Epita:Algo:Mémo-Langage

De EPITACoursAlgo.

Sommaire

Algorithme : présentation

Le langage présenté ici est celui utilisé en cours et en `td` d'algorithmique *(partie "impérative")* des deux années de classes préparatoires de l'Epita (http://www.epita.fr) . On peut distinguer deux "niveaux" d'utilisation : en `cours`, où on utilisera directement les types abstraits avec leurs opérations ; en `td`, où les **types abstraits** seront représentés à l'aide de types structurés et où les **opérations** devront être implémentées sous forme de routines *(l'étape suivante logique est alors l'implémentation machine, qui se réduit, presque, à la traduction dans le langage choisi)*.

Structure générale d'un algorithme

Voici la structure générale d'un algorithme. Comme de nombreux langages impératifs, il est composé de deux parties distinctes : les **déclarations** et les **instructions**.

```
algorithme identifiant_algo

    <partie déclarations>

debut

    <partie instructions>

fin algorithme identifiant_algo
```

La partie **déclarations** est détaillée plus loin dans cette section, la partie **instructions** le sera plus tard (voir section ***Les instructions'***).

Note typographique : dans toutes les pages de ce site, les parties algorithmiques seront décrites de la manière suivante:

- **mots clés**
- *identifiants*
- `<élément à développer>`
- `[élément facultatif]`
- ... pour indiquer que la structure en cours peut être répétée.

Règles d'écriture d'un algorithme

Alphabet et lexique -- Les "mots" du langage

Comme pour tout langage (et pas seulement de programmation, même en Klingon), il nous faut tout d'abord définir l'alphabet (les caractères utilisés) et les mots qui nous permettrons de construire des énoncés (ici des instructions par exemple). Deux types de mots seront présents dans l'algorithme : les mots clés, mots prédéfinis dans le langage, et les identifiants, mots construits pour "nommer" les variables, les types, les routines... Pour ces derniers il existe des règles très strictes de construction.

Règles de construction des identifiants

Un *identifiant* (ou *identificateur*) est un nom déclaré et valide pour :

- une constante,
- un type,
- une variable,
- une procédure,
- une fonction,
- l'algorithme principal.

Les noms d'*identifiants* ne peuvent contenir que des caractères compris dans les intervalles suivants :

- `'a'..'z'`
- `'A'..'Z'`
- `'0'..'9'`

On peut aussi utiliser le caractère `'_'` (souligné/underscore).

Pour construire un identifiant, il faudra respecter les règles suivantes :

- Il ne peut en aucun cas commencer par un chiffre.
- Le langage algorithmique ne fait pas de différence majuscules/minuscules.
- Tout identifiant doit avoir été déclaré avant d'être utilisé (plus haut dans le code source).
- Un identifiant doit bien entendu être différent d'un mot clé, ceci pour éviter tout ambiguïté.
- Enfin, pour faciliter l'écriture et la lecture des algorithmes, il est très fortement conseillé d'utiliser des identifiants explicites.

Les mots clés

Les mots clés seront utilisés pour construire les algorithmes, les déclarations et les instructions. Ceux-ci sont prédéfinis dans le langage. Comme pour les identifiants, aucune distinction n'est faite entre les majuscules et les minuscules.

La liste des mots clés du langage :

algorithme	div	globaux	parametres	tant que	alors	enregistrement	jusqu'a	procedure
types	autrement	et	locaux	pour	variables	constantes	faire	mod
selon	debut	fin	non	si	decroissant	fonction	ou	sinon

Séparateurs -- Symboles spéciaux

La structure de l'algorithme, des déclarations, des instructions est faite de telle manière qu'il n'y a pas besoin de séparateurs particuliers (les différentes parties se suivent, tout simplement). Tout ce qui est commencé est explicitement fini! Nous utilisons juste la virgule **","** comme séparateur pour les listes de paramètres dans les appels de routines. Un retour à la ligne est cependant nécessaire avant de commencer toute nouvelle instruction ou déclaration.

Un certain nombre de caractères et de combinaisons de caractères ont une signification spéciale pour le langage algorithmique. En voici la liste :

← ↑ . , : /* */ < > <= >= <> = + - * / ( )

Commentaires

Il est possible d'insérer des commentaires dans l'algorithme à n'importe quelle place. Ignoré par le compilateur, les commentaires ne font pas partie de l'algorithme et n'influent pas sur le déroulement de celui-ci, si ce n'est pour la compréhension ! Les commentaires seront délimités par les paires de symboles /* et */.

La partie déclarations

C'est ici que nous allons déclarer tout ce dont nous avons besoin pour l'algorithme : *constantes*, *types*, *variables* ainsi que les *routines* (*procédures* et *fonctions*).

```
<partie déclarations>:

    <déclarations des constantes>
    <déclarations des types>
    <déclarations des variables>
    <déclarations des routines>
```

L'ordre des déclarations est important, il ne peut être changé. Nous allons voir ici les déclarations des *constantes*, *types* et *variables* (les routines font l'objet de la section **Les procédures et fonctions**)

Déclaration des constantes

```
constantes
  ident_constante = <valeur>
      ...
```

Une constante ne peut être modifiée dans l'algorithme.

Déclaration des types

```
types
  ident_type = <définition du type>
      ...
```

C'est ici que seront décrits les types définis par l'utilisateur.

Déclaration des variables

Les variables contiennent les données manipulées par l'algorithme. Lors de la déclaration on doit leur attribuer un type. Celui-ci doit être *défini* (on applique ici une règle très importante valable partout : *on ne peut utiliser que ce qui est déjà connu!* Dès lors, lorsqu'on veut utiliser quelque chose, il faut s'assurer que cela a déjà été déclaré ou est prédéfini dans le langage.) : prédéfini dans le langage, ou défini par l'utilisateur (voir section **types**). Dans ce dernier cas le type doit obligatoirement avoir été déclaré avant !

```
variables
  ident_type   ident_variable1, ident_variable2, ...
      ...
```

Chaque ligne contient la liste des variables du type spécifié (on peut pour plus de clarté définir plusieurs listes pour un même type). Attention, il n'y a aucun symbole entre le type et les identifiants!

Les types

Les types prédéfinis

Les types prédéfinis en langage algorithmique sont :

- entier	nombres entiers signés	`42`
- reel	nombres flottants signés	`0.154`
- octet(mot)	nombres entiers non signés sur un (deux) octet(s)	
- booleen	énumération définissant les données *vrai* et *faux*	
- caractere	caractère ANSI sur un octet	`'a'`
- chaine	chaîne de caractères	`"lapin"`

Les types définis par l'utilisateur

Énumérations

Les énumérations sont des listes d'identifiants représentant de manière lisible une suite de valeurs ayant un lien logique.

Déclaration

```
types
     ident_enum = (ident_valeur1, ident_valeur2, ...)
```

Utilisation

Les identifiants sont utilisés comme des constantes et peuvent donc être directement affectés à des variables du type énuméré.

Tableaux

Les tableaux permettent d'associer dans une même variable plusieurs données de même type.

Déclaration

- **Une dimension**

```
types
     ident_tableau = <entier> ident_type_elts
```

L'entier peut être directement une valeur, ou un identifiant de constante entière. Le type des éléments peut être un type prédéfini, ou un type utilisateur qui doit avoir été déclaré avant.

- **Deux dimensions ou plus**

Dans ce cas il suffit de donner autant d'entiers que de dimensions.

```
types
     ident_tableau = <entier_1> x <entier_2> ... ident_type_elts
```

Utilisation

L'accès à un élément d'un tableau se fait en indiquant la liste des indices correspondant à chaque dimension.

```
ident_var[<indice_1>, <indice_2>, ...]
```

On obtient alors une **variable** du type des éléments.

Les enregistrements

Un enregistrement est une structure qui regroupe dans une même entité logique un ensemble de données de types différents.

Déclaration

```
types
     ident_enreg = enregistrement
                   ident_type_1     ident_champ_11, ident_champ_12, ...
                   ident_type_2     ident_champ_21, ident_champ_22, ...
                   ...
     fin enregistrement ident_enreg
```

Utilisation

Pour accéder au contenu d'un enregistrement, on utilise la notation à point : on donne le nom de l'enregistrement (l'identifiant de la variable), suivi d'un point (.) et du nom du champ auquel on désire accéder.

```
ident_var.ident_champ
```

Pointeurs typés

Un pointeur est une donnée qui contient l'adresse en mémoire d'une autre donnée. Ainsi, plusieurs pointeurs peuvent pointer sur la même donnée, s'ils contiennent la même adresse. Le pointeur typé pointe sur une donnée dont le type est défini dans le même bloc de déclaration. Particularité de cette déclaration : c'est la seule pouvant utiliser un identifiant de type non

déclaré précédemment.

Déclaration

```
types
     ident_pointeur = ↑ident_type_pointé
```

Utilisation

Accès à l'élément pointé :

```
ident_var↑
```

Les instructions

Préliminaire : Les expressions

Une expression représente une succession de calculs ; elle peut faire intervenir des constantes, des variables, des fonctions et des opérateurs. Les expressions sont utilisées dans tout l'algorithme : dans les affectations, en paramètre des routines, dans les structures de contrôles...

Expression : syntaxe

Une expression peut être :

- une valeur	`42`
- une variable	`x`
- une constante	`C`
- un appel de fonction	*cos*`(x)`
- <expression> OpérateurBinaire <expression>	`32 + x`
- OpérateurUnaire <expression>	`non A`
- (<expression>)	`(a + 15)`
...	

Les opérateurs

Les opérateurs arithmétiques

- Les classiques :

- (unaire)	Changement de signe
+	Addition
-	Soustraction

*	Multiplication
/	Division flottante

Les opérandes peuvent être du type `entier` (le résultat est `entier`) ou `reel` (le résultat est `reel`), sauf pour la division flottante, où le résultat sera toujours de type `reel` *(Voir la **concordance des types** pour les cas mixtes)*.

- La division entière :

`div`	Division entière
`mod`	Modulo (reste de la division entière)

Ces opérateurs ne fonctionnent qu'avec des entiers !

Les opérateurs logiques (et *binaires``)*

Les opérandes sont booléens, on a alors une opération logique. Le résultat est un booléen. Les expressions booléennes sont utilisées comme conditions dans les structures de contrôles.

`div`	Division entière
`non`	négation logique ($\neg$)
`et`	et logique ($\wedge$)
`ou`	ou logique ($\vee$)
`oue`	ou exclusif

Rappels : Avec `a` et `b` des booléens ou des expressions booléennes :

- `a et b` n'est vrai que si `a` est vrai et `b` est vrai — est faux dès qu'un des deux est faux
- `a ou b` n'est faux que si `a` est faux et `b` est faux — est vrai dès qu'un des deux est vrai
- `a oue b` est vrai si un des deux seulement est vrai — est équivalent à `a<>b`

Important : Les opérateurs `et` et `ou` sont séquentiels : si l'évaluation de la première opérande suffit à donner le résultat, la deuxième n'est pas évaluée. Ainsi, si *a* est *faux*, `a` et `b` sera *faux*, sans que *b* n'ait été évalué.

Note : On utilisera les mêmes opérateurs sur des entiers non signés `mot` ou `octet`, on a alors des *opérations sur bits*. Les opérateurs fonctionnent de la même manière, le vrai correspondant à 1, le faux à 0.

Les opérateurs relationnels

Les deux opérandes doivent être de types compatibles. Le résultat est toujours de type `booleen` : `vrai` ou `faux`.

=	égal
<>	différent
<	inféreur à
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

La concaténation de chaînes

L'opérateur "+" pourra être utilisé avec les chaînes de caractères et les caractères pour la concaténation.

Règles d'évaluation des expressions

Priorité des opérateurs

Par ordre décroissant :

opérateurs unaires	`- non`
opérateurs multiplicatifs	`* / div mod et`
opérateurs additifs	`+ - ou`
opérateurs relationnels	`= < <= > >= <>`

Les expressions entre parenthèses sont entièrement évaluées avant d'intervenir dans la suite des calculs.

Concordance de type

Un opérateur *binaire`` ne peut porter que sur deux valeurs du même type. Une exception a lieu lorsqu'une valeur est réelle et l'autre entière. Dans ce cas la valeur entière est convertie en une valeur réelle. Cette règle s'applique pour les opérateurs arithmétiques (+, -, *, /) et ceux de comparaisons.*

L'affectation

Cette instruction permet d'affecter une valeur à une variable. La valeur peut être n'importe quelle expression de type compatible avec la variable.

- **Syntaxe**

```
ident_var ← <expression>
```

avec *ident_var* :

- un identifiant de variable
- une référence à un élément d'un tableau
- une référence à un champ d'enregistrement
- un objet pointé

Note : dans la suite *ident_var* représentera ces quatre éléments.

- **Fonctionnement**

```
ident_var ← <valeur>
```

- Une valeur (une expression) ne peut en aucun cas figurer à gauche d'une affectation.
- Une variable figurant à droite d'une affectation (et plus généralement dans toute expression) doit obligatoirement contenir une valeur.

Les appels aux fonctions et procédures

L'appel d'une procédure ou d'une fonction (**routine**) se fait par son nom suivi, s'il y a lieu, de la liste des arguments placés entre parenthèses. Il faut respecter l'ordre de déclaration des paramètres. Lorsque le passage se fait par adresse (**paramètre global**), l'argument doit obligatoirement être une variable. S'il est passé par valeur (**paramètre local**), il peut s'agir d'une expression quelconque (voir *Les paramètres*). La distinction entre les différents paramètres sera vue en détail dans le chapitre consacré aux procédures et fonctions.

Appel de procédure : une instruction

L'appel de procédure est une instruction à part entière :

```
ident_procedure (param1, param2, ...)
```

Exemples de procédures : les entrées-sorties

- Les procédures d'affichage : `ecrire -- ecrireSansRetour`

```
ecrire ("lapin", x)
```

affiche la chaîne *lapin``, suivi du contenu de la variable* `x` *(à condition que* `x` *contienne une valeur).*

```
ecrireSansRetour (12+a)
```

affiche la valeur de l'expression `12+a`, sans retourner à la ligne.

- La procédure de lecture : `lire`

```
lire (x)
```

affecte à la variable `x` la valeur saisie.

Appel de fonction : une expression

Une fonction est une routine qui retourne une valeur. L'appel de fonction sera donc utilisable comme n'importe quelle autre valeur (dans une expression, en paramètre d'une routine, ...). Par exemple dans une affectation :

```
ident_var ← ident_fonction (param1, param2, ...)
```

Note : Un appel de fonction seul n'est pas une instruction !

Les structures de choix

L'alternative : `si ... alors ... sinon ... fin si`

- **Syntaxe**

```
si <expression booleenne> alors
       <instructions>
[sinon
       <instructions> ]
fin si
```

Remarque :La partie **`sinon`** `<instruction>` est facultative.

Attention : Le **fin si** est obligatoire ! Il en sera de même pour toutes les instructions structurées : cette marque de fin doit être présente même si il n'y a qu'une seule instruction.

- **Fonctionnement**

Si la condition (exprimée par l'expression booléenne) est vraie alors seule la suite d'instructions placée après le **alors** sera exécutée. Dans le cas contraire, si la partie **sinon** existe elle sera exécutée, si elle n'existe pas, rien ne se passe.

Choix multiples : `selon ... faire`

- **Syntaxe**

```
selon <expression> faire
       <liste_expr> : <instructions>
       ...
[autrement <instructions>]
fin selon
```

`<liste_expr>` = une liste de valeurs (séparées par des virgules) pour l'expression.

L'expression doit être de type scalaire : les types entiers, le type `caractere` et les énumérations.

- **Fonctionnement**

Les instructions exécutées seront celles correspondant à la valeur (*Attention, rien à voir avec le filtrage de `caml` !*) de l'expression. Si celle-ci n'est pas dans une des liste, alors ce sera la partie **autrement** (si elle existe) qui sera exécutée.

Structures de répétition

Les répétitives

`tant que ... fin tant que`

- **Syntaxe**

```
tant que <expression booléenne> faire
       <instructions>
fin tant que
```

- **Fonctionnement**

Les instructions sont répétées tant que la condition est vérifiée. Comme le test est au début, les instructions peuvent donc ne jamais être exécutées.

Attention : il est impératif que la condition devienne fausse à un moment. Pour cela il faut que l'expression booléenne contienne au moins une variable qui sera modifiée dans la boucle.

`faire ... tant que`

- **Syntaxe**

```
faire
       <instructions>
tant que <expression booléenne>
```

- **Fonctionnement**

La condition est placée après les instructions, elles sont exécutées donc au moins une fois, et persistent tant que la condition reste satisfaite. Bien entendu, même contrainte quant à l'expression booléenne.

L'itérative : `pour ... fin pour`

- **Syntaxe**

Elle permet de répéter une série d'instructions un nombre déterminé de fois.

```
pour ident_var ⟵ <expr_debut> jusqu'a <expr_fin> [decroissant] faire
       <instructions>
fin pour
```

- **Fonctionnement**

La variable est nécessairement de type scalaire : entier, caractère ou énumération. Les expressions de début et de fin doivent être compatibles avec elle. Elle prend successivement toutes les valeurs comprises entre les deux bornes (dans l'ordre décroissant si indiqué !). Elle ne peut pas être modifiée dans la boucle !

Les procédures et fonctions

Les paramètres

Locaux - Globaux

Toute routine peut définir des paramètres, qui sont autant de données en entrée, fournies lors de l'appel à la routine depuis un autre endroit de l'algorithme. La structure de ces paramètres est définie immédiatement après l'entête de la routine (comme indiqué dans la syntaxe des procédures et fonctions ci-dessous). On distingue deux catégories de paramètres : les *locaux* et les *globaux*.

- Les paramètres locaux indiquent que le paramètre effectif passé lors de l'appel à la routine sera en fait copié localement, et que la routine travaillera alors sur la copie locale (on parle de *passage par valeur*). De la sorte, toute modification effectuée par la routine ne sera pas répercutée sur la donnée passée en paramètre. Ce qui fait qu'un paramètre local peut recevoir une expression comme paramètre effectif, et non pas obligatoirement une variable.
- Les paramètres globaux ne gérèrent pas de copie locale du paramètre effectif. Ils ne font qu'un avec la donnée qui leur est passée. En appelant une routine qui a des paramètres globaux, il faut alors fournir pour ceux-ci un nom de variable existante, et non une expression, puisque la routine est susceptible de modifier la variable passée (on parle de *passage par variable* ou *par adresse*).

Déclaration des paramètres

La déclaration des paramètres se fait comme une déclaration de variables. Deux parties pour séparer les paramètres locaux des globaux :

```
parametres locaux
    ident_type  ident_param1, ident_param2, ...
    ...

parametres globaux
    ident_type  ident_param1, ident_param2, ...
    ...
```

Note : L'ordre de déclaration n'a pas d'importance, mais il ne peut y avoir qu'une seule déclaration de paramètres locaux et qu'une seule déclaration de paramètres globaux.

Déclaration des routines

Les procédures

Une procédure est une routine (un bloc de code) qui exécute un traitement puis rend la main. On peut ainsi isoler une partie de l'algorithme global et éventuellement l'appeler plusieurs fois en gardant un code structuré et modulaire.

- **Syntaxe**

```
algorithme procedure ident_procedure
      [<declarations parametres>]
      [<declarations>]
debut
      <instructions>
fin algorithme procedure ident_procedure
```

Les déclarations locales

Les déclarations sont les mêmes que pour un algorithme simple, à l'exception des routines : si la procédure ou fonction en cours doit en appeler une autre, cette dernière doit avoir été déclarée avant (toujours selon le même principe "je ne parle que de ce que je connais" qui devrait être beaucoup plus souvent appliqué !). On peut donc déclarer des constantes, des types et des variables. Toutefois il est très rare de déclarer des constantes et des types locaux à une routine, ceux-ci devant être généralement partagés par tout l'algorithme (voir la partie Portée des identifiants ci-dessous). Pour les variables par contre, il est fortement conseillé de n'utiliser que des variables locales (déclarées dans la routine en cours) et non pas des variables globales (en fait ce n'est pas fortement conseillé : c'est **obligatoire !**).

Les fonctions

Une fonction est une routine (un bloc de code) effectuant un traitement et renvoyant une valeur.

- **Syntaxe**

```
algorithme fonction ident_fonction : ident_type_retourné
       [<declarations parametres>]
       [<declarations>]
debut
       <instructions>
fin algorithme fonction ident_fonction
```

Le type retourné ne peut être qu'un type simple (entier et dérivés, réel, caractère, booléen, pointeur) ou une chaîne.

La procédure `retourne`

La fonction retourne une valeur au moyen de la procédure système **retourne**. Celle-ci doit donc obligatoirement figurer dans les instructions de la fonction.

Le **retourne** est débranchant : son exécution termine la fonction. Toute instruction placée après ne sera donc pas prise en compte. Cette propriété est quelquefois utilisée pour permettre la sortie d'un algorithme avant la fin. On trouvera donc la procédure **retourne** appelée sans arguments dans des procédures (voilà je l'ai dit...).

Il est très fortement déconseillé d'utiliser le *"pouvoir débranchant"* de **retourne**, cela ne plait pas du tout à votre chargée de TD ! (Arf! dit le WikiCoder Krisboul)

Portée des identifiants

La portée d'un identifiant est la partie de l'algorithme dans laquelle cet identifiant est reconnu conformément à sa déclaration, c'est-à-dire l'ensemble des lignes de codes dans lesquelles l'utilisation de cet identifiant fera référence à la donnée qu'il définit.

Un identifiant sera *"visible"* dans l'algorithme où il a été déclaré et dans tout sous algorithme appelé, mais jamais à un niveau plus haut.

Il est possible d'avoir des *"conflits"* de portée, c'est-à-dire qu'un niveau d'imbrication de routine déclare un identifiant portant le nom d'un autre identifiant existant à un niveau supérieur. La règle alors est la suivante : la version la plus proche (la plus profondément imbriquée) de l'identifiant a la priorité.

Une règle à ajouter pour la construction des identifiants : dans un même bloc de déclarations, deux identifiants ne peuvent être identiques. En revanche, s'ils sont déclarés dans des portées différentes, deux identifiants peuvent être identiques, mais ils engendrent alors un conflit de portée.

(Nathalie "Junior" Bouquet)

- Dernière modification de cette page le 28 mars 2012 à 13:23.