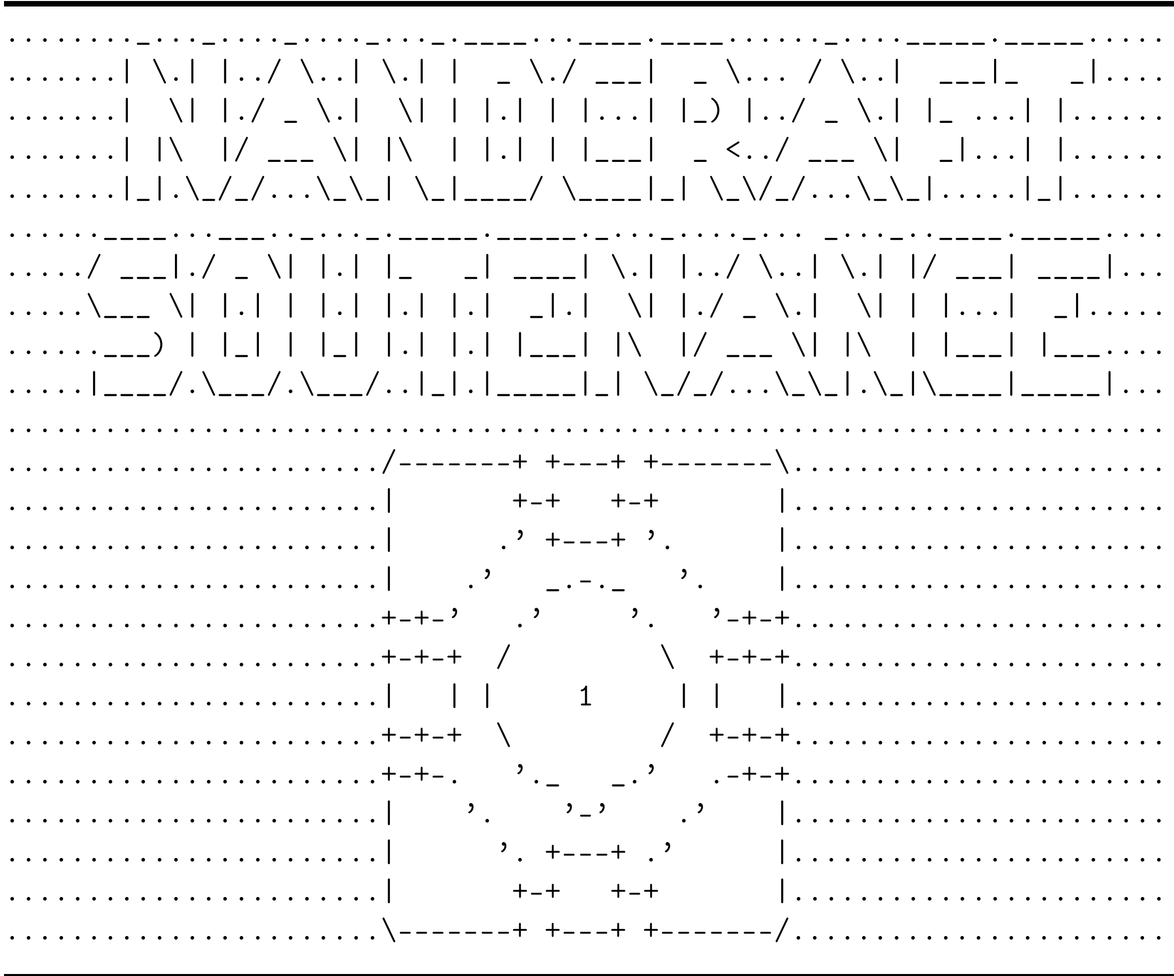
CPU - Cube Processing Unit

audebe_r - halfr

hervot_p - Dettorer

eddequ_n - nass

pruvot_a - Chaf



INFO SUP EPITA - 2011/2012

Sommaire

1 Ré-Introduction 4
1.1 NANDCRAFT 4

2 Circuits logiques :Chaf: 5
2.1 Première étape, la recherche 5
2.2 L'implémentation de portes logiques 5
2.3 L'évaluation et les portes « 16 bits » 8
2.4 Pour la suite 8

3 Assembleur :nass: 9
3.1 Assembleur 9
3.1.1 Une base: les flux 9
3.1.2 La première passe 9
3.1.3 La seconde passe 10
3.2 L'émulateur 11
3.2.1 Le simulateur (fini) 11
3.2.2 L'écran (en cours) 12
3.2.3 Le clavier (en cours) 12

4 Machine virtuelle :Dettorer: 13
4.1 L'analyseur syntaxique (parser). 13
4.2 Le traducteur. 13
4.3 Les commandes supportées actuellement. 13
4.3.1 Les commandes d'accès mémoire 14
4.3.2 Les commandes arithmétiques 15
4.4 Ce qu'il reste à faire 15

5 Compilateur :halfr: 16
5.1 Analyseur lexical 16
5.2 Analyseur syntaxique avancé 17
5.3 Arbre syntaxique abstrait 18
5.3.1 Implémentation 18
5.3.2 Visualisation de l'AST 18
5.3.3 Module générique de graphes 19
5.3.4 Combinaison de Graph et de DotAstNode 19
5.4 Conclusion 20
5.4.1 Ce qui a été fait 20
5.4.2 Ce qu'il reste à faire 20

6 Annexes 22
6.1 #nandcraft 22
6.2 CRAFTBOT 22
6.2.1 Annonce des nouvelles révisions 22
6.2.2 Rappel au code 22

6.3 nandcraft.halfr.net 22
6.4 DATA <3 22

7 Référence des langages **23**
7.1 Syntaxe de Craft 23

8 Le mot de la fin **30**

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 :

```
                               Abstraction
+-------------------------+
| Langage de haut-niveau  |        ^
+-------------------------+        |
| Machine virtuelle       |        ^
+-------------------------+        |
| Assembleur              |        ^
+-------------------------+        |
| Circuits logiques       |        ^
+-------------------------+        |
| OCaml                   |        |
+-------------------------+
```

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

2 Circuits logiques :Chaf:

2.1 Première étape, la recherche

Bien que très complet, le livre sur lequel nous nous appuyons pour réaliser notre projet (The Elements Of Computing Systems) n'en reste pas moins évasif sur la théorie de la logique. Il a donc fallu effectuer de nombreuses recherches pour découvrir les ficelles de cette nouvelle connaissance. En effet, pour un néophyte il n'est pas évident de s'y retrouver dans toute la symbolique des portes logique ainsi que dans leurs équations respectives. Pour effectuer ces recherches et ainsi comprendre et assimiler la logique élémentaire, il n'a pas été très ardu de trouver de la documentation sur Internet. La phase de recherche n'est certainement pas la plus déplaisante mais elle garde tout de même un côté frustrant : on découvre énormément de choses dans un très faible laps de temps sans vraiment les comprendre. Il faut alors lire et relire les mêmes informations pour petit à petit les assimiler et commencer à voir un cheminement dans la réflexion particulière qui est celle de la logique. Qui plus est, cela demande beaucoup de temps (que nous n'avons, hélas, pas en quantité astronomique...).

2.2 L'implémentation de portes logiques

Dans le cadre de notre projet, il était impératif de penser les différentes portes logiques comme des assemblages de porte « Nand ». Pour effectuer ceci il a donc fallu déterminer les équations des différentes portes. Pour ceci nous avons choisi de créer des fonctions simples à l'aide d'Ocaml permettant de vérifier rapidement si l'équation de la porte que nous avons déterminé était correcte.

Voici quelques exemples des dites fonctions :

```
1  (*La porte \T1\guillemotleft  nand \T1\guillemotright  est conside comme le seule porte primaire, elle
2  lue donc l\T1\textquoteright information directement*)
3
4  let my_nand a b = match (a,b) with
5    |(1,1) -> 0
6    |_ -> 1;;
7
8  (*Les portes suivantes sont des assemblages de portes \T1\guillemotleft  nand \T1\guillemotright  ou
9  d\T1\textquoteright autre portes d implnt suivant la re prdente*)
10
11 let my_not a = my_nand a a;;
12
13 let my_and a b = my_not (my_nand a b);;
```

Après cette étape de vérification des équations des différentes, la vrai implémentation des portes pouvait commencer. En effet, il était nécessaire d'implémenter les portes de façon symbolique afin d'arriver, si le temps le permet à un éditeur de circuit parfaitement fonctionnel. En d'autres termes, la définition même des portes devait comprendre les différents liens entre les entrées et sorties intermédiaires de la porte. Pour arriver à ce résultat nous avons utilisé des types enregistrement pour stocker toutes les informations nécessaires (nom de la porte, entrées, sorties et sous-portes). Nous comprendrons mieux de quoi il retourne avec un exemple détaillé :

```
1  let my_or =
2    {
3      name = "or";
4      inputs = [| "a" |> 1 ; "b" |> 1 |];
5      outputs = [| "out" |> 1 |];
6      parts = [|(my_nandio [|"a" <@ "a"; "b" <@ "a"|] [|"out" @> "out1"|]);
7                 (my_nandio [|"a" <@ "b"; "b" <@ "b"|] [|"out" @> "out2"|]);
8                 (my_nandio [|"a" <@ "out1"; "b" <@ "out2"|] [|"out" @> "out"|]) |]
9    };;
```

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 :

```
1  type gate =
2    {
3      name:string;
4      inputs:string*int array;
5      outputs:string*int array;
6      parts:part array
7    } ;;
```

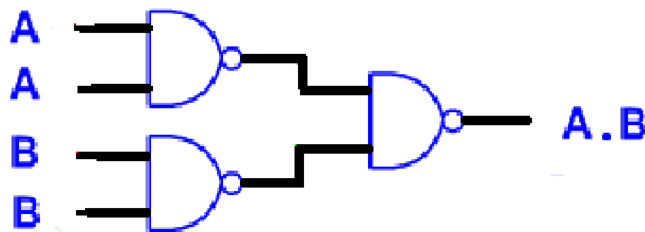
- Le type part est lui define par :

```
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 implémenté toutes les portes basiques dont nous nous servirons plus tard, à savoir les portes « not », « and », « or », « xor », « multiplexor » et « demultiplexor » La difficulté dans ce processus a été de penser les portes de façon symbolique et ainsi nous permettre d'avoir accès aux différents états des entrées et sorties à tout moment et de représenter clairement les portes comme combinaisons de portes « nand ».

2.3 L'évaluation et les portes « 16 bits »

Une fois les première portes implémentées, il était nécessaire de commencer à les évaluer, en effet, si pratiques qu'elles soient, ces portes ne produisent pour l'instant rien d'exploitable. Il a donc fallu trouver un moyen pour qu'on puisse enfin en tirer des informations. C'est là que l'évaluation prend toute sa dimension. Nous sommes maintenant capables d'évaluer les portes logiques présentées précédemment, autrement dit, nous pouvons obtenir la valeur de la sortie en passant en paramètre uniquement le nom de la porte et ses entrées. La dernière étape a été de modifier légèrement les définitions précédentes pour en tirer une définition propre des portes prenant en entrées et sorties des entiers constitués de 16 bits puis de réécrire une fonction pour les évaluer de la même manière que pour les premières portes.

2.4 Pour la suite

Maintenant que nous en sommes arrivé à une gamme plutôt complète de portes logiques fonctionnelles il nous faut créer une Unité Arithmétique et Logique (UAL) pour effectuer un certain nombre d'opérations arithmétiques. Nous construirons ici une UAL composée d'un demi-additionneur (permettant l'addition de deux bits), d'un additionneur (permettant l'addition de trois bits) et d'un additionneur complet (permettant l'addition de deux entiers de n-bits). Si nous avançons toujours aussi vite que pour la première partie nous serons peut-être même en mesure de commencer notre éditeur de circuits logiques.

3 Assembleur :nass:

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.

3.1 Assembleur

Notre assembleur permet l'utilisation d'une table de symbole 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 « 0 » et de « 1 » compréhensible par la machine. 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 la réalisation de nombreux éssais plus ou moins fructueux de chacune des deux passes.

3.1.1 Une base: 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 (« 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 seront abordées dans les chapitres suivants.

- Des améliorations nécessaires
  Le « program counter » (appelé « pointeur d'instruction » en français) correspond au registre qui contient l'adresse mémoire de l'instruction en cours d'exécution (ou la suivante) par le processeur. Ce registre est actuellement dans un système décimal ce qui n'est pas optimal pour une machine qui fonctionne sur du 16 bits. Il serait donc préférable d'implémenter un système hexadécimal pour la valeur numérique de ce registre. Cette amélioration sera implémentée pour la prochaine version de NANDCRAFT.

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 »

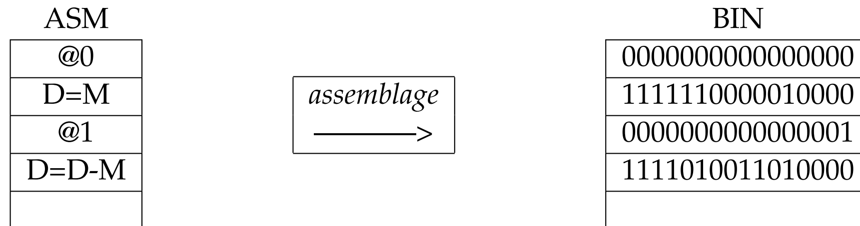
- premier bit: 0
- 15-bits restants: valeur en système binaire du nombre décimal représenté
- par la A-instruction.

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 initialement prévu pour la deuxième soutenance est en cours de réalisation. 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.

3.2.1 Le simulateur (fini)

La partie qui gère la simulation de l'exécution du code assembleur est terminée. Afin de vérifier le bon fonctionnement de l'émulateur, un executable de test donne sur la sortie standard le contenu des différents registres et de la RAM à l'adresse voulue au fur et à mesure de l'exécution des instructions.

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
a = 0, d = 0, m = 0
a = 0, d = 0, m = 0
a = 1, d = 0, m = 0
a = 1, d = 0, m = 0
a = 10, d = 0, m = 0
a = 10, d = 0, m = 0
a = 1, d = 0, m = 0
a = 1, d = 0, m = 0
a = 12, d = 0, m = 0
a = 12, d = 0, m = 0
a = 2, d = 0, m = 0
a = 2, d = 0, m = 0
```

-un saut conditionnel est effectué vers l'adresse voulue

```
a = 14, d = 0, m = 0
a = 14, d = 0, m = 0
a = 14, d = 0, m = 0
a = 14, d = 0, m = 0
a = 14, d = 0, m = 0
```

-une boucle infinie est utilisée pour éviter un « segfault »
.. suite de la boucle

3.2.2 L'écran (en cours)

La partie écran dont s'occupe l'émulateur fonctionne sur une résolution de 256 par 512 en noir et blanc. Nous utilisons ocamlSDL afin de pouvoir attribuer une couleur pixel par pixel à l'aide de la fonction set_pixel_color qui matche sur la valeur de chacun des bits. SDL prend ses valeurs directement dans la RAM

3.2.3 Le clavier (en cours)

L'action de taper une touche modifie le contenu de la mémoire à une adresse prédéfinie. Le nouveaux contenu correspond à la valeur correspondant dans le système ASCII ou, pour les caractères spéciaux, des codes que l'on redéfinit.

4 Machine virtuelle :Dettorer:

The Elements Of Computing Systems, le livre duquel le projet NANDCRAFT s'inspire, propose une implémentation de la machine virtuelle en deux sous-modules. Cependant, l'utilisation des flux en ocaml m'a permis d'imaginer le fonctionnement du traducteur d'une autre façon. Car oui, la machine virtuelle est avant tout le traducteur d'un langage basé sur une pile de données et l'accès à des segments de mémoire dont les rôles sont bien définis, vers un langage assembleur plus bas niveau, ne supportant que les accès à un emplacement précis de la RAM et les sauts de code (jump).

4.1 L'analyseur syntaxique (parser).

La machine virtuelle doit donc traduire un ensemble de fichiers écris dans un langage vers un seul fichier écris dans un autre langage. Ces fichiers n'étant que des suites de caractères, il faut d'abord les transformer en entités plus simples à manipuler. C'est le travail de l'analyseur syntaxique, que l'on divise généralement en deux parties.

La première partie, aussi appelée analyseur lexical (lexer), se charge d'enlever les commentaires, les espaces et les retours à la ligne. Il transforme alors le texte en suite de mots (un type « lexeme » que l'on aura défini de façon simple). En terme de flux ocaml, l'analyseur lexicale traduit un flux de caractères (*char Stream.t*) en flux de mots (*lexeme Stream.t*).

Le reste de l'analyseur syntaxique va alors observer l'agencement de ces mots afin de déterminer ce qu'ils veulent dire. Par exemple le mot « push » suivis du mot « constant » suivis du mot « 10 » sera interprêté comme la commande demandant d'ajouter le nombre 10 en haut de la pile. Cette partie, en terme de flux ocaml, traduit le précédent flux de mots en flux de commandes (*commandes Stream.t*), un type plus complexe que l'on aura défini.

4.2 Le traducteur.

Mon choix pour l'analyseur syntaxique s'est donc porté sur une méthode très linéaire, les flux permettant de lire un fichier au fur et à mesure de sa traduction, tout se fait en un seul passage. Ce n'est pas différent pour ce qui est de la traduction effective.

Le traducteur est le programme principal de la machine virtuelle, c'est lui qui reçoit sur l'entrée standard les fichier écris en langage VM (le langage de la machine virtuelle) et affiche le code assembleur traduit sur la sortie standard. Ce mode de fonctionnement nous permettra de relier les différents modules du projet très simplement, en redirigeant les entrées et sorties. Pour ce faire, le traducteur transforme les fichiers qu'il reçoit en entrée en flux de caractère (une simple commande fournit par camlp4), applique le parser à ce flux et enfin écrit sur la sortie standard la traduction en assembleur de chaque commande. /newpage{

4.3 Les commandes supportées actuellement.

Le gros du développement de la machine virtuelle est de trouver une traduction à chaque fonction, celles-ci sont divisées en trois parties dont deux sont actuellement supportées par la machine virtuelle.

4.3.1 Les commandes d'accès mémoire

Au nombre de deux, ces commandes permettent d'ajouter une valeur ,contenue dans un segment, sur le dessus de la pile et à l'inverse de retirer une valeur du dessus de la pile pour la placer quelque part dans un segment.

Dans le langage de la machine virtuelle (le bytecode), la ram n'est pas donnée comme un simple grand tableau, les valeurs qui ne sont pas dans la pile sont stockées dans des segments de mémoire (les constantes, les arguments, les variables locales etc. . .). Ces segments ainsi que la pile n'existent pas explicitement dans notre langage assembleur, leur base est représentée par un pointeur dont on connait l'adresse (dans le cas de la pile, le pointeur référence le haut de la pile) et leur taille n'est connue que de la machine virtuelle, qui se chargera de lever une erreur au premier dépassement.

La commande *push* <*segment*> <*index*> copie la valeur contenue à l'index donné du segment donné en haut de la pile. Du point de vue de l'assembleur, il faut récupérer la valeur choisie en passant par le pointeur associé au segment, copier cette valeur à l'adresse référencée par le pointeur de la pile (SP) et enfin incrémenter ce même pointeur (figure 3.1). La commande *pop* <*segment*> <*index*> est l'opération inverse. Elle récupère la valeur en haut de la pile (en assembleur on décrémentera le pointeur de la pile) et la place à l'index donné du segment donné.

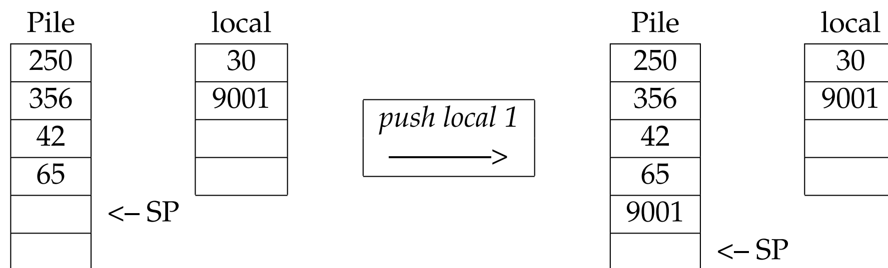


Figure 3.1 Exemple de commande push

4.3.2 Les commandes arithmétiques

Ces commandes s'appliquent aux éléments en haut de la pile, elles permettent de faire les additions, soustractions, négations et les opérations logiques *et*, *ou*, *non*, *ou exclusif*.

Les additions, soustractions, négations et les opérations logiques sont gérées par l'ALU au niveau des circuits logiques, il suffit donc de récupérer les valeurs du haut de la pile (une seule valeur pour la négation et le non logique) et de traduire l'opérateur (on n'oubliera pas de décrémenter le pointeur de pile si l'opération a deux opérandes).

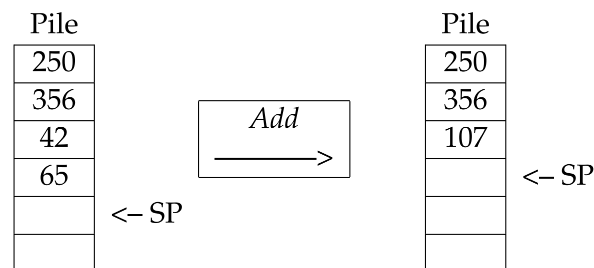


Figure 3.2 Exemple de commande arithmétique (Add).

4.4 Ce qu'il reste à faire

La prochaine étape est de traduire des commandes d'un tout autre ordre : le contrôle de flux d'exécution. Ce sont ces commandes qui permettront l'implémentation de bloc conditionnels, boucles (*via goto*, *ifgoto* et ses labels) ainsi que les définitions et appels de fonction.

Le traducteur sera alors complet, il sera temps de créer un émulateur. Celui-ci illustrera le fonctionnement de la machine virtuelle et permettra de l'observer pas à pas via une interface simple. L'émulateur s'accompagnera d'un interpréteur pouvant directement exécuter du bytecode et en afficher le résultat.

J'ai commencé au tout début du projet à réaliser cet interpréteur, afin de mieux comprendre la machine virtuelle, celui-ci n'est pas encore terminé mais supporte la quasi-totalité des fonctions arithmétiques. Puisque l'émulateur n'est pas encore une priorité, j'ai préféré ne pas aprofondir pour le moment et me concentrer sur les objectifs du cahier des charges. C'est maintenant chose faite, et ce début d'interpéteur nous procure une avance non négligeable sur nos prévisions.

5 Compilateur :halfr:

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 :

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

on peut écrire :

```
1 exp:
2   | n=NUM { n }
3   | 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)

```
1  statement:
2    | s=letstatement    { s }
3    | s=ifstatement     { s }
4    | s=whilestatement  { s }
5    | s=dostatement     { s }
6    | s=returnstatement { s }
7
8  letstatement:
9    | LET name=varname array=arrayaccess? EQUAL e=expression SEMICOLON
10   {
11     Ast.LetStatement (name, array, e)
12   }
13
14 ifstatement:
15   | IF pred=delimited(LPAREN, expression, RPAREN)
16     thens=delimited(LBRACE, statement*, RBRACE)
17     elses=option(preceded(ELSE, delimited(LBRACE, statement*, RBRACE)))
18     {
19       Ast.IfStatement (pred, thens, elses)
20     }
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` [1] pour produire des visualisations sous forme de graphe de l'AST. Il n'existe pas de bibliothèque en OCaml pour générer ces graphes[2] 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.

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

[2] 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.

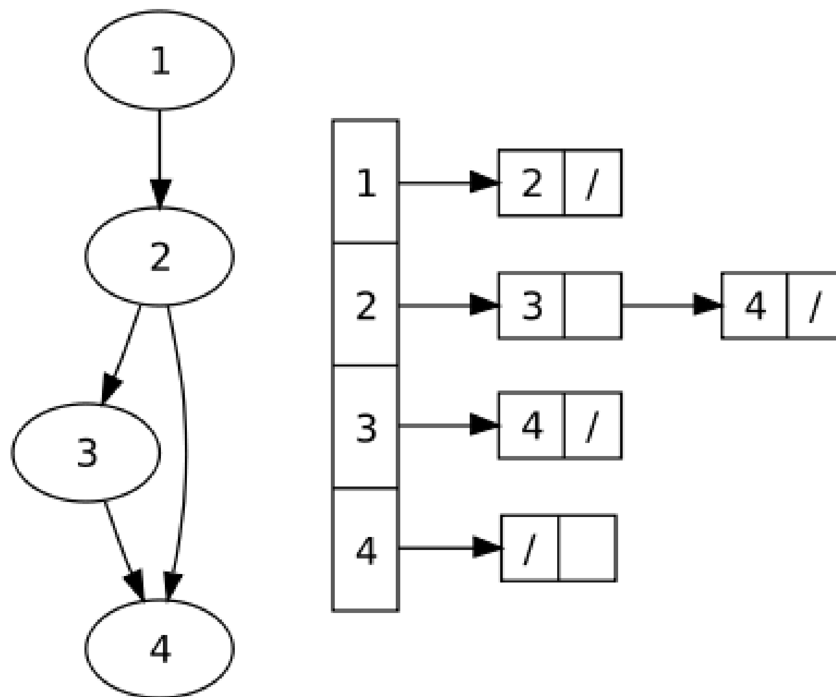


Figure 1: Illustration de la liste d'adjacence

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

5.3.4 Combinaison de Graph et de DotAstNode

Voici un exemple de fichier source dans le langage `Craft` :

```
1  class Class {
2          static int foo;
3          field char a;
4
5          constructor Class main(int a)
```

```
	{
		var char bar;

		let i = 0;

		if (true & true) {
		  do smthg();
		}
		else { }

		while (i < 10) {
			let i = i + 1;
		}
	}
}
```

Puis l'AST généré à partir de ce code :

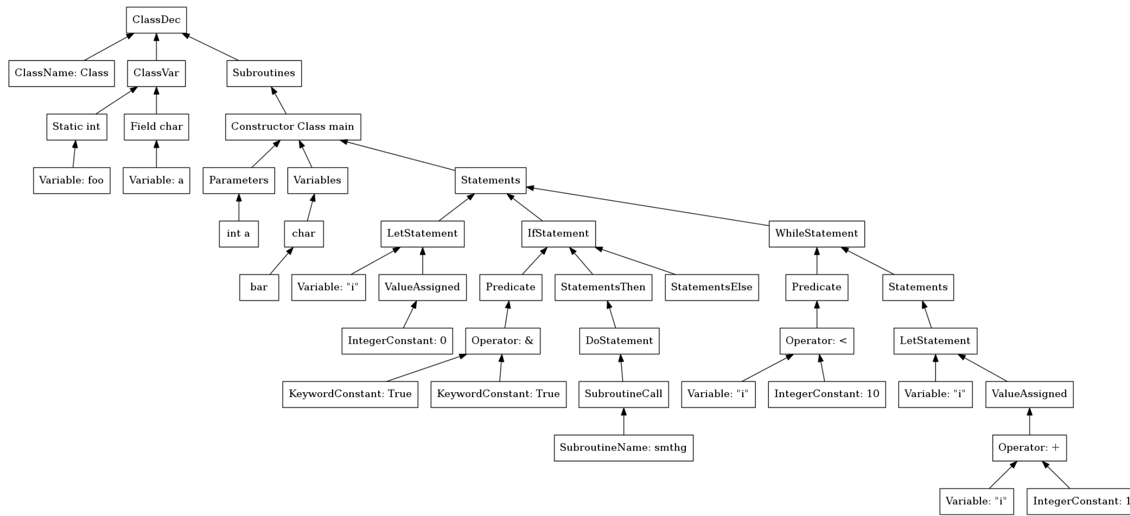


Figure 2: AST du fichier main.craft

5.4 Conclusion

5.4.1 Ce qui a été fait

En accord avec le cahier des charges l'analyseur syntaxique a été écrit et afficher sous forme d'un graphe.

L'écriture d'un module générique de graphes paramétrés nous met en avance sur les prévisions pour la soutenance 2. En effet les algorithmes de parcours et d'annotation de l'AST ont été écrit pour cette soutenance alors qu'il ne sont vraiment nécessaire que pour la génération de code, planifié pour la deuxième soutenance.

5.4.2 Ce qu'il reste à faire

Maintenant que nous disposons d'un structure de donnée adéquate ainsi que des primitives pour la manipuler pour représenter le langage `Craft` nous pouvons commencer à effectuer

des opérations plus complexes en utilisant le système d'annotation de l'AST.
Il reste donc à faire :

- Vérification d'existence des noms de variables et de fonctions
- Annotation et vérification de types
- Génération de code

6 Annexes

6.1 #nandcraft

Pour discuter de l'avancement du projet nous avons enregistré un canal IRC[3] sur le réseau rezosup.

Il est plutôt actif et fréquenté en moyenne par 8 personnes. Il compte plus de 6000 messages envoyés.

6.2 CRAFTBOT

Nous avons aussi écrit un bot (comprendre un programme qui effectue un ensemble de tâches automatisée) pour ce cannal irc. Il se nomme CRAFTBOT et est écrit en zsh[4]. Il peut effectuer deux types de script : les évènements périodiques et les évènements déclenchés par le message d'un utilisateur.

6.2.1 Annonce des nouvelles révisions

Lorsque l'un de nous envoie ses modifications sur le serveur distant, CRAFTBOT effectue une annonce indiquant le nom des nouvelles révisions ainsi que les fichiers qui ont été modifiés.

```
CRAFTBOT | 316:76b0362910d0 Add a global Makefile
CRAFTBOT | Makefile |  13 +++++++++++++
CRAFTBOT | 1 files changed, 13 insertions(+), 0 deletions(-)
```

6.2.2 Rappel au code

De temps en temps, CRAFTBOT va envoyer un message à un utilisateur aléatoire du canal pour lui rappeler qu'il faut travailler.

```
CRAFTBOT | nass: Go coder !!!
```

6.3 nandcraft.halfr.net

Nous avons réalisé un site web à l'aide de sphinx.

6.4 DATA <3

Il y a actuellement 515 révisions sur le dépot mercurial.

Nous totalisons 5296 lignes de code (tout type de sources confondue, ceci inclus donc entre autre le code des rapports) dans le projet.

[3] Internet Relay Chat, un protocole de communication sur internet très populaire
[4] Site web : http://www.zsh.org/ Zsh est un langage de script

7 Référence des langages

7.1 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[5] équivalente.

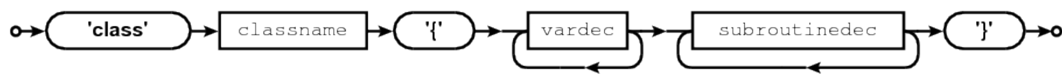


Figure 3: class

class = 'class' , classname , '{' , { vardec } , { subroutinedec } , '}';

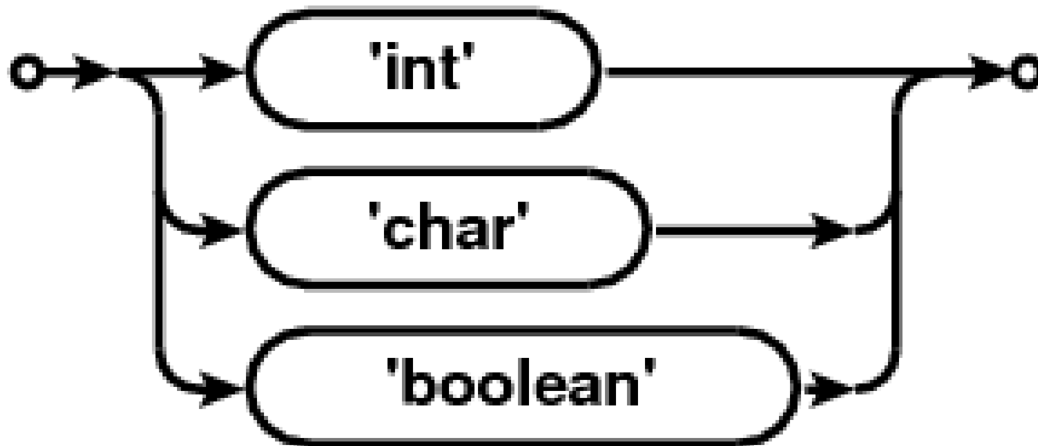


Figure 4: simpletype

simpletype = 'int' | 'char' | 'boolean';

advancedtype = simpletype | classname;

[5] Extended Backus-Naur Form

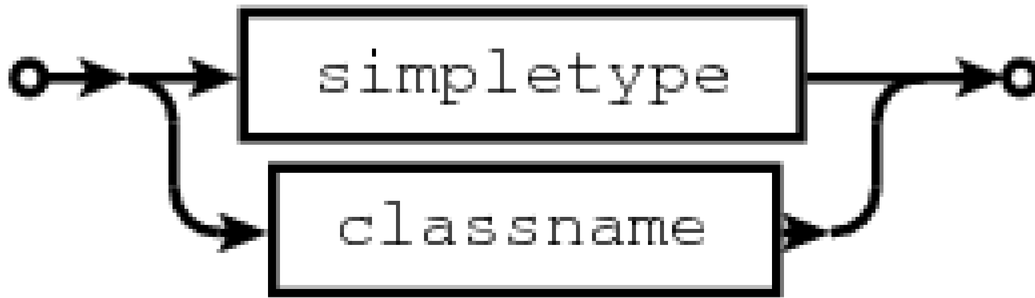


Figure 5: advancedtype

Figure 6: vardec

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

Figure 7: classvartype

classvartype = ‘static’ | ‘field’;

vartype = advancedtype;

classname = IDENTIFIER;

varname = IDENTIFIER;

Figure 8: vartype

Figure 9: classname

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

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

subroutinetype = advancedtype | 'void';

subroutinename = IDENTIFIER;

Figure 10: varname

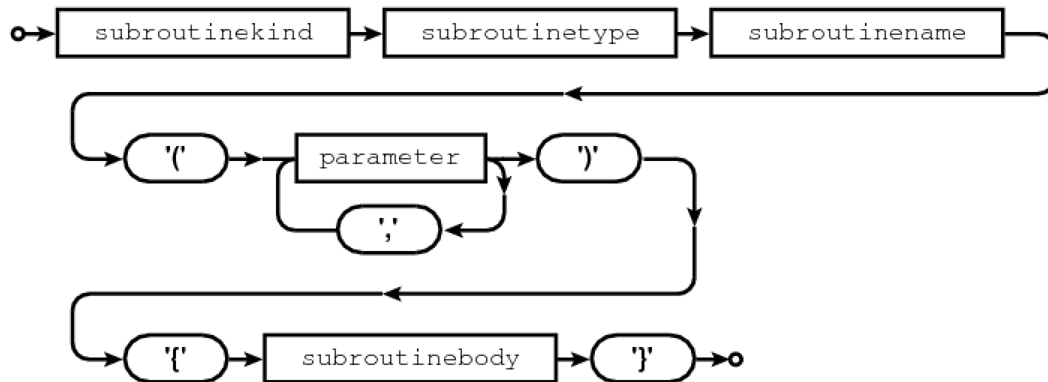


Figure 11: subroutinedec

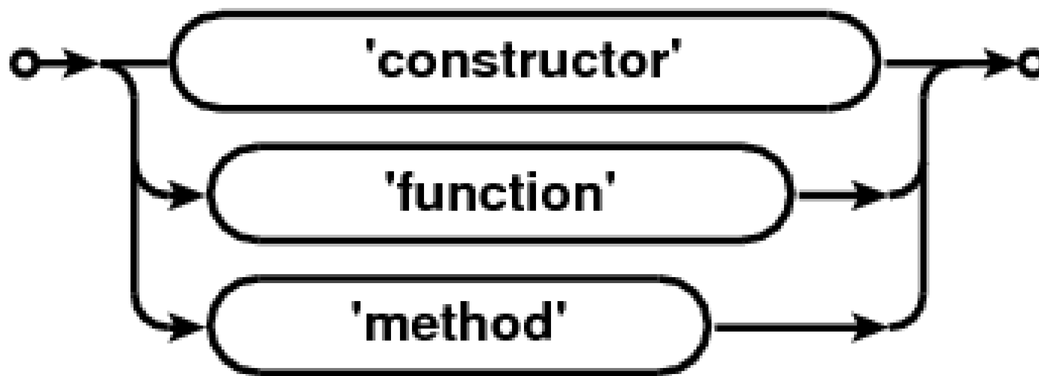


Figure 12: subroutinekind

parameter = advancetype , varname;

subroutinebody = { subroutinevar } , { statement };

subroutinevar = 'var' , varname , { ',' , varname };

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

letstatement = 'let' , varname , [arrayaccess] , '=' , expression , ';' ;

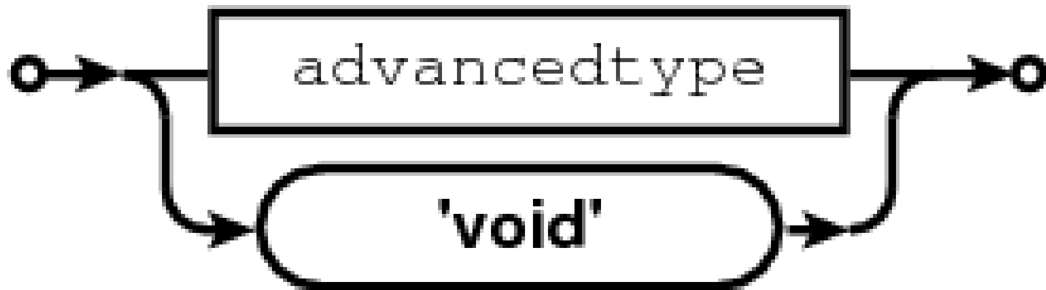


Figure 13: subroutinetype

Figure 14: subroutinename

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

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

dostatement = ‘do’ , subroutinecall;

returnstatement = ‘return’ , [expression];

Figure 15: parameter

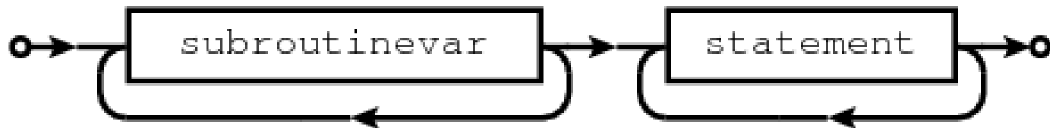


Figure 16: subroutinebody

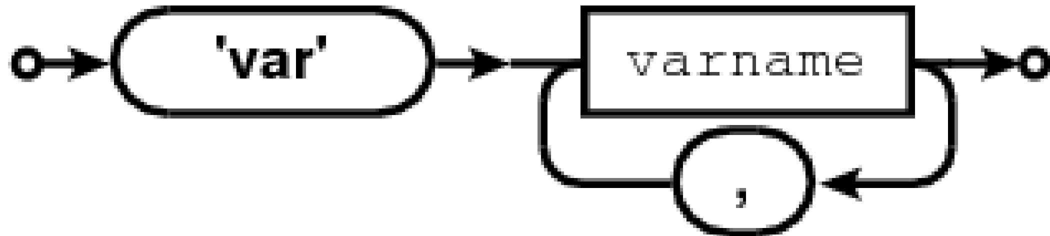


Figure 17: subroutinevar

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

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

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

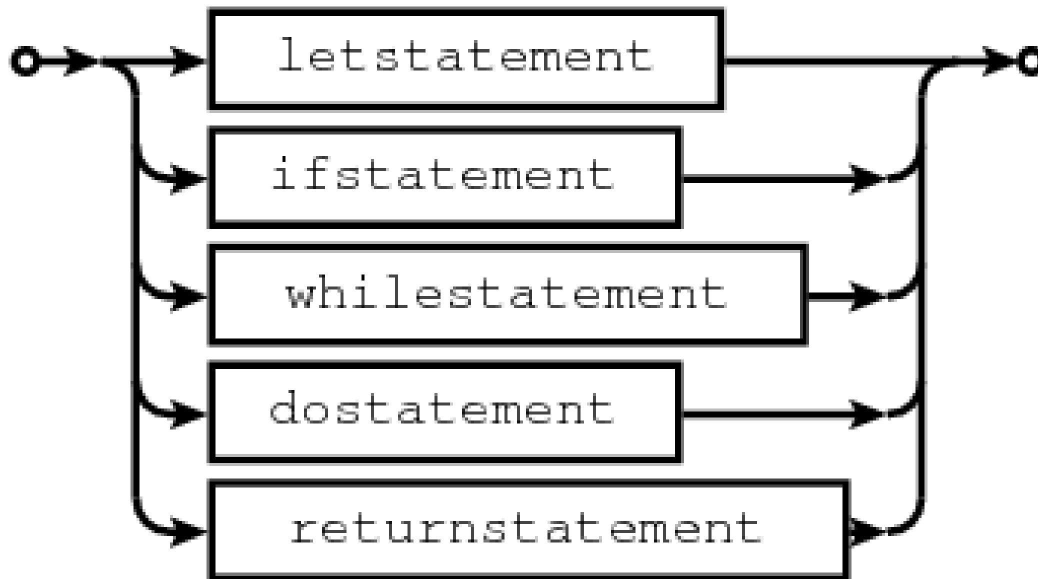


Figure 18: statement

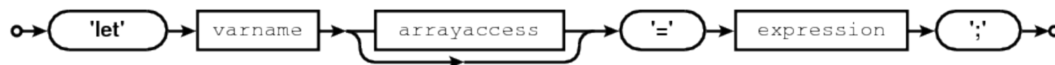


Figure 19: letstatement

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

arrayaccess = ‘[’ , expression , ‘]’;

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

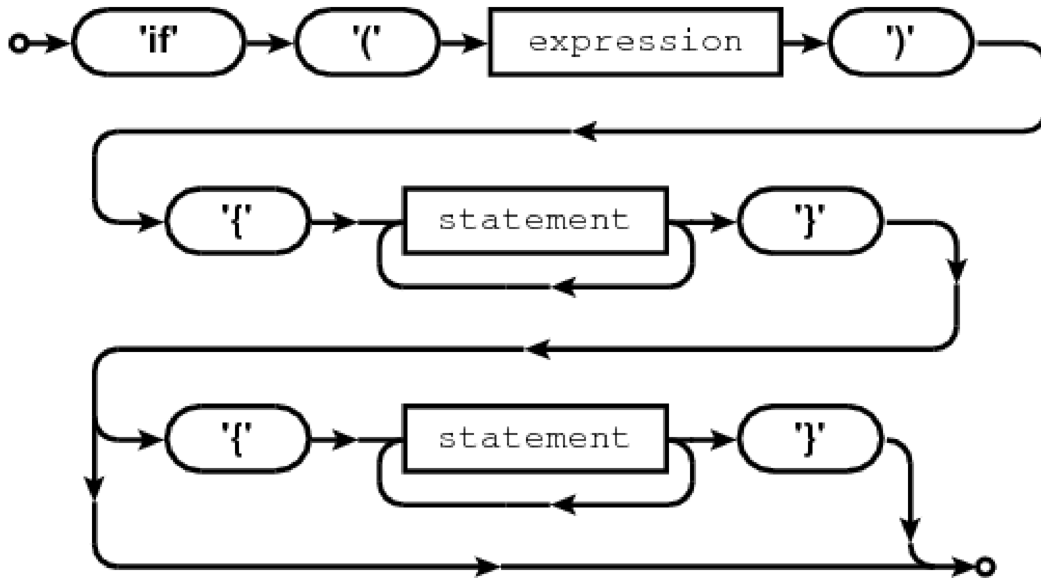


Figure 20: ifstatement

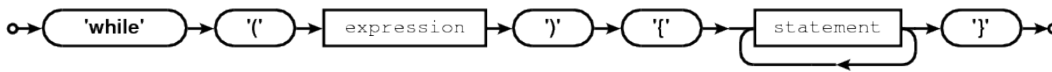


Figure 21: whilestatement

8 Le mot de la fin

```
tl;dr On est toujours très heureux.
Étrangement nous n'avons rien bullshité pour cette soutenance
```

Figure 22: dostatement

Figure 23: returnstatement

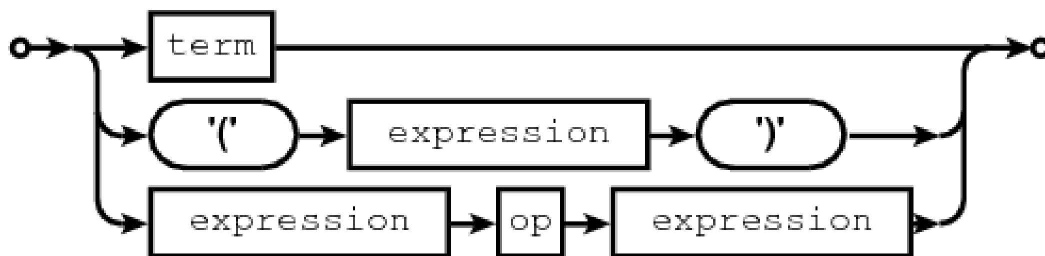


Figure 24: expression

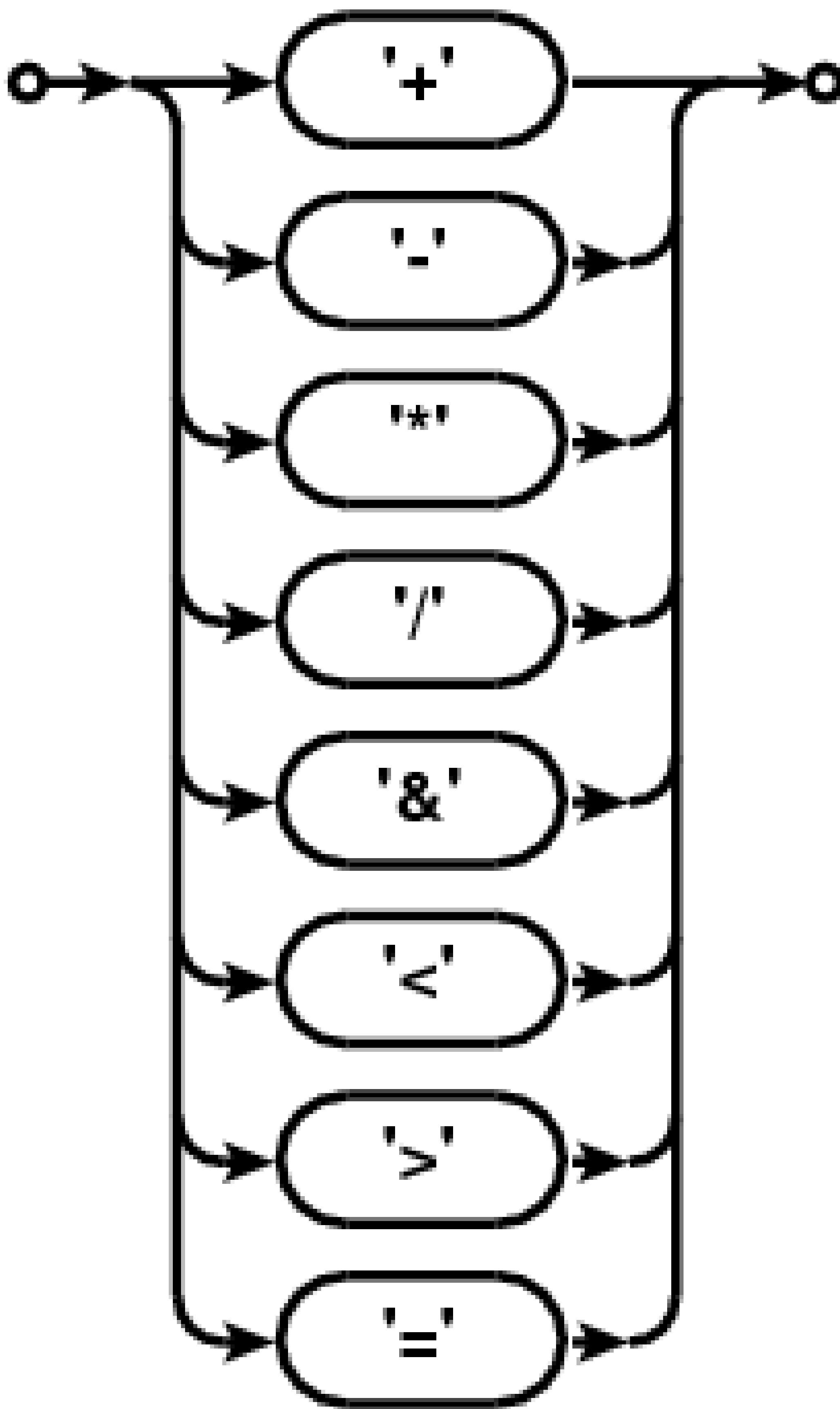


Figure 25: op

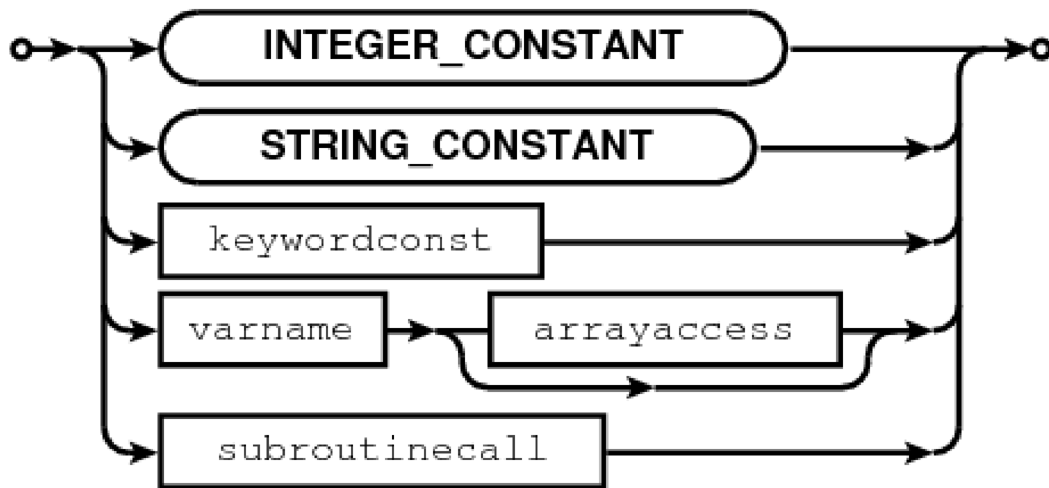


Figure 26: term

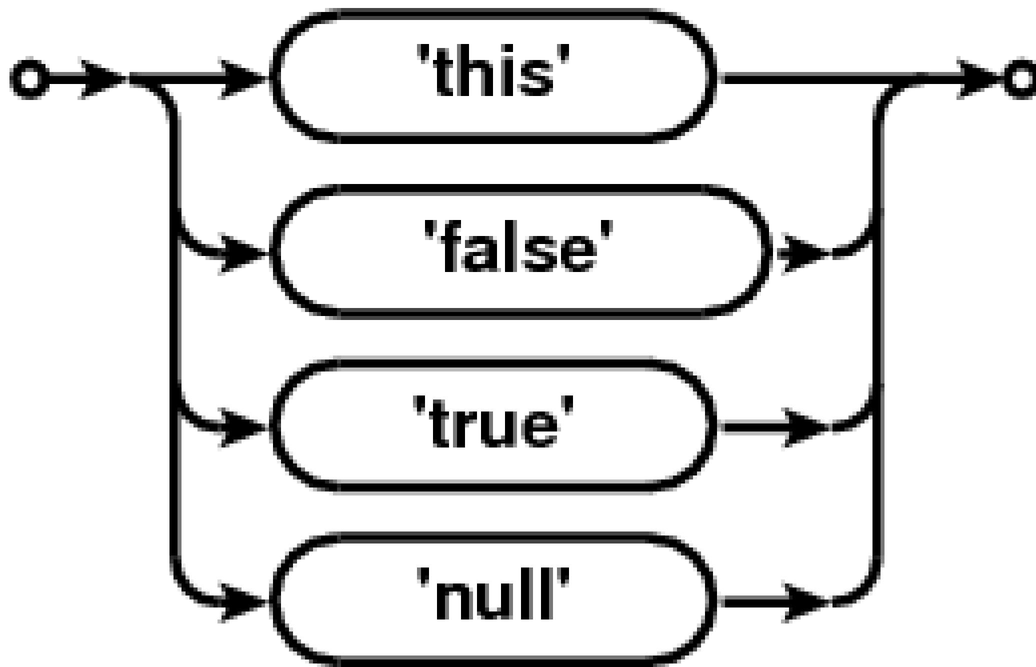


Figure 27: keywordconst

Figure 28: arrayaccess

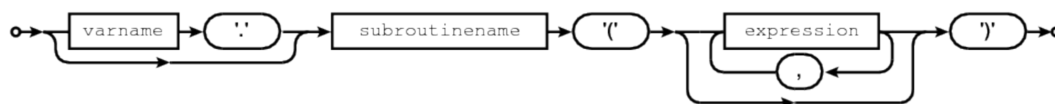


Figure 29: subroutinecall