les fonctions **`xyzList->ListOfPoints`**, **`xyList->ListOfPoints`**, et d'autres fonctions utilitaires au fichier *utils.lsp*.
- Mettre à jour la fonction **`gp:Calculate-and-Draw-Tiles`** de telle sorte que `ObjectCreationStyle` est à présent un paramètre de la fonction plutôt qu'une variable locale.
- Modifier l'appel de **`gp:Calculate-and-Draw-Tiles`** dans la fonction **`C:GPath`** à l'intérieur du fichier *gpmain.lsp*.
- Ajouter *gppoly.lsp* à votre projet, et examiner les fonctions qu'elle contient.

Faites un essai de votre application complète. Enregistrez votre travail, puis chargez les sources des projets, exécutez la fonction **Gpath** et essayez d'étirer et de déplacer le contour du sentier de jardin. Rappel : si quelque chose ne fonctionne pas et si vous n'êtes pas en mesure de résoudre le problème, vous pouvez charger le code complet à partir du répertoire *Tutorial\VisualLISP\Lesson7*.

Création d'une application

La dernière phase du didacticiel consiste à transformer le code de votre sentier en une application autonome. De cette manière, elle pourra être distribuée comme un fichier exécutable aux utilisateurs et aux clients. Heureusement, ce dernier ensemble de tâches est probablement le plus aisé de tout le didacticiel car VLISP accomplit pratiquement tout le travail à votre place.

REMARQUE Nous vous recommandons de ne créer une application que si votre code est en parfait état de marche. Assurez-vous que votre application a été testée à l'aide des fichiers source d'origine et que vous êtes satisfait du résultat.

Démarrage de l'Assistant Créer une application

Pour vous aider à créer des applications autonomes, VLISP met à votre disposition l'assistant Créer une application.

Exécution de l'assistant Créer une application

1. Pour lancer l'assistant, choisissez Fichier ➤ Créer une application ➤ Assistant Nouvelle application dans la barre de menus VLISP.
2. Sélectionnez le mode Expert et cliquez sur Suivant.

   L'assistant vous demande de préciser le répertoire dans lequel enregistrer les fichiers qui seront créés et de nommer votre application. L'assistant crée deux fichiers de sortie : un fichier *.vlx* contenant votre programme exécutable, et un fichier *.prv* contenant les options définies à l'aide de Créer une application. Le fichier *.prv* est également appelé fichier *make*. Vous pouvez l'utiliser le fichier le cas échéant pour reconstruire votre application.
3. Spécifiez votre répertoire *Tutorial\VisualLISP\myPath* comme emplacement de l'application, et nommez l'application **gardenpath**. VLISP utilise le nom de l'application parmi les noms de fichiers de sortie (dans ce cas, *gardenpath.vlx* et *gardenpath.prv*.)

Cliquez sur Suivant pour continuer.

4 Les options de l'application ne sont pas couvertes dans ce didacticiel. Acceptez les options par défaut et cliquez sur Suivant. (Pour plus d'informations sur les applications s'exécutant dans des espaces mémoire distincts, consultez "Running an Application in Its Own Namespace" dans le *Visual LISP Developer's Guide*.)

5 A ce stade, l'assistant vous demande d'identifier tous les fichiers de code source AutoLISP qui composent l'application. Vous pourriez sélectionner individuellement tous les fichiers source LISP, mais il existe un moyen plus commode. Recherchez dans le menu déroulant Type de fichier l'option "Fichier de projet Visual LISP", puis cliquez sur le bouton Ajouter. Sélectionnez le fichier de projet *Gpath* et cliquez sur Ouvrir.

REMARQUE Selon la façon dont vous avez procédé au cours des étapes de ce didacticiel, il se peut que vous ayez plusieurs fichiers de projets *Gpath* qui s'affichent. Sélectionnez le fichier le plus récent. Si vous avez copié le code source complet de la leçon 7, le nom de projet à sélectionner doit être *Gpath7.prj*.

Après avoir sélectionné le fichier de projet, cliquez sur Suivant pour continuer.

6 Un des avantages des applications VLX compilées est que vous pouvez compiler vos fichiers de contrôle des boîtes de dialogue (*.dcl*) dans votre application complète. Ceci réduit le nombre de fichiers source individuels que vos utilisateurs auront à traiter, et élimine les problèmes de chemin d'accès lors du chargement d'un fichier DCL.

Recherchez dans le menu déroulant Type de fichier l'option "Fichiers DCL", puis cliquez sur le bouton Ajouter. Sélectionnez le fichier *gpdialog.dcl*, puis cliquez sur Ouvrir.

Cliquez sur Suivant pour poursuivre la création de votre application.

7 Les options de compilation ne sont pas abordées dans ce didacticiel. Acceptez les options par défaut et cliquez sur Suivant. (Pour plus d'informations sur les options de compilation, consultez "Optimizing Application Code" dans le *Visual LISP Developer's Guide*.)

8 L'étape finale vous permet de passer en revue vos sélections. A ce stade, vous pouvez sélectionner Fin. VLISP entamera la procédure de création et affichera les résultats dans la fenêtre Générer sortie. Plusieurs fichiers intermédiaires seront produits, à mesure que vos fichiers individuels de code source sont compilés dans un format qui peut être lié dans une application VLX unique.

A la fin de la procédure, vous aurez un fichier exécutable nommé *gardenpath.vrx*. Pour le tester, procédez de la manière suivante :

- Dans le menu Outils d'AutoCAD, choisissez Charger une application.
- Chargez l'application *gardenpath.vlx* qui vient d'être créée et qui se trouve dans le répertoire *Tutorial\VisualLISP\MyPath*.
- Exécutez la commande **gpath**.

Clôture du didacticiel

Vous voici finalement au bout du sentier ! Comme vous avez pu le constater, de nombreux sujets ont été abordés dans ce didacticiel. Les concepts des opérations AutoLISP et VLISP ont été présentés. Le "sentier de jardin revisité" a été conçu pour vous donner un échantillon de sujets et de concepts. Pour en savoir plus, reportez-vous à la brève bibliographie de la page suivante où sont répertoriés quelques ouvrages traitant de Common LISP et AutoLISP.

Bibliographie LISP et AutoLISP

AutoLISP : Programming for Productivity, William Kramer, Autodesk Press, ISBN 0-8273-5832-6.

Essential AutoLISP, Roy Harkow, Springer-Verlag, ISBN 0-387-94571-7

AutoLISP in Plain English: A Practical Guide for Non-Programmers, George O. Head, Ventana Press, ISBN: 1566041406.

LISP, 3rd Edition, Patrick Henry Winston and Berthold Klaus Paul Horn, Addison-Wesley Publishing Company, ISBN 0-201-08319-1.

ANSI Common Lisp, Paul Graham, Prentice Hall, ISBN 0-13-370875-6

Looking at LISP, Tony Hasemer, Addison-Wesley Publishing Company, ISBN 0-201-12080-1.

Common LISP, The Language, Second Edition, Guy L. Steele Jr., Digital Press, ISBN 1-55558-041-6.

Index

D

G

H

I

P

Q

R

S

T

U

V