Projet d'informatique

Le but de ce projet et de compiler un fragment du langage C et de produire du code pour un processeur simulé de type RISC (*Reduced Instruction Set Computer*). Ce projet est constitué de plusieurs parties plus ou moins indépendantes :

- analyse du source C et construction d'un arbre syntaxique ;
- contrôle sémantique de l'arbre syntaxique obtenu (typage, définition des identificateurs ...) ;
- transformation (compilation) de l'arbre syntaxique du programme C en une séquence d'instructions pour le processeur considéré ;
- exécution (simulation) du code obtenu.

1 Le langage source

On s'intéresse à un petit langage impératif utilisant la syntaxe de C du genre suivant :

fib.c

```
int nb_calls;
fib1(int n)
{
  nb_calls = nb_calls + 1;
  if (n == 0) {
    return 1;
  } else if (n == 1) {
    return 1;
  } else {
    return (fib1(n-1)+fib1(n-2));
  }
}
fib2(int n)
{
  int fib[100];
  int i;
  fib[0] = 1; fib[1] = 1; i = 2;
  while (i <= n) {
    fib[i] = fib[i-2] + fib[i-1];
    i = i + 1;
  }
  return fib[n];
}
main()
{
  nb_calls = 0;
  print(fib1(12)); print(nb_calls);
  print(fib2(12));
}
```

On se limite aux idiomes suivants :

- les données sont des entiers (`int`) ou des tableaux d'entiers (`int[]`) ;
- les fonctions ont des paramètres entiers et retournent (`return`) une valeur entière (`int`) ; elle peuvent être récursives ;
- la seule instruction de base est l'affectation (`=`) sur les entiers ;
- les seules structures de contrôle sont la boucle (`while`) et la conditionnelle (`if`) ;
- les expressions sont construites avec quelques opérateurs de base sur les entiers (`+`, `*`, `==` ...), les expressions booléennes étant également de type entier (`1` pour vrai, `0` pour faux).

1.1 Syntaxe

La grammaire du langage traité s'écrit (informellement) comme suit :

program : := ϵ| declaration program | fonction program
declarations : := ϵ| declaration declarations
declaration : := `int` variable `;`
variable : := ident | ident `[` integer `]`
fonction : := ident `(` params `)` `{` declarations statements `}`
params : := ϵ| `int` ident nextparams
nextparams : := ϵ| `,` `int` ident nextparams
statements : := ϵ| statement statements
statement : := ident `=` expression `;`
: := ident `[` expression `]` `=` expression `;`
: := `while` `(` expression `)` statement
: := `{` declarations statements `}`
: := `if` `(` expression `)` statement else_
: := expression `;`
: := `return` expression `;`
else_ : := ϵ| `else` statement
expression : := expression `<` expression | expression `<=` expression | expression `==` expression
: := expression `+` expression | expression `-` expression
: := expression `*` expression | expression `/` expression
: := integer | ident | ident `(` args `)` | `(` expression `)` | ident `[` expression `]`
args : := ϵ| expression nextargs
nextargs : := ϵ| `,` expression nextargs
ident : := [`a-zA-Z`]([`a-zA-Z_0-9`]*)
integer : := [`0-9`]+

1.2 Analyse

L'analyseur correspondant à la grammaire fournie sera écrit à l'aide de `ocamllex` et `ocamlyacc`[1]. On divisera donc l'analyse en une partie lexicale (un fichier `lexer.mll`) et une partie syntaxique (un fichier `parser.mly`) .

Le rôle de l'analyseur est de fabriquer une représentation du texte source, représentation qui sera utilisée pour la compilation. Cette représentation devra donc être choisie **avant** d'écrire l'analyseur.

2 Arbre syntaxique

L'arbre syntaxique constitue la représentation du programme C traité. Cette représentation doit être suffisamment structurée pour être utilisée simplement lors de la compilation et suffisamment simple pour être fabriquée directement par l'analyseur syntaxique. Cette représentation pourra être structurée en plusieurs types (types de OCaml) :

- `c_program` le type des objets représentant les programmes (déclarations globales et définition de fonctions) ;
- `c_decl` le type des déclaration (identificateur avec son type) :
- `c_type` le type des types C (entier, tableau ou type fonctionnel) ;
- `c_function` le type des fonctions (nom, paramètres et corps) ;
- `c_statement` le type des instructions (affectation, bloc, boucle, conditionnelle, `return` ou expression) :
- `c_expr` le type des expressions (opération binaire, variable, élément de tableau, entier ou appel fonctionnel).

[1] On s'inspirera fortement de l'exemple du manuel de référence.

L'ensemble de ces définitions constituera l'interface (un fichier `syntax.mli`) entre l'analyseur et le compilateur proprement dit.

Pour des facilités de mise au point, il sera bon d'écrire une fonction (`print_program`) permettant d'écrire un arbre syntaxique sous forme de texte (cette fonction est exactement la réciproque de l'analyseur syntaxique).

3 Contrôle sémantique

Avant la phase de compilation il est nécessaire d'effectuer un minimum de contrôle sur l'arbre syntaxique fabriqué par l'analyseur :

- vérification que les identificateurs de variable (resp. de fonction) utilisés sont déclarés (resp. sont définis) ;
- vérification basique de type :
  - l'opération d'affectation (`=`) est appliquée sur une variable de type entier ;
  - dans l'expression a`[`i`]`, i doit être de type entier et a doit être l'identificateur d'un tableau d'entiers ;
  - une fonction doit être appliquée à un nombre correct d'arguments
  - ...

Cette étape de contrôle ne modifie pas l'arbre syntaxique : elle peut être facultative dans la mesure où les programmes analysés sont corrects.

Ce contrôle sera implanté avec un ensemble de fonctions (fichier `typing.ml`) dont l'interface pourra pourra avoir la forme suivante :

`typing.mli`

```
open Syntax
exception Type_error of string
exception Undefined of string
val typing : c_program -> unit
```

où la fonction `typing` ne calcule rien mais renvoie uniquement une exception si le contrôle échoue.

4 Processeur

Il serait relativement délicat d'essayer de produire du code pour un vrai processeur (le processeur des stations de travail SUN par exemple). On va donc se contenter d'un simulateur de processeur, le simulateur étant un programme OCaml.

4.1 Description

Le processeur manipule des *mots*. Dans les processeurs actuels, un mot est codé typiquement sur 32 bits. Chaque mot contiendra un entier (on ne s'intéresse pas à des opérations sur des nombres flottants).

4.1.1 Architecture

Le processeur possède :

- des registres (32) : ce sont des mémoires internes pouvant contenir un mot chacun ;
- une zone de donnée : une pile de mots ;
- une zone de code : la séquence des instructions qui sont exécutées par le processeur.

Les registres sont numérotés de 0 à 31. Le registre 0 contient **toujours** la valeur 0. Le registre 31 sera aussi appelé `ra` (*Returned Adress*). Le registre 30 (`sp` pour *Stack Pointer*) sera utilisé pour désigner le sommet de la pile de données. Par convention, le registre 1 (`res`) sera utilisé pour stocker les valeurs retournées par les fonctions.

On considérera que la zone de données est aussi grande que nécessaire. Les mots de cette zone sont désignés par des *adresses* : `0`, `1` ...

La zone de code est différenciée de la zone des données pour des raisons de simplicité (des mots dans la zone de données mais des instructions dans la zone de code). Chaque instruction de la zone de code est également adressée : `0`, `1` .... À chaque instant, le processeur pointe vers l'une des instruction (celle qui est exécutée).

4.1.2 Instructions

Les instructions du processeur sont les suivantes (r est un nom de registre, o est une opérande, nom de registre ou constante, a est une constante) :

- `add`(r_1, o, r_2), addition, r_2 reçoit $r_1 + o$ (idem pour `sub`, `mult` et `div` soustraction, multiplication et division) ;
- `slt`(r_1, o, r_2), inégalité stricte, r_2 reçoit 1 si $r_1 < o$ et 0 sinon (idem pour `sle` et `seq`, inégalité large et égalité) ;
- `load`(r_1, o, r_2), lecture mémoire, r_2 reçoit le contenu de l'adresse $r_1 + o$;
- `store`(r_1, o, r_2), écriture mémoire, le contenu de r_2 est écrit à l'adresse $r_1 + o$;
- `jmp`(o, r), branchement, saute à l'adresse o et stocke l'adresse de l'instruction qui suit le `jmp` dans r ;
- `braz`(r, a), branchement si zéro, saute à l'adresse a si $r = 0$ (idem pour `branz`, saut si non zéro) ;
- `scall`(n), appel système numéro n ;
- `stop`, arrêt du processeur.

Utilisation du registre `0` Le registre `0` dont le contenu vaut toujours 0 permet de réaliser les instructions classiques (?) de certains autres processeurs :

- `add`$(r_1$,`0`$, r_2)$ copie r_1 dans r_2 (`move`) ;
- `add`(`r 0`$, n, r)$ met la constante n dans r ;
- `load`(`r 0`$, a, r)$ lit le mot à l'adresse a ;
- `jmp`(a,`r 0`) saute à l'adresse a.

Convention L'appel et le retour d'un sous-programme utiliseront le registre `ra` :

- `jmp`(a,`ra`) pour l'appel ;
- `jmp`(`ra`,`r 0`) pour le retour.

4.1.3 Pile

Évidemment, l'appel d'un sous-programme depuis l'intérieur d'un autre ne peut pas être géré simplement avec un registre : une pile est nécessaire.

Les instructions classique de gestion de pile (`push` et `pop`) ne sont pas fournies par le processeur : elles ne sont pas nécessaires et seront simulées en utilisant des `store`, `load`, `add` et `sub` en prenant le registre `sp` comme pointeur de pile.

On pourra prendre la convention que le registre `sp` pointe toujours vers vers l'adresse de retour de la fonction courante.

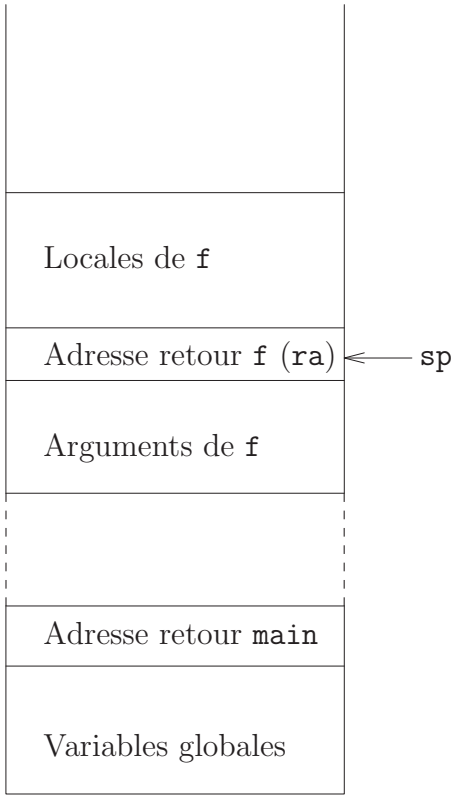
4.2 Simulation

On va écrire un simulateur pour le processeur que l'on vient de présenter.

La première chose à faire est de choisir une représentation pour les instructions : définition d'un type `instruction` (un fichier `risc.mli`). Comme pour l'arbre syntaxique (*cf* 2), pour faciliter la mise au point, il sera bon d'écrire une fonction permettant d'afficher une instruction : `print_instruction` (fichier `risc.ml`).

La simulation (fichiers `simul.ml`, `simul.mli`) fournira une fonction

```
exec : instruction array -> unit
```



- Le bas de la pile contient les variables globales dont les adresses sont fixées statiquement (pendant la compilation). On trouve donc les variables globales aux adresses 0, 1, 2, ...
- Pour le reste de la pile, seule la partie haute, relativement à la position de l'adresse de retour (pointée par `sp`) de la fonction courante est accessible.
- La position relative des variables locales par rapport à la position pointée par `sp` est fixée statiquement. On trouve donc les variables locales aux adresses (`sp`) + 1, (`sp`) + 2, ...
- Le registre `ra` contient l'adresse (dans la zone de code) à laquelle la fonction doit sauter quand elle est terminée. Ce registre est sauvé dans la pile puisqu'il sera modifié si un appel fonctionnel est exécuté depuis la fonction courante.
- Les arguments de la fonction courante se trouvent aux adresses (`sp`) − 1, (`sp`) − 2, ...

FIG. 1 – État de la pile pendant l'exécution d'une fonction `f` et commentaires.

qui exécute un code assembleur.

4.2.1 Appels systèmes

On implantera au moins une facilité d'affichage sur la sortie standard (fonction `print` du langage source) sous la forme d'un appel système du processeur (avec la convention d'appel à choisir). On pourra éventuellement aussi implanter un appel système permettant de faire une lecture sur l'entrée standard.

5 Compilation

Les types définis précédemment (`syntax.mli` et `risc.mli`) permettent de représenter respectivement des programmes en C et des programmes en assembleur. L'étape de compilation va permettre de passer de l'un à l'autre. Il s'agit de faire une transformation : une fonction `compile : c_program -> instruction list` (fichiers `compile.mli` et `compile.ml`) prend en argument un programme C et retourne un programme assembleur.

On prend comme convention que le programme source contient une fonction `main` (sans argument) qui sera la première appelée.

5.1 Données

Chaque variable entière sera stockée dans un mot de la mémoire (zone de données, c'est à dire la pile). Les éléments d'un tableau d'entiers seront stockés dans des mots contigus de la mémoire. Les variables globales (correspondant aux déclarations **en dehors** des définitions de fonction) seront stockées «en bas» de la mémoire. Les variables locales (paramètres des fonctions et déclarations

locales aux blocs) seront allouées sur la pile. La pile contiendra donc indifféremment des données et des adresse de retour.

Par exemple, avec le programme suivant :

```
int n;
f(int i, int j)
{
  int t[10];
  ...
}
main()
{
  int n;
  print(f(7,8));
}
```

la pile contiendra (dans l'ordre, du bas vers le haut) au moment de l'exécution de `...` :

- l'entier `n` (global) ;
- l'adresse de retour de la fonction `main` ;
- l'entier `n` (local à la fonction `main`) ;
- les entiers `7` et `8` (les paramètres de `f`) ;
- l'adresse de retour de `f` ;
- les éléments `t[0]` ...`t[9]`.

5.2 Convention des appels fonctionnels

Un appel fonctionnel sera réalisé comme suit :

- l'appelant empile les arguments (dans l'ordre) ;
- l'appelant positionne ler registre `sp` en sommet de pile ;
- l'appelant se branche à l'adresse de l'appelé en positionnant l'adresse de retour dans le registre `ra` ;
- l'appelé empile l'adresse de retour[2] ;
- l'appelé s'exécute (il commence éventuellement par réserver sur la pile la place nécessaire pour ses variables locales).

et symétriquement pour le retour d'un appel [3] :

- l'appelé positionne sa valeur à retourner dans le registre `res` :
- l'appelé restaure le registre `ra` avec le contenu du sommet de pile ;
- l'appelé se branche à l'adresse `ra` ;
- l'appelant restaure le registre `sp` ;
- l'appelant utilise le registre `res` qui contient la valeur retournée.

5.3 Contrôle

Le langage considéré comporte deux primitive de contrôle, la conditionnelle et la boucle.

La conditionnelle L'instruction `if (`e`)` s_1 `else` s_2 pourra se transformer en la séquence suivante :

[2] Ça n'est pas indispensable si l'appelé «sait» qu'il ne modifiera pas le registre `ra`, c'est-à-dire si l'appelé ne fait pas d'appel lui-même.

[3] La sortie d'une fonction est provoquée par l'instruction `return` ; il peut être judicieux d'ajouter systématiquement, pendant l'analyse syntaxique, une instruction `return 0 ;` à la fin de toutes les fonctions.

code pour e, résultat dans le registre r
`braz`(r, n)
code pour s_1
`braz(r 0,`m`)` (ou `jmp(`m`,r 0)`
n : code pour s_2
m : suite

La boucle L'instruction `while (`e`)` s pourra être compilée en :

n : code pour e, résultat dans le registre r
`braz`(r, m)
code pour s
`braz(r 0,`n`)` (ou `jmp(`n`,r 0)`
m : suite

5.4 Les expressions

La compilation des expressions pour une machine à registres est un peu délicate.

Les constantes, les variables entières (locales ou globales), ne posent pas de problèmes particuliers : il suffit d'aller lire la valeur dans la pile.

L'évaluation de l'application d'un opérateur binaire (`+`, `==` ...) nécessite le stockage des opérandes dans des registres. Par exemple, on évaluera e_1`+`e_2 avec la séquence :

calcul de e_1, résultat dans le registre r_1
calcul de e_2, résultat dans le registre r_2
`add`(r_1, r_2, r), addition, résultat dans r

Ce schéma de transformation peut être appliqué récursivement. Cependant, on bute sur deux problèmes :

- le nombre de registres est fini : il est donc nécessaire de choisir le **bon** ordre pour évaluer les opérandes. Une stratégie possible est d'évaluer en premier l'opérande qui nécessite le plus de registres. Le nombre de registres nécessaire pour évaluer une expression est donné par son nombre de Strahler : se référer à la littérature (algorithmes sur les arbres, et/ou compilation).
- l'évaluation d'un appel à une fonction peut modifier arbitrairement tous les registres : il est donc nécessaire de sauvegarder **tous** les registres utiles avant l'appel et de les rétablir au retour de l'appel. La mise à plat des appels fonctionnels est une solution au problème plus élégante : voir 6.1.

5.5 Calcul des branchements

Lors de la génération du code, il est judicieux pour les branchements d'utiliser des *étiquettes* symboliques en ajoutant une instruction `label` l où l est un entier ou une chaîne de caractères. La phase d'*assemblage* consiste alors à calculer les adresses des branchements et à supprimer ces pseudo-instructions `label`.

6 Améliorations

Diverses améliorations peuvent être apportées au projet tel qu'il a été présenté.

6.1 Transformation du programme source

Avant la phase de compilation, il peut être judicieux de *transformer* le programme source, ceci pour plusieurs objectifs :

- Simplifier des expressions et des instructions. Par exemple, l'expression `(2*2)` peut être transformée en `4`, l'instruction `if (0) x = 1 else x = 2` peut être transformée en `x = 2`.
- Ajouter des idiomes au langage traité. Par exemple, une boucle `for(init;cond;incr) { corps }` peut être transformée en une boucle `init; while(cond) { corps; incr }`.
- Mettre *à plat* les appels fonctionnels pour simplifier la phase de compilation. Par exemple, l'instruction `x = f(g(y)+h(x))` peut être transformée en
  `{ int v1, v2; v1 = g(y); v2 = h(x); x = f(v1+v2); }`.
- Supprimer les appels récursifs terminaux. En ajoutant un `goto` au langage, on transforme ainsi

```
fact(int n, int p)
{ if (n == 0) return p else return fact(n-1,n*p); }
```

  en

```
fact(int n, int p)
{
 fact0:
  if (n == 0) return p
  else { p = n * p; n = n - 1; goto fact0 };
}
```

  Ce genre d'optimisation est nettement plus facile à faire au niveau du source que par une modification de la phase de compilation.

6.2 Optimisation du code produit

Le code produit *naïvement* peut être optimisé *a posteriori* notamment par amélioration locale (*peephole optimisation*) :

- Suppression des instructions simples inutiles comme `add(r2, 0, r2)`.
- Compression des séquences d'instructions comme `add(sp, 1, sp); add(sp, 1, sp)`
- Suppression du code inaccessible : ceci nécessite de suivre tous les branchements accessibles depuis la première instructions du programme, de marquer ainsi les instructions atteintes et de supprimer les autres.
- ...

7 Réalisation

Un schéma résumant la structure du programme total est donné en figure 7.

Le projet sera réalisé en OCaml en utilisant le compilateur `ocamlc` ainsi que les pré-processeurs `ocamllex` et `ocamlyacc` pour l'analyseur syntaxique.

Le découpage en modules est décrit table 1 (un `Makefile` est disponible dans `/usr/local/serveur/CAML/projet`) :

La taille (calculée avec `wc`) d'une solution au projet est indiquée dans le tableau. Ceci représente la taille d'un code brut, pas forcément optimal mais non commenté ; on peut donc multiplier ces nombres par deux pour avoir une estimation d'une «vraie» solution (avec des commentaires).

Les fichiers `syntax.mli` et `risc.mli` sont les premiers à écrire. Les autres modules deviennent alors indépendants les uns des autres.

Le programme sera testé avec les exemples (fichiers `.c`) fournis dans `/usr/local/serveur/CAML/projet`.

Recommandations

1. Partager le travail au sein du binôme : l'analyseur syntaxique et le simulateur de processeur sont indépendants et ils peuvent être développés en parallèle.
2. Procéder incrémentalement : tester au fûr et à mesure le code écrit, en le compilant et en l'exécutant. La première étape sera de faire fonctionner l'analyseur syntaxique pour imprimer simplement à l'écran le résultat de l'analyse. En parallèle, le simulateur de processeur pourra être testé avec un programme en assembleur écrit à la main.

Fichier	Contenu	Taille indicative		
		Lignes	Mots	Caractères
lexer.mll	Analyseur lexical	30	146	614
parser.mly	Analyseur syntaxique	88	384	2039
syntax.mli	Type des arbres syntaxiques	28	177	1180
syntax.ml	Affichage des arbres	59	381	2407
typing.ml	Contrôle sémantique	60	325	2339
typing.mli		4	16	117
risc.mli	Type des instructions du processeur	28	114	654
risc.ml	Affichage des instructions	29	208	978
compile.ml	Compilation	141	1011	5949
compile.mli		4	15	93
simul.ml	Simulateur du processeur	49	282	1871
simul.mli		2	13	69
main.ml	Appels aux différentes passes	25	107	771
total		547	3179	19081

TAB. 1 – Découpage en modules

3. Pour la partie compilation, prévoir initialement tous les cas mais les traiter avec une exception. On écrira par exemple pour la compilation des expression :

```
...
    Call (f, args) -> failwith "Appels fonctionnels pas encore traités"
  | Var x -> failwith "Variables pour plus tard"
...
```

4. Utiliser les programmes test du fichier `test.c`.
5. Commencer par compiler le programme *vide* constitué de

```
main()
{}
```

L'analyse devra produire `main(){return 0;}` (c.f. note en bas de page 3) et la compilation devra produire le code suivant :

```
1   Add (reg 0, cst 0, sp)
2   Jmp (main, ra)
3   Stop
4   Label main
5   Store (sp, cst 0, ra)
6   Add (reg 0, cst 0, res)
7   Load (sp, cst 0, ra)
8   Jmp (ra, reg 0)
```

Les 3 premières instructions correspondent à l'initialisation du pointeur de pile (ici à 0 car il n'y a pas de variables globales), à l'appel à la fonction `main` et à l'arrêt du programme (au retour de `main`). Suit (ligne 4) le code correspondant à la fonction `main`. On commence par stocker l'adresse de retour en sommet de pile (instruction `Store`). La compilation de l'expression `0` se réduit à mettre `0` dans le registre `res`, ligne 6. Il s'agit enfin de sortir de la fonction (`return`) en rechargeant l'adresse de retour toujours pointée par `sp` et en sautant à cettre adresse (`Jmp` dernière ligne).

La compilation sera décomposée en plusieurs fonctions, exactement une par type. On aura donc la répartition suivantes pour cet exemple :

– instructions 1, 2 et 3 produites par `compile_program` ;
– instruction 4 et 5 produite par `compile_fonction` ;

– instruction 6 produite par `compile_expression` (cas *Integer*) ;
– instruction 7 et 8 produites par `compile_statement` (cas *Return*).

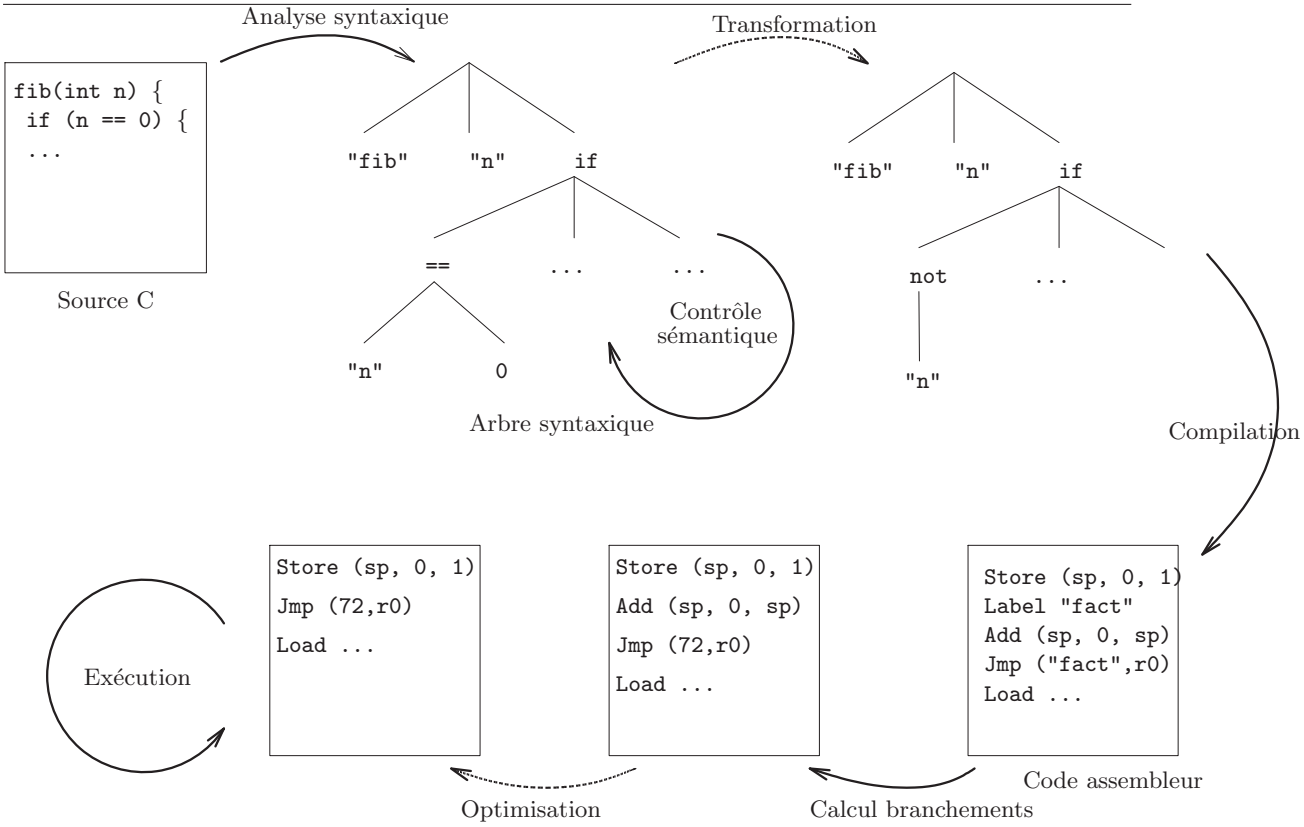


FIG. 2 – Les différentes passes

7.1 Évaluation

Le travail (effectué en binôme) sera évalué au vu de :
– le code OCaml écrit : lisibilité, utilisation des possibilités du langage, style, efficacité ... ;
– un « rapport » sur l'état d'avancement à mi-parcours ;
– une présentation de la réalisation (ce qui est fait et ce qui ne l'est pas) et une démonstration du programme (environ un quart d'heure en tout).