Compilation

CM5 - Génération de code

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

COMP, M1 info

Plan

Plan

Rappels

Analyse sémantique : 2 rôles

- Vérifications
  - table des symboles
  - vérification de types
- traduction vers le langage cible
  - = génération de code
  - 2 aspects :
    1. impératif : préservation de la sémantique
       que ça fasse ce que l'on veut
    2. secondaire : code généré efficace (optimisation)
       que ça aille aussi vite que possible
       que ça consomme le moins de ressources possible

Motivation du code intermédiaire

Les 2 aspects de la traduction motivent un découpage de la génération de code en 2 phases :

$$\text{AST décoré} \xrightarrow{1} \text{code intermédiaire} \xrightarrow{2} \text{code cible}$$

1. se concentre sur la traduction proprement dite
   - machine intermédiaire offrant une abstraction des différentes machines
     $\rightarrow$ processeur + mémoire + pile
   - la programmation de cette phase doit être assez simple pour garantir la préservation de la sémantique
     $\rightarrow$ approche compositionnelle
   - on accepte que le code produit soit inefficace
2. spécialisation du code vers une machine particulière
   - traduction relativement simple : mot à mot

Optimisations possible à chaque niveau

Motivation du code intermédiaire

Autre bénéfice du découpage en 2 phases :

- factorisation d'un compilateur à l'autre

. . . . . .

Nature de la machine intermédiaire

soit machine au sens classique

- instructions, emplacements mémoires
- mais abstrait des détails et contraintes
- ex : machine 3-adresses

soit langage pour lequel il existe déjà un compilateur

- ……

- implique 2 phases d'analyse syntaxique

Nature de la machine intermédiaire

soit machine au sens classique

- instructions, emplacements mémoires
- mais abstrait des détails et contraintes
- ex : machine 3-adresses

soit langage pour lequel il existe déjà un compilateur

- ……

- implique 2 phases d'analyse syntaxique

Plan

Machine intermédiaire

- Machine 3-adresses
  - exécutant du code 3-adresses
  - code 3-adresses = liste d'instructions 3-adresses
- instruction 3-adresses : (op, x, y, z)
  - op : opération élémentaire, type d'instruction
  - x, y, z : emplacement mémoires ou registres
    indifférenciés à ce stade
  - en général : x = résultat, y, z = opérandes
  - en fait : *au plus* 3-adresses
    toutes les opérations n'utilisent pas les 3 adresses

Opérateurs arithmétiques

Opérandes et résultats de type entier

nom	instruction	notation	arité
addition	$(+, x, y, z)$	$x = y + z$	binaire
soustraction	$(-, x, y, z)$	$x = y - z$	binaire
multiplication	$(*, x, y, z)$	$x = y * z$	binaire
division	$(/, x, y, z)$	$x = y / z$	binaire
modulo	$(\%, x, y, z)$	$x = y \% z$	binaire
- unaire	$(-unaire, x, y, \_)$	$x = -y$	unaire
constante ($N \in \mathbb{N}$)	$(const\ N, x, \_, \_)$	$x = N$	0-aire

Opérateurs logiques

Opérandes et résultats de type booléen

nom	instruction	notation	arité
et	(and, x, y, z)	x = y `and` z	binaire
ou	(or, x, y, z)	x = y `or` z	binaire
non	$(not, x, y, \_)$	x = `not` y	unaire
vrai	$(const\ true, x, \_, \_)$	x = `true`	0-aire
faux	$(const\ false, x, \_, \_)$	x = `false`	0-aire

Opérateurs de comparaison

Opérandes de type entier et résultat de type booléen

nom	instruction	notation	arité
eq	(eq, x, y, z)	$x = y == z$	binaire
neq	(neq, x, y, z)	$x = y != z$	binaire
leq	(leq, x, y, z)	$x = y <= z$	binaire
lt	$(lt, x, y, \_)$	$x = y < z$	binaire
geq	(geq, x, y, z)	$x = y >= z$	binaire
gt	$(gt, x, y, \_)$	$x = y > z$	binaire

Opérateurs de contrôle

nom	instruction	notation	arité
saut inconditionnel	(*goto L*, _, _, _)	`goto` *L*	0-aire
saut si zero	(*ifz L*, *x*, _, _)	`ifz` *x* `goto` *L*	unaire
saut si non-zero	(*ifnz L*, *x*, _, _)	`ifnz` *x* `goto` *L*	unaire
entrée fonc/proc	(*begin L*, _, _, _)	`begin` *L*	0-aire
retour fonc	(*return*, *x*, _, _)	`return` *x*	unaire
retour proc	(*return*, _, _, _)	`return`	0-aire
passage argument	(*arg*, *x*, _, _)	`arg` *x*	unaire
appel fonc	(*call L*, *x*, _, _)	*x* `=` `call` *L*	0-aire
appel proc	(*call L*, _, _, _)	`call` *L*	0-aire

où *L* est une étiquette de code

- position dans le code
- constante car connue à la compilation

Opérateurs d'affectation et d'adressage

nom	instruction	notation	arité
copie	$(=, x, y, \_)$	$x = y$	unaire
déréf. droit	$(*_D, x, y, \_)$	$x = *y$	unaire
		si y contient une adresse	
déréf. gauche	$(*_G, x, y, \_)$	$*x = y$	binaire
		si x contient une adresse	
"adresse de"	$(\&, x, y, \_)$	$x = \&y$	unaire
		si y est un emplacement mémoire	

Diagrammes mémoire de ces opérateurs : . . . . . .

Plan

Langage source

Dans le cadre de ce cours, on considère le langage BABIL

- langage impératif tel que C ou PASCAL
- contenant les principales constructions
  - expressions arithmétiques et booléennes
  - structures de contrôle : conditionnelles et boucles
  - définitions et appels de fonctions et procédures
- ordre d'évaluation des expressions non fixé (comme en C, Java)
  - sauf pour les expressions booléennes (comme en C, Java)

Syntaxe BABIL : unités lexicale

$$
\begin{array}{lcl}
\mathit{id} & \rightarrow & [a-zA-Z][a-zA-Z0-9\_]+ \\
\mathit{num} & \rightarrow & [0-9]+ \\
\mathit{bool} & \rightarrow & \underline{\mathit{true}} \mid \underline{\mathit{false}} \\
\mathit{op} & \rightarrow & + \mid - \mid * \mid / \\
\mathit{rop} & \rightarrow & == \mid =< \mid >= \mid < \mid > \mid != \\
\mathit{bop} & \rightarrow & \underline{\mathit{and}} \mid \underline{\mathit{or}}
\end{array}
$$

Syntaxe BABIL : expressions arith/bool

$$
\begin{array}{lcl}
E(xpr) & \rightarrow & num \\
 & | & id \\
 & | & E\ op\ E \\
 & | & -\ E \\
 & | & id\ (\ LA\ ) \\
L(ist)A(rgs) & \rightarrow & E\ ,\ LA\ \mid\ E\ \mid\ \epsilon \\
 & & \\
B(oolExpr) & \rightarrow & bool \\
 & | & id \\
 & | & E\ rop\ E \\
 & | & B\ bop\ B \\
 & | & \underline{not}\ B
\end{array}
$$

Syntaxe BABIL : programmes, fonctions, instructions

$$
\begin{array}{lcl}
P(rogram) & \rightarrow & LF \\
LF(unction) & \rightarrow & F\ LF \mid F \\
F(unction) & \rightarrow & \underline{define}\ id\ (\ LP\ )\ LS\ \underline{end} \\
LP(arameter) & \rightarrow & id\ ,\ LP \mid id \mid \epsilon \\
 & & \\
LS(tatement) & \rightarrow & S\ ;\ LS \mid S \\
S(tatement) & \rightarrow & id\ =\ E \\
 & \mid & \underline{if}\ B\ \underline{then}\ LS\ \underline{end} \\
 & \mid & \underline{if}\ B\ \underline{then}\ LS\ \underline{else}\ LS\ \underline{end} \\
 & \mid & \underline{while}\ B\ \underline{do}\ LS\ \underline{done} \\
 & \mid & id\ (\ LA\ ) \\
 & \mid & \underline{return} \\
 & \mid & \underline{return}\ E
\end{array}
$$

Plan

Génération de code 3-adresses

- code 3-adresses = séquence d'instructions
  $\rightarrow$ impose de linéariser le code
- exemple : `x + 2*z`
  - 3 opérations : $+$, $*$, 2
  - donc minimum 3 instructions : $+$, $*$, *const* 2
  - ordre : *const* 2 $\rightarrow * \rightarrow +$
  - code :

1	$(const\ 2, t1, \_, \_)$	`t1 = 2`
2	$(*, t2, t1, z)$	`t2 = t1 * z`
3	$(+, t3, x, t2)$	`t3 = x + t2`

  - x, y : variables du programme source

    a priori en mémoire
  - $t1, t2, t3$: variables intermédiaires
    $\rightarrow$ inventées (allouées) par le compilateur

    a priori en registre
  - $t3$ contient le résultat de l'expression

Génération de code 3-adresses

- code 3-adresses = séquence d'instructions
  $\rightarrow$ impose de linéariser le code
- exemple : `x + 2*z`
  - 3 opérations : $+$, $*$, 2
  - donc minimum 3 instructions : $+$, $*$, *const* 2
  - ordre : *const* 2 $\rightarrow * \rightarrow +$
  - code :

1	(*const* 2, *t*1, _, _)	`t1 = 2`
2	($*$, *t*2, *t*1, *z*)	`t2 = t1 * z`
3	($+$, *t*3, *x*, *t*2)	`t3 = x + t2`

  - *x*, *y* : variables du programme source
    a priori en mémoire
  - *t*1, *t*2, *t*3 : variables intermédiaires
    $\rightarrow$ inventées (allouées) par le compilateur
    a priori en registre
  - *t*3 contient le résultat de l'expression

Génération de code 3-adresses

- code 3-adresses = séquence d'instructions
  $\rightarrow$ impose de linéariser le code
- exemple : `x + 2*z`
  - 3 opérations : $+$, $*$, 2
  - donc minimum 3 instructions : $+$, $*$, *const* 2
  - ordre : *const* 2 $\rightarrow * \rightarrow +$
  - code :

1	(*const* 2, *t*1, _, _)	`t1 = 2`
2	($*$, *t*2, *t*1, *z*)	`t2 = t1 * z`
3	($+$, *t*3, *x*, *t*2)	`t3 = x + t2`

  - *x*, *y* : variables du programme source
    a priori en mémoire
  - *t*1, *t*2, *t*3 : variables intermédiaires
    $\rightarrow$ inventées (allouées) par le compilateur
    a priori en registre
  - *t*3 contient le résultat de l'expression

Génération de code 3-adresses

Comme pour la vérification de types ou la production de l'AST

- le plus simple est de procéder de façon compositionnelle
  - définir une "valeur" (ici : code généré) pour chaque construction du langage, en isolation
  - grammaire attribuée avec comme attribut synthétisé principal le code 3-adresse généré $X.code$

\+ "prises" : attributs hérités et synthétisés supplémentaires

- permettant aux constructions englobantes d'assembler les sous-séquences de code
- ex : $t3$ comme emplacement du résultat de l'expression $E.place$
- ex : $t2$ comme résultat de 2*z, utilisé dans x + 2*z
- ces prises diffèrent d'un NT à l'autre

Génération de code 3-adresses

Rappel : impératif = préserver la sémantique

- génération compositionnelle & systématique (grammaire attribuée)
- pas d'optimisation
  - on ne réutilise pas les variables intermédiaires
    $\rightarrow$ fonction `nouvar()`
  - idem pour les étiquettes de code
    $\rightarrow$ fonction `nouvetiq()`

Génération de code 3-adresses

Remarque

La génération de code est le lieu de rencontre de 3 programmes :

1. le programme source (construction de la grammaire)
2. le programme générateur (code du compilateur)
3. le programme généré (code 3-adresses)

Ils sont entremêlés dans la grammaire attribuée :

1. source : règles syntaxiques
2. générateur : actions/calculs associés aux règles
3. généré : codes 3-adresses manipulés comme valeurs par les actions/calculs

C'est sans doute la principale difficulté de la compilation !

Génération de code pour les expressions arithmétiques

. . . . . .

Grammaire attribuée pour les expressions arithmétiques

$$
\begin{array}{lcl}
E & \rightarrow & num \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & (const\ num.vallex, E.place, \_, \_) \end{array} \right. \\
 & | & id \\
 & & \left\{ \begin{array}{lcl} E.place & := & TS.place(id.vallex) \\ E.code & := & vide \end{array} \right. \\
 & | & E'\ op\ E'' \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & E'.code \mathbin{@@} E''.code \\ & \mathbin{@@} & (op.vallex, E.place, E'.place, E''.place) \end{array} \right. \\
 & | & -\ E' \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & E'.code \mathbin{@@} (-unaire, E.place, E'.place, \_ \end{array} \right.
\end{array}
$$

Exemple de génération pour les expressions arithmétiques

E = `(a + b)*(a + b)` .....

Génération de code pour les expressions booléennes

- On peut faire comme pour les expressions arithmétiques
  - attributs *B.code* et *B.place*
- mais *B* utilisé uniquement comme condition de branchement
  - le résultat ne sert qu'à décider du branchement
- de plus, dans `(a and b) or c`
  - si $a = \mathit{false}$, inutile d'évaluer b
  - si $a = \mathit{true}$ et $b = \mathit{true}$, inutile d'évaluer c

Code court-circuit pour les expressions booléennes

On va fait du code court-circuit

- branchement vers une étiquette $B.siVrai$ si $B = true$ et vers $B.siFaux$ si $B = faux$
  - plus de résultat explicite
    $\rightarrow$ l'attribut $B.place$ n'est plus défini
  - $B.siVrai$ et $B.siFaux$ sont des attributs hérités
    qui informent B de "où brancher" en fonction de la valeur de B
- en n'évaluant que ce qui est nécessaire (aspect "court-circuit")

Génération de code court-circuit pour les expressions booléennes

. . . . . .

Grammaire attribuée pour les expressions booléennes

$$
\begin{array}{lcl}
B & \rightarrow & id \\
 & & \left\{ \begin{array}{lcl} B.code & := & (ifnz\ B.siVrai, TS.place(id.vallex), \_, \_) \\ & @@ & (goto\ B.siFaux, \_, \_, \_) \end{array} \right. \\
 & | & bool \\
 & & \left\{ \begin{array}{lcl} B.code & := & si\ bool.vallex \\ & & alors\ (goto\ B.siVrai, \_, \_, \_) \\ & & sinon\ (goto\ B.siFaux, \_, \_, \_) \end{array} \right. \\
 & | & E'\ rop\ E'' \\
 & & \left\{ \begin{array}{lcl} resultat & := & nouvar() \\ B.code & := & E'.code\ @@\ E''.code \\ & @@ & (rop.vallex, resultat, E'.place, E''.place) \\ & @@ & (ifnz\ B.siVrai, resultat, \_, \_) \\ & @@ & (goto\ B.siFaux, \_, \_, \_) \end{array} \right.
\end{array}
$$

Grammaire attribuée pour les expressions booléennes

$$
\begin{array}{lcl}
B & \rightarrow & \textit{not } B' \\
 & & \left\{ \begin{array}{lcl} B'.siVrai, B'.siFaux & := & B.siFaux, B.siVrai \\ B.code & := & B'.code \end{array} \right. \\
 & | & B' \textit{ and } B'' \\
 & & \left\{ \begin{array}{lcl} B'.siVrai, B'.siFaux & := & nouvetiq(), B.siFaux \\ B''.siVrai, B''.siFaux & := & B.siVrai, B.siFaux \\ B.code & := & B'.code \\ & @@ & (label\ B'.siVrai, \_, \_, \_) \\ & @@ & B''.code \end{array} \right. \\
 & | & B' \textit{ or } B'' \\
 & & \left\{ \begin{array}{lcl} B'.siVrai, B'.siFaux & := & B.siVrai, nouvetiq() \\ B''.siVrai, B''.siFaux & := & B.siVrai, B.siFaux \\ B.code & := & B'.code \\ & @@ & (label\ B'.siFaux, \_, \_, \_) \\ & @@ & B''.code \end{array} \right.
\end{array}
$$

Exemple de génération pour les expressions booléennes

B = `(a and b) or c` .....

Génération de code pour les instructions

. . . . . .

Grammaire attribuée pour les instructions

$$
\begin{array}{lcl}
S & \rightarrow & id = E \\
 & & \left\{ S.code := E.code \mathbin{@@} (=, TS.place(id.vallex), E.place, \_) \right. \\
 & | & \underline{if}\ B\ \underline{then}\ LS'\ \underline{end} \\
 & & \left\{ \begin{array}{lcl}
B.siVrai & := & alors := nouvetiq() \\
B.siFaux & := & fin := nouvetiq() \\
S.code & := & B.code \\
 & @@ & (label\ alors, \_, \_, \_) \mathbin{@@} LS'.code \\
 & @@ & (label\ fin, \_, \_, \_)
\end{array} \right. \\
 & | & \underline{if}\ B\ \underline{then}\ LS'\ \underline{else}\ LS''\ \underline{end} \\
 & & \left\{ \begin{array}{lcl}
B.siVrai & := & alors := nouvetiq() \\
B.siFaux & := & sinon := nouvetiq() \\
fin & := & nouvetiq() \\
S.code & := & B.code \\
 & @@ & (label\ alors, \_, \_, \_) \mathbin{@@} LS'.code \\
 & @@ & (goto\ fin, \_, \_, \_) \\
 & @@ & (label\ sinon, \_, \_, \_) \mathbin{@@} LS''.code \\
 & @@ & (label\ fin, \_, \_, \_)
\end{array} \right.
\end{array}
$$

$$
\begin{array}{lcl}
S & \rightarrow & \underline{\textit{while}}\ B\ \underline{\textit{do}}\ LS'\ \underline{\textit{done}} \\
 & & \left\{ \begin{array}{lcl}
\textit{boucle} & := & \textit{nouvetiq}() \\
B.\textit{siVrai} & := & \textit{corps} := \textit{nouvetiq}() \\
B.\textit{siFaux} & := & \textit{sortie} := \textit{nouvetiq}() \\
S.\textit{code} & := & (\textit{label boucle}, \_, \_, \_) \ @@\ B.\textit{code} \\
 & @@ & (\textit{label corps}, \_, \_, \_) \ @@\ LS'.\textit{code} \\
 & @@ & (\textit{goto boucle}, \_, \_, \_) \\
 & @@ & (\textit{label sortie}, \_, \_, \_)
\end{array} \right. \\
\\
LS & \rightarrow & S\ ;\ LS' \\
 & & \left\{ \begin{array}{lcl}
LS.\textit{code} & := & S.\textit{code} \ @@\ LS'.\textit{code}
\end{array} \right. \\
 & | & S \\
 & & \left\{ \begin{array}{lcl}
LS.\textit{code} & := & S.\textit{code}
\end{array} \right.
\end{array}
$$

Exemple de génération pour les instructions

S = `i = 1; f = 1; while i<n and f<100 do i = i+1; f = f*i done`.....

Appels de fonctions/procédures

- 2 constructions
  - $E \rightarrow id\ (\ LA\ )$: appel de fonction (expression)
  - $S \rightarrow id\ (\ LA\ )$: appel de procédure (instruction)
- il existe 2 sémantiques du passage de paramètres
  - par nom : nom d'une mémoire, emplacement
    - permet à la fonction de modifier le contenu de cette mémoire
    - ex : paramètre `var` de PASCAL
    - ex : références sur objets en JAVA
  - par valeur : contenu d'une mémoire
    - la valeur du paramètre est recopiée dans une zone mémoire réservée aux arguments
      sur la pile ou dans des registres
    - ne permet pas de modifier l'argument
      mais l'argument peut être un pointeur...
    - c'est la sémantique de C... et de BABIL
- c'est l'appelant qui a la charge d'installer les paramètres avant d'appeler la fonction
  - opérateur (*arg*, *x*, ...)

Appels de fonctions/procédures

- 2 constructions
  - $E \rightarrow id\ (\ LA\ )$: appel de fonction (expression)
  - $S \rightarrow id\ (\ LA\ )$: appel de procédure (instruction)
- il existe 2 sémantiques du passage de paramètres
  - par nom : nom d'une mémoire, emplacement
    - permet à la fonction de modifier le contenu de cette mémoire
    - ex : paramètre `var` de PASCAL
    - ex : références sur objets en JAVA
  - par valeur : contenu d'une mémoire
    - la valeur du paramètre est recopiée dans une zone mémoire réservée aux arguments
      sur la pile ou dans des registres
    - ne permet pas de modifier l'argument
      mais l'argument peut être un pointeur...
    - c'est la sémantique de C... et de BABIL
- c'est l'appelant qui a la charge d'installer les paramètres avant d'appeler la fonction
  - opérateur (*arg*, *x*, ,)

Génération de code pour les appels de fonctions

. . . . . .

Grammaire attribuée pour les appels de fonctions

$$
\begin{array}{lcl}
E & \rightarrow & id\ (\ LA\ ) \\
 & & \left\{ \begin{array}{lcl} E.place & := & nouvar() \\ E.code & := & LA.code \mathbin{@@} (call\ TS.label(id.vallex), E.place, \end{array} \right. \\
S & \rightarrow & id\ (\ LA\ ) \\
 & & \{\ S.code := LA.code \mathbin{@@} (call\ TS.label(id.vallex), \_, \_, \_) \\
LA & \rightarrow & E\ ,\ LA' \\
 & & \{\ LA.code := E.code \mathbin{@@} (arg, E.place, \_, \_) \mathbin{@@} LA'.code \\
 & | & E \\
 & & \{\ LA.code := E.code \mathbin{@@} (arg, E.place, \_, \_) \\
 & | & \epsilon \\
 & & \{\ LA.code := vide
\end{array}
$$

Génération de code pour les définitions de fonctions

. . . . . .

Grammaire attribuée pour les définitions de fonctions

$$
\begin{array}{lcl}
S & \rightarrow & \underline{return} \\
 & & \{\ S.code := (return, \_, \_, \_) \\
 & | & \underline{return}\ E \\
 & & \{\ S.code := E.code \mathbin{@@} (return, E.place, \_, \_)
\end{array}
$$

$$
\begin{array}{lcl}
F & \rightarrow & id\ (\ LP\ )\ LS \\
 & & \left\{\begin{array}{rcl} F.code & := & (beginfunc\ TS.label(id.vallex), \_, \_, \_) \\ & \mathbin{@@} & LS.code \end{array}\right.
\end{array}
$$

$$
\begin{array}{lcl}
LF & \rightarrow & F\ LF' \\
 & & \{\ LF.code := F.code \mathbin{@@} LF'.code \\
 & | & F \\
 & & \{\ LF.code := F.code
\end{array}
$$

$$
\begin{array}{lcl}
P & \rightarrow & LF \\
 & & \{\ P.code := LF.code
\end{array}
$$

Exemple de génération pour les fonctions

Exemple : `define fact(n) if n==0 then return 1 else return n * fact(n-1) end`.....