Université Grenoble Alpes **UFR IM^{2}AG**
DU ISN-C2.3 **Année 2019-20**

Langages et Compilation : génération de code

1 Le langage source

Dans un premier temps, on reprend le langage `while`, sans blocs ni procédures :

p : := d ; c
d : := var x | d ; d
c : := x := e | c ; c | si e alors c sinon c | tantque e c
e : := n | x | e + e | e et e | e = e | non e

2 La machine cible (M)

On considère ici une machine cible virtuelle inspirée du processeur ARM :

— Il s'agit d'une machine à registres (notés Ri), dont le nombre est supposé non borné. Trois registres jouent un rôle particulier : `PC` est le compteur programme, `SP` contient l'adresse du sommet de pile (adresse du dernier octet occupé) et `FP` l'adresse de la base de l'environnement courant.

— Les adresses, les instructions et les entiers relatifs sont codés sur 4 octets.

— Le jeu d'instructions fournit des opérations arithmétiques et logiques `OPER`, des instructions de transfert registre/mémoire (`LD` et `ST`), de comparaison de registres (`CMP`), de chargement de registre `MOV`, de branchements conditionnels (`BRANCH`), et d'appel et retour de procédures (`CALL` et `RET`). Les opérations `OPER`, `MOV` et `CMP` modifient les codes conditions utilisés par les opérations `BRANCH`.

La sémantique informelle de ces instructions est résumée ci-dessous :

instruction	sémantique informelle
`MOV Ri, Rj`	Ri ← Rj
`MOV Ri, val`	Ri ← val
`OPER Ri, Rj, Rk`	Ri ← Rj oper Rk
`OPER Ri, Rk, val`	Ri ← Rj oper val
`CMP Ri, Rj`	Ri-Rj
`LD Ri, [adr]`	Ri ← Mem[adr]
`ST Ri, [adr]`	Mem[adr] ← Ri
`BRANCH cond label`	si cond alors PC ← adresse de l'instruction étiquetée label sinon PC ← PC + 4
`CALL label`	sauvegarde de l'adresse de retour dans la pile puis branchement à la procédure commençant à l'instruction étiquetée label
`RET`	retour de procédure en dépilant le sommet de pile dans PC

Dans ce tableau :

— `val` est une valeur entière et `adr` est une adresse de la forme :
  adr : := Ri + Rj | Ri + val | Ri | val

— `OPER` = {`ADD`, `SUB`, `AND`, ...}

— `BRANCH cond` = {`BA`, `BEQ`, `BNE`, `BGT`, ...}

3 Génération de code

Toutes les variables du programmes sont allouées sur la pile. L'adresse d'une variable `x` dans cette pile est alors représentée par l'expression `FP - depl` dans laquelle :

— `FP` contient l'adresse de la base de l'environnement dans lequel `x` est déclarée ;

— `depl` est un *déplacement* calculé statiquement et mémorisé dans la table des symboles.

On note Code* l'ensemble des séquences de codes pour la machine M, et @ l'opérateur de concaténation. Nous définissons alors trois fonctions de génération de code :

Commandes : `GenCodeInst` : Inst → Code*
`GenCodeInst(c)` *calcule le code* `C` *permettant d'exécuter la commande* `c`.

Expressions arithmétiques : `GenCodeAExp` : AExp → Code*× ℕ
`GenCodeAExp(e)` *produit un couple (*`C`, `i`*) où* `C` *est le code permettant de calculer la valeur de* `e` *et de la mémoriser dans le registre* `Ri`.

Expressions booléennes : `GenCodeBExp` : BExp× $\mathcal{L}$abel× $\mathcal{L}$abel → Code*
`GenCodeBExp(b, lvrai, lfaux)` *produit le code* `C` *permettant de calculer la valeur de* `b` *et d'effectuer un branchement sur* `lvrai` *lorsque cette valeur est "vrai" et sur* `lfaux` *sinon.*

Ces fonctions utilisent les fonctions auxiliaires suivantes :

AllouerRegistre	:	→ ℕ	alloue un nouveau registre
nouvelleEtiq	:	→ ℕ	fournit une nouvelle étiquette
GetSymbDepl	:	Noms → ℕ	renvoie le déplacement (`depl`) associé à la variable spécifiée.

3.1 Génération de code pour les expressions arithmétiques

`GenCodeAExp(x)`	=	i ← `AllouerRegistre()` ; k ← `GetSymbDepl(x)` ; return ((`LD` Ri,[FP-k]),i)
`GenCodeAExp(n)`	=	i←`AllouerRegistre()` ; return ((`MOV` Ri,n),i)
`GenCodeAExp(`a_1 `+` a_2`)`	=	(C_1,i_1)←`GenCodeAExp(`a_1`)` ; (C_2,i_2)←`GenCodeAExp(`a_2`)` ; k←`AllouerRegistre()` ; return ((C_1 @ C_2 @ `ADD` Rk, Ri_1,Ri_2),k)

3.2 Génération de code pour les expressions booléennes

`GenCodeBExp`(a_1=a_2,lvrai,lfaux)	=	(C_1,i_1)←`GenCodeAExp`(a_1) ; (C_2,i_2)←`GenCodeAExp`(a_2) ; return (C_1 @ C_2 @ `CMP` Ri_1, Ri_2 `BEQ` lvrai `BA` lfaux)
`GenCodeBExp`(b_1 et b_2),lvrai,lfaux)	=	l←`nouvelleEtiq()` ; return (`GenCodeBExp`(b_1,l,lfaux) @ l : @ `GenCodeBExp`(b_2,lvrai,lfaux))
`GenCodeBExp`(`NON` b,lvrai,lfaux)	=	`GenCodeBExp`(b,lfaux,lvrai)

3.3 Génération de code pour les instructions

`GenCodeInst` (`x := a`)	=	(C,i)←`GenCodeAExp`(a) ; k←`GetSymbDepl`(x) ; return (C @ `ST` Ri, [FP-k])
`GenCodeInst` (c_1 `;` c_2)	=	C_1 ← `GenCodeInst`(c_1) ; C_2 ← `GenCodeInst`(c_2) ; return (C_1 @ C_2)

`GenCodeInst` (`tantque b c`)	=	ldebut←`nouvelleEtiq()` ; lvrai←`nouvelleEtiq()` ; lfaux←`nouvelleEtiq()` ; return (ldebut : @ `GenCodeBExp`(b,lvrai,lfaux) @ lvrai : @ `GenCodeInst`(c) @ `BA` ldebut @ lfaux :)

`GenCodeInst` (`si b alors`	c_1 `sinon` c_2)=
	lsuivant←`nouvelleEtiq()` ;
	lvrai←`nouvelleEtiq()` ;
	lfaux←`nouvelleEtiq()` ;
	return (
	`GenCodeBExp`(b,lvrai,lfaux) @
	lvrai : @
	`GenCodeInst`(c_1) @
	`BA` lsuivant @
	lfaux : @
	`GenCodeInst`(c_2) @
	lsuivant :
	)

3.4 Génération de code pour les procédures

On étend maintenant la syntaxe du langage en ajoutant des déclarations et appels de procédures. Cette extension est prise en compte lors de la génération de code de la manière suivante :

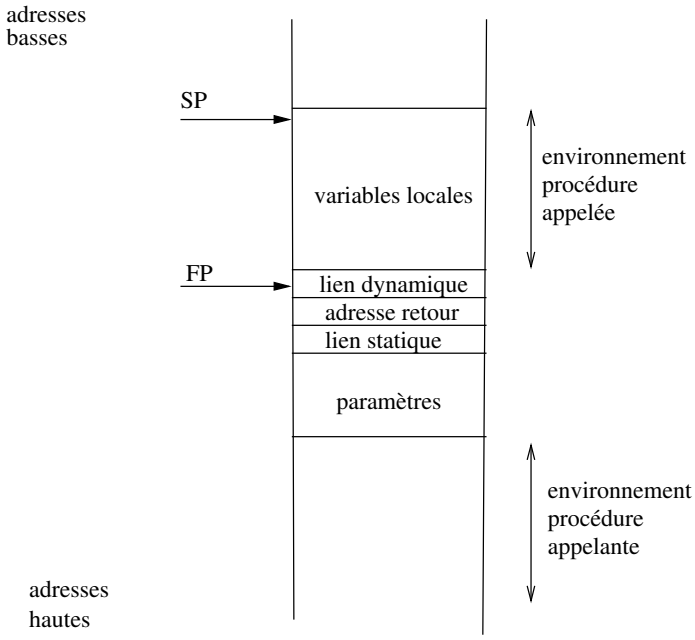
- lors d'un appel de procédure, l'appelant empile les paramètres, le lien statique (l'adresse de l'environnement de définition de la procédure appelée), l'adresse de retour ; il effectue alors l'appel.
- en début de procédure (dans un "*prologue*"), l'appelé empile le lien dynamique (l'adresse de l'environnement de la procédure appelante), et il réserve de la place sur la pile pour ses variables locales.
- en fin de procédure (dans un "*épilogue*"), l'appelé "récupère" l'espace utilisé sur la pile par ses variables locales, restaure `FP` avec l'adresse de l'environnement de la procédure appelante (en utilisant le lien dynamique) et dépile l'adresse de retour dans le `PC`.

Organisation de la pile :

Prologue et Epilogue d'une procédure

Le prologue prend en paramètre la taille nécessaire à l'ensemble des variables locales sur la pile (`Taille`). Cette taille dépend du nombre et du type de ces variables locales, elle est donc connue à la compilation.

`prologue(Taille)`	`epilogue`
`ADD SP, SP, -4`	`MOV SP, FP`
`ST FP, [SP]`	`LD FP, [SP]`
`MOV FP, SP`	`ADD SP, SP, +4`
`ADD SP, SP, -Taille`	



Appel d'une procédure

L'appel d'une procédure `p` de paramètre `y` (dont l'adresse est supposée être `FP -8`) est codé par :

Passage par valeur	Passage par adresse	Passage par résultat
`LD R2, [FP-8]`	`ADD R2, FP, -8`	`ADD SP, SP, -4`
`ADD SP, SP, -4`	`ADD SP, SP, -4`	`CALL p`
`ST R2, [SP]`	`ST R2, [SP]`	`LD R2, [SP]`
`CALL p`	`CALL p`	`ST R2, [FP-8]`
`ADD SP, SP, +4`	`ADD SP, SP, +4`	`ADD SP, SP, +4`