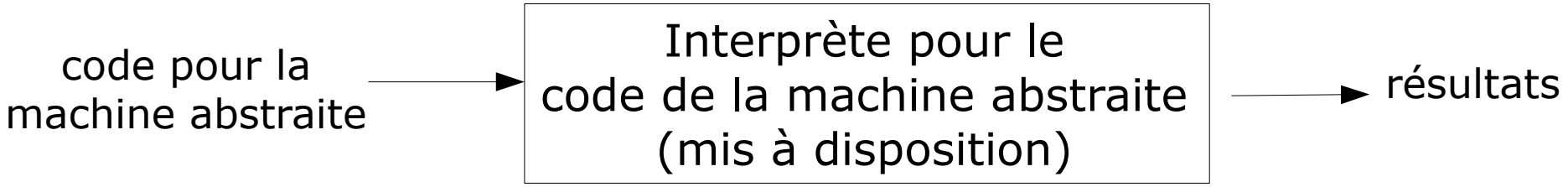
Le projet 2020-2021



Interprète pour la machine abstraite

Archive avec le source de l'interprète disponible sur ma page

L'archive contient des exemples de programmes écrits dans le langage de la machine (mais sans rapport direct avec le TP et le projet)

Appel de l'interprète ?

```
interp fichier-de-code
```

ou `interp -d fichier-de-code` (mode debug)

Mode debug ?

- Commandes pour visualiser le contenu de la mémoire, l'instruction courante, etc.
- Mode pas-à-pas, pose de points d'arrêts, etc.

Code de la machine abstraite

Semblable à un assembleur (pouvoir d'expression limité)

Structure de « **machine à pile** » :

- Les **opérandes d'un opérateur sont implicites** (en sommet de pile)
- Les instructions dépilent les opérandes et empilent le résultat

Exemple : traduction de `3 + 5`

```
PUSHI 3
PUSHI 5
ADD
```

Format des instructions ?

* *étiquette : Instruction + argument éventuel -- commentaire*

Le symbole * est facultatif et sert à indiquer un point d'arrêt

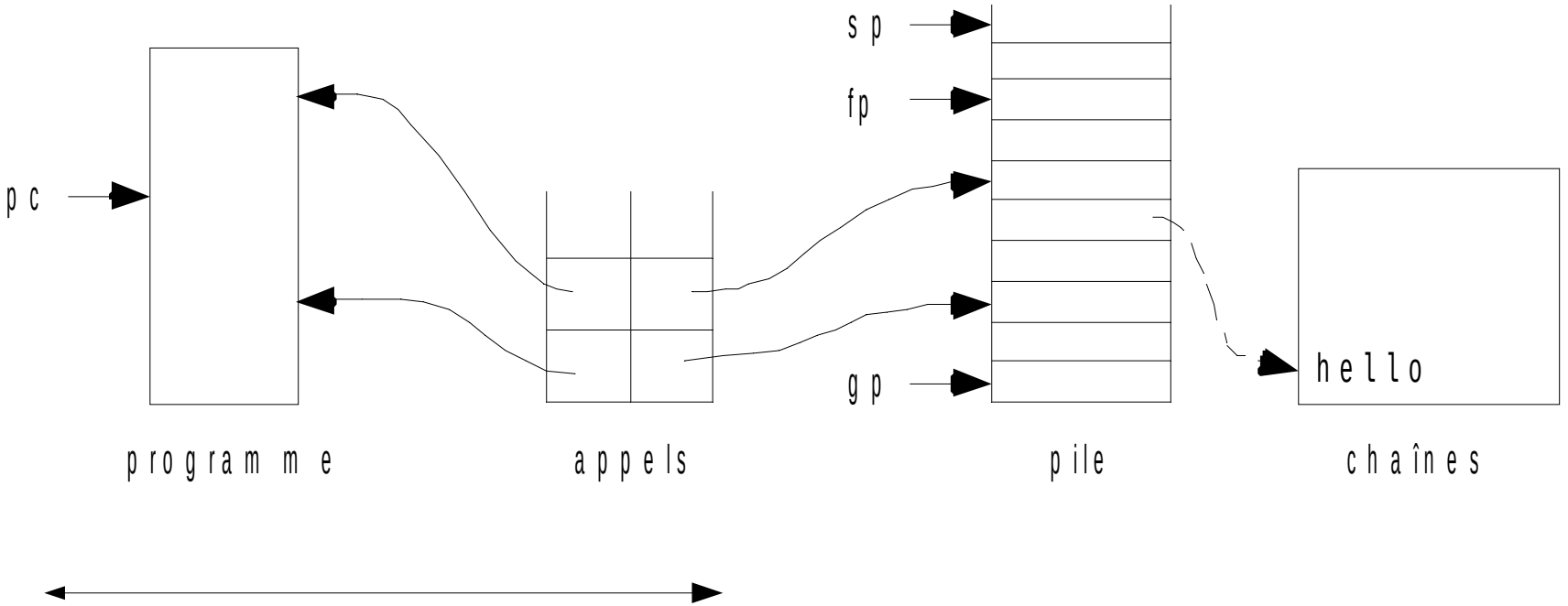
L'étiquette (et le `:` qui suit) est facultative

Si le commentaire est présent dans le code, en mode debug il sera affiché avec l'instruction courante

Exemples d'instructions de base

- `NOP`
- `ADD, SUB, …, INF, INFEQ, EQUAL, …`
- `PUSHI` *entier*`, WRITEI, PUSHS` *string*`, WRITES`
- `JUMP` *label*`, JZ` *label*
- …

Architecture de la machine abstraite



Partie cachée

Architecture de la machine abstraite

Une **pile** pour les appels de fonctions, les variables locales, …

Le fond de la pile peut servir à stocker des objets permanents (variables globales par exemple) en faisant attention à ce que le pointeur de pile ne descende jamais sous ce niveau.

Un « **tas** » pour gérer les objets dont la **durée de vie** est inconnue (objets alloués dynamiquement), les chaînes de caractères.

Une **pile cachée** pour la gestion sécurisée des appels de fonctions.

Les « mots » sont capables de stocker des valeurs de n'importe quel type de base (entiers, références d'objets ou de chaînes)

3 pointeurs pour gérer/accéder à la pile :

- SP (Stack Pointer) — manipulé par toutes les instructions
- GP (Global Pointer) — Fond de pile. Reste inchangé
- FP (Frame Pointer) — Modifié par `CALL` et `RETURN`
  Pointe sur le « tableau d'activation » courant

D’autres instructions

Instructions relatives au « tas » :

- ALLOC *n*
- LOAD *offset*
- STORE *offset*
- Pas d’instruction de désallocation, ni de « garbage collector »

Instructions relatives à FP (offset positif, nul ou négatif) :

- STOREL *offset*
- PUSHL *offset*

Instructions relatives à GP (offset positif ou nul) :

- STOREG *offset*
- PUSHG *offset*

« *offset* » : relatif à une adresse stockée en sommet de pile.

Quelques autres instructions

Gestion de la pile :

- `DUPN` *n*
- `POPN` *n*
- `PUSHN` *n*

Gérer l'exécution :

- `START`
- `STOP`

L'exécution démarre par la première instruction du fichier

Attention à ne pas essayer d'accéder à des adresses au dessus de `SP` (même via `FP` ou `SP`) : les adresses sont contrôlées.

Ne pas oublier que les instructions dépilent leurs opérandes. Les dupliquer à l'avance via `DUPN` si besoin.

Appels de fonctions

Appel de fonction : une séquence de deux instructions :

- `PUSHA` *label* *label* correspond à une **étiquette dans le code**
- `CALL`

Retour de fonction :

- `RETURN`

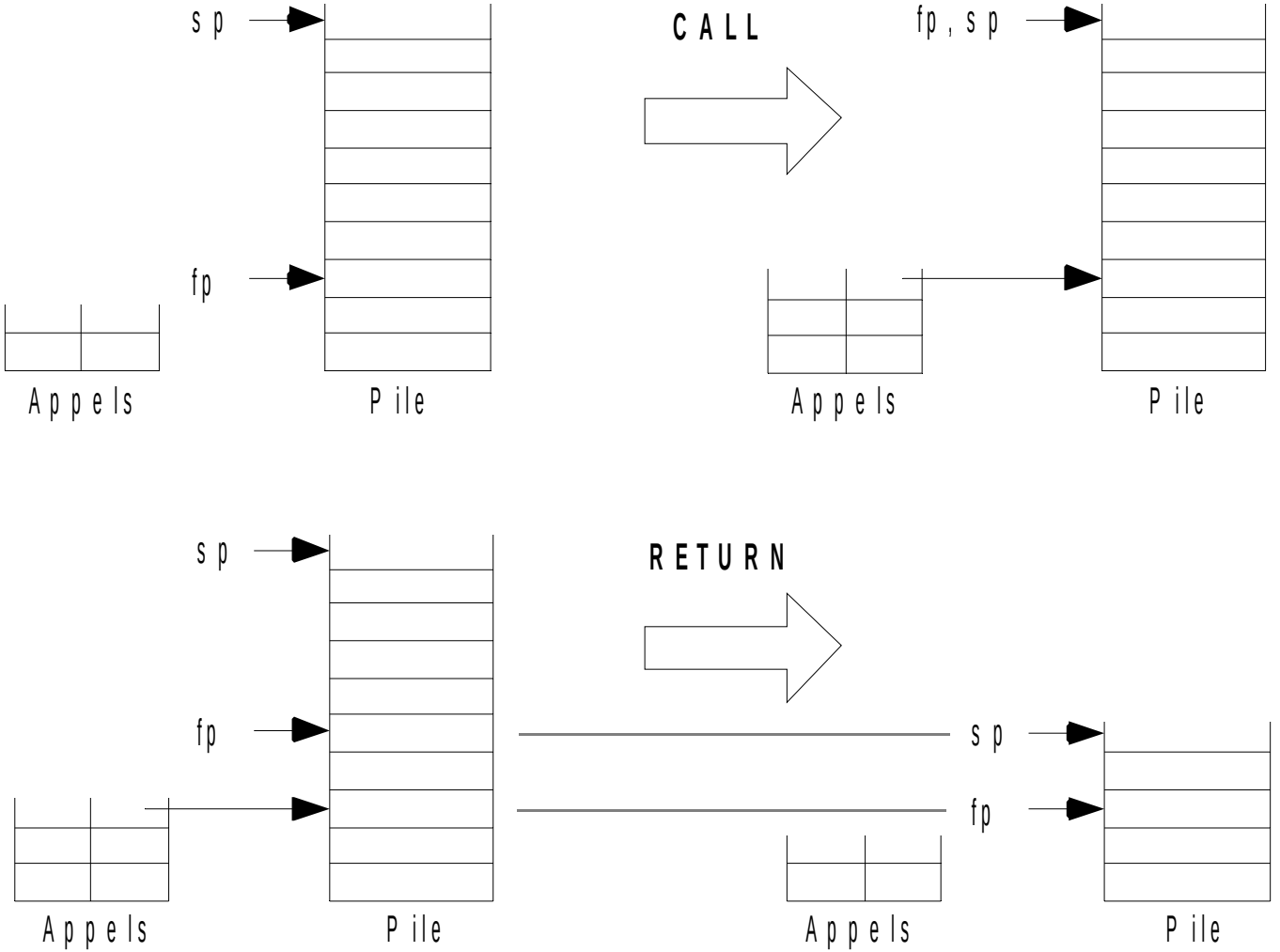
Gestion automatique de `SP` et `FP` ainsi que de l'adresse de retour chez l'appelant (« *program counter* »)

À votre charge : empilement/dépilement des arguments, réservation d'emplacements pour les variables locales, sauvegarde du résultat

Être sûr de garder la pile « sous contrôle » :

- De quelle situation on part, quelle situation on doit trouver au retour
  Fixer qui fait quoi entre appelant et appelée !
- Distinguer ce qui laisse des valeurs dans la pile et ce qui n'en laisse pas !

Gestion des appels de fonctions



Organisation dynamique de la mémoire

Ou « comment retrouver à l'exécution l'emplacement de tout identificateur, quelle que soit sa catégorie » !

Catégories d'identificateurs (déterminée lors de l'analyse de portée, en tenant compte des règles de masquage et de visibilité) :

- Variable globale
- Paramètre formel de la méthode courante
- Attribut d'instance
- Variable locale à un bloc (y compris avec des blocs emboîtés)
- Pseudo-variables : `this`, `super`, `result`

Déterminer les informations nécessaires selon la catégorie.

Les « vérifications contextuelles » doivent fournir ces informations au fur et à mesure qu'elles contrôlent la correction du programme

Méthodologie de réalisation

- Analyseurs lexical et syntaxique corrects ! Vérifier les précédences entre les constructions
- Représentation du programme source (AST en ocaml ?)
- Vérifications contextuelles :
  - Résolution de portée : *qui est qui* ? Gestion du masquage et de la visibilité
  - Typage des expressions **modulo héritage**
  - Vérification des appels de fonctions, des types de retour, etc.
  - Vérification des autres règles : bon usage de `this`, `super`, etc.
  - Calcul des informations nécessaires pour la génération de code
- Génération de code et représentation des classes prédéfinies

Méthodologie de réalisation (suite)

- **Utilisez à bon escient les mécanismes de ocaml et les librairies du langage. Se souvenir que le système de types est une aide, pas une contrainte** (bref, ne ralez pas!)
- Vérifications contextuelles **très** importantes :
  - Déterminer les programmes vraiment corrects
  - Déterminer les informations nécessaires pour la génération de code
- Le lancement du programme met en place les mécanismes nécessaires :
  - « tables virtuelles » pour gérer la liaison dynamique
  - Variables globales, initialisations, etc.
- **Pas d'intégration « big bang » :**
  - Mise en place progressive des mécanismes
  - Parallélisez vos tâches, faites-vous confiance mais relisez-vous !
  - Testez d'abord des programmes simples, dont on est sûr qu'ils sont corrects
  - Constituez vous votre base d'exemples significatifs
  - Pensez « général » : vérifiez que vos mécanismes se composent

Quelques exemples à traiter « à la main »

- `new` C()
- `var` x : C ; – selon le statut de x : champ, var locale d'un bloc
- x -- selon le statut de x
- x:= e ; -- où e est une expression arbitraire
- e1.v1 := e2.v2 ;
- x:= y.f(1, 2) ; -- liaison virtuelle de fonction
- `result`:= e ;
- `this`.f(1, 2) ;
- `super`.f(1, 2) ;
- `return` ou fin du corps d'une méthode