Génération de code

INFO010 – Théorie des langages – Partie pratique

S. COLLETTE et G. GEERAERTS

Génération de code

Deux questions :

- Quel code générer ? Pour quelle architecture cible ?
  - P-langage pour une P-Machine
- À quel moment de l'analyse faut-il le générer ? Comment spécifier formellement cela ?
  - Grammaires attribuées

P-Machine

Dans le cadre des TPs et du projet, nous allons générer du P-langage.

- Langage intermédiaire développé pour la compilation de Pascal
- Interprété par une P-Machine :
  - Même idée que la JVM
  - C'est une machine à *stack*

Machine à *stack* ?

Idée : Pas de registres de travail, on utilise un *stack* à la place.

Exemple réaliser la somme de 4 et 5

- Dans une machine avec registres :
  - `mov AX, 4 mov BX, 5 add AX, BX`
  - Le résultat est dans `AX`.
- Dans une machine à pile :
  - `push 4 push 5 add`
  - Le résultat est au sommet du *stack*.

P-Machine – Mémoire et registres

La mémoire est divisée en deux parties :

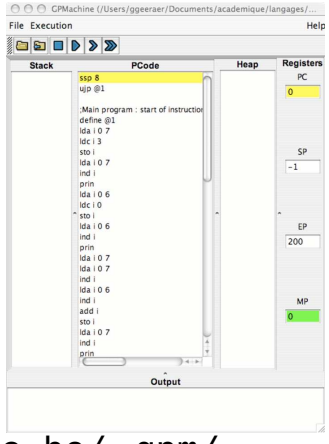
- Zone de code : contient le programme.
  - Le registre PC indique l'adresse de la prochaine instruction.
- Zone de données : divisée en deux parties
  - le *stack* : va de l'adresse 0 (bas) à l'adresse stockée dans le registre SP (compris).
  - le *heap* : va de l'adresse contenue dans le registre EP à la fin de la zone.

P-Machine – La zone de données

Données de la pile	Mémoire inutilisée	Données du tas
0 ↑ SP		↑ Max EP

GPMachine

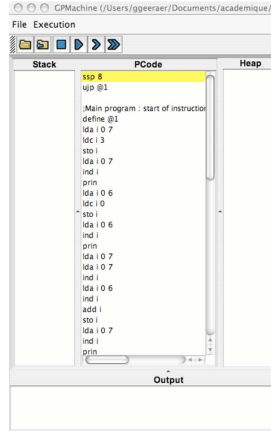
- P-Machine développée aux FUNDP.
- Licence GPL.
- Interface pratique.
- *Cross-platform* : écrite en Java
- Mode d'emploi sur la page des TPs.



`http ://www.info.fundp.ac.be/~gpm/`

GPMachine – 2

- Cellules du *stack* et du *heap* typées (entier ou booléen).
- Code à exécuter fourni dans un fichier texte.
- lignes de commentaires = lignes commençant par ;

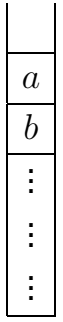


GPMachine – Instructions du *stack*

- `ldc i` q: *Push* de la constante entière q.
- `ldc b` q: idem avec booléen
- `ldo i` q: *Push* de l'entier qui est à l'adresse q.
- `ldo b` q: idem avec booléen
- `sro i` q: *Pop* et met la valeur à l'adresse q
- `sro b` q: idem avec booléen
- `pop` : *Pop*

GPMachine – Instructions arith. / logiq

- `div i` : *Pop* a et b, et *push* b/a
- `mul i` : idem avec $b * a$.
- `add i` : idem avec $b + a$.
- `sub i` : idem avec $b - a$.
- `and b` : idem avec $a \wedge b$
- `or b` : idem avec $a \vee b$.
- `neg i` : *Pop* a et *push* $-a$
- `not b` : *Pop* a et *push* $\neg a$



a et b doivent être entiers pour `div`, `mul`, `add`, `sub`, `neg`, et booléens pour `and`, `or`, `not`.

GPMachine – Instructions d'I/O

- `prin` : Affiche la valeur entière au sommet de la pile sur la sortie et la supprime de la pile.
- `read` : Lit une valeur entière sur l'entrée et la place au sommet de la pile.

Exercice 1

Écrivez le code qui calcule et affiche la valeur de :

$$(3 + x) * (9 - y)$$

où x est une valeur lue sur *input* et y est la valeur qui se trouve en mémoire à l'adresse 0 (on verra plus tard comment placer correctement une valeur en mémoire).

Exercice 1 – Correction

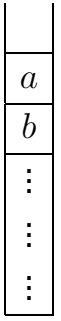
```
; calcul de 3+x
ldc i 3
read
add i
; calcul de 9 - y
ldc i 9
ldo i 0
sub i
```

```
; calcul du
; produit
mul i
; affichage
prin
```

Suppose qu'une valeur a été correctement placée à l'adresse 0.

GPMachine – Comparaisons

- `geq i` : *Pop* a et b et *push* la valeur booléenne correspondant à $b \geq a$.
- `leq i` : Même chose avec $b \leq a$.
- `les i` : Même chose avec $b < a$.
- `grt i` : Même chose avec $b > a$.
- `equ i` : Même chose avec $b = a$.
- `neq i` : Même chose avec $b \neq a$.



a et b doivent être entiers.

GPMachine – Branchements

- `define` L: Place l'étiquette L.
- `ujp` L: Saut non conditionnel. La prochaine instruction sera celle qui suit `define L`.
- `fjp` L: Saut conditionnel. *Pop* un booléen. Si *faux*, la prochaine instruction sera celle qui suit `define L`. Sinon, continue normalement.

Exercice 2

Écrire le code qui affiche toutes les valeurs impaires dans l'intervalle $[7, 31]$. Pour ce faire vous aurez besoin de l'instruction `dpl i` qui duplique le sommet (entier) de la pile.

Exercice 2 – Correction

```
ldc i 7

;debut boucle
define debutboucle

; test boucle
dpl i
ldc i 31
leq i
fjp fin
```

```
; corps boucle
dpl i
prin
ldc i 2
add i
ujp debutboucle

; fin boucle
define fin
stp
```


GPMachine – Gestion mémoire

- Quand on a des variables statiques dans le programme source, il faut se réserver de la place sur le *stack* pour stocker leurs valeurs.
- L'instruction `ssp` q donne au *stack* la taille q.
- À n'appeler qu'une fois au début du programme sinon on perd le contenu du *stack*.
- Quand on appelle `ssp` q au début du programme, on se réserve donc les cases d'adresses $0, \ldots, q-1$.

Exercice 3

Écrire le code qui :

- Se réserve de la place pour deux variables que nous appelerons a et b.
- Initialise a et puis b avec des valeurs lues sur *input*.
- Ajoute 5 à a.
- Divise b par 2.
- Si $a > b$, affiche a, sinon affiche b.

Tout au long de l'exercice, les valeurs stockées dans les cases correspondant à a et b devront rester cohérentes.

Exercice 3 – Correction (1)

```
; a == adresse 0
; b == adresse 1
ssp 2

; Initialisation
read
sro i 0
read
sro i 1
```

```
; a+=5
ldo i 0
ldc i 5
add i
sro i 0

; b/=2
ldo i 1
ldc i 2
div i
sro i 1
```

Exercice 3 – Correction (2)

Structure d'un **if** :

⋮	Évaluation de la condition
`fjp else`	Saut si condition fausse
⋮	Code **then**
`ujp fin`	Pour éviter le **else**
`define else`	
⋮	Code **else**
`define fin`	Fin du **if**

Exercice 3 – Correction (3)

```
; if (a>b)
ldo i 0
ldo i 1
grt i
fjp else

; branche then
ldo i 0
prin
ujp fin

; branche else
define else
ldo i 1
prin

; fin
define fin
stp
```


Grammaires attribuées

Il reste à faire le lien avec les grammaires. . . ce sont les Grammaires attribuées.

- But : décorer la grammaire avec des attributs, de manière à générer «quelque chose».
  - Calculer la valeur d'une expression arithmétique. . .
  - Générer du code. . .
- Cela revient à appliquer un traîtement à l'arbre de dérivation de la *string*.

Grammaires attribuées (2)

- On distingue deux types d'attributs :
  - Les attributs hérités : dont la valeur dépend des valeurs des attributs du père et des frères d'un nœud.
  - Les attributs synthétisés : dont la valeur dépend des valeurs des attributs des fils d'un nœud.
- Les attributs sont donc de la forme :

$$b = f(c_1, c_2, \ldots, c_n)$$

où b est le nouvel attribut et $c_1 \ldots c_n$ sont des attributs quelconques de la production. f peut être une fonction *stricto sensu* ou une procédure.

Grammaires attribuées – Exemple

- Considérons un fragment de grammaire qui permet de calculer des expressions arithmétiques.
- Ajoutons un attribut val à ses non-terminaux !

$$
\begin{array}{lll}
E \rightarrow E_1 + T & \{E.val \leftarrow E_1.val + T.val\} \\
\rightarrow T & \{E.val \leftarrow T.val\} \\
T \rightarrow \texttt{INTLIT} & \{T.val \leftarrow \texttt{INTLIT}.val\}
\end{array}
$$

Exercice 4

- Réécrivez la grammaire suivante en tenant compte de la priorité et de l'associativité des opérateurs :

E ⟶ E op E | (E) | nb

op ⟶ + | - | * | /

- Associez à cette grammaire les règles et attributs nécessaires pour calculer la valeur d'une expression E.
- Dérécursifiez cette grammaire et adaptez les règles.

Exercice 4 – Solution (1)

$$
\begin{array}{lll}
E \longrightarrow E_1 \texttt{ + } T & \{E.val \leftarrow E_1.val + T.val\} \\
\longrightarrow E_1 \texttt{ - } T & \{E.val \leftarrow E_1.val - T.val\} \\
\longrightarrow T & \{E.val \leftarrow T.val\} \\
T \longrightarrow T_1 \texttt{ * } F & \{T.val \leftarrow T_1.val * F.val\} \\
\longrightarrow T_1 \texttt{ / } F & \{T.val \leftarrow T_1.val / F.val\} \\
\longrightarrow F & \{T.val \leftarrow F.val\} \\
F \longrightarrow (E) & \{F.val \leftarrow E.val\} \\
\longrightarrow \texttt{nb} & \{F.val \leftarrow nb.val\}
\end{array}
$$

Exercice 4 – Solution (2)

$$
\begin{array}{lll}
E \longrightarrow T & \{E'.h \leftarrow T.val\} \\
E' & \{E.val \leftarrow E'.s\} \\
E' \longrightarrow \texttt{+} \\
T & \{E'_1.h \leftarrow E'.h + T.val\} \\
E'_1 & \{E'.s \leftarrow E'_1.s\} \\
\longrightarrow \texttt{-} \\
T & \{E'_1.h \leftarrow E'.h - T.val\} \\
E'_1 & \{E'.s \leftarrow E'_1.s\} \\
\longrightarrow \varepsilon & \{E'.s \leftarrow E'.h\}
\end{array}
$$

Exercice 4 – Solution (3)

T	$\longrightarrow$	F	$\{T'.h \leftarrow F.val\}$
		T'	$\{T.val \leftarrow T'.s\}$
T'	$\longrightarrow$	*	
		F	$\{T'_1.h \leftarrow T'.h * F.val\}$
		T'_1	$\{T'.s \leftarrow T'_1.s\}$
	$\longrightarrow$	/	
		F	$\{T'_1.h \leftarrow T'.h/F.val\}$
		T'_1	$\{T'.s \leftarrow T'_1.s\}$
	$\longrightarrow$	ε	$\{T'.s \leftarrow T'.h\}$
F	$\longrightarrow$	(E)	$\{F.val \leftarrow E.val\}$
	$\longrightarrow$	nb	$\{F.val \leftarrow nb.val\}$

Exercice 5

Voici une règle pour définir le **if** d'un langage impératif :

⟨Si⟩	$\longrightarrow$	**if**⟨Cond⟩**then**⟨Code⟩⟨SuiteSite⟩
⟨SuiteSite⟩	$\longrightarrow$	**else**⟨Code⟩**endif**
⟨SuiteSite⟩	$\longrightarrow$	**endif**

Indiquez quelles instructions en P-langage vous devez générer pour traduire ce code dans un compilateur en descente récursive. Faites l'hypothèse que les règles correspondant à ⟨Cond⟩ et ⟨Code⟩ sont déjà décorées pour générer le code correct.

Exercice 6 – Solution

cpt est un compteur global initialisé à 0.

⟨Si⟩	$\longrightarrow$	**if** ⟨Cond⟩	{cpt++}
			{Produire(`fjp else`cpt)}
		then ⟨Code⟩⟨SuiteSite⟩	
⟨SuiteSite⟩	$\longrightarrow$	**endif**	{Produire(`define else`cpt)}
	$\longrightarrow$	**else**	{Produire(`ujp fin`cpt)}
			{Produire(`define else`cpt)}
		⟨Code⟩ **endif**	{Produire(`define fin`cpt)}