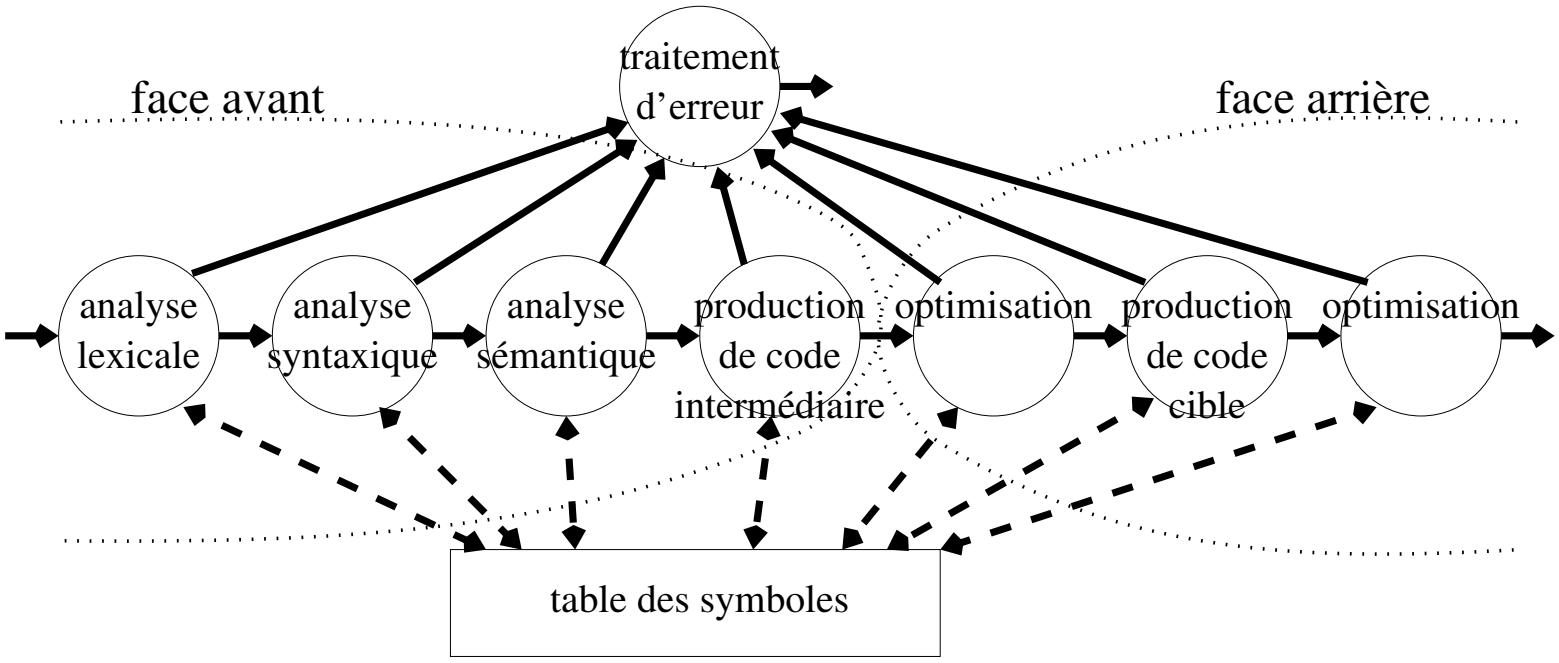
Programmation Dirigée par la Syntaxe (PDS)

CM5 - Génération de code

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

PDS, M1 info

Les phases de la compilation



Plan

Plan

Phase : Génération de code cible

Traduction de l'AST source vers le langage cible

- vient après les analyses sémantiques (vérification de types)
- 2 aspects (rappel) :
  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
     $\rightarrow$ machine = processeur + mémoire + pile
   - la programmation de cette phase doit être assez simple pour garantir la préservation de la sémantique
     $\rightarrow$ approche compositionnelle (via ASD attribuée)
   - 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

(1)……

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

- (2)……

- 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

- (2)……

- 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	$(-\texttt{UNAIRE}, x, y, \_)$	$x = -y$	unaire
constante ($N \in \mathbb{N}$)	$(\texttt{CONST}\ N, x, \_, \_)$	$x = N$	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	$(\texttt{EQ}, x, y, z)$	$x = y == z$	binaire
neq	$(\texttt{NEQ}, x, y, z)$	$x = y != z$	binaire
leq	$(\texttt{LEQ}, x, y, z)$	$x = y <= z$	binaire
lt	$(\texttt{LT}, x, y, z)$	$x = y < z$	binaire
geq	$(\texttt{GEQ}, x, y, z)$	$x = y >= z$	binaire
gt	$(\texttt{GT}, x, y, z)$	$x = y > z$	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	$(*\mathtt{D}, x, y, \_)$	$x = *y$	unaire
		si y contient une adresse	
déréf. gauche	$(*\mathtt{G}, x, y, \_)$	$*x = y$	binaire
		si x contient une adresse	
"adresse de"	$(\&, x, y, \_)$	$x = \&y$	unaire
		si y est un emplacement mémoire	

Diagrammes mémoire de ces opérateurs : (3). . . . . .

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 abstraite BABIL : types de base et types énumérés

- types de base : *string*, *int*, *bool*
- types synonymes et énumérés

$$
\begin{array}{lcl}
\textit{id} & ::= & \textit{string} \\
 & & \\
\textit{op} & ::= & + \mid - \mid * \mid / \\
\textit{rop} & ::= & == \mid =< \mid >= \mid < \mid > \mid != \\
\textit{bop} & ::= & \textbf{And} \mid \textbf{Or}
\end{array}
$$

Syntaxe abstraite BABIL : expressions arith/bool

$$
\begin{array}{lcl}
\mathit{expr}(\mathit{ession}) & ::= & \mathbf{Const}(\mathit{int}) \\
& | & \mathbf{Var}(\mathit{id}) \\
& | & \mathbf{Binop}(\mathit{op}, \mathit{expr}, \mathit{expr}) \\
& | & \mathbf{Minus}(\mathit{expr}) \\
& | & \mathbf{Call}(\mathit{id}, \mathit{arg}*) \\
\mathit{arg}(\mathit{ument}) & ::= & \mathit{expr} \\
\mathit{cond}(\mathit{ition}) & ::= & \mathbf{Bool}(\mathit{bool}) \\
& | & \mathbf{Comp}(\mathit{rop}, \mathit{expr}, \mathit{expr}) \\
& | & \mathbf{Logop}(\mathit{bop}, \mathit{cond}, \mathit{cond}) \\
& | & \mathbf{Not}(\mathit{cond})
\end{array}
$$

Syntaxe abstraite BABIL : programmes, fonctions, instructions

$$
\begin{array}{lcl}
\mathit{prog}(\mathit{ram}) & ::= & \textbf{Prog}(\mathit{function}*) \\
\mathit{func}(\mathit{tion}) & ::= & \textbf{Func}(\mathit{id}, \mathit{param}*, \mathit{stat}) \\
\mathit{param}(\mathit{eter}) & ::= & \mathit{id} \\
\mathit{stat}(\mathit{ement}) & ::= & \textbf{Assign}(\mathit{id}, \mathit{expr}) \\
 & | & \textbf{IfThenElse}(\mathit{cond}, \mathit{stat}, \mathit{stat}?) \\
 & | & \textbf{While}(\mathit{cond}, \mathit{stat}) \\
 & | & \textbf{SCall}(\mathit{id}, \mathit{arg}*) \\
 & | & \textbf{Return}(\mathit{expr}?) \\
 & | & \textbf{Bloc}(\mathit{stat}*)
\end{array}
$$

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, $t1$, _, _)	`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, $t1$, _, _)	`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

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
  - ASD attribuée avec comme attribut synthétisé principal le code 3-adresse généré $\tau.code$

\+ "prises" : attributs hérités et synthétisés supplémentaires

- permettant aux constructions englobantes d'assembler les sous-séquences de code
- ex : *t*3 comme emplacement du résultat de l'expression *expr*.*place*
- ex : *t*2 comme résultat de 2*z, utilisé dans x + 2*z
- ces prises diffèrent d'un type à l'autre

Génération de code 3-adresses

Rappel : impératif = préserver la sémantique

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

Génération de code 3-adresses

Remarque

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

1. le programme source (types et variants ASD)
2. le programme générateur (code du compilateur)
3. le programme généré (code 3-adresses)

Ils sont entremêlés dans l'ASD attribuée :

1. source : variants ASD
2. générateur : actions/calculs associés aux variants
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

(4)……

Grammaire attribuée pour les expressions arithmétiques

$$
\begin{array}{lll}
expr & ::= & \mathbf{Const}(int) \\
 & & \left\{ \begin{array}{lll} expr.place & := & nouvar() \\ expr.code & := & (\texttt{CONST}\ int.vallex, expr.place, \_, \_) \end{array} \right. \\
 & | & \mathbf{Var}(id) \\
 & & \left\{ \begin{array}{lll} expr.place & := & TS.place(id.vallex) \\ expr.code & := & vide \end{array} \right. \\
 & | & \mathbf{Binop}(op, expr_1, expr_2) \\
 & & \left\{ \begin{array}{lll} expr.place & := & nouvar() \\ expr.code & := & expr_1.code \ @@ \ expr_2.code \\ & @@ & (op.vallex, expr.place, expr_1.place, expr \end{array} \right. \\
 & | & \mathbf{Minus}(expr_1) \\
 & & \left\{ \begin{array}{lll} expr.place & := & nouvar() \\ expr.code & := & expr_1.code \\ & @@ & (\texttt{-UNAIRE}, expr.place, expr_1.place, \_) \end{array} \right.
\end{array}
$$

Exemple de génération pour les expressions arithmétiques

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

(5).....

Génération de code pour les expressions booléennes

- On peut faire comme pour les expressions arithmétiques
  - attributs *cond*.*code* et *cond*.*place*
- mais *cond* utilisé uniquement comme condition de branchement
  - le résultat ne sert qu'à décider du branchement
- de plus, dans `(`c_1 `and` c_2`) or` c_3
  - si $c_1 = \mathit{false}$, inutile d'évaluer c_2
  - si $c_1 = \mathit{true}$ et $c_2 = \mathit{true}$, inutile d'évaluer c_3

Code court-circuit pour les expressions booléennes

On va fait du code court-circuit

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

Génération de code court-circuit pour les expressions booléennes

(6)……

Grammaire attribuée pour les expressions booléennes

$$
\begin{array}{lll}
\mathit{cond} & ::= & \mathbf{Bool}(\mathit{bool}) \\
 & & \left\{ \begin{array}{lll}
\mathit{cond.code} & := & \mathit{si}\ \mathit{bool.vallex} \\
 & & \mathit{alors}\ (\texttt{GOTO}\ \mathit{cond.siVrai}, \_, \_, \_) \\
 & & \mathit{sinon}\ (\texttt{GOTO}\ \mathit{cond.siFaux}, \_, \_, \_)
\end{array} \right. \\
 & | & \mathbf{Comp}(\mathit{rop}, \mathit{expr}_1, \mathit{expr}_2) \\
 & & \left\{ \begin{array}{lll}
\mathit{resultat} & := & \mathit{nouvar}() \\
\mathit{cond.code} & := & \mathit{expr}_1\mathit{.code}\ @@\ \mathit{expr}_2\mathit{.code} \\
 & @@ & (\mathit{rop.vallex}, \mathit{resultat}, \mathit{expr}_1\mathit{.place}, \mathit{expr}_2\mathit{.place}) \\
 & @@ & (\texttt{IFNZ}\ \mathit{cond.siVrai}, \mathit{resultat}, \_, \_) \\
 & @@ & (\texttt{GOTO}\ \mathit{cond.siFaux}, \_, \_, \_)
\end{array} \right. \\
 & | & \mathbf{Not}(\mathit{cond}_1) \\
 & & \left\{ \begin{array}{lll}
\mathit{cond}_1\mathit{.siVrai} & := & \mathit{cond.siFaux} \\
\mathit{cond}_1\mathit{.siFaux} & := & \mathit{cond.siVrai} \\
\mathit{cond.code} & := & \mathit{cond}_1\mathit{.code}
\end{array} \right.
\end{array}
$$

Grammaire attribuée pour les expressions booléennes

| **Logop**(**And**, $cond_1$, $cond_2$)

$$\begin{cases} cond_1.siFaux & := & cond.siFaux \\ cond_2.siFaux & := & cond.siFaux \\ cond_1.siVrai & := & nouvetiq() \\ cond_2.siVrai & := & cond.siVrai \\ cond.code & := & cond_1.code \\ & @@ & (\texttt{LABEL}\ cond_1.siVrai, \_, \_, \_) \\ & @@ & cond_2.code \end{cases}$$

| **Logop**(**Or**, $cond_1$, $cond_2$)

$$\begin{cases} cond_1.siFaux & := & nouvetiq() \\ cond_2.siFaux & := & cond.siFaux \\ cond_1.siVrai & := & cond.siVrai \\ cond_2.siVrai & := & cond.siVrai \\ cond.code & := & cond_1.code \\ & @@ & (\texttt{LABEL}\ cond_1.siFaux, \_, \_, \_) \\ & @@ & cond_2.code \end{cases}$$

Exemple de génération pour les expressions booléennes

B = `(a = d and b > e) or c < f`
(7).....

Génération de code pour les instructions

(8)……

Grammaire attribuée pour les instructions

$$
\begin{array}{lll}
stat & ::= & \textbf{Assign}(id, expr) \\
 & & \left\{ \begin{array}{lll} stat.code & := & expr.code \\ & @@ & (=, TS.place(id.vallex), expr.place, \_) \end{array} \right. \\
 & | & \textbf{IfThenElse}(cond, stat_1, \epsilon) \\
 & & \left\{ \begin{array}{lll} cond.siVrai & := & alors := nouvetiq() \\ cond.siFaux & := & fin := nouvetiq() \\ stat.code & := & cond.code \\ & @@ & (\texttt{LABEL}\ alors, \_, \_, \_) \ @@\ stat_1.code \\ & @@ & (\texttt{LABEL}\ fin, \_, \_, \_) \end{array} \right. \\
 & | & \textbf{IfThenElse}(cond, stat_1, stat_2) \\
 & & \left\{ \begin{array}{lll} cond.siVrai & := & alors := nouvetiq() \\ cond.siFaux & := & sinon := nouvetiq() \\ fin & := & nouvetiq() \\ stat.code & := & cond.code \\ & @@ & (\texttt{LABEL}\ alors, \_, \_, \_) \ @@\ stat_1.code \\ & @@ & (\texttt{GOTO}\ fin, \_, \_, \_) \\ & @@ & (\texttt{LABEL}\ sinon, \_, \_, \_) \ @@\ stat_2.code \\ & @@ & (\texttt{LABEL}\ fin, \_, \_, \_) \end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
\mid \quad \textbf{While}(cond, stat_1) \\
\quad \left\{ \begin{array}{lcl}
boucle & := & nouvetiq() \\
cond.siVrai & := & corps := nouvetiq() \\
cond.siFaux & := & sortie := nouvetiq() \\
stat.code & := & (\texttt{LABEL}\ boucle, \_, \_, \_)\ @@\ cond.code \\
 & @@ & (\texttt{LABEL}\ corps, \_, \_, \_)\ @@\ stat_1.code \\
 & @@ & (\texttt{GOTO}\ boucle, \_, \_, \_) \\
 & @@ & (\texttt{LABEL}\ sortie, \_, \_, \_)
\end{array} \right. \\
\mid \quad \textbf{Bloc}(stat*) \\
\quad \left\{ \begin{array}{lcl}
stat.code & := & @@(stat*.code)
\end{array} \right.
\end{array}
$$

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`

(9).....

Appels de fonctions/procédures

- 2 constructions
  - *expr* ::= **Call**(*id*, *arg*∗) : appel de fonction (expression)
  - *stat* ::= **SCall**(*id*, *arg*∗) : 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
  - *expr* ::= **Call**(*id*, *arg*∗) : appel de fonction (expression)
  - *stat* ::= **SCall**(*id*, *arg*∗) : 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

(10). . . . . .

Grammaire attribuée pour les appels de fonctions

$$
\begin{array}{lcl}
\textit{expr} & ::= & \textbf{Call}(\textit{id}, \textit{arg}*) \\
 & & \left\{ \begin{array}{lcl} \textit{expr.place} & := & \textit{nouvar}() \\ \textit{expr.code} & := & @@(\textit{arg} * .\textit{code}) \\ & @@ & (\texttt{CALL}\ \textit{TS.label}(\textit{id.vallex}), \textit{expr.place}, \_, \_) \end{array} \right. \\
\textit{stat} & ::= & \textbf{SCall}(\textit{id}, \textit{arg}*) \\
 & & \left\{ \begin{array}{lcl} \textit{stat.code} & := & @@(\textit{arg} * .\textit{code}) \\ & @@ & (\texttt{CALL}\ \textit{TS.label}(\textit{id.vallex}), \_, \_, \_) \end{array} \right. \\
\textit{arg} & ::= & \textit{expr} \\
 & & \{ \ \textit{arg.code} := \textit{expr.code} @@ (\texttt{ARG}, \textit{expr.place}, \_, \_)
\end{array}
$$

Génération de code pour les définitions de fonctions

(11)……

Grammaire attribuée pour les définitions de fonctions

$$
\begin{array}{lcl}
\textit{stat} & ::= & \textbf{Return}(\epsilon) \\
 & & \left\{ \begin{array}{lcl} \textit{stat.code} & := & (\texttt{RETURN}, \_, \_, \_) \end{array} \right. \\
 & | & \textbf{Return}(\textit{expr}) \\
 & & \left\{ \begin{array}{lcl} \textit{stat.code} & := & \textit{expr.code} \\ & @@ & (\texttt{RETURN}, \textit{expr.place}, \_, \_) \end{array} \right. \\
 & & \\
\textit{func} & ::= & \textbf{Func}(\textit{id}, \textit{param}*, \textit{stat}) \\
 & & \left\{ \begin{array}{lcl} \textit{func.code} & := & (\texttt{BEGINFUNC}\ \textit{TS.label}(\textit{id.vallex}), \_, \_, \_) \\ & @@ & \textit{stat.code} \end{array} \right. \\
 & & \\
\textit{prog} & ::= & \textbf{Prog}(\textit{func}*) \\
 & & \left\{ \begin{array}{lcl} \textit{prog.code} & := & @@(\textit{func} * .\textit{code}) \end{array} \right.
\end{array}
$$

Exemple de génération pour les fonctions

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

(12).....