Université Joseph Fourier **Polytech'Grenoble**
Département RICM 4 **Année 2009-00**

Langages et Traducteurs

Génération de Code

Le langage source

On reprend la syntaxe (abstraite) du 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

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. Le registre `R0` contient toujours la valeur 0.
- Les adresses, les instructions et les entiers relatifs sont codés sur 4 octets.
- Le jeu d'instructions fournit des opérations arithmétiques et logiques `OPER`, de transfert registre - mémoire (`LD` et `ST`), de comparaison de registres (`CMP`), de branchement inconditionnels (`BRANCH`), et d'appel et retour de procédures (`CALL` et `RET`). Les opérations `OPER` et `CMP` modifient les codes conditions utilisés par les opérations `BRANCH`.

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

instruction	sémantique informelle
`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 label`	si cond alors PC ← adresse de l'instruction étiquetée label sinon PC ← PC + 4
`CALL label`	branchement à la procédure commençant à l'instruction étiquetée label
`RET`	retour de procédure

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` = {`BA`, `BEQ`, `BNE`, `BGT`, ...}

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× ℒabel× ℒabel → 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.

Génération de code pour les instructions

`GenCodeInst (x := e)`	=	`Soit`	(C,i)=`GenCodeAExp`(e), k=`GetSymbDepl`(x)
		`dans`	C‖ `ST` Ri, [FP-k]
`GenCodeInst` (c_1 `;` c_2)	=	`Soit`	C_1 = `GenCodeInst`(c_1), C_2 = `GenCodeInst`(c_2)
		`dans`	C_1 ‖ C_2
`GenCodeInst (tantque e c)`	=	`Soit`	ldebut=`nouvelleEtiq()`, lvrai=`nouvelleEtiq()`, lfaux=`nouvelleEtiq()`
		`dans`	ldebut :‖ `GenCodeBExp`(e,lvrai,lfaux)‖ lvrai :‖ `GenCodeInst`(c)‖ `BA` ldebut‖ lfaux :
`GenCodeInst (si e alors` c_1 `sinon` c_2`)`	=	`Soit`	lsuivant=`nouvelleEtiq()`, lvrai=`nouvelleEtiq()`, lfaux=`nouvelleEtiq()`
		`dans`	`GenCodeBExp`(e,lvrai,lfaux)‖ lvrai : `GenCodeInst`(c_1)‖ `BA` lsuivant ‖ lfaux :‖ `GenCodeInst`(c_2)‖ lsuivant :

Génération de code pour les expressions arithmétiques

GenCodeAExp(x)	=	Soit	i=AllouerRegistre() k=GetSymbDepl(x)
		dans	((LD Ri,[FP-k]),i)
GenCodeAExp(n)	=	Soit	i=AllouerRegistre()
		dans	((ADD Ri,R0,n),i)
GenCodeAExp(e_1 + e_2)	=	Soit	(C_1,i_1)=GenCodeAExp(e_1), (C_2,i_2)=GenCodeAExp(e_2), k=AllouerRegistre()
		dans	((C_1‖C_2‖ ADD Rk, Ri_1,Ri_2),k)

Génération de code pour les expressions booléennes

GenCodeBExp (e_1 = e_2,lvrai,lfaux)	=	Soit	(C_1,i_1)=GenCodeAExp(e_1), (C_2,i_2)=GenCodeAExp(e_2),
		dans	C_1‖C_2‖ CMP Ri_1, Ri_2 BEQ lvrai BA lfaux
GenCodeBExp (e_1 et e_2,lvrai,lfaux)	=	Soit	l=nouvelleEtiq()
		dans	GenCodeBExp(e_1,l,lfaux)‖ l :‖ GenCodeBExp(e_2,lvrai,lfaux)
GenCodeBExp(NON e,lvrai,lfaux)	=		GenCodeBExp(e,lfaux,lvrai)

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.

On détaille ci-après le code machine correspondant à ces différentes opérations.

Instructions CALL et RET

CALL label	RET
ADD R1, CP, +4	LD CP, [SP++]
ADD SP, SP, -4	(adressage post-incrémenté :
ST R1, [SP]	SP est incrémenté de 4 à la fin du LD)
BA label	

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`
`ST FP, [SP]`
`ADD FP, SP, 0`
`ADD SP, SP, -Taille`
	`ADD SP, FP, 0`
	`LD FP, [SP]`
	`ADD SP, SP, +4`

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 SP, SP, -4`
`ST R2, [SP]`
`CALL p`
`ADD SP, SP, +4`
	`ADD R2, FP, -8`
	`ADD SP, SP, -4`
	`ST R2, [SP]`
	`CALL p`
	`ADD SP, SP, +4`
		`ADD SP, SP, -4`
		`CALL p`
		`LD R2, [SP]`
		`ST R2, [FP-8]`
		`ADD SP, SP, +4`

Organisation de la pile :