

Liste primitives nécessaires pour le mode Warbot classique

Introduction :	2
If & co :	3
Si	3
Ou	3
Variables :	4
Actions :	5
Définir objectif	5
Supprimer objectif	5
Conditions :	6
A un objectif	6
Niveau de vie	6
Niveau de ressource	6
Inférieur, supérieur, égal...	7
Primitives à supprimer ou à remplacer :	8
Task	8
SelectUnite	8
TargetNearest	8
Near	8
Turn	9
HalfTurn	9
RandomTurn	9
Not full, not empty	9
Reloaded	9
Idée de conception	10
Bloc comportementale	10

Introduction :

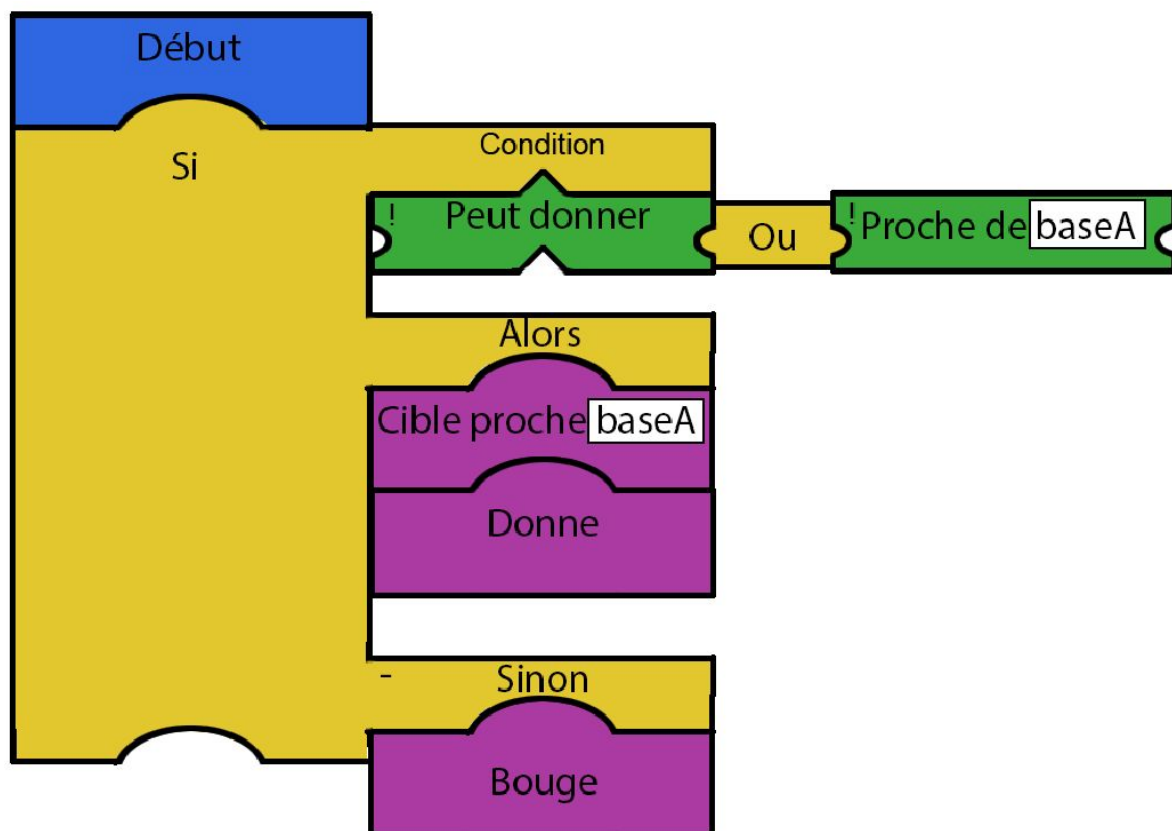
Actuellement, Warbot ne comprend que 3 catégories de primitives (Actions, Condition et Control). Au vu de nos attentes pour le logiciel, il nous semble obligatoire de créer une quatrième catégorie : Variable. Cette dernière contiendra les getters de l'unité.

Le nom de la catégorie "Control" (qui comprend les if et task) ne me plaît pas et il faudra lui trouver un nouveau nom. Pour le moment, elle a été rebaptisé "If & co" parce que c'est déjà plus logique. Le principal problème est que le nom le plus approprié pour cette catégorie serait "Condition" et elle est déjà utilisée par une autre catégorie qui, elle, est composé des primitives qui retourneront des booléens. Ce nom me semble plus approprié à ce groupe qu'à celui des "If & co".

Au vu du public visé qui s'est étendu pour viser des personnes plus jeunes qui ne maîtrise ni l'informatique ni l'anglais, il a été proposé de changer la langue de Warbot pour le français.

Les noms des primitives listés dans ce document seront donc en français. **En revanche, nous ne proposons pas de traduction du nom des primitives déjà existantes dans ce document.**

Exemple de code après la refonte du langage :



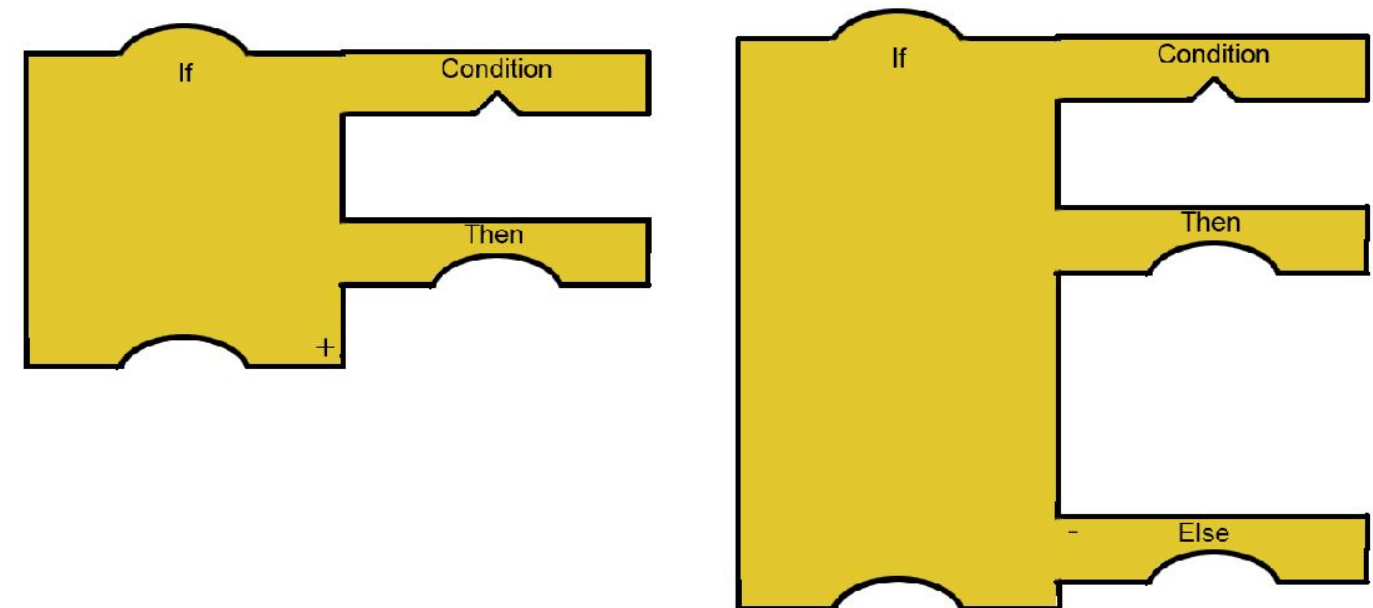
If & co :

Si

Le if classique. Par défaut, la primitive Si est composé des blocs sections “condition” (dans laquelle on place notre condition), “alors” (dans laquelle on place les actions qu’on souhaite exécuter si la condition est vraie). En cliquant sur le bouton “+”, on ajoute un troisième bloc :

“sinon” où l’on place les actions que l’on souhaite exécuter si la condition est fausse.

On peut à tout moment supprimer le bloc “sinon” en cliquant sur le bouton “-”, ce qui supprimera les actions qui étaient attachées à ce bloc.



Ou

L’opérateur or classique. Comme l’opérateur and est exprimé par le fait de placer des primitives de conditions les une en-dessous des autres, nous avons décidé d’exprimer les ou en posant une primitive entre deux primitives alignées. Les primitives à droite de la primitive “ou” perdent leurs “pics”, pour éviter que les utilisateurs puissent exprimer le même and en plaçant une primitive à plusieurs endroits différents.



Variables :

- Vie actuelle : la vie restante de l'unité.
- Ressource actuelle : la quantité de ressource de l'unité.
- Porté tir : la portée de l'arme de l'unité
- Distance objectif : la distance entre l'unité et son objectif
- Distance target : la distance entre l'unité et sa target
- Nombre membre : compte le nombre d'unité au sein d'un groupe dont le nom a été passé en paramètre
- Objectif : position de l'objectif sur le long terme de l'unité
- Cible : position de l'objectif à court terme de l'unité.
- Constante : permet d'écrire un nombre ou du texte. Vaut null si rien n'est écrit dans la constante. **Les constantes ne permettent pas de faire de calcul.**

Actions :

Les actions sont les primitives qui agissent sur l'unité. Elles peuvent se placer un peu partout : en dessous de la primitive "Départ" ou d'une primitive "si" ainsi que dans les blocs "alors" et "sinon" de la primitive "si". Certaines actions terminent le tour de l'unité (Donner, Avancer, Tirer...) et ont donc une forme différente qui suggère le fait que l'unité ne pourra rien faire après cette action :



Définir objectif

L'objectif est essentiel à la création de meilleures équipes car ce qui s'en rapproche le plus à l'heure actuelle est la cible mais elle se perd à chaque début de tour.

"Définir objectif" permet d'affecter la valeur présente dans la variable cible dans la variable objectif afin que l'unité ne perde pas son objectif de vue.

Supprimer objectif

Une fois un objectif accompli, il doit être possible d'exprimer le fait que l'unité n'a plus d'objectif. Comme il n'existe pas de moyen de dire que l'on a plus de cible et qu'une telle primitive serait inutile vu que la cible se réinitialise à chaque nouveau tour, il vaut mieux modifier directement la valeur.

Conditions :

Les conditions se mettent dans le bloc “condition” des primitives “si”. Pour exprimer ce fait, la forme des primitives est différente de celle des actions.

Il est possible d’exprimer l’inverse de la condition en cliquant sur le “!” eu haut à gauche de la primitive. Après un clic, la couleur de la primitive change :



A un objectif

Renvoi true si l’unité a un objectif, false sinon.

Niveau de vie

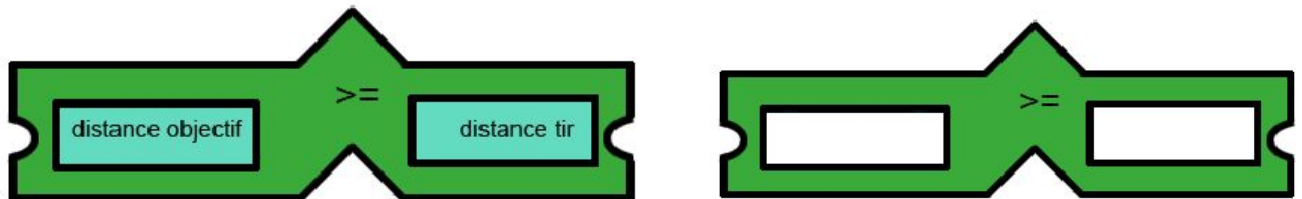
Prend en paramètre un chiffre entre 0 et 1. Retourne true si la vie de l’unité est supérieure ou égale à ce chiffre multiplié par la vie maximale de l’unité.

Niveau de ressource

Prend en paramètre un chiffre entre 0 et 1. Retourne true si la quantité de ressource de l’unité est supérieure ou égale à ce chiffre multiplié par la capacité maximale de l’inventaire de l’unité.

Inférieur, supérieur, égal...

Prend deux variables en paramètre. Renvoie le résultat du test d'égalité/supériorité/infériorité.



L'opérateur se modifie via un menu déroulant sous cette forme :

Condition 1	Opérateur	Condition 2
	< > =	

Les opérateurs disponibles sont : égalité, supérieur, supérieur ou égal, inférieur, inférieur ou égal. Le test de non égalité se fait en inversant l'opération d'égalité en cliquant sur le "!".

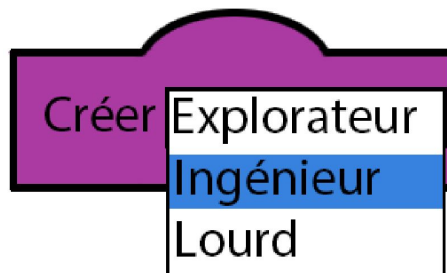
Primitives à supprimer ou à remplacer :

Task

Task est une primitive qui a actuellement la même fonction que la primitive “si” mais ne dispose pas de bloc “sinon”. Au vu des modifications sur cette dernière, task est devenue inutile : il faut donc la supprimer.

SelectUnite

Les primitives SelectWarExplorer, SelectWarHeavy, SelectWarEngineer... ne servent que pour les primitives “peut créer” et “créer unité”. Il est plus intéressant de modifier les primitives “peut créer” et “créer unité” pour qu’elles prennent en paramètre un type d’unité, rendant ainsi les primitives de sélection inutiles :



TargetNearest

Plutôt que d’avoir près de vingt primitives différentes qui ont en réalité le même objectif, il est plus intéressant de créer une simple primitive “Cibler le plus proche” qui prend en paramètre le type de la cible voulu (base alliée, message, explorateur ennemi...) sous la même forme que ce qui est prévu pour la création d’unité.

Near

Pour les mêmes raisons que pour TargetNearest, il est plus judicieux d’ajouter un paramètre à une simple primitive “Proche de” plutôt que de conserver autant de primitives silaires.

Turn

Actuellement, cette primitive fait tourner l'unité dans la direction de sa cible. Cependant, nous avons maintenant deux variables vers lesquelles nous souhaitons que l'unité se tourne : la cible et l'objectif. Il faut donc rajouter un paramètre sous la forme d'un menu déroulant pour laisser l'utilisateur choisir s'il veut que son unité se tourne vers l'objectif ou la cible.

HalfTurn

Il est important de pouvoir faire demi-tour. Il nous semble tout aussi important de pouvoir tourner d'un angle précis, sans forcément cibler quelque chose. Nous proposons d'ajouter un paramètre à cette primitive et de la renommer "tourner". Le paramètre sera un chiffre compris entre 0 et 360.

RandomTurn

Nous ne voyons aucun intérêt à rajouter de l'aléatoire dans ce jeu. Les seuls moments où il peut être intéressant de tourner dans un sens quelconque est lorsque l'on souhaite faire fuir notre unité. Il est alors plus intéressant de contrôler parfaitement le changement d'angle pour fuir efficacement.

Not full, not empty

Vu qu'il sera possible d'exprimer la négation d'une condition, ces primitives sont devenues inutiles.

Reloaded

Plutôt que de tester si l'unité est rechargé, il est plus intéressant de savoir si l'unité peut tirer : en plus du test de rechargement, cette nouvelle primitive fera également un test pour savoir si la distance entre la cible et l'unité est inférieure à la portée maximale de l'arme.

Idée de conception

Bloc comportementale

Ce bloc se diviserait en deux parties : un premier bloc s'intégrerait dans le comportement de base. Une fois ajoutée, une nouvelle zone permettant de définir un comportement s'afficherait. Fonctionnalité pouvant être utile dans le cas d'une unité ayant des comportements complexes différents en fonction du groupe auquel il appartient par exemple.

