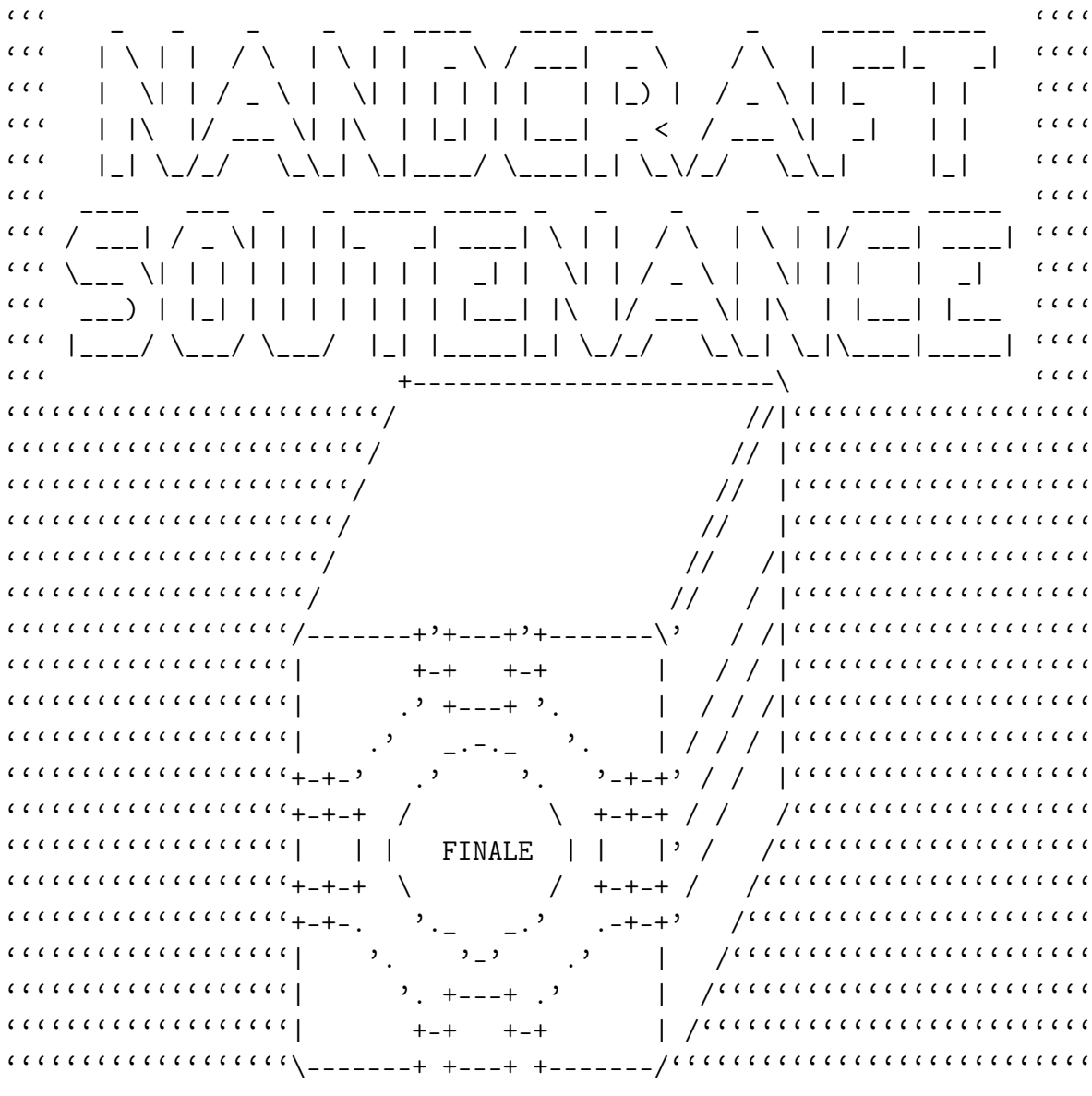
CPU - Cube Processing Unit

audebe_r - halfr

hervot_p - Dettorer

eddequ_n - nass

pruvot_a - Chaf



Sommaire

1 Introduction

1.1 NANDCRAFT

What I hear, I forget; What I see, I remember; What I do, I understand.

– Confusius, 551-479 av. J.-C.

Il y a fort longtemps, tous les ingénieurs en informatique comprenaient parfaitement le fonctionnement des ordinateurs. L'ensemble des intéractions entre la machine, les logiciels, les compilateurs et le système d'exploitation étaient simples et suffisament transparents pour pouvoir comprendre l'ensemble des opérations de l'ordinateur. Mais la technologie a évoluée, elle est devenue de plus en plus complexe et cette simplicité a été perdue. Les fondements de l'informatique, l'essence de la discipline, sont maintenant cachés par des interfaces obscures et des implémentations propriétaires.

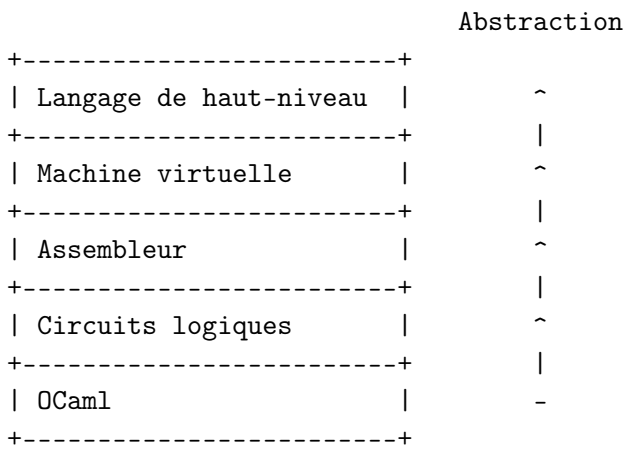
C'est pourquoi nous avons construit un ordinateur en partant d'un élément atomique de logique : la porte `NAND`. Avec elle nous avons assemblé une machine capable d'exécuter du code stocké dans une mémoire interne (elle même créée à l'aide de la porte `NAND`). Puis nous avons créé un language qui a facilité l'écriture du code machine. D'abstractions en abstractions, on a créé des langages toujours plus simples à utiliser. Chaque langage se basant sur le précédent.

Dans le cadre de notre projet chaque membre du groupe a travaille sur une partie différente de la machine. Chacunes sont directement dépendantes de la partie directement inférieure à celle-ci. Ce fut l'occasion de s'entraider et de profiter réellement de la dimension humaine du projet. La diversité dans les projets a impliqué une importante source de découverte et de partage entre les membres du groupe. En effet, malgré l'indépendance des projets, nous avons toujours eu besoin de nous assurer du fonctionnement global de la machine.

Nous avons choisi d'utiliser le langage OCaml pour la réalisation de `NANDCRAFT` car des nazis nous l'avaient ordonné dans un songe. En effet, nous avons principalement manipulé des structures de données sous forme de graphes ainsi que des langages, ce pourquoi OCaml est très adapté.

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 :



1.2 Présentation du groupe

1.2.1 Amaury *Chaf*

Ma passion pour l'informatique est apparue dès mon plus jeune âge en parcourant de nombreux jeux vidéo. Contrairement à beaucoup de personnes à l'Epita, je ne suis pas venu pour créer des jeux vidéo plus tard, j'ai toujours voulu explorer la sécurité réseau. C'est pourquoi cette année j'ai rejoins un groupe ayant un projet atypique mais néanmoins très attrayant : quoi de mieux que de fabriquer un ordinateur de A à Z pour comprendre son fonctionnement et pouvoir mieux le protéger ?

1.2.2 Nassim *nass*

NANDCRAFT m'a sauvé ! Sans ce projet, je serai très certainement entrain de dealer de la drogue, en prison ou bien mort dans une rixe de gangs. Je lui dois la vie à tout jamais...

Nassim Eddequiouaq, né le 26 mai 1990 en plein Paris. Je me suis découvert une grande passion pour les systèmes informatiques cette année. Le travail effectué sur ce projet m'a amené à chercher et trouver des informations extrêmement intéressantes sur la manière dont un ordinateur fonctionne. Ce projet m'a également apporté énormément en terme de reflexion sur le code et l'impact d'optimisations sur les performances d'un logiciel. Je pense que celà m'a convaincu de continuer les projets dans ce domaine afin d'y poursuivre une carrière professionelle.

Je me considère désormais comme membre du projet NANDCRAFT avant d'être un citoyen européen.

1.2.3 Paul *Dettorer*

Je suis arrivé à EPITA sans avoir la prétention d'être un passioné, mais avec la certitude que beaucoup de domaines de l'informatique allaient m'intéresser.

J'avais rencontré Rémi pendant les vacances d'été, je passe mes soirées en salle machine avec Nassim depuis le séminaire et j'ai appris à connaitre Amaury lors de nos nombreuses

soirée à coder. Je savais que nous partagions beaucoup d'intérets et que nous prendrions beaucoup de plaisir à travailler sur ce projet.

Ce que j'attendais de ce projet s'est entièrement réalisé Je souhaitais participer à mon premier « grand » projet et surtout beaucoup apprendre. Je pense aujourd'hui avoir acquis les bases en informatiques que je comptais acquérir cette année et peux maintenant m'orienter vers des projets plus ambitieux.

tl;dr : J'aime les chats

1.2.4 Rémi *halfr*

Je suis venu à EPITA car je pense que pour prétendre être un ingénieur en informatique compétent il est nécessaire de s'investir pleinement, sans compter les heures, ni les bols de café.

J'ai beaucoup trop joué aux jeux vidéos pour pouvoir en coder un, je pense que c'est une source infinie de frustration et d'échecs. C'est pourquoi j'ai proposé à mon groupe la création d'un ordinateur en OCaml. Pour leur prouver la faisabilité de ce projet je leur ai montré un bout de code que j'avais écrit il y a quelques années. Il était de conception simple et arrivait à simuler un CPU en se basant uniquement sur la porte logique `NAND`.

J'ai rencontré Dettorer avant même d'être entré à EPITA. Nous nous sommes instantanément bien entendus et c'est avec plaisir que je lui ai proposé de rejoindre le projet. De même pour Nass, avec qui nous parlions d'idées de projet dès Août. Quant à Chaf, nous sommes dans la même classe, nous avons sympathisé et ainsi fut formé notre groupe de projet.

Ils sont tous géniaux, je les aime.

> *tl;dr Un jeune connard de geek. Ne l'invitez jamais en soirée. C'est un troll, c'est comme ça. Personne ne l'aime et nous pensons qu'il a insulté ta mère.*
>
> – Fichier des Renseignement Généraux d'une planète lointaine

2 Logique par *Amaury*

2.1 Description et premiers pas

Le but du projet NANDCRAFT étant de construire un ordinateur de A à Z, nous avons besoin de définir un certains nombre de portes logiques. En effet, les portes logiques sont les éléments de base de l'ensemble des appareils numériques d'aujourd'hui.

Bien que ces portes puissent prendre plusieurs formes physiques, leur comportement est le même dans tout les systèmes, ainsi nous pouvons construire et simuler toutes les portes nécessaires. Cette partie du projet est celle de plus bas-niveau d'abstraction puisqu'elle traite directement de l'architecture de l'ordinateur. De plus, c'est donc dans cette partie que le comportement de la machine sera imité de façon à créer une architecture virtuelle.

Pour réaliser ce projet nous nous appuyons sur « The Element of Computing Systems » qui décris sucintement les étapes à suivre afin de simuler notre ordinateur. Cependant, bien qu'il fournsse des renseignements très important, il n'en reste pas moins évasif sur la théorie en général.

C'est pourquoi la première étape de cette partie consacrée à la logique a été de rassembler des informations pour mieux cerner le sujet. Cette recherche d'informations peut paraître rébarbative mais elle reste cependant nécessaire pour éviter de perdre un temps considérable en s'embrouillant l'esprit avec des équations logique compliquées qui pourraient être simplifiées facilement à l'aide de quelques formules de logique élementaire. C'est donc uniquement après cette longue phase de recherche que nous pouvons commencer à définir nos portes logiques.

2.2 Logique combinatoire

2.2.1 Implémentation des portes simples

Dans ce projet, il est impératif de définir nos différentes portes commes des assemblages de portes NAND. Pour ceci, il a fallu déterminer les équations logiques de chacunes des portes en fonction de la porte NAND. Pour vérifer les équations que nous avons trouver il nous a simplement fallu tester les différentes équations obtenues en établissant leur table de vérité et en les comparant aux tables de vérité classiques des portes.

Une fois les équations établies et vérifiées, la vraie implémentation des portes pouvait commencer. Nous avons besoin de représenter nos portes de façon symbolique pour rester au plus proche des puces qui composent un ordinateur. En d'autres termes, chacune des portes quue nous allons construire doit comprendre dans sa définition les différents liens entre les entrées et les sorties des portes intermédiaires qui la compose. pour se faire nous avons utilisé les types enregistrement afin de stocker toutes les informations nécessaires (nom de la porte, entrées, sorties et liens intermédiaires)

L'exmeple qui suit permettra de mieux comprendre notre démarche :

```
let my_or =
  {
    name = "or";
    inputs = [| "a" |> 1 ; "b" |> 1 |];
    outputs = [| "out" |> 1 |];
    parts = [|(my_nandio [|"a" <@ "a"; "b" <@ "a"|] [|"out" @> "out1"|]);
               (my_nandio [|"a" <@ "b"; "b" <@ "b"|] [|"out" @> "out2"|]);
               (my_nandio [|"a" <@ "out1"; "b" <@ "out2"|] [|"out" @> "out"|]) |]
  };;
```

Pour comprendre cet exemple, il est nécessaire d'expliquer le fonctionnement des différents opérateurs et les types utilisés :

- L'opérateur « |> » permet de lier l'entrée « a » à son nombre de bits : ainsi « a |> 1 » signifie que l'entrée « a » est un entier d'un seul bit, tandis que « a |> 16 » sera un entier constitué de 16 bits. Il renvoie un couple (string*int).
- Les opérateurs « <@ » et « @> » permettent eux de lier la nouvelle entrée à une précédente/suivante, ainsi « a <@ a » permet de donner au premier « a » la valeur du « a » entré en paramètre et « out @> out1 » donne à « out1 » la valeur de sortie de la porte utilisée. Il renvoie un couple (string*string).
- « my_or » est une valeur du type « gate » défini comme ceci :

```
type gate =
  {
    name:string;
    inputs:string*int array;
    outputs:string*int array;
    parts:part array
  } ;;
```

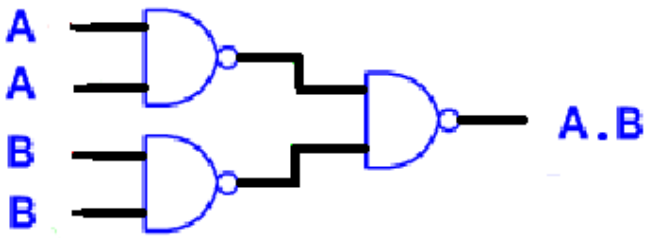
- Le type part est lui défini par :

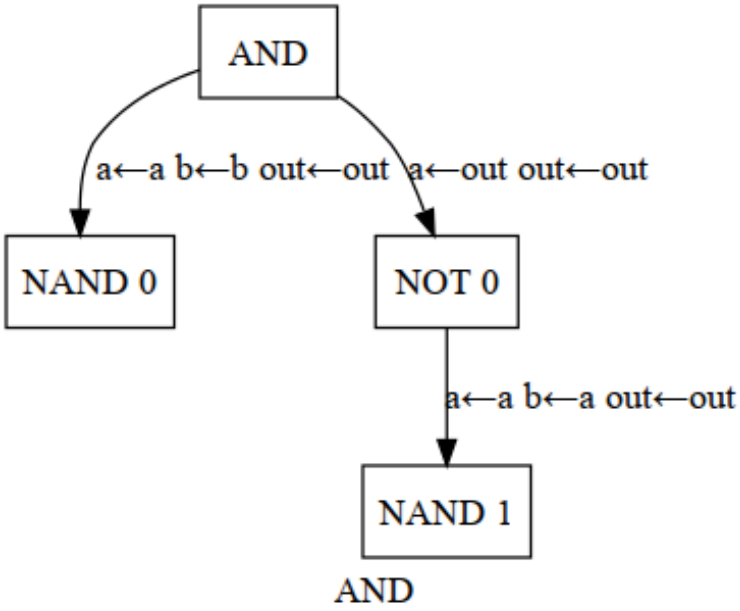
```
type part =
  {
     gate:gate;
     inputs_bind:string*string array;
     outputs_bind:string*string array
  };;
```

La fonction my_nandio permet de lier les entrées et les sorties à la porte comme ceci :

```
let my_nandio inputs outputs =
  {
    gate = my_nand;
    inputs_bind = inputs;
    outputs_bind = outputs
  };;
```

Revenons maintenant à la porte « or », comme nous l'avons implémentée nous avons maintenant accès à son nom, ses entrées et sorties et surtout à la façon dont elle est créée (son squelette en quelques sortes). Nous voyons ainsi qu'elle est composée de trois portes « nand » de la façon suivante :



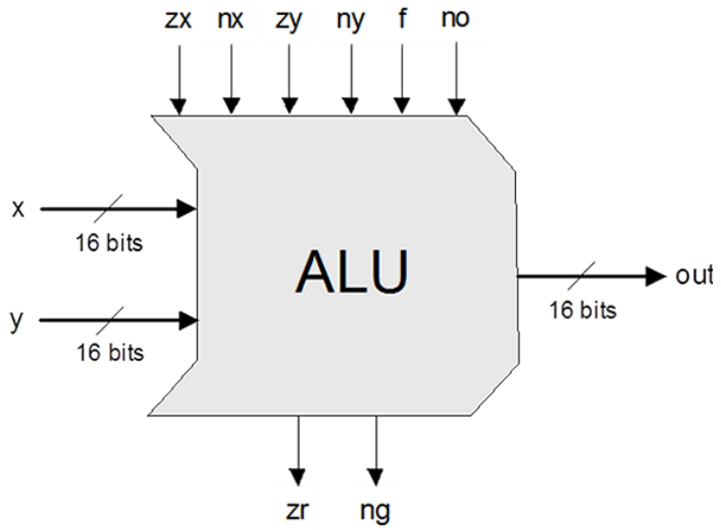


Nous avons ainsi défini toutes les portes combinatoires usuelles de cette manière.

2.2.2 Unité Arithmétique et Logique

- 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.

- 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

2.3 Logique séquentielle

Un ordinateur doit être capable non seulement de calculer des valeurs grâce aux portes combinatoires que nous avons définies précedement mais il doit aussi pouvoir stocker des données pour les réutiliser ultérieurement. C'est pourquoi nous avons aussi besoin de portes séquentielles dont le sorties dépendent à la fois des combinaisons de variables d'entrées mais aussi des sorties précédentes. Nous avons donc aussi défini un certains nobmres de portes séquentielles (horloges, bascules, registres, ...) afin de créer des mémoires pour notre ordinateur.

Malheureusement, avec notre mode de simulation de porte, nous ne pouvons que créer des registres bits par bits composés chacuns de quatre portes NAND. Ainsi, il nous est simplement impossible de créer de « grosses » mémoires puisqu'elles nécessiteraient une bien trop grande quantité de mémoire de la part de nos machines personnelles pour être simulées. C'est pourquoi, dans le cadre de notre projet, nous nous sommes arrêtés à des mémoires RAM 16K.

2.4 Des outils supplémentaires

Au fur et à mesure que nous avons avancé dans l'établissement de notre éventail de portes logiques nous avons eu besoins de quelques outils supplémentaires pour simplifier notre tâche.

2.4.1 Génération de graphes et tables de vérité

Pour vérifier les équations que les équations de nos portes étaient correctes nous avons écris un petit supplément dans l'implémentaion des portes qui nous permet de dresser la table de

vérité de la porte automatiquement et de l'afficher à l'écran. Grâce à ce système nous avons gagné un temps considérable puisque jusque là nous devions écrire les tables de vérité à la main pour chacune des portes puis seulement les comparer aux tables de vérité usuelles.

Pour construire les portes les plus complexes (l'UAL par exemple…) nous avons rapidement eu besoin de pouvoir vérifier nos équations de façon automatique. Ainsi, nous avons modifié le code d'implémentation de nos portes pour qu'en plus de créer la porte désirée, il trace aussi un graphe des liens en les différentes portes dont elle est composée. Nous avons alors pu détecter les redondances plus facilement et réduire considérablement le nombre de porte NAND utilisée pour simuler les portes complexes. De cette manière nous avons réduit le nombre de portes NAND utilisée pour définr notre UAL pour le faire passer d'un peu plus de 1500 portes NAND à « seulement » 647.

2.4.2 VHDL

Pour se rapprocher de ce qui est utilisé ailleurs en Europe et donc de ce qui pourra nous servir en regardant après la Sup nous avons décidé d'utiliser un intermédiaire entre la description de la porte en CAML et son utilisation réelle. Pour celà nous avons choisi d'utiliser le langage VHDL. Ainsi notre implémentation de porte en CAML pourra génèrer un fichier VHDL qui pourra être utilisé plus simplement par la suite.

VHDL est un langage de description de matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. Son nom complet est VHSIC Hardware Description Language.

Le but d'un langage de description matériel tel que le VHDL est de faciliter le développement d'un circuit numérique en fournissant une méthode rigoureuse de description du fonctionnement et de l'architecture du circuit désirée. L'idée est de ne pas avoir à réaliser un composant réel, en utilisant à la place des outils de développement permettant de vérifier le fonctionnement attendu. Ce langage permet en effet d'utiliser des simulateurs, dont le rôle est de tester le fonctionnement décrit par le concepteur.

En VHDL, tout composant (dans le sens logiciel) est décrit sous deux aspects :

- L'interface avec le monde extérieur, décrite dans une section dénommée entity.
- L'implémentation elle-même, décrite dans une section dénommée architecture.

C'est donc la section architecture qui contient la description de la fonction matérielle désirée :

- soit sous forme de description structurelle précise de l'architecture matérielle (les portes logiques à utiliser et leurs interconnexions), ici c'est cette caractéristiques qui est cruiciale dans notre projet, qui je le rappelle, consiste à construire les différentes portes logiques existantes à l'aide de la seule porte NAND.
- soit sous forme de comportement attendu, c'est-à-dire orienté fonctionnel.

Voici le code CAML que nous avons déjà et le code VHDL qui devra être généré:

- Le code CAML

```
let my_and =
{
  name = "and";
  inputs = [| "a" |> 1 ; "b" |> 1 |];
  outputs = [| "o" |> 1 |];
  parts = [| (my_nandio [|"a" <@ "a"; "b" <@ "b"|] [|"o" @> "o1"|]);
             (my_notio [|"a" <@ "o1"|] [|"o" @> "o"|]) |]
};;
```

- Le code VHDL

```
entity My_Nand is
  port (
    a, b : in  std_logic;
    o  : out std_logic);
end My_Nand;

entity My_Not is
  port (
    a : in  std_logic;
    o  : out std_logic);
end My_Not;

-- Porte AND

entity My_And is
  port (
    a, b : in  std_logic;
    o  : out std_logic);
end My_And;

architecture Logique of My_And is

  component My_Nand
    port (
      a, b : in  std_logic;
      o  : out std_logic);
  end component;

  component My_Not
    port (
      a, b : in  std_logic;
      o    : out std_logic);
  end component;

  signal o1 : std_logic := '0';

begin
  n1 : My_Nand port map (
    a => a,
    b => b,
    o => o1);
  n2 : My_Not port map (
    a => o1,
    o => o);
end Logique;
```

Nous voyons ici que la porte AND est créée grâce à l'association des entrées et sorties de portes NOT et NAND dont on rapelle la définition à chaque fois que l'on en a besoin comme montre l'exemple ci-dessus. De plus nous voyons bien les relations entre le CAML et le VHDL, les liens entres les portes qui servent à définir le AND sont les mêmes grâce au port map ainsi que les entrées et les sorties.

3 Assembleur par *Nass*

Le language que le programmeur utilise est bien différent de celui lu par la machine. En effet, les différentes étapes de traduction amènent à un language binaire représenté symboliquement par des « 0 » et des « 1 » lorsqu'on le veut explicite car sans celà, il est impossible d'utiliser des caractères pour le lire. Le language de haut niveau « craft » est donc traduit en language intermédiaire pour la machine virtuelle. Ce dernier est à son tour traduit en language assembleur qui donne des instructions directes sur les flux de données. Malgré tout, une vraie machine ne peut lire ce type de fichier car il est composé de caractères encore incompréhensibles.

C'est à ce moment que l'assembleur intervient. L'assembleur est l'outil qui gère la traduction de programmes en langage symbolique assembleur vers le langage machine. L'assembleur s'occupe également de la gestion des symboles définis par le système ainsi que par l'utilisateur et leur assigne à des adresses de la mémoire physique selon les besoins. On récupère donc de vrais fichiers binaires en sortie de notre assembleur pour une rapidité optimale de l'émulateur CPU.

3.1 Assembleur

L'assembleur que nous avons réalisé permet l'utilisation d'une table de symboles et la gestion d'une RAM (mémoire vive), d'une ROM (mémoire morte) et de registres. Le fichier reçu en entrée a pour extension « .asm » et a pour sortie un fichier en binaire (dont l'extension est « .hack ») composé de bits compréhensible par la machine ainsi qu'un « .dbg » qui est un fichier de débug comprenant toutes les informations liées au debugging (méta-données et commentaires). La réflexion sur la façon de structurer cet assembleur est ce qui a pris le le plus de temps. Mais l'importance capitale d'un bon fonctionnement de l'assembleur a entraîné un important travail.

Au départ, notre assembleur ne nous donnait que des fichiers en binaire symbolique composé de « 0 » et de « 1 » en ASCII. On a donc cherché à lexer et parser les fichiers ASCII avec OCaml et sommes tombés sur une extension de syntaxe du language appelée « Stream » (flux en français).

La syntaxe de notre langage assembleur est la suivante:

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

3.1.1 Une première technique : les flux

Le premier réflexe de l'ensemble de l'équipe a été de rechercher un moyen efficace de traîter des flux de données afin de pouvoir gérer et travailler sur les fichiers de programmes. C'est ainsi que nous avons décidé d'utiliser les flux pour lire des fichiers ASCII (« streams » en OCaml).

Les flux fonctionnent comme une liste de caractères à laquelle on accèderait de manière récursive en celà qu'ils permettent l'accès au premier caractères du flux et le détruisent une fois testé. Il est donc possible de stocker les éléments du flux au fur et à mesure de son avancement mais également d'en rajouter voir même de stocker des sous-ensembles du flux principal, c'est à dire des sous-flux et les gérer de la même façon. La gestion approfondie des contenus de fichiers est donc passée par le système de flux lors de la première étape de ce projet. Les fonctionnalités plus techniques permises par les flux ont été abordées lors des précédents rapports.

- Des améliorations nécessaires et fructueuses
  Le lexing et parsing du fichier « .asm » se fait toujours à l'aide des streams mais étant donné que le format « .hack » en sortie de l'assembleur est désormais du vrai binaire, nous utilisons les fonctions d'input et d'output binaire de

  OCaml. Ceci se fait octet par octet à l'aide des fonctions :

  En sortie de l'assembleur :

  - output_byte (open_out_bin nom_fichier) $\text{valeur}_{\text{decimaledubit}}$
  - flush (open_out_bin nom_fichier)
  - close_out (open_out_bin nom_fichier)

  En entrée de l'émulateur :

  - input_byte (nom_fichier)
  - close_in (open_in_bin nom_fichier)

  Notre ordinateur tourne sur du 16bits, il faut donc output deux par deux en utilisant le masque de bits sur les 8 premiers puis les 8 autres au niveau de l'assembleur. De la même façon, l'émulateur récupère input deux octets par deux octets. Ce système de vrai binaire a permis de passer le temps de chargement du pong de 40 secondes à 10 ainsi que multiplier la vitesse du jeu par 30. De plus de nombreuses erreurs liées aux opérations logiques sur du binaire ont été éradiquées. C'est certainement l'une des optimisations majeures de notre projet.

  Voici un récapitulatif du fonctionnement de notre assembleur à deux passes:

3.1.2 La première passe

La première passe correspond à une phase de parsing, d'enregistrement des adresses dans la mémoire morte des instructions et de construction d'une table de symboles associés à leurs valeurs pour les pseudo-commandes. Les pseudo-commandes sont de la forme « (Xxx) ». Le parsing s'effectue grâce aux flux. Le test de caractères et la possibilité de stocker ces caractères progressivement dans des buffers en font un outil idéal. Le fait de rencontrer une A-instruction ou une C-instruction (adressement ou calcul) incrémente le pointeur d'instruction de 1 et met la valeur du premier des 16 bits à respectivement 0 et 1. La table des symboles est représentée par une hashtable dans laquelle on stocke au fur et à mesure les labels parsés associés à leur valeur au niveau du pointeur d'instruction.

Afin de faciliter la seconde passe qui comprend des sauts conditionnels ou non à certaines adresses du registre du pointeur d'instruction, la solution d'un tableau s'est présentée. L'ensemble des instructions sont donc stockées une-à-une dans le tableau lors de l'étape du parsing du flux. Le tableau est redimensionné pour éviter que le programme ne pointe vers parties vides ou non-autorisées de la mémoire (ce qui entrainerait un « segfault »).

3.1.3 La seconde passe

La seconde passe correspond à une phase de parcours du tableau, de remplacement des instructions d'adressement par leur valeur numérique, de gestion des variables et de leur adresse dans la mémoire vive et enfin de traduction des instructions en language machine. Le système de saut à certaines adresses est implémenté ce qui permet de gérer des boucles nécessaires au bon fonctionnement de tout programme. Les différents sauts conditionnels sont gérés par cette passe. Ces sauts sont définis sur 3 bits et fonctionnent selon un système de comparaison de la valeur annoncée dans le segment « computation » par-rapport à 0. Le système de registres et de mémoires fonctionne, de façon assez simple, à l'aide de tableaux et de variables qui prennent différentes valeurs progressivement au cours du parcours du programme. Les calculs effectués sur les valeurs contenues dans les différents registres sont définis sur 7 bits. Ces mêmes calculs sont réalisés par l'ALU et stockés dans le registre mémoire.

Les valeurs des A-instructions sont envoyées dans le registre A d'une part et peuvent par la suite être également envoyées soit dans le registre mémoire D, soit dans à la place A de M qui correspond à la RAM ou associées à d'autres valeurs pour être calculées avant d'être placées dans ces registres.Ces destinations sont définies sur 7 bits.

Récapitulatif de la répartition des bits pour les instructions.

A-instruction: forme « @Xxx »

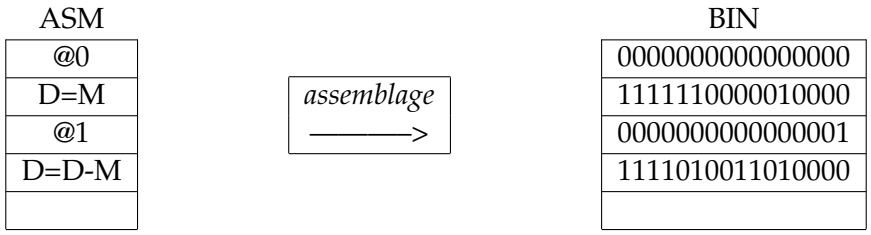
- premier bit: 0
- 15-bits restants: valeur en système binaire du nombre décimal représenté
- par la A-instruction.

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

Remarque: un bit étant occupé pour la définition du type d'instruction, les nombres sont en réalité définis sur 15 bits ce qui diminue le nombre de possibilités.

Exemple:



Exemple de traduction en binaire

3.2 L'émulateur

L'émulateur est totalement terminé. Il comprend la gestion de l'ensemble des registres, des sauts à certains pointeurs d'instruction (et donc des boucles), du clavier et de l'affichage à l'écran de certaines adresses de la RAM.

Prenons le programme de calcul du maximum entre deux entiers:

```
// Calcul M[2] = max(M[0], M[1])  où M est la RAM
    @0
    D=M
    @1
    D=D-M
    @OUTPUT_FIRST
    D;JGT
    @1
    D=M
    @OUTPUT_D
    0;JMP
(OUTPUT_FIRST)
    @0
    D=M
(OUTPUT_D)
    @2
    M=D
(INFINITE_LOOP)
    @INFINITE_LOOP
    0;JMP
```

-une boucle infinie est utilisée par les JMP pour éviter un « segfault »

3.3 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.

3.3.1 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 a longtemps été le 4ème résultat sur Google de « OCamlSDL ».

3.3.2 Exemples de programmes émulés:

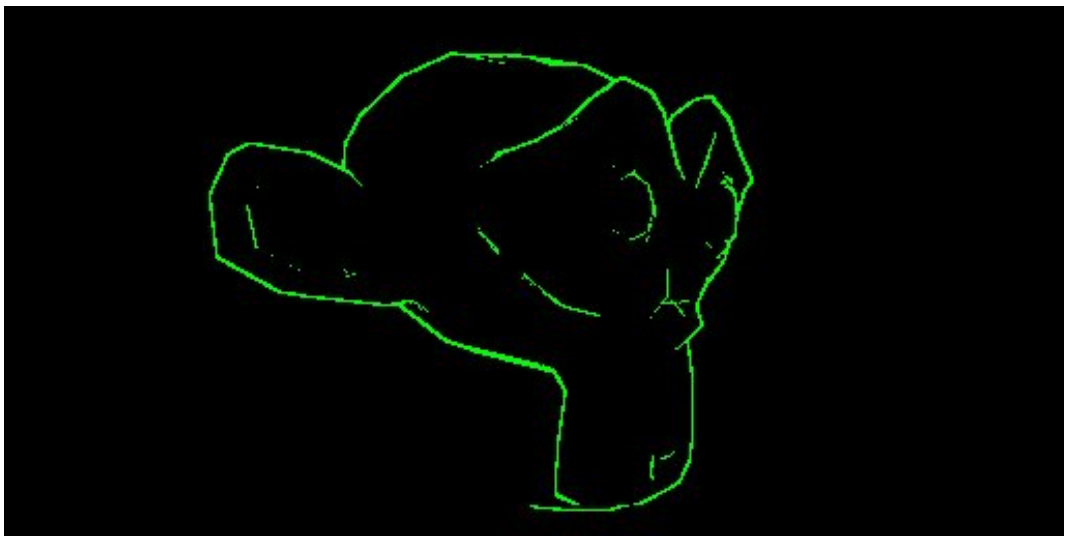


Figure 1: animation 1

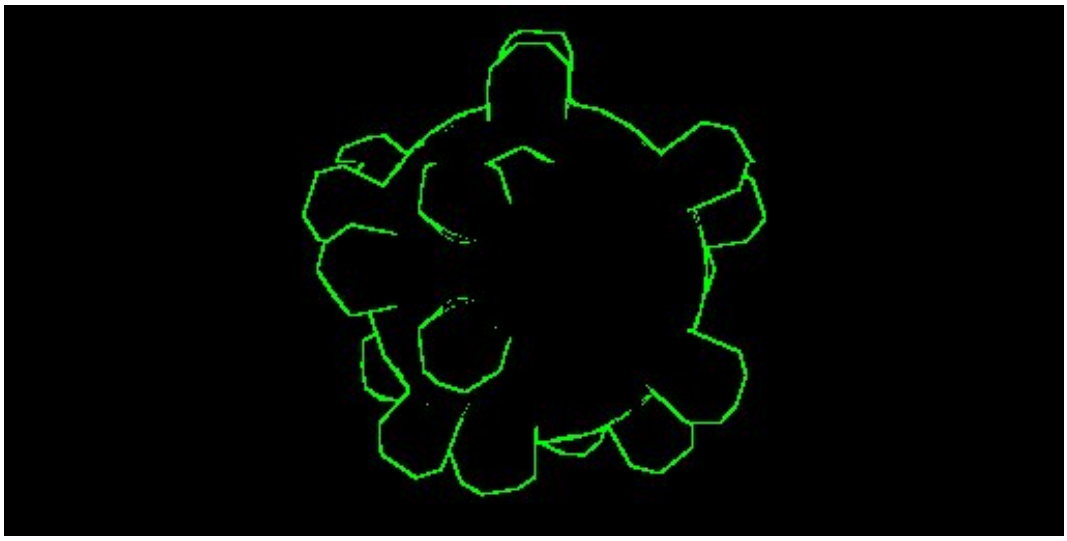


Figure 2: animation 2

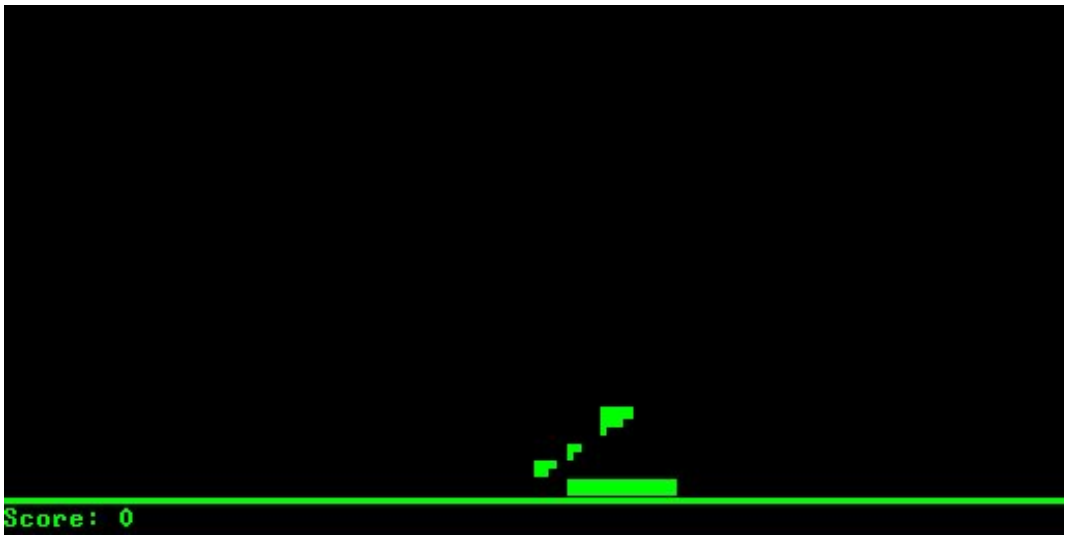


Figure 3: pong

- 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 RAM 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

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

3.3.3 Le fonctionnement

L'émulateur récupère les instructions une à une 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 (land et lor sous OCaml)
- additions
- soustractions (les nombres négatifs sont également gérés)
- « NON » logique (lnot sous OCaml)

Récapitulatif de la répartition des bits pour les instructions Nos opcodes sont définis comme suit :

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

  En terme de code, la gestion par la ram de binaire sous forme d'entiers

plutôt que de chaînes de caractères permet une optimisation en terme de poids d'instructions énorme.

3.4 Le débuggeur

3.4.1 Rappels: Langague assembleur

Nous avons dû perfectionner à la dernière soutenance notre assembleur qui traduit des programmes en langage assembleur vers du langage binaire avec une table des symboles. La syntaxe de notre langage assembleur est la suivante:

[2]Le simulateur disponible lors de la précédente soutenance 2 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.

- 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. Tous ces commentaires sont stockés dans un « .dbg » avec le pointeur d'instruction qui leur correspond.

Nous avons également ajouté un nouvel élément dans le language assembleur:

- Opérations sur les éléments principaux marquée par le

Nous pouvons par exemple avoir :

ASM	comentaire
D=M	// D=R13
@3	
A=D-A	// A=R13-3
~A=(*R13)-3	
D=M	// D=M(R13-3)
@ARG	
M=D	// ARG <- M(R13-3)

Le débuggeur 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).

Le débuggeur écrit sur la sortie standard à chaque début de cycle le type de saut, les valeurs des différents registres ainsi que la valeur contenue dans le pointeur de stack et la fonction dans laquelle nous nous trouvons lors de l'instruction en cours.

Nous pouvons également poser des breakpoints c'est à dire demander à notre programme de s'arrêter à une certaine instruction et nous donner la valeur des différentes mémoires à ce moment ou alors des watchpoints qui consistent en l'observation de toutes les assignations de valeurs à une adresse donnée de la RAM.

L'ensemble de ces informations permet au programmeur de résoudre de nombreux problèmes. Et pour ce qui est des erreurs de syntaxes, des messages d'erreurs spécifiques sont donnés au parsing.

3.4.2 Le désassembleur

Un module pour le célèbre et on ne peut plus moderne désassembleur IDA a été codé. Celui-ci nous permet de voir quel code assembleur correspond à le binaire en question puis ségmenter le code assembleur selon les portions qui s'executent d'un coup où celles qui nécessitent un saut conditionnel ou non. C'est un module extrêmement pratique qui aide énormément à l'optimisation du code grâce aux fonctionnalités de IDA qui génère des graphiques à partir de programme et renseignent donc sur d'éventuelles portions redondantes de code.

Le loader n'est finalement plus nécessaire du fait des importantes optimisations qui ont été réalisées en amont pour la factorisation de code.

4 Machine Virtuelle par *Paul*

4.1 Description

La machine virtuelle est au centre de la chaine qu'est la traduction du code. Son rôle est de traduire les programmes générés par le compilateur (en langage intermédiaire, appelé aussi bytecode) en langage assembleur (qui sera donc pris en charge par l'assembleur par la suite). Pourquoi donc cette étape intermédiaire ? Pourquoi le compilateur ne traduit-il pas directement en assembleur ? La machine virtuelle est l'endroit rêvé pour effectuer des optimisations sur le code traduit : le langage intermédiaire permet de reconnaitre des schémas facilement optimisables dans le code. La machine virtuelle reconnait donc ces schémas et utilise des astuces du langage assembleur pour réduire la taille du code généré. Elle implémente en outre un émulateur capable de simuler l'execution des programmes écrits en bytecode, il est utilisé pour des tests bien entendu, mais il permet aussi l'optimisation la plus importante de la machine virtuelle : la pré-execution.

4.2 Le Bytecode

Le langage intermédiaire utilisé par la machine virtuelle fonctionne sur le principe de pile, c'est à dire que les instructions qui forment ce langage manipulent la mémoire sous l'apparence d'une pile maitresse et de segments de mémoire.

Les fonctions *PUSH* et *POP* déplacent des valeurs entre la pile et les segments de mémoire. Il existe huit segments de mémoire : Static, Argument, Local, This, That, Pointer, Temp et Constant, tous représentés par leur pointeur, indiquant leur position dans la ram.

Les fonctions arithmétiques telles que l'addition, la soustraction, les comparaisons ou les opérations logiques manipulent uniquement les valeurs en haut de la pile. *ADD* supprimera les deux valeurs les plus hautes de la pile et y mettra le résultat de leur addition, *EQ* y placera un booléen représentant si oui ou non les deux valeurs sont égales, *NOT* remplacera le booléen en haut de la pile par son complément, etc…

Enfin, des fonctions de contrôle de flux *FUNCTION*, *CALL*, *RETURN*, *GOTO* et *IF-GOTO* permettent de définir des fonctions, de les appeler et de créer des branchements conditionnels.

4.3 Le traducteur

La fonction principale de la machine virtuelle est donc de traduire un programme écrit dans ce langage intermédiaire en langage assembleur, elle se base pour cela sur des équivalences permettant de traduire chaque instructions bytecode en une série d'instruction assembleur. Ainsi un *PUSH CONSTANT 42* sera traduit comme tel :

```
1 @42
2 D=A
3 @SP
4 AM=M+1
5 A=A-1
6 M=D
```

4.4 L'émulateur

L'émulateur se présente comme un module Ocaml. Le type qu'il manipule est un environement dont voici la description :

```
type environment =
{
    (* RAM *)
    ram : int array;

    (* Various pointers *)
    mutable sp : int;
    mutable arg : int;
    mutable lcl : int;

    call_stack : string Stack.t;
    mutable pc : int;

    (* Names tables *)
    mutable labels : (string, int) Hashtbl.t;
    mutable functions : (string * string, int * int) Hashtbl.t;
    (*                  filename *funname, pc  * localnum    *)

    mutable il_program : il_code
}
}
```

Cet environnement contient la mémoire, les pointeurs du bytecode, une pile d'appel afin de savoir à tout moment dans quelle fonction nous somme, un pointeur d'instruction, deux table renseignant la position des labels et des fonctions dans le code et enfin, le programme.

L'émulateur fournit donc des fonctions permettant de charger un programme dans cet environnement, de construire les tables de fonctions et de labels, d'executer une instruction ou tout le programme, et enfin d'afficher le contenu de la ram (à des fin de test). Voici la signature de ce module :

```
module VM_interpretor(D : Displayor) :
  sig
    type environment =
    {
        (* RAM *)
        ram : int array;

        (* Various pointers *)
        mutable sp : int;
        mutable arg : int;
        mutable lcl : int;

        call_stack : string Stack.t;
        mutable pc : int;

        (* Names tables *)
        mutable labels : (string, int) Hashtbl.t;
        mutable functions : (string * string, int * int) Hashtbl.t;
        (*                  filename *funname, pc  * localnum    *)

        mutable il_program : il_code
    }
```

```
23
24      val initial_environment : environment
25
26      (* IL commands *)
27      val push : environment -> Parser.segment -> int -> unit
28      val pop : environment -> Parser.segment -> int -> unit
29      val operation : environment -> Parser.operation -> int -> unit
30      val goto : environment -> string -> unit
31      val ifgoto : environment -> string -> int -> unit
32      val skip_function : environment -> unit
33      val call : environment -> string -> string -> int -> int -> unit
34      val return : environment -> int -> unit
35
36      (* Basic informations *)
37      val print_ram : environment -> int -> unit
38      val print_stack_top : environment -> unit
39
40      (* Prepare and execute programs *)
41      val make_label_table : environment -> unit
42      val load_program : il_code -> environment
43      val reload_program : environment -> il_code -> environment
44      val run : environment -> string
45      val run_script_like : environment -> unit
46
47      (* Precompilation tools *)
48      val extract_init : il_code -> (il_code * il_code)
49      val pre_exec : char Stream.t -> int array
50    end
```

Ici, « Displayor » est un module à passer en paramètre à l'émulateur, ce module se chargera d'afficher diverses informations et l'émulateur est complètement indépendant de son implémentation. La fonction extract_init sert à la dernière optimisation, présenté plus bas.

4.5 Les optimisations

Objectif de la machine virtuelle, les optimisations réduisent la taille du code assembleur généré

4.5.1 Push et Pop

Parfois, une séquence d'instruction Bytecode fait transiter des valeurs sur la pile alors que l'opération attendu est un simple transfert d'un segment de mémoire à un autre. En bytecode, c'est le cas quand une instruction *POP* suit immédiatement un *PUSH*, la valeurs que l'on manipule quitte son segment d'origine, s'arrête sur la pile et est transférée de nouveau vers un autre segment. Une optimisation est donc de détecter cette suite d'instruction et de la traduire d'un bloc, sans passer par la pile. Cette optimisation nous fait passer de vingt-deux instructions assembleur (*PUSH* et *POP* séparés) à quatorze, ce qui, sur un programme entier, est une réduction considérable car ce cas de figure se répète assez souvent.

4.5.2 Factorisation du code assembleur

Les fonctions de comparaison *GT*, *LT* et *EQ* prennent environ dix-huit instructions assembleurs, car elle doivent se faire via les instructions de saut de code de l'assembleur, seul

outil permettant la comparaison de valeurs. Ce code n'a en fait pas besoin d'être réécris à chaque fois que l'on souhaite traduire une comparaison. En effet une optimisation possible est, à la rencontre d'une fonction de comparaison, de sauvegarder le numéro de l'instruction courrante dans un registre temporaire, puis de sauter à un endroit du code écrit au préalable (lors de la phase d'initialisation) qui va alors récupérer les valeurs en haut de la pile, les comparer, mettre le bon résultat en haut de la pile, puis retourner à l'instruction dont le numéro se trouve dans le registre. Nous gagnons ainsi encore une dizaine d'instruction assembleur à chaque comparaison, ce qui est considérable.

Plus compliqué maitenant, les fonctions d'appel et de retour de fonction généraient une quantitée colossale d'instruction assembleur. Dans le cas de *return*, la procédure est la même dans tous les cas, il n'était donc pas excessivement compliqué de garder le bloc de code au début du programme et d'y faire appel à chaque fois que l'on en avait besoin. Le vrai tour de force était d'optimiser l'instruction *call*. Celle-ci a besoin de trois informations n'étant quasiment jamais les mêmes : le nom de la fonction appelée, le nombre d'arguments, et une addresse de retour. Le problème étant que nous disposons de trois registres temporaires pour y mettre ces valeurs, mais que l'un d'eux est déjà utilisé lors de la procédure d'appel. Il a donc fallu ruser et garder une des informations dans le registre D, un registre processeur à très court terme car constament utilisé. L'adresse de retour se trouve être la seule information qui, après quelques ajustements, peut être utilisée avec une telle contrainte.

Grâce à ces optimisation, nous avons par exemple, sur un programme à la base de 31 000 instructions assembleur, économisé plus de 6 000 instructions.

4.5.3 Pré-execution

L'optimisation la plus importante de la machine virtuelle utilise son émulateur. Beaucoup de programmes commencent leur execution par charger des bibliothèques en mémoire, ils initialisent des morceaux de mémoire permettant d'utiliser des tableaux par exemple ou des chaines de caractère. Le comportement de ces initialisations est invariable, il ne requiert aucune information exterieur et n'a pas besoin d'en fournir. Autrement dit il est totalement invisible au moment de l'execution, si ce n'est qu'il prend du temps et de la place dans le code. L'idée est donc de simuler l'execution de ces fonctions d'initialisation dans l'émulateur, de récupérer l'état mémoire après l'execution, puis de fournir à l'assembleur la mémoire initialisée d'une part et le reste du programme d'autre part. Pour cela, l'émulateur fournit la fonction extract_init, qui à partir d'un programme, fournit un programme contenant les fonctions d'initialisation et un autre contenant le reste, il suffit alors de simuler le premier et de traduir le second.

4.6 Le debogage

L'assembleur peut à tout moment connaitre la valeur contenue dans ses registres, seulement quand on cherche une erreur, ces valeurs peuvent se révéler complêtement obscure. C'est pourquoi nous avons ajouté au traducteur une fonctionnalité lui permettant de donner à l'assembleur la signification de ces valeurs.

4.6.1 Informations sur les fonctions

Tout d'abord, chaque portion de code correspondant à une instruction en bytecode est précédée d'un commentaire renseignant le nom de cette instruction, afin qu'à tout endroit du code assembleur l'on puisse savoir dans quelle fonction nous somme. L'information sera précédée d'un croisillon. Exemple : la traduction en assembleur de l'instruction *function Main.fibo 2* sera précédée de la ligne **#function Main.fibo 2**

4.6.2 Informations sur les registres

Ensuite, une information encore plus utile consiste à renseigner régulièrement la signification du contenu des registres du processeur, c'est à dire par exemple pendant un séquence correspondant à un *PUSH*, de préciser que la valeur actuellement contenue dans le registre *D* est en fait la valeur que l'on veut push. C'est cette fois-ci un tilde qui introduira la ligne donnant l'information. Exemple : la ligne correspondant à l'exemple ci-dessus sera **∼D=push_value**

5 Compilateur - syntaxe par *Rémi*

Le travail du compilateur est de transformer un langage en un autre, généralement dans le but de créer un exécutable. Le compilateur du projet `NANDCRAFT`, nommé `craft`, doit traduire le langage Craft vers le langage de la machine virtuelle.

C'est un procédé complexe que nous avons divisé en deux étapes : la création de l'arbre syntaxique abstrait puis la génération de code.

5.1 Analyseur lexical

Les streams d'ocaml sont des outils puissants et très simple d'utilisation, cependant pour le projet du compilateur il est nécessaire d'utiliser des outils adaptés, plus robuste et qui permettent d'éviter les bugs que l'on aurait pu introduire si on avait écrit l'analyseur à la main.

Nous avons utilisé l'outil `ocamllex` pour décrire l'ensemble des lexèmes du langage. Il permet de définir des lexèmes comme des expressions rationnelles, ainsi la définition du lexème représentant un chiffre est :

```
let digit = ['0'-'9']
```

Voici un exrait de `lexer.mll` :

```
1  let digit = ['0'-'9']
2  let ident = ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
3
4  rule token = parse
5    | [' ' '\t' '\n'] { token lexbuf }
6    | "//" { comment_eof lexbuf }
7    | "/*" { comment lexbuf }
8
9    | "class"
10   | "constructor"
11   | "method"
12   | "function"
13   | "int"
14   | "boolean"
15   | "char"
16   | "void"
17   | "var"
18   | "static"
19   | "field"
20   | "let"
21   | "do"
22   | "if"
23   | "else"
24   | "while"
25   | "return"
26   | "true"
27   | "false"
28   | "null"
29   | "this" as kwd { Hashtbl.find keywords kwd }
30
31   | '"' ([^ '"']* as str) '"' { STRING_CONSTANT str }
32   | digit* as int_const { INTEGER_CONSTANT (int_of_string int_const) }
33   | ident as word { IDENTIFIER word }
```

```
34
35    (* a lonely char must be a symbol *)
36    | _ as c { Hashtbl.find symbols c }
37    | eof  { raise End_of_file }
```

5.2 Analyseur syntaxique avancé

De même que nous aurions pu écrire l'analyseur syntaxique à l'aide des flots d'ocaml mais nous avons plutôt choisi d'utiliser `menhir` pour réaliser cette tâche.

`menhir` est une amélioration de l'outil `ocamlyacc`, il y ajoute certaines fonctionnalités qui facilitent et rendent les grammaires écrites plus expressives. On citera par exemple l'existence le nommage des valeurs sémantiques à l'intérieur des règles. Ainsi, au lieux d'écrire :

```
exp:
  | NUM { $1 }
  | exp + exp { $1 + $2 }
```

on peut écrire :

```
exp:
  | n=NUM { n }
  | e1=exp + e2=exp { e1 + e2 }
```

Voici un extrait de l'analyseur syntaxique, on notera l'utilisation des fonctionnalités de menhir suivantes :

- valeurs sémantiques nommées
- quantificateurs EBNF (?, +, *)
- symboles paramétrés (delimited, option, preceded)

```
statement:
  | s=letstatement    { s }
  | s=ifstatement     { s }
  | s=whilestatement  { s }
  | s=dostatement     { s }
  | s=returnstatement { s }

letstatement:
  | LET name=varname array=arrayaccess? EQUAL e=expression SEMICOLON
  {
    Ast.LetStatement (name, array, e)
  }

ifstatement:
  | IF pred=delimited(LPAREN, expression, RPAREN)
    thens=delimited(LBRACE, statement*, RBRACE)
    elses=option(preceded(ELSE, delimited(LBRACE, statement*, RBRACE)))
    {
      Ast.IfStatement (pred, thens, elses)
    }
```

```
21
22 whilestatement:
23   | WHILE pred=delimited(LPAREN, expression, RPAREN)
24     s=delimited(LBRACE, statement*, RBRACE)
25     {
26       Ast.WhileStatement (pred, s)
27     }
28
29 dostatement:
30   | DO c=subroutinecall SEMICOLON
31   {
32     Ast.DoStatement c
33   }
34
35 returnstatement:
36   | RETURN e=expression? SEMICOLON
37   {
38     Ast.ReturnStatement e
39   }
```

5.3 Arbre syntaxique abstrait

L'arbre de syntaxe abstraite (ou *AST* pour *Abstract Syntax Tree*) est une structure arborescente permettant de manipuler le langage source. Chaque noeud correspond à un élément syntaxique du langage.

5.3.1 Implémentation

Nous avons d'abord implémenté l'AST en utilisant un arbre composé de nombreux types OCaml. Ceci nous oblige à écrire de façon très rigoureuse l'arbre et nous assure que les fils de chaque noeud correspondent à un type de donné spécifique. On aurait pu en effet n'utiliser qu'un seul type somme pour définir l'ensemble des types de noeuds. Cependant cette structure est difficile à parcourir car à chaque nouveaux noeud rencontré dans le parcours de l'arbre il aurait fallu utiliser un filtrage de motif spécifique au type de ce noeud.

5.3.2 Visualisation de l'AST

Nous avons utilisé le langage `dot` [3] pour produire des visualisations sous forme de graphe de l'AST. Il n'existe pas de bibliothèque en OCaml pour générer ces graphes[4] nous avons donc du en écrire une.

La première version de cette bibliothèque ne faisait que traverser l'AST en générant pour chaque noeud qu'elle rencontrait le code `dot` équivalent. Ce code devait être stocké en mémoire puis réorganisé pour respecter l'ordre :

1. Définition des noeuds
2. Définition dest arrêtes

Cela fonctionnait très bien mais cette procédure n'était pas suffisamment flexible pour que mous puissions implémenter des algorithmes avancés facilement.

[3] Site web :http://www.graphviz.org/doc/info/lang.html

[4] si on néglige ocamlgraph qui est une usine totalement non documentée et non adaptée à nos besoin

5.3.3 Module générique de graphes

Nous avons donc écris une autre bibliothèque sous forme d'un module OCaml nommé `Graph` ayant pour but la manipulation de graphes. Il est générique, c'est à dire qu'il s'adapte à tout type de graphes, peut importe leur domaine d'utilisation. Cette généricité est atteinte car le module est un foncteur prenant en paramètre le type des noeuds du graphe. Par exemple pour la génération du code `dot` on applique le foncteur `Graph` sur le type de noeud `DotAstNode`. Ce type contient toutes les informations nécessaire à la génération du code `dot`.

Le module `Graph` est implémenté avec une liste d'adjacences. Dans ce type de structure, on a un tableau de liste de sommet. Le tableau comporte autant de cases qu'il y a de sommets. Chacune des cases pointe vers une liste de sommet. Cette liste n'est rien d'autre que les successeurs du sommet considéré.

Ceci permet l'implémentation d'algorithme tels que le parcours profondeur (ou DFS pour *Depth First Search*) trivial.

6 Compilateur - génération de code par *Rémi*

6.1 Présentation du compilateur

6.1.1 Compilation du langage craft

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

6.1.2 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.

6.1.3 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` :

```
class DebugMe {
    field int a, b;
    static bool lol;

    constructor DebugNew new () {
        let a = 42 + 2;
        let b = 2 / Math.pow(2, 4);

        let lol = a = b;
    }
}
```

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
```

6.1.4 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 :

```
class Main {}

class Main {}
```

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

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

Il existe bien d'autres warnings (certains sont cachés, à vous de les trouver !)[5]

6.2 Fonctionnement du compilateur

6.2.1 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 :

```
module H = Hashtbl

type name = string

type ftype =
  | Constructor
  | Procedure (* aka. function *)
  | Method

type locals = int

type stype =
  | Int
  | Char
  | String
  | Boolean
  | Void
  | UserType of string
  | Function of (ftype * rettype * locals * stype list)
and rettype = stype

type kind =
  | Static
  | 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;

```

[5] astuce : coder comme un porc

```
    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 :

```
(* Nouvelle 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
(* le nom de la classe courante *)
val new_class : symtable_data -> string -> unit
(* nouveau nom dans la table de symboles *)
val define : symtable_data -> name -> stype -> kind -> unit
(* retourne un symbole a partir de son nom *)
val get_sym : symtable_data -> name -> symbol
(* retourne la table des symbole appartenant a un 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 l'index d'un nom (utile pour obtenir des labels dans le code en sortie) *)
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 la 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.

6.2.2 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 segments *)
type segment =
    Static
  | Argument
  | Local
  | This
  | That
```

```
  | Pointer
  | Temp
  | Constant

(* Les instructions *)
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 intermediaire *)
val comment : string -> string
(* retourne le code intermediaire pour l'operation 'op' *)
val write : op -> string
```

6.2.3 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
val compile_letarray : A.statement -> unit
val compile_if : A.statement -> unit
val compile_while : A.statement -> unit
val compile_do : A.statement -> unit
val compile_return : A.statement -> unit
val compile_statements : A.statements -> unit
val compile_subroutine_body : A.statements -> unit
val compile_subroutine_locals : A.vardec list -> unit
val compile_subroutine_args : A.parameter list -> S.stype list
val compile_subroutines : A.subroutinedec list -> unit
val compile_sub_declaration : A.subroutinedec list -> unit
val compile_class_vars : A.classvardec list -> unit
val compile_class : A.top -> unit
val compile_module : A.top -> unit
val compile_include : A.top -> unit
val compile_top : A.top -> unit
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` [6]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`.

6.2.4 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 intermediaire"
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.

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

7 Système d'exploitation

Le système d'exploitation doit faire le lien entre la couche matérielle et logicielle de l'ordinateur. Il doit donner au programmeur un moyen simple de développer des applications sans se soucier du fonctionnement de la machine.

Dans le cadre de notre projet il fournit surtout un ensemble de bibliothèque qui permette de s'abstraire du fonctionnement concret de la machine. Par exemple on multiplie deux entiers `x` et `y` en appelant `Math.multiply(x, y)` car la multiplication n'est pas gérée au niveau matériel. Si dans le futur on utilisait un composant dédié on pourrait juste remplacer l'implémentation de cette méthode par un appel à lui. Ainsi le code utilisant cette biliothèque n'aurait pas à être changé mais pourrait quand même bénéficier du gain de vitesse.

7.1 Les fonctionnalités du système d'exploitation CraftOS

- Primitives de dessin sur l'écran
- Affichage de texte à l'écran
- Gestion de l'allocation et de la libération de la mémoire

7.2 Algorithmes utilisés

Dans cette section on va présenter quelques algorithmes imlémentés dans le CraftOS.

7.2.1 Multiplication

L'ordinateur nandcraft ne dispose pas d'un circuit dédié à la multiplication, on doit donc implémenter un algorithme utilisant les opérations élémentaires dont nous disposons pour accomplier ma même tâche.

On utilise la méthode de la multiplication égyptienne pour calculer la le produit de deux nombres.

(Pour rappel le code donné ici est écrit en Craft, le langage pour lequel on a déjà écrit un compilateur et un assembleur)

```
1  function int multiply(int x, int y) {
2    var int result;
3
4    let result = Math.multiplyAbs(x, y);
5    if(x < 0) {
6      let result = -result;
7    }
8    if(y < 0) {
9      let result = -result;
10   }
11
12   return result;
13 }
14
15 function int multiplyAbs(int x, int y) {
16   var int i, shiftedX, result;
17
```

```
18    let x = Math.abs(x);
19    let y = Math.abs(y);
20
21    let i = 0;
22    let shiftedX = x;
23    let result = 0;
24    while(i < 16) {
25      if(Math.bit(y, i) = 1) {
26        let result = result + shiftedX;
27      }
28      let shiftedX = shiftedX + shiftedX;
29      let i = i + 1;
30    }
31
32    return result;
33  }
```

Elle s'effectue en $O(n)$ additions pour des nombres de n bits.

7.2.2 Division euclidienne

La méthode naïve pour diviser qui consiste à retrancher successivement le diviseur au dividende tant quand que le résultat est supérieur à zéro. Cependant on voit bien que la complexité de cet algorithme croit avec la taille du quotient.

On l'optimise en retranchant le plus grand multiple de deux du diviseur à chaque fois. Il a été implémenté ici de manière récursive, mais on aurait aussi pu le faire de manière itérative.

```
1   function int divideRec(int x, int y) {
2     var int q;
3
4     if(y < 0) {
5       return 0;
6     }
7     if(y > x) {
8       return 0;
9     }
10    let q = Math.divideRec(x, y + y);
11    if((x - ((q + q) * y)) < y) {
12      return q + q;
13    }
14    else {
15      return q + q + 1;
16    }
17  }
```

On ne parlera pas des opérations avec des nombres à virgule flottante car on ne souhaite pas les gérer dans l'OS de base. Ils pourront cependant être utilisés *via* un bibliothèque supplémentaire.

7.2.3 Racine carrée

Il existe plusiseurs moyens pour calculer efficacement la racine carré d'un nombre, par exemple en utilisant la méthode de Newton ou un développement limité. Pour le CraftOS on utilisera un algorithme simple qui un se base sur les proriétés de la racine carré $y = \sqrt{x}$:

- c'est une fonction croissante
- son inverse $x = y^2$ se base sur la multiplication (ce dont nous disposons déjà)

```
function int sqrt(int x) {
  var int left, right, mid, midSq, result;

  if(x < 0) {
    do Sys.error(4);
  }

  let left = 0;
  let right = Math.min(x, 181);
  let result = 0;
  while(left < (right + 1)) {
    let mid = (left + right) / 2;
    let midSq = mid * mid;
    if(midSq > x) {
      let right = mid - 1;
    }
    else {
      let left = mid + 1;
      let result = mid;
    }
  }

  return result;
}
```

Cet algorithme s'exécute en un temps au pire de $n/2$, soit une complexité de $O(n)$ opérations arithmétiques.

7.2.4 Traduction/Interprétation ASCII

Les applications doivent souvent manipuler des chaines de caractères, que ce soit pour les afficher ou car l'utilisateur lui en donne *via* le clavier.

Elles doivent également gérer la traduction d'un caractère ASCII vers un entier (par exemple dans un jeu quand l'utilisateur entre le nombre d'ennemis) et inversement (quand il faut afficher le score du joueur).

Pour cela certains processeurs possèdent des instructions dédiées au calcul arithmétiques sous forme BCD[7] ou ASCII[8]. Le processeur `nandcraft` n'en dispose pas, il faut donc recoder des fonctions pour passer d'un format à l'autre.

Caractères	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Code ASCII	48	49	50	51	52	53	54	55	56	57

```
method void setInt(int j) {
  var int d;
  var boolean sign;

  let sLength = 0;
  let sign = false;
```

[7] Binary Coded Decimal
[8] American Standart for Information Interchange

```
  if(j < 0) {
    let sign = true;
    let j = -j;
  }
  if(j = 0) {
    do appendChar(48);
  }

  while(j > 0) {
    let d = j - ((j / 10) * 10);
    do appendChar(d + 48);
    let j = j / 10;
  }

  if(sign) {
    do appendChar(45);
  }

  return reverse();
}
```

```
method int intValue() {
  var int value, i;
  var boolean sign;

  let i = 0;
  let value = 0;
  let sign = false;
  if((sLength > 0) & (string[0] = 45)) {
    let sign = true;
    let i = i + 1;
  }
  while(true) {
    if(i = sLength) {
      if(sign) {
        let value = -value;
      }
      return value;
    }
    if((string[i] > 47) & (string[i] < 58)) {
      let value = (10 * value) + (string[i] - 48);
    }
    else {
      return value;
    }
    let i = i + 1;
  }
}
```

7.2.5 Gestion de la mémoire

Les programmes déclarent et allouent des variables, la plupart sont scalaires (un entier, un caractère) ou bien leur taille est connue. Ils peuvent également utiliser des types de données dont la taille n'est pas connue à la compilation ou bien créer des objets à volée pendant l'exécution.

Les systèmes d'exploitation utilisent différentes techniques pour gérer la mémoire allouée (`Memory.alloc`) et libérée (`Memory.free`) dynamiquement.

Un bon algorithme d'allocation mémoire doit avoir les caractéristiques suivantes :

- minimise la fragmentation mémoire, c'est à dire qu'on doit perdre le moins de place possible ;
- pouvoir trouver le plus rapidement un bloc de mémoire libre disponible

```
function Array alloc(int size) {
  var Array currentPointer;
  var Array result;
  var Array temp;

  if(size < 1) {
    do Sys.error(5);
  }

  let currentPointer = freeListHead[1];
  while(true) {
    if(currentPointer = freeListTail) {
      do Sys.error(6);
    }
    if(currentPointer[0] > (size + 2)) {
      let temp = (currentPointer[0] - (size + 1)) + currentPointer;
      let temp[0] = size + 1;
      let result = temp + 1;
      let currentPointer[0] = currentPointer[0] - (size + 1);
      return result;
    }
    let currentPointer = currentPointer[1];
  }
}
```

```
function void deAlloc(Array o) {
  var Array newNode;
  var Array previous;
  var Array current;

  let newNode = o - 1;
  let previous = freeListHead;
  let current = previous[1];

  while(true) {
    if(current > newNode) {
      let newNode[1] = current;
      let previous[1] = newNode;
      do Memory.mergeBlocks(newNode, current);
      do Memory.mergeBlocks(previous, newNode);
      return;
    }
    let previous = previous[1];
    let current = current[1];
  }
}
```

7.3 Spécification

On a implémenté les bibliothèques suivantes :

- Math

  - `function void init()` (utilisée en interne)
  - `function int abs(int x)` retourne la valeur absolue de `x`
  - `function int multiply(int x, int y)` retourne le prduit de `x` et `y`
  - `function int divide(int x, int y)` retourne le quotient de la division euclidienne de `x` par `y`
  - `function int min(int x, int y)` retourne le minimum de `x` et `y`
  - `function int max(int x, int y)` retourne le maximum de `x` et `y`
  - `function int sqrt(int x)` retourne la partie entière de la racine carrée de `x`

- String
  - `constructor String new(int maxLength)` construit une chaine vide (de taille 0) qui peut contenir jusqu'à `maxLength` caractèrs
  - `method void dispose()` libère cette chaine
  - `method int length()` retourne la taille de cette chaine
  - `method char charAt(int i)` retourne le caractère à la position `j` de la chaine
  - `method void setCharAt(int i, char c)` remplce le caractère à la position `i` de la chaine par `c`
  - `method String appendChar(char c)` ajoute `c` à la chaine et la renvoit
  - `method void eraseLastChar()` efface le dernier caractère de la chaine
  - `method int intValue()` retourne la valeur entière contenue dans la chaine
  - `method setInt(int i)` place dans la chaine la représentation `ASCII` de l'entier `j`
  - `method char backSpace()` retourne le caractère `backSpace`
  - `method char doubleQuote()` retourne le caractère = »=
  - `method char newLine()` retourne le caractère `newline`

- Array
  - `constructor Array new(int size)` construit un tableau de la taille `size`
  - `method void dispose()` libère la mémoire occupée par l'objet

- Output
  - `function void init()` (utilisée en interne)
  - `function void moveCursor(int i, int j)` déplace le curseur à la position `(i, j)`
  - `function void printChar(char c)` affiche la caractère `c`
  - `function void printString(String s)` affiche la chaine `s`
  - `function void printInt(int i)` affiche l'entier `i`
  - `function void printLn()` effectue un retour à la ligne
  - `function void backSpace()` déplace le curseur d'un caractère vers l'arrière

- Screen
  - `function void init()` (utilisée en interne)
  - `function void clearScreen()` efface l'écran
  - `function void setColor(boolean b)` spécifie la couleur des opérations `drawXXX` (`false` = blanc, `true` noir)
  - `function void drawPixel(int x, int y) dessigne le pixel =(x, y)`
  - =function void drawLine(int x1, int y1, int x2, int y2) dessine une ligne du pixel (x1, y1) au pixel (x2, y2)
  - `function void drawRectangle(int x1, int y1, int x2, int y2) dessine un rectangle plein de =(x1, y1)` à `(x2, y2)`
  - `function void drawCircle(int x, int y, int r) dessine un cercle de rayon =r` (r < 181) autour de `(x, y)`
- Keyboard
  - `function void init()` (utilisée en interne)
  - `function char keyPressed()` retourne le caractère pressé sur le clavier, 0 si aucun ne l'est actuellement
  - `function char readChar()` attend qu'une touche soit pressée et relachée, affiche le caractère à l'écran puis retourne sa valeur
  - `function String readLine(String message)` affiche le message `message` à l'écran puis attends que l'utilisateur entre une chaine de caractère au clavier terminée par la touche `Entrée`, affiche cette chaine à l'écran puis la retourne.
  - `function int readInt(String message)` affiche le message `message` à l'écran puis attends que l'utilisateur entre un entier au clavier terminé par la touche `Entrée`, l'affiche à l'écran puis le retourne
- Memory
  - `function void init()` (utilisée en interne)
  - `function int peek(int address)` retourne la valeur stockée à l'emplacement mémoire `address`
  - `function void poke(int address, int value)` met la valuer `value` à l'adresse `address`
  - `function Array alloc(int size)` trouve et alloue un bloc de mémoire libre du tas de la taille `size`, retourne son adresse de base
  - `function void deAlloc(Array o)` libère la mémoire occupée par l'object `o` et la rend disponible
- Sys
  - `function void init()` (utilisée en interne, appelle toutes les autres fonctions `init()`)

- `function void halt()` arrete le programme
- `function void error(int errorCode)` affiche le message d'erreur à l'écran puis quitte
- `function void wait(int duration)` attends approximativment `duration` millisecondes et retourne

8 Annexe : Référence du langage craft

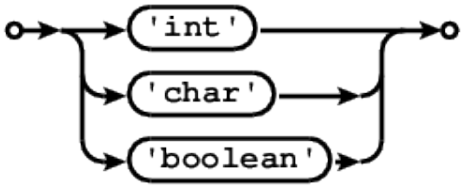
8.1 Syntaxe de Craft

Voici la description de la syntaxe du langage Craft dans deux représentations :

- Digrammes en rail, générés en hackant un outil en `tcl` à la base utilisé pour les digrammes de SQLite;
- Notation EBNF[9] équivalente.

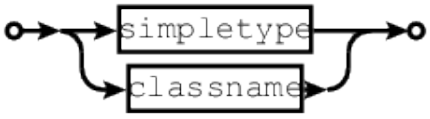
class = ‘class’ , classname , ‘{’ , { vardec } , { subroutinedec } , ‘}’;

Figure 4: class



simpletype = ‘int’ | ‘char’ | ‘boolean’;

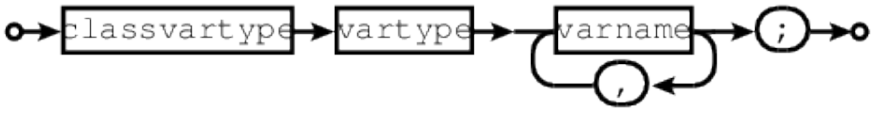
Figure 5: simpletype



advancedtype = simpletype | classname;

Figure 6: advancedtype

[9] Extended Backus-Naur Form

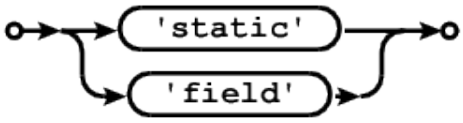


vardec = ('static' | 'field') , vartype , varname , { ',' , varname } ;

Figure 7: vardec

classname = IDENTIFIER;

Figure 8: classname

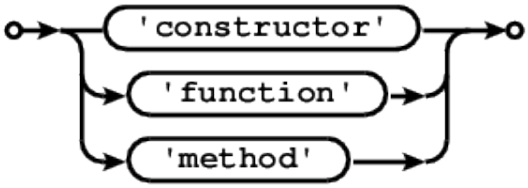


classvartype = 'static' | 'field';

Figure 9: classvartype

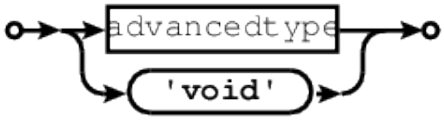
varname = IDENTIFIER;

Figure 10: varname



subroutinekind = 'constructor' | 'function' | 'method';

Figure 11: subroutinekind



subroutinetype = advancedtype | 'void';

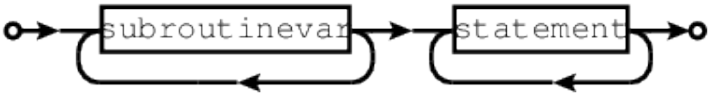
Figure 12: subroutinetype

subroutinename = IDENTIFIER;

Figure 13: subroutinename

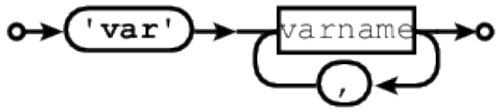
parameter = advancetype , varname;

Figure 14: parameter



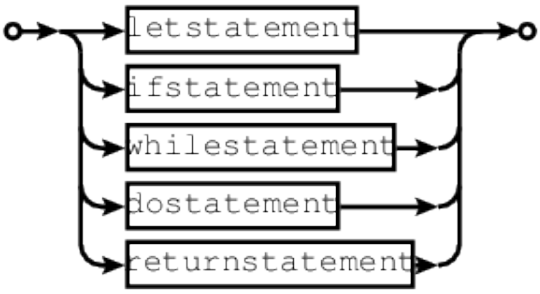
subroutinebody = { subroutinevar } , { statement };

Figure 15: subroutinebody



subroutinevar = ‘var’ , varname , { ‘,’ , varname };

Figure 16: subroutinevar

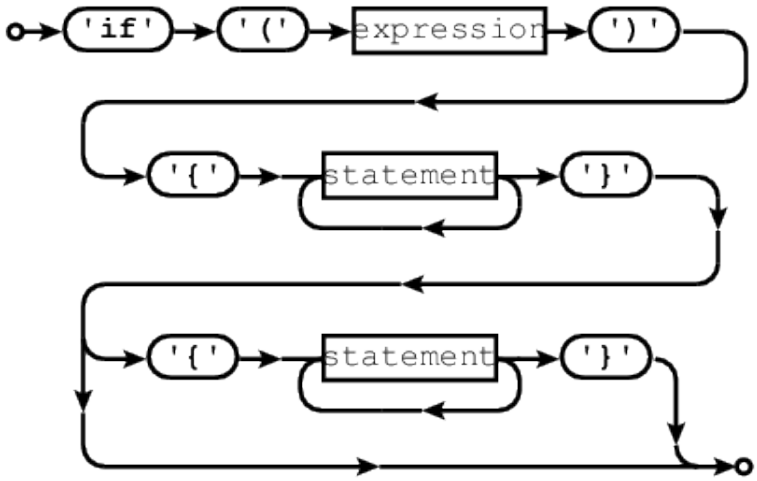


statement = letstatement | ifstatement | whilestatement | dostatement | returnstatement;

Figure 17: statement

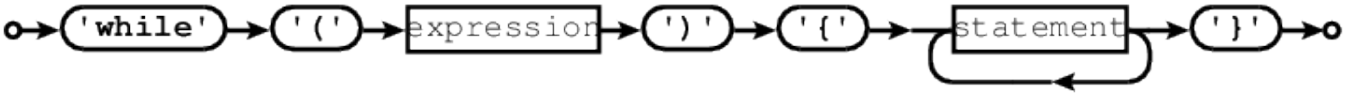
letstatement = ‘let’ , varname , [arrayaccess] , ‘=’ , expression , ‘;’ ;

Figure 18: letstatement



ifstatement = ‘if’ , ‘(’ , expression , ‘)’ , ‘{’ , { statement } , ‘}’ , [‘{’ , statement , ‘}’];

Figure 19: ifstatement



whilestatement = ‘while’ , ‘(’ , expression , ‘)’ , ‘{’ , { statement } , ‘}’;

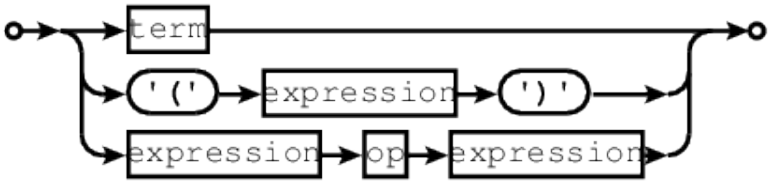
Figure 20: whilestatement

dostatement = ‘do’ , subroutinecall;

Figure 21: dostatement

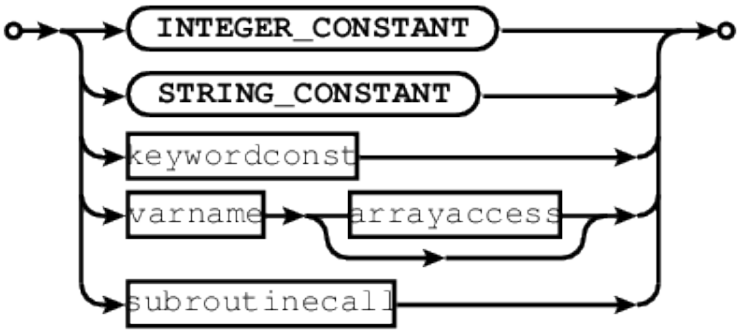
returnstatement = ‘return’ , [expression];

Figure 22: returnstatement



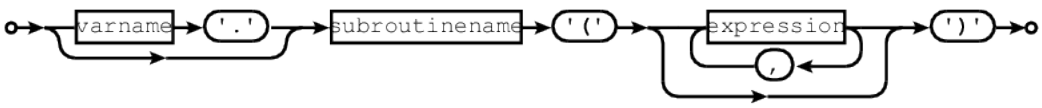
expression = term | ‘(’ expression ‘)’ | expression op expression;

Figure 23: expression



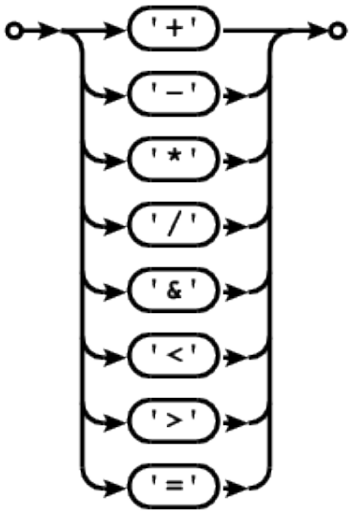
term = INTEGER_CONSTANT | STRING_CONSTANT | keywordconst | varname ,
[arrayacces] | subroutinecall;

Figure 24: term



subroutinecall = [varname , ‘.’] , subroutinename , ‘(’ , [expression , { ‘,’ , expression }];

Figure 25: subroutinecall

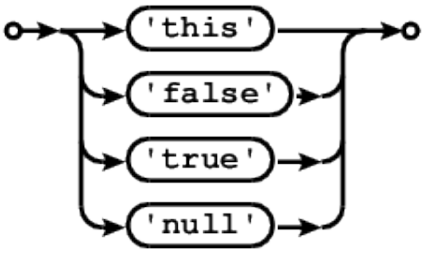


op = ‘+’ | ’-’ | ‘*’ | ‘/’ | ‘<’ | ‘>’ | ‘=’;

Figure 26: op

vartype = advancedtype;

Figure 27: vartype

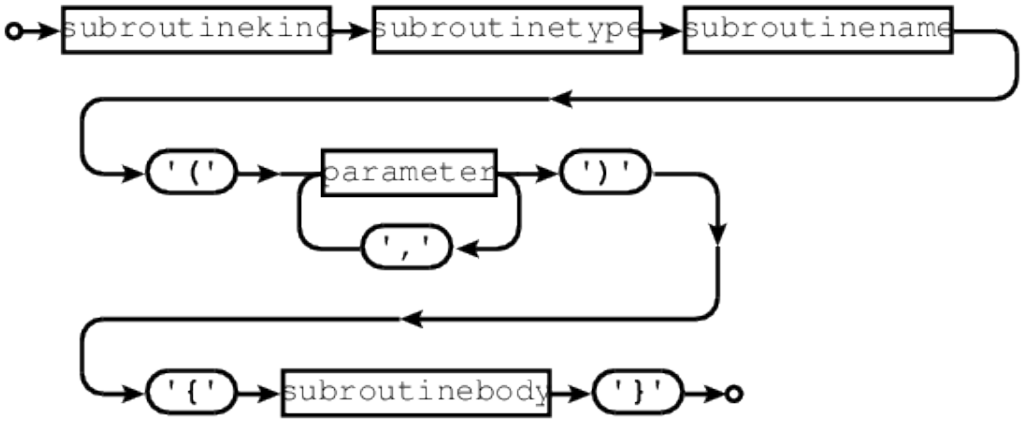


keywordconst = ‘this’ | ‘false’ | ‘true’ | ‘null’;

Figure 28: keywordconst

arrayaccess = ‘[’ , expression , ‘]’;

Figure 29: arrayaccess



subroutinedec = subroutinekind , subroutinetype , subroutinename , '(' , paramater? , { ',' , parameter } , '{' , subroutinebody , '}';

Figure 30: subroutinedec

9 Le mot de la fin

On est maintenant très heureux et en vacances.