Spécifications du langage While

Ce langage est une variante mineure du langage While initialement proposé par Olivier Ridoux.

Table des matières

A. Grammaire du langage While

Dans cette grammaire, les non terminaux sont en gras, les lexèmes sont en rouge.

Program ⟶**Function Program** | **Function**

Function⟶ 'function' Symbol ':' **Definition**

Definition⟶ 'read' **Input** '%' **Commands** '%' 'write' **Output**

Input⟶ **InputSub** | ε

InputSub ⟶Variable ',' **InputSub** | Variable

Output ⟶Variable ',' **Output** | Variable

Commands⟶**Command** ';' **Commands**

Commands⟶**Command**

Command ⟶ 'nop'

Command ⟶**Vars** ':=' **Exprs**

Vars ⟶Variable ',' **Vars** | Variable

Exprs⟶**Expression** ',' **Exprs** | **Expression**

Command ⟶ 'if' **Expression** 'then' **Commands** ['else' **Commands**] 'fi'

Command ⟶ 'while' **Expression** 'do' **Commands** 'od'

Command ⟶ 'for' **Expression** 'do' **Commands** 'od'

Command ⟶ 'foreach' Variable 'in' **Expression** 'do' **Commands** 'od'

ExprBase⟶ 'nil' | Variable | Symbol

ExprBase⟶ '(' 'cons' **Lexpr** ')' | '(' 'list' **Lexpr** ')'

ExprBase⟶ '(' 'hd' **ExprBase** ')' | '(' 'tl' **ExprBase** ')'

ExprBase⟶ '(' Symbol **LExpr** ')'

Expression ⟶**ExprBase**

Expression ⟶**ExprBase** '=?' **ExprBase**

LExpr⟶**ExprBase LExpr** | ε

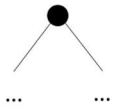
Variable = *Maj (Maj|Min|Dec)*('!'|'?') ?*

Symbol = *Min (Maj|Min|Dec)*('!'|'?') ?*

B. Domaine de calcul

I. Type de données et simulation des autres types

Le langage While ne permet de manipuler qu'un seul type de données : des arbres binaires.



Dénote d'un nœud interne de l'arbre ayant deux fils

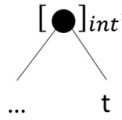
$\otimes$ Dénote d'un arbre vide

symb Dénote d'un arbre feuille contenant un symbole (lexème Symbol de la grammaire) – peut être assimilé à une chaine.

Les types standards, tels que les entiers, les booléens ou encore les chaines de caractères sont simulés i.e. encodé sous la forme d'un arbre binaire qui sera interprété en fonction des besoins.

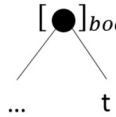
Simuler un entier :

$[\bullet]_{int} = [t]_{int} + 1 \; avec \; [symb]_{int} = 0 \; et \; [\otimes]_{int} = 0$



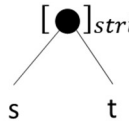
Simuler un booléen :

$[\bullet]_{bool} = true \; et \; [symb]_{bool} = false \; et \; [\otimes]_{bool} = false$



Simuler une chaine de caractères :

$[\bullet]_{string} = [s]_{string} . [t]_{string}, [symb]_{string} = symb.str \; et \; [\otimes]_{string} = \varepsilon$



II. Les expressions

Le rôle des expressions est de dénoter des arbres binaires.

- **nil** correspond à l'arbre vide (⊗)
- Si **Var** est une variable, elle est remplacée par sa valeur quand elle est utilisée
  - Note : une variable non initialisée prend la valeur **nil** par défaut.
- **Symb** : un symbole i.e. un arbre feuille (lexeme Symbol)
- **cons** : construction d'arbres binaires
  - **(cons) = nil** construit un arbre vide
  - **(cons T) = T** retourne l'arbre T
  - **(cons A B)** construit un arbre binaire ayant A pour fils gauche et B pour fils droit
  - **(cons T_1 T_2 … T_n) = (cons T_1 (cons T_2 … (cons T_{n-1} T_n) …)**
- **list : construction d'une liste**
  - **(list) = nil** construit une list vide
  - **(list T) = (cons T nil)** construit une liste a un élément
  - **(list T_1 T_2 … T_n) = (cons T_1 (cons T_2 … (cons T_n nil) …)** construit une liste à n éléments
- **(hd T)**
  - si **T = (cons A B)** alors retourne **A**
  - si **T = Symb** alors retourne **nil**
  - si **T = nil** alors retourne **nil**
- **(tl T)**
  - si **T = (const A B)** alors retourne **B**
  - si **T = Symb** alors retourne **nil**
  - si **T = nil** alors retourne **nil**
- **(f v_1 … v_n)** : appel de la fonction f avec les paramètres v_1 à v_n.
  - f doit être une fonction définie par l'utilisateur
  - l'appel de f doit comporter le bon nombre de paramètres
  - vérification de type à effecteur dans le compilateur…

La gestion des erreurs à l'exécution peut s'avérer assez complexe et ne pas les gérer proprement est irresponsable. Le langage While prend le parti de s'arranger pour qu'il n'y ait pas d'erreur à l'exécution. Pour ce faire, **nil** devient une valeur par défaut **pour les variables non initialisées** et **pour les opérations non définies** : (hd nil) = (tl nil) = (hd symb) = (tl symb) = nil.

III. Mémoire et variables

La mémoire du processus stocke :

1. Des arbres binaires
2. Des relations entre variables et arbres binaires

Il n'y a pas de variable dans les arbres binaires et ces derniers ne peuvent pas être modifiés.

Une variable s'évalue dans la mémoire courante et a nil pour valeur par défaut.

En mémoire, on stocke l'association entre une variable et l'arbre binaire qui y est associé.

Les variables sont locales à la fonction dans laquelle elles apparaissent, pas de variable globale.

Les variables ne permettent pas les effets de bord. Soit le code suivant :

1. X := (**cons nil nil**)
2. Y := X
3. X := **nil**

Après l'exécution de la ligne 3, Y vaut (cons nil nil). Le seul moyen de changer la valeur de Y est d'utiliser la syntaxe Y := expression. Cela empêche, entre autres, la création de structures circulaires.

IV. Commandes et structures de contrôle

Le rôle fondamental des commandes en While est de modifier la relation entre les variables et les arbres binaires. Rien d'autre n'est modifiable. Ci-dessous, vous trouverez une liste des commandes et des structures de contrôle du langage While.

- **nop**
  - Une commande qui ne fait rien. Ceci est utile pour tester la génération de code.
- C1 **;** C2
  - Exécute la commande C1 puis la commande C2. Le point-virgule est un opérateur correspondant à l'enchainement de commandes.
- **if** E **then** C1 **fi**
  - si $[E]_{bool}$, exécute C1, sinon ne fait rien.
- **if** E **then** C1 **else** C2 **fi**
  - si $[E]_{bool}$, exécute C1, sinon exécute C2.
- **while** E **do** C **od**
  - si $[E]_{bool}$, exécute C. Répète l'opération tant que $[E]_{bool}$.
  - *Cette structure de contrôle peut boucler indéfiniment*
- **for** E **do** C **od**
  - Répète $[E]_{int}$fois la commande C.
  - Attention : l'exécution de C ne doit pas perturber le décompte du for.
    - Soit v = $[X]_{int}$
    - **for** X **do** X := (**cons nil** X) **od** double la longueur de X
    - En sortie de boucle X -> (**cons nil** (**cons nil** … X …))
      v fois
  - *Cette structure de contrôle ne peut pas boucler indéfiniment*.
- **foreach** X **in** E **do** C **od**
  - Pour chaque élément X de E, répéter C.
  - Exécute $[E]_{int}$fois la commande C
  - Soit v la valeur de E en entrée dans la boucle
  - Exécute C avec X ← (hd v) puis recommencer avec v = (tl v)
- $V_1, V_2, \ldots, V_n = E_1, E_2, \ldots, E_n$
  - Evalue toutes les expressions $E_1, E_2, \ldots, E_n$ En puis stocke les résultats dans les variables $V_1, V_2, \ldots, V_n$
  - Attention le comportement escompté est le suivant
    - $R_1 := E_1, R_2 := E_2, \ldots, R_n := E_n$
    - Puis
    - $V_1 := R_1, V_2 := R_2, \ldots, V_n := R_n$
- $V_1, V_2, \ldots, V_n = (f\ E_1\ E_2 \ldots E_m)$
  - Evalue la fonction $(f\ E_1\ E_2 \ldots E_m)$ et stocke les valeurs de retour dans $V_1, V_2, \ldots, V_n$
  - Attention : la fonction f doit prendre m paramètres et retourner n valeurs
    - **function** f : **read** X1, …, Xm % … % **write** R1, …, Rn
  - Vérification de type faite à la compilation

V. Définitions de fonctions

La syntaxe de définition d'une fonction est la suivante :

function symbol :

read I1, …, In

%

C

%

write O1, …, Om

Dans cette définition, symbol est le nom de la fonction I1, …, In sont les paramètres de la fonction et O1, …, Om sont les valeurs de retour de la fonction. C est un bloc de code contenant une succession de commandes dans le format présenté dans la section précédente.

Il est à noter que dans le langage While, le nombre de paramètres d'une fonction peut être 0 (ici, il s'agit d'une fonction renvoyant une valeur constante). Par contre le nombre de valeur de retour est toujours supérieur ou égal à 1. Il ne peut y avoir de fonction sans valeur de retour.

VI. Le programme principal

Le programme principal est décrit dans une fonction nommée *main*. Le nombre de paramètres et de valeurs de retour de la fonction main est choisi par l'utilisateur.

Deux questions se posent :

1. d'où viennent les paramètres de main ?
2. où vont les valeurs retournées par main ?

i. Paramètres en entrée

Les paramètres en entrée du programme seront passés via la ligne de commande. Soit monProgramme le programme While compilé par votre compilateur. Ce programme pourra accepter deux types de paramètres :

1. Des entiers. Dans ce cas, il s'agira de transformer l'entier en un arbre représentant sa valeur dans le monde while et passer cet arbre en paramètre de la fonction main compilée.
    - Exemple d'appel du programme :
      monProgramme 2
      ici 2 sera transformé en (cons nil (cons nil nil))
2. Des arbres représentés par une formule basée sur cons
    - Exemple d'appel du programme :
      monProgramme "(cons (cons (cons (cons ceci est) une) liste) nil)"
      ici la chaine de caractère sera interprétée pour construire l'arbre correspondant

Dans les deux cas, il s'agira donc de faire l'analyse syntaxique des paramètres de manière à fournir au programme main leur représentation dans le format interne de votre langage compilé.

Bien sûr lors de son lancement, le programme compilé vérifiera que le nombre de paramètres fourni est correct et affichera une erreur dans le cas contraire.

ii. Valeurs de sortie

La convention veut que les valeurs de sortie du programme principal soient affichées à l'écran en utilisant un pretty printer. Soit pp(T) le pretty printing de T, pp est défini comme suit :

- pp(nil) -> nil
- pp(symb) -> représentation de symb
- pp((cons int A)) -> $[A]_{int}$
- pp((cons bool A)) -> $[A]_{bool}$
- pp((cons string A)) -> $[A]_{string}$
- pp((cons A B)) avec $A \notin \{int, bool, string\}$ -> (cons pp(A) pp(B))

Exemple : le prettry printing de (cons int (cons nil (cons nil))) -> 1

C. Exemples de fonctions While

I. Fonctions sur les booléens

```
// true constant (for lisibility)
function true :
read
%
        Result := (cons nil nil)
%
write Result

// false constant (for lisibility)
function false :
read
%
        Result := nil
%
write Result

// Logical not
function not :
read Op1
%
        if Op1 then Result := (false) else Result := (true) fi
%
write Result

// Logical and
function and :
read Op1, Op2
%
        if (not Op1) then
                Result := (false)
        else
                if (not Op2) then
                        Result := (false)
                else
                        Result := (true)
                fi
        fi
%
write Result
```

II. Fonctions travaillant sur des entiers

```
// Addition of two numbers
function add :
read Op1, Op2
%
        Result := Op1 ;
        for Op2 do
                Result := ( cons nil Result )
        od
%
write Result

// Soustraction of two numbers (there is no negative number…)
function sub :
read Op1, Op2
%
        Result := Op1;
        for Op2 do
                Result := (tl Result)
        od
%
write Result

// Multiplication
function mul :
read Op1, Op2
%
        for Op1 do
                Result := (add Result Op2)
        od
%
write Result
```