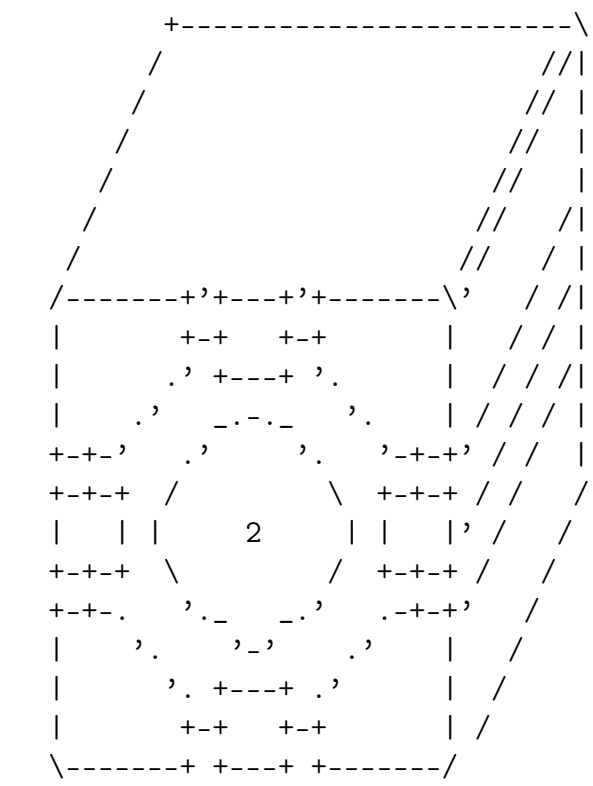
CPU - Cube Processing Unit

audebe_r - halfr
hervot_p - Dettorer
eddequ_n - nass
pruvot_a - Chaf

```
 _   _    _    _   _ ____   ____ ____      _    _____ _____
| \ | |  / \  | \ | |  _ \ / ___|  _ \    / \  |  ___|_   _|
|  \| | / _ \ |  \| | | | | |   | |_) |  / _ \ | |_    | |
| |\  |/ ___ \| |\  | |_| | |___|  _ <  / ___ \|  _|   | |
|_| \_/_/   \_\_| \_|____/ \____|_| \_\/_/   \_\_|     |_|
 ____   ___  _   _ _____ _____ _   _    _    _   _  ____ _____
/ ___| / _ \| | | |_   _| ____| \ | |  / \  | \ | |/ ___| ____|
\___ \| | | | | | | | | |  _| |  \| | / _ \ |  \| | |   |  _|
 ___) | |_| | |_| | | | | |___| |\  |/ ___ \| |\  | |___| |___
|____/ \___/ \___/  |_| |_____|_| \_/_/   \_\_| \_|\____|_____|
```



Sommaire

1 Ré-Introduction **4**
1.1 NANDCRAFT 4

2 Circuits logiques **5**
2.1 Quelques petits rappels 5
2.2 Demi-additionneur, additionneur et additionneur complet 5
2.3 Unité Arithmétique et Logique (UAL) 5
2.4 Pour la suite… 6

3 Assembleur **7**
3.1 Émulateur 7
3.1.1 OCamlSDL 7
3.1.2 Exemples de programmes émulés: 8
3.1.3 Le fonctionnement 10
3.2 Le debugger 11

4 Machine virtuelle **12**
4.1 Rappels 12
4.1.1 Le rôle de la machine virtuelle 12
4.1.2 Le langage intermédiaire 12
4.2 Ce qui a été fait et comment 12
4.2.1 LABEL label 13
4.2.2 GOTO label 13
4.2.3 IF-GOTO label 13
4.2.4 CALL function nb_arg 14
4.2.5 FUNCTION nb_local_var 14
4.2.6 RETURN 15
4.2.7 Recapitulatif de la structure d'une commande 15
4.3 Ce qu'il reste à faire 16

5 Compilateur **17**
5.1 Ce qui a été fait 17
5.1.1 Avance prise sur la prochaine soutenance 17
5.2 Présentation du compilateur 17
5.2.1 Compilation du langage craft 17
5.2.2 Compilation séparée 17
5.2.3 Informations de *debug* 17
5.2.4 Les avertissements 19
5.3 Fonctionnement du compilateur 19
5.3.1 La table des symboles 19
5.3.2 Générateur de code intermédiaire 21
5.3.3 Le cœur du compilateur 21
5.3.4 Gestion des entrées sorties 22

5.4 Présentation du langage `craft` 22
5.4.1 Les directives 23
5.4.2 Les classes 23
5.4.3 Les variables 24
5.4.4 Les instructions 26
5.4.5 Les structures de contrôle du flux d'exécution 26
5.4.6 Les expressions 27
5.4.7 Construction et destruction des objets 28
5.5 Ce qu'il reste à faire 29

6 Annexes **30**
6.1 Web cloud green it technologies agiles et peer programming 30
6.2 « Bitbucket c'est pas stable » 30
6.3 Le chan IRC, repaire des gens d'une autre époque™ 30

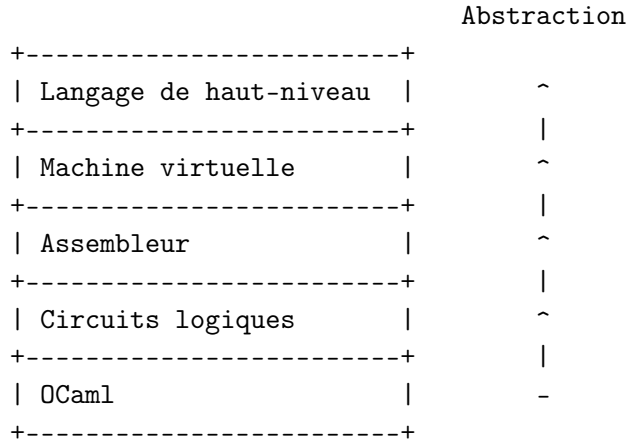
7 Le mot de la fin **32**

1 Ré-Introduction

1.1 NANDCRAFT

NANDCRAFT est un projet de création d'ordinateur en passant par toutes les abstractions, de la porte logique jusqu'au langage de haut niveau.

Un beau schéma vaut mieux qu'un long discours, résumant chacun de nos sous-projets :



Nous vous présentons dans ce rapport notre avancement sur chacune de ces parties.

2 Circuits logiques

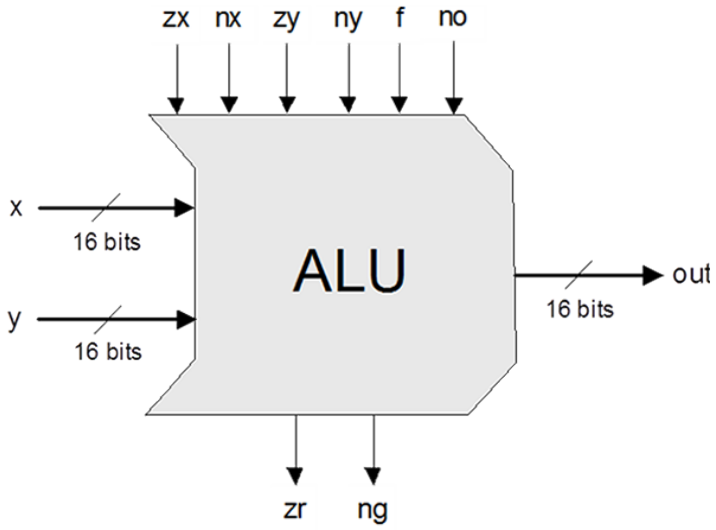
2.1 Quelques petits rappels

Lors de la dernière soutenance nous avons créé une gamme de portes logiques en partant de la simple porte NAND. Nous avons ainsi à notre disposition des portes simples comme les portes AND ou OR ou comme les portes complexes que sont les multiplexeurs/démultiplexeurs… Je rappelle qu'ici nous ne travaillons que sur du binaire de 16 bits pour des raisons évidentes de simplicité. Nos portes sont donc capables de gérer au maximum des entiers de cette taille.

2.2 Demi-additionneur, additionneur et additionneur complet

Ici, nous avons construit trois portes différentes pour nous permettre d'effectuer des additions binaires. En effet, nous remarquerons que bon nombre d'opérations se rapportent souvent à une addition (x - y = x + (-y)). Premièrement le demi-additionneur va nous permettre d'additionner deux bits entres eux. Ensuite nous avons fabriqué un additionneur afin d'additionner trois bits entre eux. Finalement, nous avons mis en place un additionneur complet capable d'additionner deux nombres de 16 bits. La notion de surcharge au niveau de la taille n'est jamais détectée ni prise en charge.

2.3 Unité Arithmétique et Logique (UAL)



Nous arrivons maintenant à la pièce centrale de notre unité logique : l'unité arithmétique et logique. Elle va nous permettre d'effectuer la plupart des opérations nécessaires à notre machine pour fonctionner. L'UAL prend en entrée deux entiers de 16 bits ainsi que ce qu'on appelle des instructions de contrôle (zx, nx, zy, ny, f, no) qui, comme leur nom l'indique donne à l'utilisateur plus de contrôle sur l'opération en cours :

- nx / ny permettent d'obtenir la négation des entrées x ou y
- zx / zy permettent de fixer les entrées x ou y a zéro

- f permet de spécifier l'opération désirée, ici 1 pour une addition et 0 pour un AND
- no permet d'obtenir la négation de la sortie

En sortie de l'UAL nous aurons bien sûr le résultat de l'opération mais aussi deux informations supplémentaires (zr et ng) qui spécifient à l'utilisateur si la sortie est négative (ng) ou nulle (zr).

Voici le tableau du fonctionnement de notre UAL en fonction des différents paramètres passés en entrée:

These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(X,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

2.4 Pour la suite...

Dans l'optique des prochaines soutenances, nous continuerons l'implémentation des portes logiques séquentielles que nous avons déjà entamé dès la première soutenance et que nous avons encore avancé.

3 Assembleur

Le processeur, ou CPU (pour Central Processing Unit), est le composant dans un ordinateur qui exécute les programmes informatiques. Le processeur récupère les informations dans les différentes mémoires mises à sa disposition, effectue les calculs requis par le programme et peut dialoguer avec les appareils périphériques de l'ordinateur via la technique de `MMIO` (`Memory-maped Input/Output`) qui consiste en une association de la mémoire et des registres des périphériques à des adresses mémoires prédéfinies dans la `RAM`.

3.1 Émulateur

Notre processeur virtuel est composé d'une Unité arithmétique et logique (en anglais ALU) qui gère les calculs arithmétiques et les tests pour des comparaisons de valeur et de deux registres. Il lit les informations contenues dans la ROM et a un accès en lecture et écriture dans les autres mémoires c'est à dire le registre de pointeur d'instructions, de destination qui récupère les informations à la sortie de l'ALU et permet un stockage « intermédiaire » de valeurs si nécessaire et la RAM représentée par un tableau dont l'accès se fait via le registre de pointeur d'instruction.

La gestion du clavier et de l'écran se fait par MMIO. L'optimisation de l'écran et autres modifications ont été nécessaires pour le bon fonctionnement de l'émulateur.

3.1.1 OCamlSDL

- L'écran
  Notre écran, fonctionnant en `MMIO`, a pour résolution 512 par 256. Nous l'avons donc associé aux adresses RAM allant de 16384 à 24575 ce qui permet d'associer chaque bit de chaque contenu d'adresse RAM à un pixel. Ceci est possible via la fonction `put_pixel_color` d'OCamlSDL[1].

  Un bit à une place n et à l'adresse A définie la couleur d'un pixel aux coordonnées:

  ```
  x = n + 16 * ((A - 16384) modulo 32)
  y = (A - 16384) / 32
  ```

  Refaire les calculs de correspondance pour chaque bit de chaque adresse ram lors de tous les cycles du cpu étant évidemment très long. C'est pourquoi nous ne refaisons les correspondances que dans les adresses RAM modifiées au fur et à mesure ce qui divise le nombre de $\text{put}_{\text{pixelcolor}}$ effectué par environ 90000. Le rafraîchissement de l'écran a également été une problématique car rafraîchir l'écran à chaque instruction modifiant la RAM reste énorme et bien trop lourd. Nous avons donc choisi d'actualiser l'écran à chaque fois que la case 16383 sera modifiée.

[1] **Cassedédi** Le TP d'OCamlSDL de M. Burelle Marwan est tout de même le 4ème résultat sur Google de « OCamlSDL ».

3.1.2 Exemples de programmes émulés:

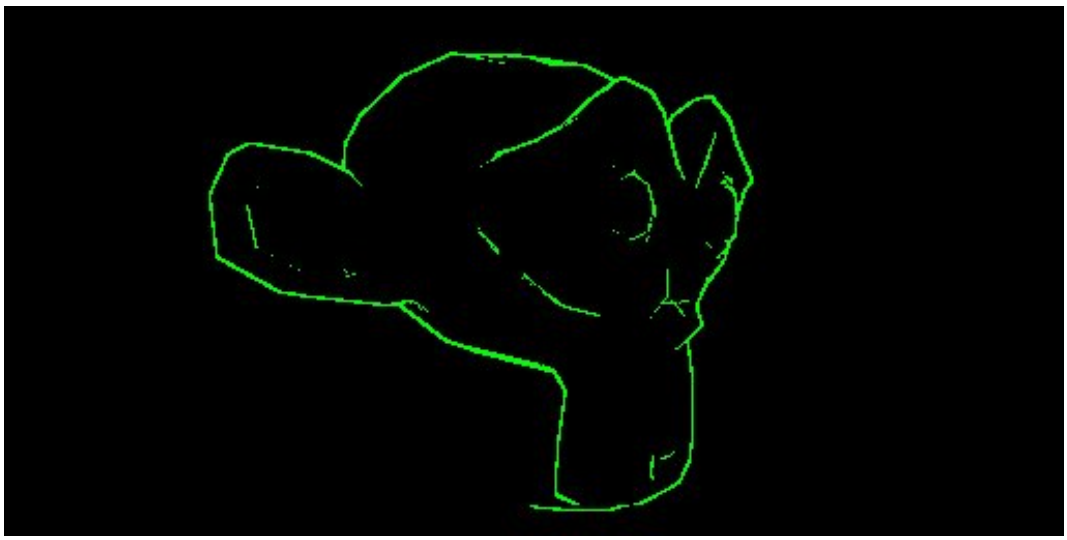


Figure 1: animation 1

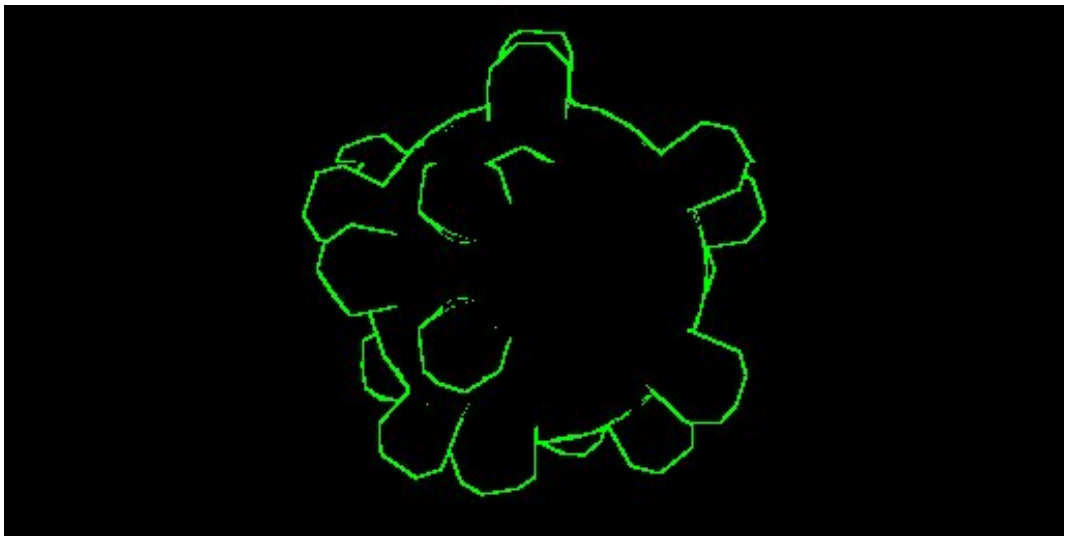


Figure 2: animation 2

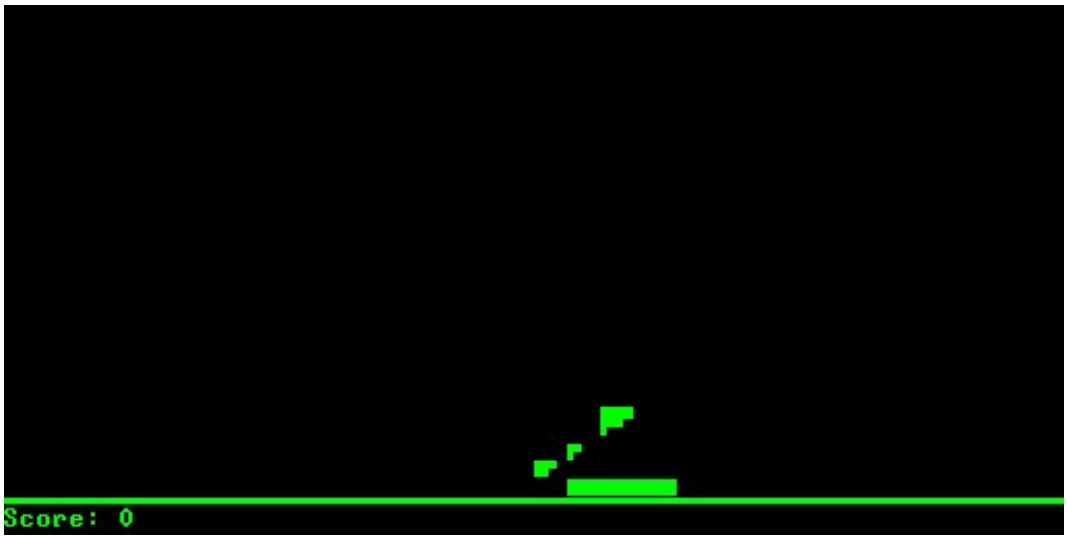


Figure 3: pong

- Le clavier
  Notre clavier interagit avec la RAM à l'adresse 24576. En effet, le code ASCII de la touche enfoncée est traduit en binaire sur 16 bits puis insérée dans la RAM à la place prédéfinie. Ainsi, le processeur peut accéder à tout moment à la lettre donnée par l'utilisateur. Nous utilisons également SDL pour récupérer les évenements du clavier. Lorsque aucune touche n'est appuyée, la RA:xM à l'adresse 24576 contient 0 sur 16 bits. Pour les lettres, les majuscules sont gérées. Les caractères spéciaux gérés sont les suivants:
  - newline -> 128
  - backspace -> 129
  - left, up, right, down -> 130,131,132,133
  - home -> 134
  - end -> 135
  - page up, down -> 137
  - instert -> 138
  - delete -> 139
  - escape -> 140
  - f1-f12 -> 141-152
- Rappels: Assembleur et langague assembleur
  Nous avons dû perfectionner notre assembleur qui traduisait des programmes en langage assembleur vers du langage binaire avec une table des symboles. La syntaxe de notre langage assembleur est la suivante:
  - Commentaires : **//commentaire**

- Pseudo-commande : **(label)**
- A-instruction : **@label** ou **@1123**
- C-instruction : **destination=calcul;saut**

Pour plus de facilités, les commentaires peuvent désormais être sur la même ligne qu'une instruction

3.1.3 Le fonctionnement

L'émulateur récupère les instructions unes à unes dans l'ordre puis analyse le type de l'instruction en cours. Si le premier bit est à « 0 » c'est une instruction d'adressage, la valeur des 15 bits suivant est donc stockée sous forme d'entier dans le registre A. Sinon, sont analysés les bits de 4 à 10 pour savoir quoi calculer (les opérations élémentaires effectuées par l'ALU) puis les bits de 11 à 13 pour savoir où placer le resultat de l'opération et enfin les bits 14 à 16 pour connaître le besoin de saut conditionnel (et si oui à quelle condition).

Les calculs arithmétiques élémentaires gérés par notre processeur sont les suivants[2] :

- « ET » et « OU » logiques
- additions
- soustractions (les nombres négatifs sont également gérés)
- « NON » logique

Récapitulatif de la répartition des bits pour les instructions A-instruction: forme « @Xxx » :

- premier bit: 0
- 15-bits restants: valeur en système binaire du nombre décimal représenté
- par la A-instruction.

C-instruction: forme « destination=computation;jump » :

- bit 1: 1
- bits 2,3: 1 et 1 (inutilisés)
- bits 4,5,6,7,8,9,10: fonction de l'opération logique à effectuer par l'ALU
- bits 11,12,13: fonction des registres dans lesquels placer la valeur calculée
- bits 14,15,16: fonction de l'absence de saut, de saut conditionnel ou inconditionnel

[2]Le simulateur disponible lors de la précédente soutenance était très peu fonctionnel et comportait de nombreuses erreurs qui s'accumulaient sur d'importants fichiers, rendant impossible l'émulation de gros programmes. Tout ceci est du passé, notre émulateur rockse actuellement de la patate frite suédoise.

Des améliorations possibles Actuellement, les instructions de calculs (par opposition aux instructions d'adressage) permettent soit un calcul par l'ALU soit un jump mais pas les deux à la fois, ces nouvelles instructions seront implémentées à la prochaine version de NAND-CRAFT en utilisant très certainement les deux bits non utilisés aux places 2 et 3.

En terme de code, la gestion par la ram de binaire sous forme d'entiers plutôt que de chaînes de caractères permettrait une optimisation en terme de poids d'instructions.

3.2 Le debugger

Le debugger initialement prévu pour la troisième soutenance est actuellement en cours de réalisation. La première version rudimentaire permet l'affichage de l'état des différents registres à chaque cycle du processeur virtuel et l'affichage de l'ensemble des opérations, flux de données et sauts effectués à chaque instruction. Nous préparons actuellement, en plus de ces informations élémentaires, un format de debuggage nous permettant de nous transmettre des méta-données au fil des étapes de modification du programme. L'idéal étant de connaître au plus bas niveau les noms et les emplacements des fonctions ainsi que leur portée au sein du fichier.

4 Machine virtuelle

4.1 Rappels

4.1.1 Le rôle de la machine virtuelle

La machine virtuelle a plusieurs utilités, mais son rôle principale est toujours d'être un intermédiaire entre le compilateur et la véritable machine.

Afin de simplifier la tache du compilateur, celui-ci traduit le langage de haut niveau non pas directement en langage machine, mais en un langage intermédiaire bas niveau : celui da la machine virtuelle. Celle-ci va alors traduire le programme obtenu en un langage directement exécutable par la machine.

L'intéret d'un tel système est le suivant : Le compilateur n'a besoin que d'une implémentation quelque soit la machine sur laquelle il compile, celle en langage intermédiaire. C'est la machine virtuelle qui aura besoin d'une version par machine, mais le langage source étant beaucoup plus simple, les différences entre ces versoins sont beaucoup plus petites.

4.1.2 Le langage intermédiaire

C'est un langage de bas niveau organisé autour d'une pile et de segments abstraits de mémoire, un programme en langage intermédiaire est une suite d'instruction sur cette pile.

Les segments utilisables sont :
`STATIC` : Les variables globales.
`ARG` : Les arguments de la fonction courrante.
`LCL` : Les variables locales de la fonction courrante.
`THIS` et `THAT`: Ces segments n'ont qu'un seul élément et peuvent se trouver n'importe où dans la ram, leur emplacement est définit par le segment suivant.
`POINTER` : Ce segment a deux éléments: les adresses de `THIS` et `THAT`.

Les instructions `PUSH segment index` et `POP segment index` permettent de déplacer des données entre la pile et les segments, les fonctions `ADD, SUB, NEG, AND, OR, EQ, GT, LT` et `NOT` réalisent des opérations arithmétique sur le haut de la pile.
`LABEL label` donne un nom à une ligne dans le programme, `GOTO label` et `IF-GOTO label` permettent de sauter à un label ou directement à l'adresse d'une instruction, sans condition avec `GOTO`, ou en vérifiant que la valeur du haut de la pile est `VRAIE` (on la représentera par un nombre positif) avec `IF-GOTO`, cette valeur est alors supprimée de la pile.

`FUNCTION nom nb_local` définit une fonction « nom » et la taille de son segment `LOCAL` (initialisé à 0).
`RETURN` finit une fonction, la valeur en haut de la pile est alors la valeur de retour.
Enfin, `CALL nom nb_arg` appel la fonction « nom » avec comme arguments les nb_arg valeurs les plus en haut de la pile, ce qui équivaut à `POP` ces nb_arg valeurs et à `PUSH` la valeur de retour de « nom ».

4.2 Ce qui a été fait et comment

L'implémentation de la machine virtuelle pour notre propre machine est entièrement opérationnelle. À la première soutenance, les fonctions permettant de déplacer des données entre

la pile d'execution et les segments de la ram (`PUSH` et `POP`) ansi que les opération arithmétiques et logiques sur ladite pile (`ADD, SUB, AND, OR`, etc.) étaient correctement traduites vers le langage assembleur de notre machine.

Pour la deuxième soutenance, il restait à implémenter les fonctions de contrôle de flux, à savoir : `LABEL, GOTO, IF-GOTO, FUNCTION, RETURN` et `CALL`.

4.2.1 LABEL label

La définition de label en langage assembleur est direct, il suffisait donc de réécrire la même chose dans une sytaxe différente.

```
label lol
```

devient alors simplement

```
(lol)
```

4.2.2 GOTO label

Elle aussi très simple à traduire, cette fonction se résume en assembleur à un jump sans condition. Comme nous pouvons directement utiliser les labels ou le numéro de l'instruction à laquelle sauter, un goto à l'adresse « adresse » se traduit en assembleur par :

```
@address
0;JMP
```

4.2.3 IF-GOTO label

C'est un goto avec un booleen en condition. Tout dans la ram est (de notre poin de vue) nombre, nous représentons donc le `VRAI` par un nombre positif (strictement) et le `FAUX` par un nombre négatif ou nul. Il faut donc récupérer la valeur en haut de la pile (cette opération est concidérée comme un pop, on décrémente alors le pointeur de pile de 1) puis sautons à l'adresse désirée avec comme condition le fait que le nombre soit strictement positif, ce qui donne en assembleur :

```
@SP
AM=M-1
D=M
@adress
D;JGT
```

Soit 5 instructions assembleur

4.2.4 CALL function nb_arg

L'appel à une fonction est une des opérations les plus compliquées que la machine virtuelle doit réaliser, mais raisonner avec une pile permet de s'en sortir efficacement.

Il faut sauvegarder l'état actuel de la fonction appelante, pour cela nous allons pousser les pointeurs propres à cette fonction sur la pile ainsi que l'adresse de retour, la fonction appelée pourra ensuite agir en concidérant que sa pile est initialement vide, l'instruction `RETURN` se chargera de récupérer la valeur de retour et de rétablire l'état de la fonction appelante. Les valeurs en haut de la pile sont les arguments de la fonction, leur nombre est spécifié avec `CALL`, ce qui permettra de bien définir le segment `ARGUMENT` de la fonction appelée.

La version simplifiée du code traduit en assembleur ressemble à :

```
1  M(SP)   <- return address (* On pousse l'adresse de retour *)
2  M(SP+1) <- LCL            (* Le pointeur du segment LOCAL *)
3  M(SP+2) <- ARG            (* Le pointeur du segment ARGUMENT *)
4  M(SP+3) <- THIS           (* Le pointeur THIS *)
5  M(SP+4) <- THAT           (* Le pointeur THAT *)
6  ARG     <- SP-nb_arg      (* On dnit le nouveau segment ARGUMENT *)
7  LCL     <- SP+5           (* et LOCAL *)
8  SP      <- SP+5           (* On place le pointeur de pile au dessus de tout *)
9  Jump to called function
10 (return address)          (* On dnit l'adresse de retour *)
```

Cette fonction prend en tout 59 instruction assembleur.

4.2.5 FUNCTION nb_local_var

Lors de la définition d'une fonction, tout ce qu'il y a à faire est de définir son début par un label et d'initialiser son segment `LOCAL`, le reste des segments étant préparé par `CALL` de façon normalisée.

Initialiser le segment `LOCAL` consiste à pousser autant de 0 sur la pile qu'il y a de variable local (ce qui est définit avec `FUNCTION`), on peu donc directement exécuter `PUSH CONSTANT 0` autant de fois que necessaire (une solution simple et temporaire, facilement optimisable)

Le label prend une instruction et un push de constante en prend sept, la traduction de `FUNCTION` se fait donc en 1 + 7 * nb_local_var.

4.2.6 RETURN

Deuxième fonction la plus grosse, son rôle est de récupérer la valeur de retour de la fonction appelée et de rétablire l'état de la fonction appelante.

En assembleur simplifié, cette fonction ressemble à

```
1 R13 <- LCL         (* Les registres de R13 15 servent de variables temporaires, ici on garde LCL comme point de rep *)
2 R14 <- M(R13-5)    (* R14 contient alors l'adresse de retour *)
3 M(ARG) <- M(SP-1) (* On place la valeur de retour a base du segment ARGUMENT (futur haut de la pile) *)
4 SP <- ARG-1        (* On rednit le haut de la pile *)
5 THAT <- M(R13-1)   (* Le pointeur THAT *)
6 THIS <- M(R13-2)   (* Le pointeur THIS *)
7 ARG <- M(R13-3)    (* Le segment ARGUMENT *)
8 LCL <- M(R13-4)    (* Le segment LOCAL *)
9 goto R14           (* Enfin, on saute 'adresse de retour, la fonction appelante reprend son cours *)
```

La traduction complète d'un return se fait en 48 instructions assembleur.

4.2.7 Recapitulatif de la structure d'une commande

L'ensemble des segments d'une fonction et les informations sur l'état de la fonction appelante est appelé cadre de la fonction. Pour récapituler, le cadre d'une fonction à n arguments et k variables locales est donc :

	Cadres des fonctions plus hautes dans la file d'appel	
ARG –>	argument 0	
	argument 1	
	...	Les arguments de la fonction courrante
	argument n-1	
	adresse de retour	
	LCL sauvegardé	
	ARG sauvegardé	Pointeurs de la fonction appelante
	THIS sauvegardé	
	THAT sauvegardé	
LCL –>	local 0	
	local 1	
	...	Variables locales de la fonction courrante
	local k-1	
SP –>		Pile de la fonction courrante

4.3 Ce qu'il reste à faire

La prochaine étape est une implémentation de la machine virtuelle pour les machines de l'école sous linux et windows, c'est-à-dire ne pas traduire les fonctions et rendre du code sous forme de texte en sortie, mais directement manipuler une représentation de la ram en ocaml (un simple mais grand tableau), compiler le code ocaml suffit à rendre executable le langage intermédiaire. Nous utiliserons SDL pour afficher l'écran et récupérer les information clavier, ainsi que ncurses pour afficher des informations sur le programme en cours.

Cette machine virtuelle est déjà bien avancée, une bonne représentation de la ram et de ses informations utiles est en place et les fonctions du palier 1 (`PUSH`, `POP` et les fonctions arithmétiques) sont déjà implémentées. Cette avance sur le cahier des charges nous permettra de bien nous concentrer sur SDL, ocaml étant très peu adapté à son utilisation.

5 Compilateur

5.1 Ce qui a été fait

Le langage `craft` a été défini.

On a écrit un compilateur pour ce langage vers un langage intermédiaire.

Ces deux éléments sont présentés dans les paragraphes qui suivent.

5.1.1 Avance prise sur la prochaine soutenance

Pour la prochaine soutenance on doit écrire une bibliothèque standard pour le language, ainsi que l'ensemble de fonctions qui composent le système d'exploitation. Cependant il s'avère que ces fonctions ont déjà été écrites pour cette soutenance, car elles étaient nécéssaires à l'écriture des tests.

5.2 Présentation du compilateur

5.2.1 Compilation du langage craft

Le compilateur `craft` traduit le language `craft` vers un langage intermédiaire qui sera ensuite traduit en langage assembleur, qui sera lui-même traduit vers un format binaire.

5.2.2 Compilation séparée

Pour faciliter le développement de façon séparée on permet la compilation de plusieurs fichiers séparés. Cela évite des temps de compilation trop longs.

Cette fonctionnalité est implémentée dans les directives `include` et `import` du langage.

5.2.3 Informations de *debug*

Pour faciliter la compréhension du code intermédiaire généré par le compilateur, on a implémenté un mode spécial qui affiche en plus du code généré des informations en commentaire.

- Exemple
  On compile la source suivante avec l'option `-t debug` :

```
1  class DebugMe {
2      field int a, b;
3      static bool lol;
4
5      constructor DebugNew new () {
6          let a = 42 + 2;
7          let b = 2 / Math.pow(2, 4);
8
9          let lol = a = b;
10     }
11 }
```

Le code **avec les informations de debug** produit alors :

```
function DebugMe.new 0
// this subroutine is a constructor, it must allocate memory for the object
// push the size th the object = the number of fields
push constant 2
call Memory.alloc 1
// pointing 'this' to the allocated space
pop pointer 0
// -- begining of statement #0
// let statement #0
// rvalue of let #0
// operation #0 : +
// left operand of op #0
// term: int const
push constant 42
// right operand of op #0
// term: int const
push constant 2
// operator of #0
add
// lvalue of let #0: name = a
pop this 0
// -- end of statement #0
// -- begining of statement #1
// let statement #1
// rvalue of let #1
// operation #1 : /
// left operand of op #1
// term: int const
push constant 2
// right operand of op #1
// calling #0
// term: int const
push constant 2
// term: int const
push constant 4
call Math.pow 2
// operator of #1
call Math.divide 2
// lvalue of let #1: name = b
pop this 1
// -- end of statement #1
// -- begining of statement #2
// let statement #2
// rvalue of let #2
// operation #2 : =
// left operand of op #2
// term: var, name = a
push this 0
// right operand of op #2
// term: var, name = b
push this 1
// operator of #2
eq
// lvalue of let #2: name = lol
pop static 0
// -- end of statement #2
// end of subroutine
return
```

5.2.4 Les avertissements

Le compilateur effectue de nombreuses vérifications sur le code écrit par le programmeur. Il peut ainsi détecter de nombreuses erreurs de programmation.

Par défaut elle ne sont pas , leur affichage est activé par l'option « -w » du compilateur.

- Exemple
  Le compilateur détecte la double déclaration de classe, ainsi si on compile le code suivant avec les avertissements :

```
1  class Main {}
2
3  class Main {}
```

On obtient sur la sortie standard d'erreur le message :

```
1  Warning: class Main is defined at least twice.
```

Il existe bien d'autres warnings (certains sont cachés, à vous de les trouver !)[3]

5.3 Fonctionnement du compilateur

5.3.1 La table des symboles

La table des symboles est une structure de données qui a pour but la gestion du nom, du type et de la portée des variables, fonctions et classes. Elle est implémentée par le type OCaml suivant :

```
1  module H = Hashtbl
2
3  type name = string
4
5  type ftype =
6    | Constructor
7    | Procedure (* aka. function *)
8    | Method
9
10 type locals = int
11
12 type stype =
13   | Int
14   | Char
15   | String
16   | Boolean
17   | Void
18   | UserType of string
19   | Function of (ftype * rettype * locals * stype list)
20 and rettype = stype
21
22 type kind =
23   | Static
```

[3] astuce : coder comme un porc

```
  | Field
  | Arg
  | Var
  | Fun

type index = int

type symbol = (stype * kind * index)

type symtable = (name, symbol) H.t

type module_symtable = (name, symtable) H.t

type symtable_data =
  {
    mutable class_name: string option;

    class_scope: symtable;
    subroutine_scope: symtable;
    module_scope: module_symtable;

    mutable static_index: int;
    mutable field_index: int;
    mutable arg_index: int;
    mutable var_index: int;
  }
```

On a également codé (entre autre) les fonctions suivantes pour la manipuler :

```
(* crla table des symboles *)
val create : unit -> symtable_data
(* demande l'effacement des variables locales *)
val new_subroutine : symtable_data -> unit
(* retourne le nom de la classe courante *)
val get_class : symtable_data -> string
(* dnit le nom de la classe courante *)
val new_class : symtable_data -> string -> unit
(* dnit un nouveau nom dans la table de symboles *)
val define : symtable_data -> name -> stype -> kind -> unit
(* retourne un symbole artir de son nom *)
val get_sym : symtable_data -> name -> symbol
(* retourne la table des symbole appartement n module *)
val get_module : symtable_data -> name -> symtable
(* retourne le genre d'une variable *)
val kind_of : symtable_data -> name -> kind
(* retourne le type d'une variable *)
val type_of : symtable_data -> name -> stype
(* retourne le type d'une sous-routine *)
val ftype_of : symtable_data -> name -> ftype
(* retourne le num d'un nom (utile pour grer des labels dans le code gr*)
val index_of : symtable_data -> name -> index
(* retourne le nombre d'argument que prend une sous-routine *)
val arity_of : symtable_data -> name -> int
(* retourne le nombre de variables locales a sous-routine courante *)
val current_locals : symtable_data -> int
(* retourne le nombre de champs que contient un objet de la classe courante *)
val size_of_current_class : symtable_data -> int
```

C'est la partie la plus importante du compilateur, elle permet de s'assurer du type des variables et de leur existence dans la portée courante.

5.3.2 Générateur de code intermédiaire

Le générateur de code intermédiaire permet de découpler la partie compilation de la écriture du code sous-jacent. Ainsi le compilateur ne manipule que des types haut niveau, tandis que le générateur écrit proprement le texte sur la sortie.

Son interface est la suivante :

```
type symbol = string
type argc = int
(* les diffnts segments dans le langage intermire *)
type segment =
    Static
  | Argument
  | Local
  | This
  | That
  | Pointer
  | Temp
  | Constant

(* les diffntes instructions du langage intermaire  *)
type op =
    Add
  | Sub
  | Neg
  | Eq
  | Gt
  | Lt
  | And
  | Or
  | Not
  | Push of (segment * int)
  | Pop of (segment * int)
  | Label of symbol
  | Goto of symbol
  | IfGoto of symbol
  | Function of (symbol * argc)
  | Call of (symbol * argc)
  | Return

(* retourne le nom d'un segment *)
val string_of_segment : segment -> string
(* retourne un commentaire dans le langage intermaire *)
val comment : string -> string
(* retourne le code intermaire pour l'option 'op' *)
val write : op -> string
```

5.3.3 Le cœur du compilateur

Le compilateur traduit récursivement l'ensemble du programme en parcourant récursivement l'arbre de syntaxe abstrait (construit lors de la soutenance 1).

Un sous-ensemble des fonctions utilisée est présenté ci-dessous :

```
val compile_term : A.term -> unit
val compile_expression : A.arraypos -> unit
val compile_call : A.subroutinecall -> unit
val compile_let : A.statement -> unit
```

```
5  val compile_letarray : A.statement -> unit
6  val compile_if : A.statement -> unit
7  val compile_while : A.statement -> unit
8  val compile_do : A.statement -> unit
9  val compile_return : A.statement -> unit
10 val compile_statements : A.statements -> unit
11 val compile_subroutine_body : A.statements -> unit
12 val compile_subroutine_locals : A.vardec list -> unit
13 val compile_subroutine_args : A.parameter list -> S.stype list
14 val compile_subroutines : A.subroutinedec list -> unit
15 val compile_sub_declaration : A.subroutinedec list -> unit
16 val compile_class_vars : A.classvardec list -> unit
17 val compile_class : A.top -> unit
18 val compile_module : A.top -> unit
19 val compile_include : A.top -> unit
20 val compile_top : A.top -> unit
21 val compile : A.top list -> (string, out_channel) Output.H.t -> unit
```

La compilation commence lors de l'appel de la fonction `compile` qui prend une forêt d'éléments de premier niveau (`A.top list`, `A` étant le module qui gère l'`AST` [4]et écrit sur le fichier de sorti *via* le module Output.

La fonction `compile` crée une table des symboles vide qui est ensuite remplie par les fonctions `compile_class_vars`, `compile_sub_declaration` et `compile_subroutine_locals`.

5.3.4 Gestion des entrées sorties

Le module `Output` gère l'écriture dans les fichiers. Il permet la définition de canaux de haut niveau dans lesquels les autres modules peuvent écrire. Il lie ensuite ces canaux à des fichiers pour y écrire à proprement parlé les données.

Ainsi dans le module compile on trouve :

```
1 out "debug" "// la fonction main en langage intermaire"
2 out "out" "function Main.main"
```

Où `out` est la fonction du module Output, le premier argument le canal dans lequel écrire et en troisième argument la chaîne à écrire.

Dans le mode de compilation par défaut le canal « debug » n'est relié à aucun fichier, donc les informations de *debug* sont simplement perdue. Lorsque l'on active le mode *debug*, le canal « debug » est relié au fichier qui est utilisé pour écrire le code. On retrouve ainsi dans le même fichier les informations de *debug* et le code généré.

Ce système est utilisé partout dans le compilateur, il permet le découplage des fichiers et des fonctions d'écriture et il facilite leur gestion.

5.4 Présentation du langage `craft`

Quoi de mieux pour présenter un langage qu'un `hello world` ?

```
1 include OS.all
2
```

[4] *Abstract Syntax Tree*, arbre de syntaxe abstraite

```
class Main {
    function void main () {
        do Output.println("hello, world!");
    }
}
```

Présentons maintenant les différents aspects du langage, ils sont tous supportés par le compilateur.

5.4.1 Les directives

On peut inclure un autre fichier `craft` via la directive `import`. Ce fichier sera compilé comme si son contenu était présent dans le premier fichier. On peut utiliser une notation pointée pour sélectionner des fichiers dans d'autres répertoires.

Exemple, pour charger la classe Map, située dans le dossier Game :

```
import Game.Map
```

On peut inclure un fichier déjà compilé (pour des raisons d'optimisation manuelle par exemple) en langage intermédiaire avec la directive `include`.

Exemple, pour charger toutes les fonctionnalités de l'OS :

```
include OS.all
```

5.4.2 Les classes

Chaque fichier `craft` contient au moins une classe.

Une classe a l'allure suivante :

```
class name {
    Field and static variable declarations

    Subroutines declarations
}
```

Chaque classe spécifie un nom par lequel la classe peut être désignée de manière globale.

Puis on déclare un ensemble (possiblement vide) de `field` et de `static variables` (elles sont expliquées plus tard dans la section *Types de variables*).

On déclare ensuite des sous-routines, chacune définissant une *méthode* (`method`), une *fonction* (`function`) ou un *constructeur* (`constructor`). Le *méthodes* « appartiennent » aux objets instanciés, tandis que les fonctions « appartiennent » aux classes et n'ont pas besoin d'être appelées sur un objet. Un *constructeur* « appartient » à une classe, lorsqu'il est appelé alloue l'espace nécessaire au stockage d'un objet sur le tas[5] et renvoie une instance de la classe.

Toutes les sous-routines sont définies de la même façon :

[5] *heap* en anglais

```
subroutine_type return_type name (parameters) {
   local variables declaration

   statements
}
```

Où `subroutine_type` est `constructor`, `method` ou `function`. Chaque sous-routine a un nom, par lequel elle peut être appelée et un `type` qui décrit le type de la valeur retournée. Si la sous-routine ne retourne rien elle doit être déclarée comme retournant `void`, sinon tout les types primitifs peuvent être retourné, de même que les toutes les autres déjà classes définies (une déclaration de classe étant une déclaration de type). Les constructeurs peuvent avoir un nom arbitraire (on leur préfèrera quand même le nom `new`) et doivent retourner le type de la classe dans laquelle ils sont définis.

Chaque sous-routine peut définir un ensemble de variables locales, puis une suite d'expressions.

Le point d'entrée d'un programme `craft` est la fonction `main` de la classe `Main`.

5.4.3 Les variables

Les variables en `craft` doivent être explicitement déclarées et typées avant d'être utilisées. Il existe quatre genre de variables : `field`, `static`, `local` et `parameter`.

Les types Chaque variable a un type qui peut être soit scalaire (c'est à dire ne contenant qu'une seule valeur) soit objet (le nom d'une classe).

Les types scalaires Le langage `craft` possède trois types scalaires :

- `int` : un entier signé sur 16 bits complément à 2 ;
- `boolean` : `false` ou `true` ;
- `char` : un caractère sur 16 bits[6].

Les variables scalaires sont allouées en mémoire lorsqu'elles sont déclarées.

Les types objets Chaque classe définit un type. La déclaration d'un type objet cause l'allocation d'un pointeur qui stockera l'adresse de l'objet sur le tas, lorsque le programmeur décidera de construire l'objet en appelant son constructeur.

La bibliothèque standard de `craft` contient deux classe très importante pour tout programme :

- `Array` : pour l'allocation et la gestion des tableaux ;
- `String` : pour la gestion des chaînes de caractères.

[6] on se restreint dans les fonctions de l'OS aux caractères `ASCII`

Les tableaux Les tableaux sont déclarés avec la classe `Array`. Les tableaux sont à une dimension et indicés à 0. Les tableaux mutli-dimensionnels peuvent être obtenus à l'aide de tableaux de tableaux. Les éléments du tableaux ne sont pas typés, mais doivent tous avoir la même taille. La déclaration d'un tableau ne crée qu'une référence, la construction du tableau étant effectuée lors de l'appelle du constructeur : `Array.new(taille)`. On peut accéder aux éléments du tableau avec la notation `a[j]`.

Les chaînes de caractères Les chaînes de caractères sont déclarées avec la classe `String`. Le compilateur reconnaît la syntaxe

```
"hello, world!"
```

pour définir une chaîne de manière littérale, qui sera automatiquement traduite en un objet `String`.

Conversions de type Le langage `craft` est typé faiblement, et les informations de type ne sont vérifiées qu'aux endroits critiques du code. On peut ainsi changer le type d'une variable sans avertissement du compilateur, tant que la taille des objets correspond.

- Caractères simples et chaînes
  Les caractères et les entiers sont convertis implicitement d'un type vers l'autre. Par exemple :

```
1  var char c;
2  var String s;
3  let c = 65; (* 'A' *)
4
5  (* Pareillement *)
6  let s = "A";
7  let c = String.charAt(0);
```

- Tableaux et indirection
  Un entier peut être assigné à une variable (peu importe le type) pour être utilisé comme une adresse mémoire. Exemple :

```
1  var Array a;
2  let a = 5000;
3  let a[100] = 77; (* L'adresse mire 5100 vaut maintenant 77 *)
```

- Tableaux et objets
  Un objet peut être converti en un objet de type `Array` (et *vice versa*).

  Une telle conversion permet d'accéder aux champs de l'objet comme à une case d'un tableau (et *vice versa*). Exemple :

```
1  (* Nous avons une classe Complex ayant deux champs : re et im *)
2  var Complex c;
3  var Array a;
```

```
4  let a = Array.new (2);
5  let a[0] = 7;
6  let a[1] = 8
7  let c = a; (* c == Complex (7, 8) *)
```

Portée et genre des variables Il existe quatre genres de variables dans un programme écrit en `craft`.

Les *variables statiques* sont définies au niveau de la classe et accessibles partout dans les sous-routines de celle-ci. Elle sont partagées par tous les objets qui instancient cette classe.

Les *champs* (ou *attributs*) sont utilisés pour définir les propriétés des objets. C'est eux qui définissent la taille d'un objet en mémoire.

Les *variables locales*, utilisée dans les sous-routines n'existent en mémoire que lors de l'exécution de la sous-routine.

Les *paramètres* sont utilisé en tant qu'arguments pour les sous-routines. Ils sont passés par référence.

5.4.4 Les instructions

L'assignation L'opération d'assignation se fait à travers la construction `let`.

```
1  let variable = expression;
2  (* ou *)
3  let variable [expression] = expression;
```

`variable` est sois une valeur scalaire où un tableau.

L'exécution d'une routine L'instruction `do` est utilisée pour appeler une fonction ou une méthode en ignorant la valeur de retour.

```
1  do function-or-method-call;
```

La fin de sous-routine L'instruction `return` permet de sortir d'une sous-routine (où d'une structure de contrôle de flux), renvoyant éventuellement une valeur.

```
1  return;
2  (* ou *)
3  return expression;
```

5.4.5 Les structures de contrôle du flux d'exécution

Le branchement conditionnel Un `if` classique avec un clause `else` optionnelle. Les accolades sont obligatoires même si le `if` ne contient qu'une seule instruction. `expression` doit pouvoir être assimilée à une valeur booléenne.

```
if (expression) {
    statements
}
else {
    statements
}
```

La répétition Une boucle `while` classique. Les accolades sont obligatoires même si le `while` ne contient qu'une seule instruction. `expression` doit pouvoir être assimilée à une valeur booléenne.

```
while (expression) {
    statements
}
```

5.4.6 Les expressions

Une expression peut être :

- une constante (`42`, `"foobarlol"`, `true`, `false`, `null`) ;
- un nom de variable à portée ;
- le mot clef `this`, désignant l'objet courant ;
- une case d'un tableau, accédé par la syntaxe `nom[expression]`, où nom est une variable de type `Array` à portée ;
- un appel à une sous-routine n'étant pas de type `void` ;
- une expression préfixée par un opérateur unaire :
  - `-` `expression` : la négation arithmétique ;
  - `~` `expression` : la négation logique ;
- une expression de la forme `expression opérateur expression` où `opérateur` peut être :
  - `+ - * /` opérateurs arithmétiques sur des entiers ;
  - `& |` opérateurs booléen ET et OU (pouvant agir sur les entiers pour les même opérations) ;
  - `< > =` opérateurs de comparaison ;
- `(expression)` une expression parenthésée.

La priorité des opérateurs est celle communément admise.

Appel des sous-routines La syntaxe générale pour appeler une sous-routine est la suivante :

```
1 subroutineName(argumentList)
```

Le nombre et les types des arguments doit correspondre à ceux donnés dans la déclaration de la sous-routine. Les parenthèses doivent doivent être présentes même si la sous-routine ne prend pas d'argument. Chaque argument peut être d'une complexité infinie.

À l'intérieur d'une classe, les méthodes peuvent être appelée en utilisant la syntaxe :

```
1 methodName(argumentList)
```

tandis que les fonctions et les constructeurs doivent être appelés en utilisant leur forme générale :

```
1 className.subroutineName(argumentList)
```

En dehors d'une classe, les fonctions et le constructeurs doivent être appelé avec leur nom complet tandis que les méthodes sont appelées par la syntaxe :

```
1 varName.methodName(argumentList)
```

où `varName` est un objet déjà définis.

5.4.7 Construction et destruction des objets

On construit un objet en deux étapes. Premièrement on doit déclarer une référence vers cet objet (un pointeur), et uniquement la mémoire celle-ci est allouée. Deuxièmement on construit l'objet en appelant le constructeur de sa classe. Ainsi une classe qui implémente un type doit forcement avoir un constructeur. Les constructeurs peuvent avoir un nom arbitraire mais on préfèrera souvent utiliser `new`. Les constructeurs sont appelés comme toute autre sous-routine :

```
1 let varName = className.constructorName(argument-list)
```

Par exemple :

```
1 let c = Circle.new(x, y, 42)
```

où `x` et `y` sont la position du cercle sur l'écran et `42` son rayon. Quand le constructeur est appelé, le compilateur va effectuer un appel à la routine d'allocation mémoire du système d'exploitation[7] pour allouer suffisamment de place en mémoire pour contenir l'objet. Puis

[7] *Operating System* ou OS

l'OS retourne l'adresse de base de la zone mémoire allouée, qu'il assigne à la variable `this`. Dans l'exemple du cercle, `this` est assigné à `c`. Puis le constructeur est appelé pour initialiser l'objet et le mettre dans un état initial valide.

Quand un objet n'est plus nécessaire dans le programme on doit d'en débarrasser. En particulier la mémoire qui avait été allouée doit être libérée et rendue disponible à l'usage pour de futur allocations. Cette opération est effectuée par la fonction `Memory.deAlloc(object)` fournie par l'OS.

5.5 Ce qu'il reste à faire

Le compilateur est aujourd'hui complètement fonctionnel on peut ainsi s'atteler aux tâches suivantes :

- continuer d'écrire la bibliothèque standard ;
- améliorer l'algorithme d'allocation mémoire (pour l'instant c'est un `first fist`) ;
- compiler vers d'autres langages intermédiaires (MSIL, bytecode python ou même llvm) ;
- coder d'autres programmes en `craft` !

6 Annexes

6.1 Web cloud green it technologies agiles et peer programming

Le site web[8]a été mis-à-jour, il contient désormais les articles relatifs à cette soutenanace.

6.2 « Bitbucket c'est pas stable »

Nous utilisons bitbucket pour stocker notre dépot mercurial[9]. Celui-ci a été deux fois corrompus sur leurs serveurs[10] :

- https://bitbucket.org/site/master/issue/3401/cant-clone-a-repo-abort-http-error-500
- https://bitbucket.org/site/master/issue/3695/abort-http-error-500-internal-server-error

6.3 Le chan IRC, repaire des gens d'une autre époque™

Nous tenons à remercier les habitués du cannal `#nandcraft` sur `irc.rezosup.org` (sans ordre de préférence) :

- aurag
- Chewie
- delroth
- Flash11
- Horgix
- kalenz
- kushou
- Magicking
- Mareo
- Nebryum
- PM
- Staphylo
- Thiel
- thrashboul

[8]http://nandcraft.halfr.net

[9]http://code.halfr.net/nandcraft/

[10]le dernier datant de la veille de cette soutenance (comprendre « en plein rush »), nous avons du mettre en place un dépot local à la SM dans laquelle nous travaillions pour pouvoir continuer à coder…

- underflow
- yoopo
- yroeht

C'est grâce à eux que le canal a reçu depuis sa création plus de 10000 messages !

7 Le mot de la fin

```
tl;dr On est toujours très très heureux.
```