

Compiler pour une petite machine réelle

CHIP-8, registres, appels et sauts

CHIP-8

allocation de registres

appel terminal

Pourquoi un compilateur passe plus de temps à décider où ranger les valeurs qu'à choisir ses instructions, et ce qu'une console de 1977 nous apprend sur LLVM et la BEAM.

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Backend CHIP-8 » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Le projet « backend CHIP-8 » reprend les arbres d'AP2 et d'AP3 et les traduit pour une machine qui existe vraiment : la console virtuelle CHIP-8, dont les programmes s'ouvrent dans un émulateur du navigateur. Vous y compilez des calculs, des liaisons, des choix et trois primitives d'écran et de clavier, jusqu'à afficher un niveau de Sokoban ; le palier avancé ajoute les appels de fonctions et transforme l'auto-appel terminal de la boucle de jeu en un saut.

Ce fascicule ne contient aucune solution du projet. Il raconte d'où vient cette machine, puis ce que fait la partie d'un compilateur qu'on appelle le *backend* : choisir les instructions, décider quelle valeur vit dans quel registre, organiser les appels et leurs retours, et polir le code produit. Les mêmes questions se posent, à une autre échelle, dans LLVM, dans les moteurs WebAssembly et dans le compilateur à la volée de la BEAM, et c'est là que nous finirons. Lisez-le un soir, par curiosité ; rien n'est à rendre.

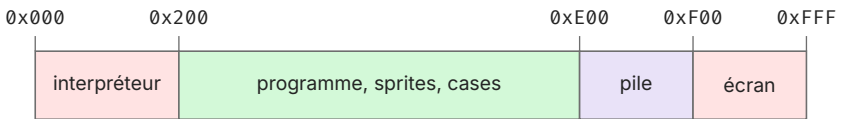
2 Une console de 1977 dans votre navigateur

CHIP-8 n'est pas un processeur. C'est un *langage interprété*, conçu en 1977 par Joseph Weisbecker, ingénieur chez RCA, pour le COSMAC VIP, un ordinateur en kit bâti autour du microprocesseur CDP1802. Le VIP n'avait que quelques kilo-octets de mémoire et un pavé de seize touches ; écrire un jeu directement en code machine 1802 était long et pénible. Weisbecker y a donc logé un petit interpréteur, cinq cent douze octets, qui lit des instructions de deux octets pensées pour les jeux : dessiner un sprite, lire une touche, tirer un nombre au hasard, décompter un délai. Il en a fait la publicité dans BYTE en décembre 1978, sous le titre « An Easy Programming System ».

L'idée a survécu à la machine. Dans les années 1990, elle a été portée sur la calculatrice HP-48, avec CHIP-48 d'Andreas Gustafsson en 1990 puis SUPER-CHIP d'Erik Bryntse en 1991, qui a ajouté un écran de 128 sur 64. En 1997, Thomas P. Greene a rédigé, sous le pseudonyme Cowgod, la référence technique que la plupart des émulateurs suivent encore. Depuis 2014, l'outil Octo de John Earnest et sa game jam annuelle lui ont donné une seconde vie : écrire un émulateur CHIP-8 est devenu l'exercice classique de celui qui veut comprendre une machine, et écrire un compilateur pour elle, l'exercice de celui qui veut comprendre un backend.

2.1 Ce que la machine offre, et ce qu'elle refuse

Le contrat tient en quelques lignes, et c'est ce qui rend la machine si agréable à cibler. Seize registres d'un octet, $v0$ à VF , dont le dernier sert aussi de *drapeau* : il reçoit la retenue d'une addition, l'absence d'emprunt d'une soustraction, la collision d'un dessin. Un registre d'adresse I , de seize bits, qui désigne des données en mémoire. Un compteur ordinal PC , une pile de retours, deux minuteurs qui décomptent à soixante hertz. Quatre kilo-octets de mémoire, trente-cinq instructions de deux octets, un écran monochrome de 64 sur 32 pixels et un pavé hexadécimal.



512 octets réservés : nos ROM commencent à $0x200$, là où le PC démarre.

Le projet borne code, sprites et cases à 3 Ko, avant $0xE00$.

Sur le VIP, la pile de retours vit entre $0xEA0$ et $0xEFF$, l'image entre $0xF00$ et $0xFFF$.

Les 4 096 octets de CHIP-8, largeurs non proportionnelles. Les cinq cent douze premiers octets abritaient l'interpréteur sur le VIP ; les émulateurs modernes y rangent une police de chiffres hexadécimaux.

Deux choses manquent, et elles vont nous occuper tout le fascicule. Il n'y a aucune pile de calcul : contrairement à notre machine d'AP3, aucune instruction ne dit « empile » ou « dépile », et toute valeur intermédiaire doit vivre dans un registre ou dans une case mémoire que vous aurez choisie. Il n'y a pas non plus d'instruction qui lise la mémoire depuis un registre général : tout passe par I , que l'on charge avec une adresse, et que l'on peut au mieux décaler d'un registre ; $LD [I], v0$ et $LD v0, [I]$ recopient ensuite entre $v0$ et la case désignée. Une variable du langage source devient donc, au mieux, un registre, et plus souvent une case et deux instructions.

Chaque instruction tient sur deux octets, et ses champs se lisent au demi-octet près, exactement avec le pattern matching binaire d'AP5. $6XNN$ charge la constante NN dans VX ; $2NNN$ appelle le sous-programme à l'adresse NNN :

```
1 <<op::4, x::4, nn::8>> = <<0x60, 0x2A>>
2 IO.inspect({op, x, nn})
3 IO.puts(Base.encode16(<<0x2::4, 0x24A::12>>))
```

CONSOLE {6, 0, 42} puis 224A

Le premier motif décode $LD v0, 42$, l'instruction que le squelette produit pour le littéral 42 ; le second encode $CALL 0x24A$. L'assembleur fourni dans le projet ne fait

rien d'autre, instruction par instruction, après avoir résolu les étiquettes en deux passes comme celui d'AP3.

2.2 Dessiner, c'est faire un ou exclusif

L'écran se commande par une seule instruction, `DXYN` : dessiner `N` lignes de huit pixels lues à partir de `I`, aux coordonnées `(VX, VY)`. Chaque pixel du sprite est combiné avec le pixel déjà affiché par un *ou exclusif* (XOR) : allumé sur allumé donne éteint, et `VF` passe à un si au moins un pixel s'est éteint. Ce choix de 1977 n'est pas une bizarrerie : il permet d'effacer un sprite en le redessinant au même endroit, sans mémoriser ce qu'il recouvrait, et le drapeau offre une détection de collision gratuite. Sur une ligne de huit pixels, le calcul se refait en Elixir :

```
1 import Bitwise
2 ligne = 0b1111_0000
3 sprite = 0b1010_0000
4 une_fois = bxor(ligne, sprite)
5 deux_fois = bxor(une_fois, sprite)
6 IO.inspect(for <<bit::1 <- <<une_fois>> >>, do: bit)
7 IO.inspect(for <<bit::1 <- <<deux_fois>> >>, do: bit)
```

CONSOLE [0, 1, 0, 1, 0, 0, 0, 0] puis [1, 1, 1, 1, 0, 0, 0, 0]

Le second dessin restitue exactement la ligne de départ. C'est aussi pour cela que le projet fournit une tuile par combinaison visible, caisse sur cible ou joueur sur cible : superposer deux sprites éteindrait leurs pixels communs.

PIÈGE

Les deux octets d'une instruction sont lus en *big endian*, poids fort d'abord, comme en AP3, et le PC avance de deux. Mais `I` désigne des octets, et un sprite de quatre lignes occupe quatre octets consécutifs. Confondre l'adresse d'une instruction et celle d'une donnée produit une ROM qui dessine du code : l'image est absurde, mais la machine ne proteste pas.

3 Ce qu'un backend décide

Un compilateur se découpe classiquement en deux moitiés. Le *frontend* lit le texte, construit l'arbre et le vérifie ; c'est AP2, et le mini-typeur d'AP9 en est le prolonge-

ment. Le *backend* part de l'arbre, ou d'une représentation intermédiaire plus proche de la machine, et produit les instructions d'une cible précise. En AP3, le backend était presque transparent : une machine à pile accepte un parcours postordre de l'arbre sans autre décision, parce que la pile conserve les valeurs intermédiaires à notre place. C'est exactement ce que CHIP-8 ne fait pas, et c'est pourquoi le projet mérite son nom.

3.1 Choisir les instructions

La *sélection d'instructions* associe à chaque nœud de l'arbre une petite séquence d'instructions de la cible. Pour un littéral, un `LD VX, NN` suffit. Pour une addition, CHIP-8 offre `ADD VX, VY`, qui écrit la somme dans `VX` et détruit donc l'un des deux opérandes ; pour une comparaison `<`, il n'offre rien de direct, et il faut la fabriquer à partir d'une soustraction et du drapeau `VF`, ou d'un saut conditionnel. Un backend contient ainsi une table qui dit, motif par motif, ce que la machine sait faire et ce qu'il faut composer. Sur les processeurs réels, où une même opération a souvent dix encodages possibles, cette table devient un problème d'optimisation à part entière ; sur CHIP-8, elle tient sur une page.

3.2 Garder une valeur pendant qu'on en calcule une autre

Le vrai souci apparaît avec $40 - (8 + 3)$. Pour soustraire, il faut disposer des deux opérandes au même instant ; or calculer la droite après la gauche, avec une convention qui laisse chaque résultat dans `V0`, écrase la gauche. Il faut donc *préserver* une valeur pendant un calcul : dans un autre registre si l'on en a un de libre, en mémoire sinon. La question « combien de registres faut-il pour évaluer cet arbre ? » a une réponse précise depuis 1970 : Ravi Sethi et Jeffrey Ullman ont montré que, pour un arbre d'expression binaire, on obtient le minimum en évaluant d'abord le sous-arbre qui en demande le plus. Une feuille demande un registre ; un nœud dont les deux enfants demandent n en demande $n + 1$, car il faut garder le premier résultat pendant le second calcul ; un nœud dont les enfants demandent des nombres différents demande seulement le plus grand des deux, en commençant par ce côté-là. Pour $40 - (8 + 3)$, la gauche demande un registre et la droite deux : deux registres suffisent, à condition de calculer $8 + 3$ en premier.

À RETENIR

L'invariant du projet est le pendant de celui d'AP3. En AP3, le code d'une expression ajoutait exactement une valeur au sommet de la pile. Ici, le code d'une expression laisse sa valeur dans `v0` et peut détruire `v1` à `v3`, `VF` et `I`, mais rien d'autre. Un sous-arbre compilé est ainsi une boîte noire dont on connaît les effets, et c'est ce qui permet de composer les traductions sans relire le code produit.

Notre langage source respecte l'ordre d'évaluation gauche puis droite, et le projet vous demande de le conserver : réordonner les sous-arbres à la Sethi et Ullman n'est permis que lorsqu'aucun des deux n'a d'effet, ce qui n'est plus vrai dès que `read_key()` ou `tile(...)` entrent dans une expression. Un compilateur réel passe beaucoup de temps à prouver qu'un réordonnement ne change rien.

4 Quinze registres pour tout le monde

Un programme entier contient bien plus de valeurs qu'un arbre d'expression : chaque `let`, chaque temporaire, chaque paramètre. Les ranger dans les quinze registres utilisables, `v0` à `VE`, est le problème de l'*allocation de registres*, et c'est, avec la sélection d'instructions, le cœur d'un backend. Le projet le contourne en gardant chaque liaison dans une case mémoire ; voyons ce qu'il faudrait pour faire mieux.

4.1 Qui est vivant en même temps que qui ?

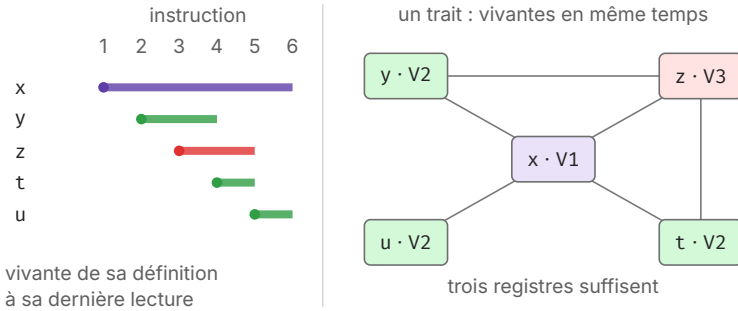
Deux valeurs peuvent partager un registre si elles ne sont jamais utiles au même moment. Une valeur est *vivante* entre l'instruction qui la définit et la dernière qui la lit ; calculer ces intervalles s'appelle l'*analyse de durée de vie*. Prenons ce programme, numéroté par liaison :

```

1 let x = 8           (1)
2 let y = 3           (2)
3 let z = 40          (3)
4 let t = x + y       (4)
5 let u = z - t       (5)
6 u + x               (6)
```

`x` vit de la ligne 1 à la ligne 6, `y` de 2 à 4, `z` de 3 à 5, `t` de 4 à 5 et `u` de 5 à 6. Deux valeurs *interfèrent* si leurs intervalles se chevauchent : `x` interfère avec toutes les

autres ; y et z interfèrent ; z et t aussi, puisque t naît à la ligne 4 alors que z est encore attendue à la ligne 5. En revanche t ne gêne pas y : y est lue pour la dernière fois dans l'instruction même qui définit t , et les deux peuvent occuper le même registre, l'une sortant quand l'autre entre.



À gauche, les durées de vie ; à droite, le graphe d'interférence et une coloration à trois couleurs. Deux nœuds reliés reçoivent des registres distincts ; y , t et u , jamais vivantes ensemble, se partagent $v2$.

4.2 Colorier un graphe

En 1981, Gregory Chaitin et ses collègues d'IBM, dont John Cocke, ont formulé le problème ainsi : les valeurs sont les sommets d'un graphe, les interférences ses arêtes, et allouer k registres, c'est colorier le graphe avec k couleurs sans que deux voisins partagent la leur. Le *coloriage de graphe* est un problème NP-complet en général, Richard Karp l'a montré en 1972, mais Chaitin a proposé une heuristique qui marche remarquablement : tant qu'il existe un sommet de moins de k voisins, on le retire du graphe, car on pourra toujours lui trouver une couleur une fois les autres fixées ; si l'on se retrouve bloqué, on choisit une valeur à *décharger* (*spill*) en mémoire, ce qui coupe ses arêtes, et l'on recommence. Les valeurs reviennent ensuite dans l'ordre inverse, chacune prenant une couleur libre parmi ses voisins déjà placés.

Sur notre exemple, le triangle x, y, z impose trois couleurs, et trois suffisent. Avec seulement deux registres, il faudrait décharger x , la plus gênante, en mémoire : chaque lecture de x coûterait alors un $\text{LD } I$ et un $\text{LD } V0, [I]$, et le code grossirait. Voici l'étape finale de l'heuristique, l'attribution gloutonne, écrite en quelques lignes :

```

1 defmodule Coloration do
2   # voisins : les valeurs vivantes en même temps que chacune
3   def registres(voisins, ordre) do
4     Enum.reduce(ordre, %{}, fn valeur, choix ->
5       pris = Enum.map(voisins[valeur], &choix[&1])
6       libre = Enum.find(1..15, &("V#{&1}" not in pris))
7       Map.put(choix, valeur, "V#{libre}")
8     end)
9   end
10 end
11
12 voisins = %{x: [:y, :z, :t, :u], y: [:x, :z],
13            z: [:x, :y, :t], t: [:x, :z], u: [:x]}
14 IO.inspect(Coloration.registres(voisins, [:x, :y, :z, :t, :u]))

```

```
CONSOLE  %{t: "V2", u: "V2", x: "V1", y: "V2", z: "V3"}
```

Tout l'art est dans l'ordre : l'heuristique de Chaitin le choisit en retirant d'abord les sommets peu connectés, pour que les sommets difficiles soient placés en dernier, quand on sait exactement quelles couleurs leurs voisins ont prises. Le projet, lui, décharge tout en mémoire, ce qui est toujours correct et jamais rapide ; réutiliser les cases mortes, l'optimisation suggérée par le README, est précisément une analyse de durée de vie appliquée aux cases plutôt qu'aux registres.

EN PRATIQUE

Les compilateurs à la volée, qui doivent produire du code en quelques millisecondes, n'ont pas le temps de construire un graphe. Massimiliano Poletto et Vivek Sarkar ont décrit en 1999 l'*allocation par balayage linéaire* : on trie les intervalles par début, on les parcourt une fois, et l'on décharge l'intervalle qui finit le plus tard quand les registres manquent. Moins bon que la coloration, beaucoup plus rapide ; c'est le compromis de nombreux compilateurs à la volée.

5 Appeler, revenir, sauter

Dès qu'un programme a des fonctions, le backend doit fixer une *convention d'appel* : où l'appelant dépose les arguments, où l'appelé laisse son résultat, quels registres chacun a le droit de détruire, et comment on retrouve son chemin après le retour.

Le projet répond : paramètres dans des cases dédiées à chaque fonction, résultat dans v_0 , registres v_1 à v_3 et i sans garantie. Les processeurs réels répondent de la même façon, avec plus de détails ; sur x86-64, la convention des systèmes Unix réserve six registres aux premiers arguments et un au résultat, et trie le reste en registres que l'appelé doit préserver et registres qu'il peut écraser.

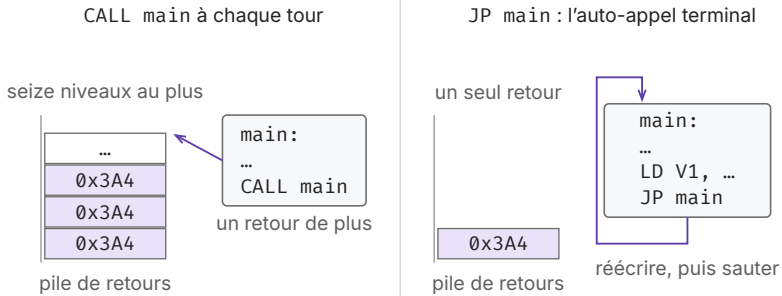
5.1 Une pile de retours de seize niveaux

CALL NNN empile l'adresse de l'instruction suivante, puis saute ; RET dépile cette adresse et y revient. La pile de CHIP-8 n'a que seize niveaux dans la référence de 1997, et l'interpréteur original du VIP n'en réservait que douze : un dix-septième appel imbriqué écrase ou refuse. C'est une pile de *retours*, pas une pile de calcul, et le projet garde les variables locales de chaque fonction dans des cases fixes. Ce choix est possible parce que le profil interdit la récursion non terminale et la récursion mutuelle : deux activations d'une même fonction ne coexistent jamais, et une seule copie de ses cases suffit.

Ce n'est pas une astuce de cours. Le premier FORTRAN, livré par l'équipe de John Backus chez IBM en 1957, allouait toutes ses variables statiquement et interdisait la récursion pour la même raison. C'est Edsger Dijkstra qui, en 1960, a décrit comment une pile de *cadres d'appel*, un bloc de mémoire par activation, permet à une fonction de se rappeler librement, et c'est le mécanisme qu'a adopté ALGOL 60, puis à peu près tout le monde. L'atelier facultatif d'AP3 vous l'a fait construire ; la BEAM le fait pour chaque processus.

5.2 L'appel terminal devient un saut

Reprenons la notion d'AP1 : un appel est *terminal* si rien ne reste à faire après son retour. Dans `sum(n - 1.0, total + n)`, le résultat de l'appel est directement celui de la fonction. Dans ce cas, empiler une adresse de retour est inutile : la fonction appelée pourrait revenir directement là où nous serions revenus. Pour un auto-appel, il suffit donc de réécrire les paramètres puis de sauter au début de la fonction avec JP, sans CALL. La pile ne grandit pas, et une boucle de jeu peut tourner indéfiniment avec le seul retour de son appel initial.



À gauche, chaque tour de `main` ajoute un retour, et la pile déborde au dix-septième. À droite, l'auto-appel terminal réécrit les paramètres et saute : la pile garde un seul retour, quel que soit le nombre de tours.

Guy Steele a défendu cette idée en 1977 dans un article au titre resté célèbre, « Lambda: The Ultimate GOTO » : un appel de fonction n'est qu'un saut qui transmet des arguments, et c'est la pile de retours, pas l'appel, qui coûte. Le langage Scheme, qu'il concevait alors avec Gerald Sussman, impose depuis cette optimisation à toute implémentation. La BEAM la pratique aussi : c'est elle qui permet à `loop(state)` d'AP8 de recevoir des messages sans fin avec une pile bornée, et Elixir compte sur vous pour écrire vos boucles ainsi.

```
1 defmodule Boucle do
2   def tours(0, total), do: total
3   def tours(n, total), do: tours(n - 1, total + n)
4 end
5 IO.inspect(Boucle.tours(1_000_000, 0))
```

CONSOLE 500000500000

Un million d'appels, aucun débordement : la BEAM a remplacé chaque appel terminal par un saut, exactement ce que votre `transfer/2` décide pour `main`.

PIÈGE

Réécrire les paramètres avant de sauter cache un problème classique, celui de l'*affectation parallèle*. Pour `loop(b, a, n - 1.0)`, le nouveau `a` est l'ancien `b` et le nouveau `b` est l'ancien `a` : écrire `a` en premier détruit la valeur dont `b` a besoin. Il faut calculer tous les arguments avant d'écrire le moindre paramètre, ou trouver un ordre d'écriture sans conflit, qui n'existe pas toujours. Les compilateurs réels construisent le graphe des dépendances entre les copies et cassent ses cycles avec un temporaire ; le projet choisit la solution simple, tout calculer d'abord.

6 Polir le code : les optimisations peephole

Une traduction nœud par nœud produit du code correct mais redondant, parce que chaque sous-arbre ignore ce que son voisin vient de faire. William McKeeman a proposé en 1965 l'*optimisation par trou de serrure (peephole)* : faire glisser une petite fenêtre, deux ou trois instructions, sur le code produit, et remplacer les motifs connus par un équivalent plus court. Voici le genre de séquence qu'une liaison suivie de sa lecture immédiate engendre :

```
1 LD I, case_x      ; I désigne la case de x
2 LD [I], V0        ; mémoire[I] prend V0
3 LD I, case_x      ; I n'a pas changé
4 LD V0, [I]        ; V0 reprend ce qu'il contenait déjà
```

Les deux dernières instructions ne changent rien : `V0` contient encore la valeur rangée, et l'on peut les supprimer. De même, un `JP suite` immédiatement suivi de l'étiquette `suite` : ne fait que perdre deux octets, et deux chargements consécutifs de `I` à la même adresse n'en valent qu'un. Une *propagation de constantes* va plus loin : `case True { ... }` peut se réduire à sa première branche, et `8 + 3` à `11`, sans produire aucune instruction.

La règle d'or est que chaque remplacement doit préserver le sens dans *tous* les cas, pas seulement dans ceux que l'on a en tête. Supprimer la branche inactive d'un `case True` est légitime ; supprimer `LD V0, [I]` ne l'est que si l'on est sûr que `I` et la case n'ont pas bougé entre-temps, et certaines variantes de CHIP-8 incrémentent `I` après `LD [I], vx`. C'est précisément pour ne dépendre d'aucune variante que le projet recharge `I` avant chaque accès, et c'est le genre de précaution qu'un optimi-

seur doit connaître avant de toucher au code. Mesurer vient ensuite : le README vous invite à comparer la taille de la même ROM avant et après, tests identiques.

7 Au-delà du projet

Le backend que vous écrivez a un grand frère : LLVM, commencé en 2000 à l'université de l'Illinois par Chris Lattner, et présenté par Lattner et Vikram Adve en 2004. Son idée est de séparer frontends et backends par une représentation intermédiaire commune, l'IR LLVM, une sorte d'assembleur idéalisé à registres infinis. Clang pour C et C++, rustc pour Rust, Swift et bien d'autres produisent cet IR ; les backends x86-64, ARM, RISC-V ou WebAssembly le prennent de là. Chacun refait, en grand, ce que vous faites en petit : sélection d'instructions par motifs sur un graphe, allocation de registres, avec plusieurs allocateurs au choix dont un glouton fondé sur les intervalles de vie, conventions d'appel propres à chaque système, et des centaines de passes peephole.

WebAssembly, défini en 2017 par Andreas Haas, Andreas Rossberg et leurs coauteurs, puis standardisé par le W3C en 2019, est une machine virtuelle pensée comme cible de compilation pour le navigateur. Vous la reconnaitrez : c'est une machine à pile, comme celle d'AP3, avec des instructions typées et un contrôle structuré, blocs et boucles plutôt que sauts arbitraires, afin qu'un vérificateur puisse garantir avant exécution qu'aucun programme ne lira hors de sa mémoire. Le navigateur compile ensuite ce bytecode à pile vers les registres du processeur, c'est-à-dire qu'il refait, à la volée, l'allocation de registres que vous avez contournée.

Cela mène à la dernière distinction. Un *émulateur* CHIP-8 est un interprète : il lit une instruction, la decode, l'exécute, et recommence, comme la VM d'AP3. Un *compilateur à la volée*, ou JIT (*just in time*), traduit plutôt le bytecode en code machine au moment du chargement ou de la première exécution, pour supprimer le coût du décodage. La BEAM a interprété son bytecode pendant plus de vingt ans ; avec Erlang/OTP 24, en mai 2021, elle a reçu BeamAsm, un JIT écrit par l'équipe OTP avec la bibliothèque asmjit. Son principe est modeste et efficace : chaque instruction BEAM possède un gabarit de code machine écrit à la main, et le chargeur recopie ces gabarits à la suite, sans profilage ni recompilation. D'abord réservé à x86-64, BeamAsm couvre les processeurs ARM 64 bits depuis OTP 25. Le code Elixir que vous avez écrit cette année a donc, sans que vous le sachiez, traversé un backend dont les décisions ressemblent fort à celles de votre Codegen.

8 Pour aller plus loin

Cowgod's Chip-8 Technical Reference v1.0, Thomas P. Greene (1997).

<http://devernay.free.fr/hacks/chip8/C8TECH10.HTM>

La référence que suivent les émulateurs : registres, carte mémoire et les trente-cinq instructions, une page chacune. À garder ouverte pendant le projet.

Guide to making a CHIP-8 emulator, Tobias V. Langhoff (2020).

<https://tobiasvl.github.io/blog/write-a-chip-8-emulator/>

L'autre côté du miroir : comment on écrit la machine que vous ciblez, avec l'histoire des variantes et de leurs pièges, dont ceux de `I` et des décalages.

Octo, manuel, John Earnest.

<https://johnearnest.github.io/Octo/docs/Manual.html>

L'émulateur du projet et son assembleur, pour écrire quelques programmes à la main et voir ce que produit un humain face à ce que produit votre backend.

Register Allocation via Coloring, Gregory Chaitin, Marc Auslander, Ashok Chandra, John Cocke, Martin Hopkins et Peter Markstein (1981).

[https://doi.org/10.1016/0096-0551\(81\)90048-5](https://doi.org/10.1016/0096-0551(81)90048-5)

L'article qui a fait de l'allocation de registres un problème de coloriage. Court, concret, lisible après la section 4.

Debunking the « Expensive Procedure Call » Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO, Guy L. Steele Jr. (1977).

<https://doi.org/10.1145/800179.810196>

Le manifeste de l'appel terminal, écrit avec verve. Il explique pourquoi un appel n'est qu'un saut, et ce qui coûte vraiment.

Peephole Optimization, William M. McKeeman (1965).

<https://doi.org/10.1145/364995.365000>

Deux pages de 1965 qui suffisent à comprendre l'idée, avec des exemples sur une machine aussi simple que la nôtre.

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Chris Lattner et Vikram Adve (2004).

<https://llvm.org/pubs/2004-01-30-CGO-LLVM.html>

La présentation de LLVM et de son IR. La section 2, qui décrit la représentation intermédiaire, se lit avec le bagage d'AP3.

BeamAsm, the Erlang JIT, documentation Erlang/OTP.

<https://www.erlang.org/doc/apps/erts/beamasm.html>

Comment la BEAM traduit son bytecode en code machine : gabarits, chargement,

allocation des registres du processeur. Pour voir votre Elixir de cette année sous un autre angle.

9 Une question pour la route

Le projet transforme en saut l'appel terminal d'une fonction à *elle-même*. Steele affirme pourtant que tout appel terminal, y compris vers une autre fonction, peut devenir un saut. Qu'est-ce qui rend ce cas général plus délicat quand les deux fonctions n'ont pas le même nombre de paramètres, ni les mêmes cases ? Et si l'appelante doit encore libérer quelque chose après le retour, l'appel est-il toujours terminal ? Cherchez ce que `prepare/4` devrait garantir de plus, sans l'écrire.

Pour toute question, vous pouvez me contacter par email, comme pour les autres séances.