Réalisation d'un compilateur

T.P. de l'unité « Théorie des Langages & Compilation »

(voir aussi `www.dil.univ-mrs.fr/~garreta/Licence/compil.html`)

L'objet de ce travail est la réalisation d'un compilateur pour traduire des programmes écrits en *L*, qui est un langage de programmation à définir. Ce compilateur produit du code exécutable pour la machine *M*, qui est une machine virtuelle à pile à réaliser.

Le travail à faire se compose de quatre tâches :
- la définition du langage *L*, c'est-à-dire l'écriture de sa grammaire,
- la réalisation de l'analyseur syntaxique correspondant à cette grammaire,
- la transformation de l'analyseur en un compilateur, par l'ajout des opérations de génération de code,
- l'écriture de la machine *M* et la validation de l'ensemble.

Les éléments imposés de ce travail sont :
- un ensemble minimal de fonctionnalités du langage *L*, donné ci-dessous,
- la spécification de la machine *M*, donnée plus loin.

Lorsque les éléments imposés auront été implémentés, chaque étudiant pourra réaliser les extensions de son choix. Il est fortement recommandé de demander l'avis d'un enseignant avant de se lancer dans une extension d'origine personnelle.

1 *L* minimum

On prendra pour modèles les langages comme Pascal ou C, dont on pourra remplacer les mots-clés par d'autres plus expressifs ou plus sympathiques, avec les contraintes suivantes :

Types simples. Le langage *L* connaît un seul type simple, le type entier. Toutes les variables simples sont entières, et il n'y a pas d'équivalent de la déclaration `typedef` de C ou `type` de Pascal.

Types dérivés. *L* possède un unique type dérivé, le type « tableau d'entiers à un indice ». Une variable de type tableau est nécessairement globale.

Chaînes de caractères. On ne s'attardera pas sur les chaînes de caractères, ni constantes ni variables.

Opérateurs. On s'en tiendra aux opérateurs arithmétiques (au moins : `+`, `-`, `*`, `div`, `mod`), de comparaison (au moins : `==`, `<`, `<=`) et logiques (au moins : `non`, `et`, `ou`) usuels.

Instructions. On s'en tiendra aux instructions les plus basiques :
- instruction composée « `début`...`fin` » ou « { ... } » et instruction vide, comme en C ou en Pascal ;
- affectation, comme en Pascal (c.-à-d. a `:=` b n'est pas une expression et ne représente pas une valeur) ;
- instructions « `si`...`alors`... » et « `si`...`alors`...`sinon`... », comme en C ou en Pascal ;
- instruction « `tantque`...`faire`... », comme en C ou en Pascal ;
- instruction « `rendre` *expression* `;` » dans l'esprit du *return* de C.

Sous-programmes. En *L* tous les sous-programmes sont des fonctions à résultat entier. Leurs arguments sont simples (i.e. des entiers) et le passage se fait par valeur. Les fonctions possèdent des variables locales.

Une fonction ne peut pas être déclarée à l'intérieur d'une autre : les fonctions sont toutes globales.

L'appel d'une fonction comme si c'était une procédure – c'est-à-dire en ignorant le résultat rendu – est permis, comme en C. Pour limiter les difficultés syntaxiques, on pourra décider qu'une instruction réduite à un tel appel doit commencer par un mot réservé spécifique. Exemple : « `appel` *fonction*`(`*arguments*`)` »

Procédures prédéfinies. Les entrées-sorties de base se présentent comme des appels à des fonctions prédéfinies `lire` et `ecrire`. Par exemple :
- lecture : *variable* `:= lire() ;`
- écriture : `appel ecrire(`*expression*`) ;`

A part le fait qu'elles sont prédéfinies, ces deux fonctions sont traitées comme les fonctions ordinaires, écrites par les programmeurs *L*. En particulier, le passage de l'argument de la fonction `ecrire` et le retour du résultat de la fonction `lire` se fait comme pour les fonctions ordinaires.

Compilées à la main, ces deux fonctions seront codées « en dur », c'est-à-dire déposées dans l'espace du code et intégrées au dictionnaire pendant l'initialisation de ces éléments.

2 Analyse lexicale

L'analyseur lexical se présente comme une fonction, appelée dans nos exemples `lireUnite()` (si on utilise Lex pour l'écriture de l'analyseur lexical, cette fonction s'appellera `yylex()`), qui à chaque appel renvoie comme résultat l'unité lexicale dont c'est le tour.

Unités lexicales. Les unités lexicales de notre langage sont de plusieurs sortes :
- les mots-clés (`si`, `alors`, ...),
- les symboles simples (`+`, `;`, ...),
- les symboles doubles (`<=`, `!=`, ...),
- les identificateurs et
- les nombres, uniquement entiers.

Chaque unité lexicale est représentée par un nombre entier. Les unités dont le lexème n'est pas un caractère unique sont représentées par la valeur d'une constante symbolique, définie comme ceci (en C) :

```
#define TANTQUE 300
```

ou bien (en Java) :

```
static final int TANTQUE = 300;
```

Ces valeurs sont toutes supérieures ou égales à 256. Les unités lexicales dont le lexème est un caractère unique sont codées par [le code interne de] ce caractère lui-même, c'est-à-dire un entier inférieur à 256.

Mots clés. Si on écrit l'analyseur lexical « à la main », c'est-à-dire sans utiliser Lex, la reconnaissance d'un mot-clé commence comme celle d'un identificateur. Une fois le lexème formé, celui-ci est recherché dans une *table des mots clés* qui associe à chaque [lexème d'un] mot-clé [le code de] l'unité lexicale correspondante.

Cette table de mots clés *ne doit pas être confondue* avec le dictionnaire des identificateurs (cf. § 4), ces deux structures de données servent dans des couches différentes du compilateur, analyse lexicale pour l'une, analyse sémantique pour l'autre. Dit autrement : les mots clés *ne sont pas* des identificateurs particuliers !

Identificateurs. Lorsque l'analyseur lexical rencontre un identificateur *il ne lui appartient pas* de le rechercher dans le dictionnaire des identificateurs pour en vérifier la présence ou l'absence ou pour en trouver les attributs. Le travail de l'analyseur lexical doit se limiter à
- déterminer qu'il s'agit d'un identificateur (ce qui correspond, dans le cas d'un analyseur écrit « à la main », à l'échec de sa recherche dans la table des mots-clés),
- donner comme résultat la valeur `IDENTIFICATEUR` (si c'est comme cela qu'on a nommé la constante correspondante),
- affecter une variable globale, nommée par exemple `lexeme`, avec la chaîne de caractères reconnue, afin que d'autres couches du compilateur puissent la rechercher *ultérieurement* dans le dictionnaire des identificateurs.

Lecture du texte source. L'analyseur lexical accède au texte source caractère à caractère. Pour la clarté des idées, on s'interdira de lire le texte source par lignes.

Un petit problème posé par les analyseurs qu'on écrit soi-même est celui des unités dont la reconnaissance implique la lecture d'un caractère « en trop » : les nombres, les identificateurs, les opérateurs à deux caractères. On pourra lui donner la solution habituelle (la variable `calu`, toujours en avance) ou bien, si on écrit l'analyseur en C, utiliser la fonction `ungetc` de la bibliothèque standard.

S'agissant de développer un compilateur il faudra rapidement que les lectures se fassent dans un fichier au lieu de l'entrée standard. Pour éviter que cela entraîne des modifications un peu partout dans le programme, on remplacera dès le début les appels de `getchar()` par des appels d'une fonction spécialisée (dont la première version est triviale) :

```
int lireCar() {
    return getchar();
}
```

Cette fonction est pratiquement *le seul* point du programme à modifier lorsque, ultérieurement, on voudra lire dans un fichier à la place de l'entrée standard, ou bien compter les lignes ou connaître la position dans la ligne courante (en vue d'afficher des messages d'erreur très précis), ou encore produire un « fichier listing » de la compilation.

N.B. Si l'analyseur lexical provient de Lex, pour obtenir que les lectures se fassent dans un fichier au lieu de l'entrée standard il suffit de réaffecter, avant toute lecture, la variable `yyin` :

```
yyin = fopen(nom du ficher source, "r");
```

Commentaires. Il vous est conseillé, pour commencer, de ne pas permettre des commentaires dans les programmes écrits en L. En tout cas, cette question ne doit concerner que l'analyseur lexical, et il ne doit rester aucune trace des commentaires au niveau de l'analyseur syntaxique. Notamment, les commentaires ne sont mentionnés nulle part dans la grammaire.

3 Analyse syntaxique

Spécification du langage. On définira le langage L en rédigeant la grammaire non contextuelle correspondante. Exemple, avec une grammaire dans laquelle les non terminaux sont écrits en *italique* et les terminaux en `MAJUSCULES` ou entre `'apostrophes'` (les métasymboles {, }, [et] sont expliqués plus loin) :

programme → `PROGRAMME IDENTIF` { *déclaration* } *bloc* `'.'`
déclaration → `CONST` { `IDENTIF '=' NOMBRE ';'` }
  | *déclaration-de-variables*
  | `TABLEAU IDENTIF '[' NOMBRE ']'` { `',' IDENTIF '[' NOMBRE ']'` } `';'`
  | `FONCTION IDENTIF '('` *arguments-formels* `')'` [[*déclaration-de-variables*] *bloc*] `';'`
déclaration-de-variables → `ENTIER IDENTIF` { `',' IDENTIF` } `';'`
etc.

Méthodologie générale Pour obtenir un compilateur il faut d'abord posséder un analyseur syntaxique correct et complet auquel on ajoute *ensuite* les fonctions de production de code.

Un analyseur syntaxique lit – à l'aide d'un analyseur lexical – le texte source et n'en extrait aucune sorte de traduction, il ne fait que répondre à la question « ce texte appartient-il au langage engendré par la grammaire donnée ? » ou, plus familièrement, « ce texte est-il correct ? ».

Notre analyseur syntaxique sera écrit selon la technique de la *descente récursive*, qu'on peut présenter comme une méthode presque mécanique pour produire un analyseur syntaxique à partir d'une grammaire. En voici les principales idées :

• L'ensemble des règles de la forme « $S \rightarrow ...$ », c'est-à-dire ayant le même non terminal S pour membre gauche, donne lieu à une fonction *reconnaitre_S()*, ou simplement *S()*, qui n'est autre que l'analyseur correspondant à une grammaire qui aurait S pour symbole de départ. Chaque non terminal de la grammaire donne lieu à une telle fonction.

• Le corps de la fonction correspondant à une règle simple « $S \rightarrow \alpha_1 \; ... \; \alpha_k$ » (où $\alpha_1 ... \alpha_k$ sont terminaux ou non terminaux) est une *séquence* de groupes d'instructions, chacun correspondant à la reconnaissance d'un élément α_i.

• Chaque apparition d'un non terminal dans le membre droit d'une règle se traduit par l'appel de la fonction correspondante. Ainsi, par exemple, la règle $S_0 \rightarrow S_1 ... S_n$ (où S_1, ... S_n sont tous non terminaux) se traduit par une fonction de la forme :

```
reconnaitre_S0() {
    reconnaitre_S1();
    ...
    reconnaitre_Sn();
}
```

• L'apparition d'un terminal dans le membre droit d'une règle se traduit par l'instruction : « si l'unité courante n'est pas le terminal attendu annoncer une erreur, sinon acquérir l'unité suivante ». On constatera que c'est dans des situations de cette sorte que se produisent toutes les lectures et la plupart des détections d'erreurs. Ainsi une règle de la forme

instruction-tantque → `TANTQUE` *expression* `FAIRE` *instruction*

donne la fonction suivante (où, puisque la fonction `erreur` est sans retour – elle « tue » l'analyseur – les `else` peuvent être omis) :

```
reconnaitre_instruction_tantque() {
    if (uniteCourante != TANTQUE) erreur("'tantque' attendu");
    else lireUnite();
    reconnaitre_expression();
    if (uniteCourante != FAIRE) erreur(" 'faire' attendu");
    else lireUnite();
    reconnaitre_instruction();
}
```

• En présence d'une disjonction, les différents ensembles (appelés ci-après *PREMIER*) de symboles qui peuvent commencer la réécriture de chacun des termes de la disjonction permettent d'écrire la condition (portant sur l'unité lexicale courante) qui fera choisir le terme de la disjonction adéquat. Par exemple, puisque

$PREMIER(instruction_si) = \{$ `IF` $\}$
$PREMIER(instruction_tantque) = \{$ `WHILE` $\}$
...
$PREMIER(instruction_retour) = \{$ `RETURN` $\}$

la règle

$$instruction \rightarrow instruction_si \mid instruction_tantque \mid ... \mid instruction_retour$$

donne :

```
reconnaitre_instruction() {
    switch (uniteCourante) {
        case IF : reconnaitre_instruction_si(); break;
        case WHILE : reconnaitre_instruction_tantque(); break;
        ...
        case RETOUR : reconnaitre_instruction_retour(); break;
        else erreur("debut d'instruction incorrect");
    }
}
```

Métasymboles additionnels. Les métasymboles de base $\rightarrow$ (symbole fondamental) et $|$ (disjonction) sont suffisants pour écrire la grammaire non contextuelle de tout langage L. Par commodité on pourra employer en plus les métasymboles suivants :

• Les accolades, qui expriment la présence un nombre quelconque de fois de la suite de symboles qu'elles encadrent : $\{\beta\}$ représente toute séquence $\beta\beta...\beta$. La condition suivante devra être vérifiée : pour que la règle

$A \rightarrow \alpha \; \{ \; \beta \; \} \; \gamma$

soit légitime, il faut qu'une réécriture de γ ne puisse pas être vide et que $PREMIER(\beta) \cap PREMIER(\gamma) = \emptyset$.

Sous cette condition, la séquence $\{\beta\}$ se traduit très naturellement, lors de l'écriture de l'analyseur syntaxique, par une boucle `while` dont la condition porte sur l'unité lexicale courante, de la forme :

reconnaitre α
`while (uniteCourante` $\in$ *PREMIER(*β*)* `)`
    reconnaitre β
reconnaitre γ

ou bien de la forme :

reconnaitre α
`while (uniteCourante` $\notin$ *PREMIER(*γ*)* `)`
    reconnaitre β
reconnaitre γ

• Les crochets, qui expriment un élément facultatif, c'est-à-dire la présence ou l'absence de la suite de symboles qu'ils encadrent : $[\beta]$ représente soit β soit rien. Une condition analogue à la précédente devra être vérifiée : pour que la règle

$A \rightarrow \alpha \; [\; \beta \;] \; \gamma$

soit légitime, il faut qu'une réécriture de γ ne puisse pas être vide et que $PREMIER(\beta) \cap PREMIER(\gamma) = \emptyset$. Sous cette condition, la séquence $[\beta]$ se traduit, lors de l'écriture de l'analyseur, par une instruction `if` dont la condition porte sur l'unité lexicale courante et un des ensembles $PREMIER(\beta)$ ou $PREMIER(\gamma)$, au choix.

• Les parenthèses permettent de mettre en facteur des parties communes à plusieurs termes d'une disjonction : « $\alpha \; (\; \beta \mid \gamma \;) \; \delta$ » représente la même chose que « $\alpha \; \beta \; \delta \mid \alpha \; \gamma \; \delta$ ».

La séquence « $\alpha \; (\; \beta \mid \gamma \;) \; \delta$ » se traduit facilement, lors de l'écriture de l'analyseur syntaxique, à la condition que $PREMIER(\beta) \cap PREMIER(\gamma) = \emptyset$.

Erreurs de syntaxe. La rencontre d'une erreur dans le texte source doit provoquer l'affichage d'un message adéquat et *l'abandon de la compilation*. Une fonction auxiliaire, appelée `erreur` sur nos exemples, prend en charge ces deux opérations.

4 Dictionnaire des identificateurs

Le *dictionnaire des identificateurs* est une structure de données associant à chaque identificateur « connu » (une chaîne de caractères) un ensemble d'attributs :

- la *classe*, parmi `VARIABLE_GLOBALE`, `VARIABLE_LOCALE`, `ARGUMENT`, `FONCTION`[1] (ces noms représentent des constantes entières à définir,
- le *type*, parmi `ENTIER` et `TABLEAU_ENTIERS` (des constantes entières à définir),
- l'*adresse*, un entier identifiant une cellule de la mémoire de la machine virtuelle,
- le *complément*, un nombre entier dont l'interprétation dépend de la classe et du type : la dimension, dans le cas d'un tableau, le nombre d'arguments, dans le cas d'une fonction.

Dictionnaire global et local. Lorsque le compilateur examine l'intérieur d'une fonction, deux dictionnaires coexistent : le *dictionnaire local* contient les identificateurs locaux à la fonction, le *dictionnaire global* les autres identificateurs. Lorsque que le compilateur examine les éléments extérieurs aux fonctions, seul le dictionnaire global existe.

Le dictionnaire global est créé au début de la compilation, il existe jusqu'à la fin de celle-ci, il ne fait que grandir. *Un* dictionnaire local est créé, vide, chaque fois que commence la compilation d'une fonction. Ce dictionnaire existe durant la compilation de la fonction, et est détruit lorsque cette compilation se termine.

La recherche d'un identificateur dans le(s) dictionnaire(s) se produit dans trois circonstances différentes :

- lorsque le compilateur traite la partie exécutable (c.-à-d les instructions) d'une fonction. La recherche se fait alors *d'abord* dans le dictionnaire local, *ensuite* dans le dictionnaire global. En principe, une telle recherche doit aboutir, son échec déclenche une erreur « *Identificateur non déclaré* » ;
- lorsque le compilateur traite la partie déclarative d'une fonction. La recherche se fait alors *uniquement* dans le dictionnaire local. En principe, une telle recherche doit échouer, son succès déclenche une erreur « *Identificateur déjà déclaré dans cette fonction* » ;
- lorsque le compilateur se trouve en dehors de toute fonction (il est donc certainement en train de compiler une déclaration globale). La recherche se fait alors dans le dictionnaire global, qui est le seul à exister à ce moment. En principe une telle recherche doit échouer, son succès déclenche une erreur « *Identificateur déjà déclaré* ».

Implémentation. Une implémentation, simple mais suffisante, du dictionnaire : un tableau de couples (*identificateur*, *attributs*) contrôlé par deux indices `base` et `sommet` comme le montre la figure 1.

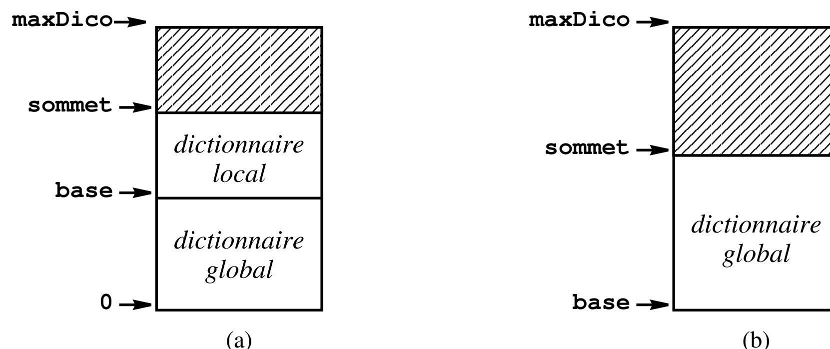

FIG. 1 – Dictionnaires, quand on est à l'intérieur (a) et à l'extérieur (b) des fonctions

Les ajouts dans le dictionnaire se font toujours dans la case d'indice `sommet`, en incrémentant cet indice. Les recherches se font :

- durant la compilation d'une partie exécutable, en parcourant *à reculons* les indices `[0,sommet[`,
- dans une partie déclarative, en parcourant les indices `[base,sommet[`.

Avec cette implémentation des dictionnaires, les opérations de création d'un dictionnaire local (lorsque le compilateur « rentre » dans une fonction) et de destruction du dictionnaire local (lorsque le compilateur « sort » de la fonction) se réduisent à des opérations élémentaires sur les indices `base` et `sommet` :

- entrée dans une fonction : `base` ← `sommet ;`
- sortie d'une fonction : `sommet` ← `base ; base` ← `0 ;`

A quel moment le compilateur entre-t-il dans une fonction et à quel moment en sort-il ?

[1] On notera que les noms des tableaux et des fonctions sont toujours des identificateurs globaux, tandis que les noms des arguments sont toujours des identificateurs locaux.

– le compilateur « entre » dans la fonction lorsque l'unité lexicale courante est la parenthèse ouvrante qui suit immédiatement le nom de la fonction. En effet, ce dernier est un identificateur global, tandis qu'après la parenthèse on trouve, le cas échéant, les arguments qui sont déjà des identificateurs locaux, puis les éventuelles variables locales ;
– le compilateur « sort » de la fonction lorsque l'unité lexicale courante est le symbole (`'}'`, `end`, etc.) qui termine la partie exécutable de la fonction.

5 Production de code

L'analyseur devient un compilateur par l'*ajout* (uniquement des ajouts !) aux endroits pertinents des opérations, généralement très simples, qui génèrent les codes-machine constituant le programme compilé.

Puisque dans notre cas la compilation ne sera pas suivie d'une phase d'édition de liens ou de chargement :
– le compilateur dépose le code produit directement dans l'espace mémoire où celui-ci se trouvera durant son exécution par la machine virtuelle ;
– la convention sur le point d'entrée (l'instruction par laquelle l'exécution du programme commence) doit être connue du compilateur. On décidera, comme en C, que l'exécution commence par la fonction qui a un certain nom convenu (par exemple, `main`). A la fin de la compilation, ce nom sera recherché dans le dictionnaire et ne pas le trouver sera une erreur.

Le résultat de la compilation est constitué de quatre éléments :

– une suite de codes (cf. table 1 à la page 9), rangés au début de la mémoire de la machine cible, constituant la traduction en langage machine du programme source,

– un entier `TC` (pour *Taille du Code*) qui indique le nombre de cases occupées par ces codes.

  `TC` est la valeur finale d'un indice qui, pendant la compilation, indique constamment la « prochaine case à garnir » : initialisé à 0, il augmente de 1 ou de 2 à chaque génération d'une instruction machine,

– un entier `PE` (pour *Point d'Entrée*) qui est l'indice de la cellule mémoire contenant la première instruction à exécuter.

– un entier `TEG` (pour *Taille de l'Espace Global*) qui exprime la somme des tailles des variables globales du programme.

  `TEG` est la valeur finale d'un indice qui, pendant la compilation, indique constamment l'adresse de la prochaine variable globale qui sera déclarée. Initialisé à zéro, il augmente de 1 à l'occasion de la déclaration d'une variable entière, et de n à l'occasion de la déclaration d'un tableau de n éléments.

Important. Il vous est fermement recommandé d'écrire une fonction auxiliaire pour lister, *avec les mnémoniques en clair*, le code produit par le compilateur. Écrire cette fonction vous fera gagner du temps, car sans elle vous seriez obligés de déboguer ensemble le compilateur et la machine cible, ce qui est beaucoup trop compliqué.

6 La machine *M*

La mémoire de la machine M est constituée de cellules contiguës identiques. Dans notre implémentation, ce sera un tableau de `int`. La figure 2 montre la structure générale de la mémoire.

Les « pointeurs » (en réalité des indices) suivants jouent un rôle capital pendant le fonctionnement de la machine M :

`CO` (*Compteur Ordinal*) indique la cellule contenant la prochaine instruction à exécuter.

`BEG` (*Base de l'Espace Global*) indique la première cellule de l'espace réservé aux variables globales (autrement dit, `BEG` « pointe » la variable globale d'adresse 0).

`BEL` (*Base de l'Espace Local*) indique la cellule autour de laquelle est organisé l'espace local de la fonction en cours d'exécution ; la valeur de `BEL` change lorsque l'activation d'une fonction commence ou finit.

`SP` (*Sommet de la Pile*) indique constamment le sommet de la pile, ou plus exactement la première cellule libre au-dessus de la pile. `SP` est aussi le nombre total de cellules occupées dans la mémoire.

La table 1, à la page 9, donne la liste des op-codes de la machine M.

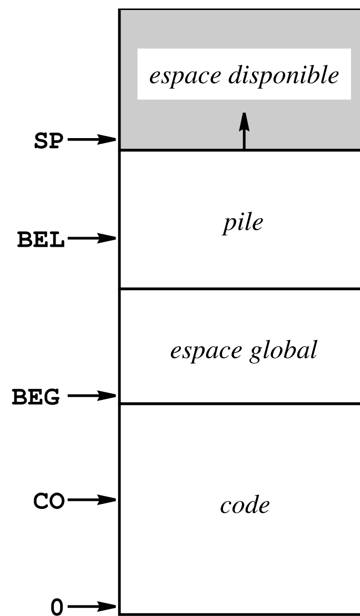

Fig. 2 – Structure générale de la mémoire de la machine M

Espace local d'une fonction. Voyez la figure 3. La création et la destruction de l'espace local des fonctions a lieu à quatre moments bien déterminés : l'activation de la fonction, d'abord du point de vue de la fonction appelante (1), puis de celui de la fonction appelée (2), ensuite le retour, d'abord du point de vue de la fonction appelée (3) puis de celui de la fonction appelante (4) :

1. La fonction appelante réserve un emplacement vide sur la pile, où sera déposé le résultat de la fonction, puis elle empile les valeurs des arguments effectifs. Ensuite, elle exécute une instruction `APPEL` (qui empile l'adresse de retour).
2. La fonction appelée empile le « `BEL` courant » (qui est en train de devenir « `BEL` précédent »), prend la valeur de `SP` pour `BEL` courant, puis alloue l'espace local.

   Pendant la durée de l'activation de la fonction :
   - les variables locales sont atteintes à travers des déplacements (positifs ou nuls) relatifs à `BEL` : $0 \leftrightarrow$ première variable locale, $1 \leftrightarrow$ deuxième variable locale, etc.
   - les arguments formels sont également atteints à travers des déplacements (négatifs, cette fois) relatifs à `BEL` : $-3 \leftrightarrow (n-1)$-ème argument (le dernier), $-4 \leftrightarrow (n-2)$-ème argument, etc.
   - la cellule où la fonction doit déposer son résultat est atteinte elle aussi à travers un déplacement relatif à `BEL`. Ce déplacement est $-(n+3)$, n étant le nombre d'arguments formels.
     Cela suppose l'accord entre la fonction appelante et la fonction appelée au sujet du nombre d'arguments de la fonction appelée.

   L'ensemble de toutes ces valeurs forme la structure de contrôle de la fonction en cours. Au-delà de cet espace, la pile est utilisée pour le calcul des expressions courantes.
3. Terminaison du travail : la fonction appelée remet `BEL` et `SP` comme ils étaient lorsqu'elle a été activée, puis effectue une instruction `RETOUR`.
4. Reprise du calcul interrompu par l'appel. La fonction appelante libère l'espace occupé par les valeurs des arguments effectifs. Elle se retrouve alors avec une pile au sommet de laquelle est le résultat de la fonction qui vient d'être appelée. Globalement, la situation est la même qu'après une opération arithmétique : les opérandes ont été dépilés et remplacés par le résultat.

Qui fait le ménage, au retour de la fonction ? Il y a une différence importante entre les arguments d'une fonction et ses (autres) variables locales :
- les arguments sont installés dans la pile par la fonction appelante, car ce sont les valeurs d'expressions qui doivent être évaluées dans le contexte de la fonction appelante. Il est donc naturel de donner à cette dernière la responsabilité de les enlever de la pile, au retour de la fonction appelée,
- les variables locales sont allouées par la fonction appelée, qui est la seule à en connaître le nombre ; c'est donc cette fonction qui doit les enlever de la pile, lorsqu'elle se termine (l'instruction `SORTIE` fait ce travail).

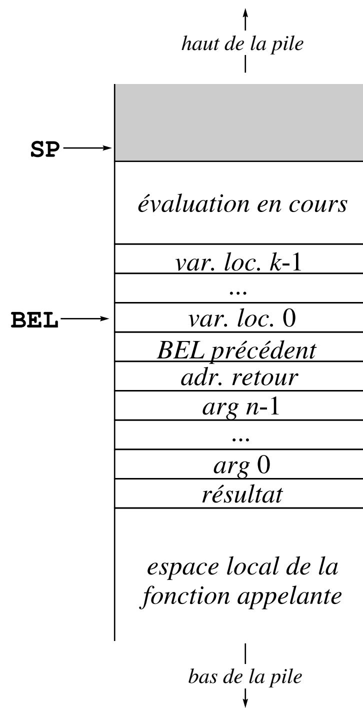

Fig. 3 – Espace local d'une fonction

Réalisation de la machine M. Plus le programme est simple, plus la simulation est crédible. Exemple :

```
for (;;) {
    switch (memoire[CO++]) {
        case EMPC:
            memoire[SP++] = memoire[CO++];
            break;
        ...
        case ADD:
            val = memoire[--SP];
            memoire[SP - 1] += val;
            break;
        ...
    }
}
```

Lancement de l'exécution. L'initialisation de la machine se fait à partir des valeurs `TC` (*Taille du Code*), `PE` (*Point d'Entrée*) et `TEG` (*Taille de l'Espace Global*) qui sont les résultats de la compilation :

```
BEG = TC;           // L'espace global se place immédiatement après le code
SP  = BEG + TEG;    // La pile fait suite à l'espace global
BEL = -1;           // Le programme principal n'a pas d'espace local
CO  = PE;           // L'exécution démarre au point d'entrée
```

TAB. 1 – Les instructions de la machine M

opcode	*opérande*	*explication*
EMPC	valeur	*EMPiler Constante.* Empile la valeur indiquée.
EMPL	adresse	*EMPiler la valeur d'une variable Locale.* Empile la valeur de la variable déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
DEPL	adresse	*DEPiler dans une variable Locale.* Dépile la valeur qui est au sommet et la range dans la variable déterminée par le déplacement relatif à BEL donné par adresse (entier relatif).
EMPG	adresse	*EMPiler la valeur d'une variable Globale.* Empile la valeur de la variable déterminée par le déplacement (relatif à BEG) donné par adresse.
DEPG	adresse	*DEPiler dans une variable Globale.* Dépile la valeur qui est au sommet et la range dans la variable déterminée par le déplacement (relatif à BEG) donné par adresse.
EMPT	adresse	*EMPiler la valeur d'un élément de Tableau.* Dépile la valeur qui est au sommet de la pile, soit i cette valeur. Empile la valeur de la cellule qui se trouve i cases au-delà de la variable déterminée par le déplacement (relatif à BEG) indiqué par adresse.
DEPT	adresse	*DEPiler dans un élément de Tableau.* Dépile une valeur v, puis une valeur i. Ensuite range v dans la cellule qui se trouve i cases au-delà de la variable déterminée par le déplacement (relatif à BEG) indiqué par adresse.
ADD		*ADDition.* Dépile deux valeurs et empile le résultat de leur addition.
SOUS		*SOUStraction.* Dépile deux valeurs et empile le résultat de leur soustraction.
MUL		*MULtiplication.* Dépile deux valeurs et empile le résultat de leur multiplication.
DIV		*DIVision.* Dépile deux valeurs et empile le quotient de leur division euclidienne.
MOD		*MODulo.* Dépile deux valeurs et empile le reste de leur division euclidienne.
EGAL		Dépile deux valeurs et empile 1 si elles sont égales, 0 sinon.
INF		*INFerieur.* Dépile deux valeurs et empile 1 si la première est inférieure à la seconde, 0 sinon.
INFEG		*INFerieur ou EGal.* Dépile deux valeurs et empile 1 si la première est inférieure ou égale à la seconde, 0 sinon.
NON		Dépile une valeur et empile 1 si elle est nulle, 0 sinon.
LIRE		Obtient de l'utilisateur un nombre et l'empile
ECRIV		*ECRIre Valeur.* Extrait la valeur qui est au sommet de la pile et l'affiche
SAUT	adresse	*Saut inconditionnel.* L'exécution continue par l'instruction ayant l'adresse indiquée.
SIVRAI	adresse	*Saut conditionnel.* Dépile une valeur et si elle est non nulle, l'exécution continue par l'instruction ayant l'adresse indiquée. Si la valeur dépilée est nulle, l'exécution continue normalement.
SIFAUX	adresse	Comme ci-dessus, en permutant nul et non nul.
APPEL	adresse	*Appel de sous-programme.* Empile l'adresse de l'instruction suivante, puis fait la même chose que SAUT.
RETOUR		*Retour de sous-programme.* Dépile une valeur et continue l'exécution par l'instruction dont c'est l'adresse.
ENTREE		*Entrée dans un sous-programme.* Empile la valeur courante de BEL, puis copie la valeur de SP dans BEL.
SORTIE		*Sortie d'un sous-programme.* Copie la valeur de BEL dans SP, puis dépile une valeur et la range dans BEL.
PILE	nbreMots	*Allocation et restitution d'espace dans la pile.* Ajoute nbreMots, qui est un entier positif ou négatif, à SP
STOP		La machine s'arrête.