Génération de code trois adresses

Alexis Nasr
Franck Dary
Pacôme Perrotin

Compilation – L3 Informatique
Département Informatique et Interactions
Aix Marseille Université

Représentations intermédiaires

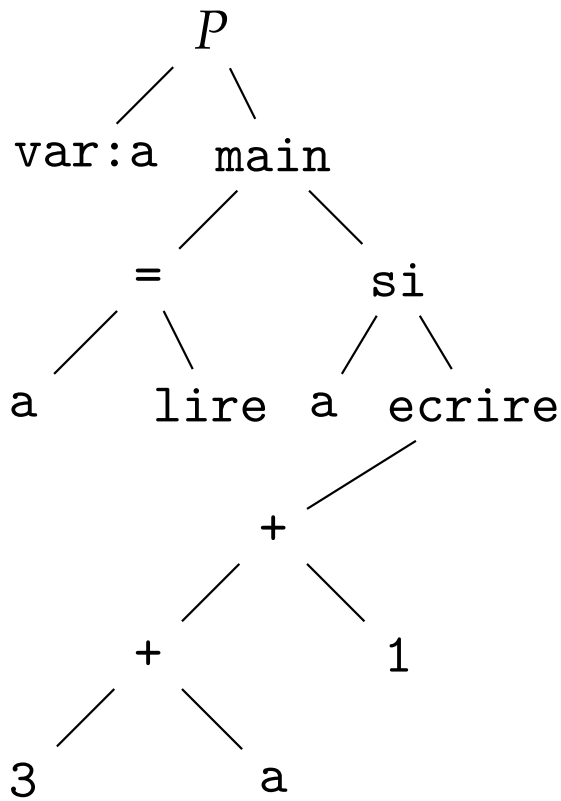
- Arbre abstrait → représentation intermédiaire “haut niveau”
  Assez structurée, proche du langage source
- Code trois adresses → représentation intermédiaire “bas niveau”
  Linéaire, peu structurée, proche du langage machine
- La génération de code trois adresses s’effectue pendant le parcours de l’arbre abstrait

Exemple : code trois adresses

Code-source en L :

```
entier a;
main(){
  a = lire();
  si a alors {
    ecrire(3+a+1);
  }
}
```

Arbre abstrait :



Code 3 adresses :

```
01 : fbegin
02 : t0 = read
03 : a = t0
04 : if a == 0 goto 8
05 : t1 = 3 + a
06 : t2 = t1 + 1
07 : write t2
08 : fend
```

Intérêt du code trois adresses

- Se concentrer sur la linéarisation du programme :
  - dérouler les boucles
  - désimbriquer les blocs de code
  - décomposer les expressions complexes en une séquence d'opérations
- Faire abstraction des détails propres à chaque architecture :
  - Organisation de la mémoire (segments mémoire de pile, tas, ...)
  - Taille (nombre d'octets) et emplacement des variables
  - Nombre de registres disponibles
  - Transferts registres $\leftrightarrow$ mémoire
  - Particularités des instructions (p.ex. `idiv` opère sur le registre `eax`)
- **Compromis** : facile à convertir en code machine mais suffisamment générique pour s'adapter aux différentes architectures

Représentation en mémoire du code trois adresses

- Le code trois adresses est une séquence d'instructions
- Toute instruction correspond à un indice dans la séquence, son adresse
- Chaque instruction prend la forme :

```
opcode a1 a2 r
```

- Le code de l'opération `opcode` est obligatoire
- Chaque instruction a au plus trois opérandes
  - `a1` et `a2` sont généralement les opérandes d'une opération
  - `r` est généralement le résultat d'une opération
- Les opérations sont simples : arithmétiques, logiques, sauts, etc.

Les Opérandes

- Les opérandes des instructions peuvent être :
  1. Des constantes : *1, -5, 3455*
  2. Des variables du programme source : `max, a`
  3. Des étiquettes du code trois adresses : `fin, e1`
  4. Des fonctions
  5. Des variables temporaires générées lors de la traduction : `t0, t1`
- Les constantes, les variables et les noms de fonctions existent déjà dans le programme source (donc dans l'arbre abstrait)
- Les étiquettes et les variables temporaires sont créées lors du processus de génération

Les variables temporaires

- Dans le langage source, les expressions peuvent être complexes :
  `delta = b * b - 4 * a * c`
- Dans le code trois adresses, ces expressions complexes ne sont pas autorisées (plus de trois opérandes)
- Il faut les décomposer et stocker des valeurs intermédiaires dans des variables temporaires :

```
01 : t0 = b × b
02 : t1 = 4 × a
03 : t2 = t1 × c
04 : delta = t0 - t2
```

- Les variables temporaires (ou simplement temporaires) sont générées à la demande pendant la traduction
- Nous les noterons `t0, t1, t2 ...` par convention

Etiquettes

- Une étiquette est un nom symbolique unique donné à une adresse
- Cela permet de rendre le code plus lisible
- Les étiquettes sont généralement représentées par un identifiant qui précède l'instruction

Au lieu d'écrire :

```
04 : if a == 0 goto 8
05 : t1 = 3 + a
06 : t2 = t1 + 1
07 : write t2
08 : fend
```

nous écrirons :

```
04      : if a == 0 goto fin
05      : t1 = 3 + a
06      : t2 = t1 + 1
07      : write t2
08 fin : fend
```

Les variables I

- Les variables sont des noms appartenant au langage source
- Dans le code trois adresses, nous avons besoin d'informations sur ces variables contenues dans la table des symboles
- Les variables sont représentées par des pointeurs vers la table des symboles

Les variables II

Le cas des tableaux

- Les variables de type tableau sont toujours indicées
- En L ces indices peuvent être des expressions quelconques
- En code 3 adresses, ces indices sont uniquement des temporaires ou des constantes
- Par exemple, `tab[t2]` et `tab[5]`, mais pas `tab[i]` ou `tab[t0+1]`

Les instructions

Type	Opérations	a1	a2	r	Syntaxe
arithmétique	+ - * /	ctv	ctv	tv	r = a1 op a2
affectation	=	ctv		tv	r = a1
saut test	== < <= > >=	ctv	ctv	e	if a1 op a2 goto r
saut direct	goto	e			goto a1
appel fonction	call	e		tv	r = call a1
lecture	read			tv	r = read
écriture	write	ctv			write a1
e/s fonction	param ret	ctv			op a1
début/fin fonc.	fbegin fend				fbegin, fend

- Chaque instruction accepte certains types d'opérandes `a1`, `a2` et `r` :

  constantes (`c`), temporaires (`t`), variables (`v`) ou étiquettes (`e`)
- Exemples :
  - l'opérande `a1` d'un saut direct est une étiquette `e`
  - l'opérande `r` d'une op. arith. est un temporaire ou variable (`tv`).
- Pas d'opérateurs de comparaison, pas d'opérateurs logiques

Les instructions arithmétiques et d'affectation

- Format :
  - `r = a1 op a2`
  - `r = a1`
- Le résultat `r` ne peut pas être une constante
- Sémantique identique aux opérations arithmétiques classiques
- Limitation aux expressions simples, pas plus de trois adresses
- Le parcours de l'arbre abstrait génère des temporaires pour stocker les variables intermédiaires des expressions
- Les expressions complexes sont transformées en instructions arithmétiques simples dans le code trois adresses
- Cela s'applique aussi aux cases de tableaux :
  `a = tab[ i ]` $\rightarrow$ `t0 = i; a = tab[t0]`

Les sauts

- Formats :
  - `if a1 op a2 goto r`
  - `goto r`
  - `call r`
- Les sauts changent le cours de l'exécution du programme
- Sauts conditionnels : vont à la ligne cible `r` si une condition portant sur les adresses `a1` et `a2` est vraie :

```
if t4 < 10 goto e12
```

- Sauts inconditionnels : la prochaine instruction est la cible `r`
  - `goto r` : pas de mémorisation de la ligne courante
- La traduction en code trois adresses crée des nouvelles étiquettes `e0, e1, e2...` qui seront la cibles des sauts

Déclarations et appels de fonctions

- Instructions gardant la trace des fonctions du langage source :
  - `fbegin` marque le début d'une déclaration de fonction. Cette instruction n'a aucun effet sur l'état du programme. Sa ligne est toujours étiquetée, p.ex. `fmain: fbegin`.
  - `ret` et `param` ont une opérande (`ctv`). Elles communiquent un paramètre ou une valeur de retour entre la fonction appelante et la fonction applée, p.ex. `ret t5`.
  - `call r` mémorise la prochaine ligne à exécuter et change le cours de l'exécution vers la cible `r`.
  - `fend` marque la fin d'une déclaration de fonction, changeant le cours de l'exécution vers la dernière adresse sauvegardée par `call`.
- La cible `r` d'un `call r` est toujours une instruction `fbegin` (début de fonction) étiquetée `r`

Exemple d'appel de fonction

```
f(entier a, entier b)
{
ecrire(a);
ecrire(b);
}
main()
{
f(1, 456);
}
```

```
      t0 = call main
      stop t0
f     fbegin
      write a
      write b
      fend
main  fbegin
      param 1
      param 456
      t1 = call f
      fend
```

Instructions spéciales

- `r = read` lit un entier depuis le clavier et le stocke dans `r` (temporaire ou variable)
- `write a1` écrit un entier `a1` (temporaire, constante ou variable) sur le terminal

Traduction

- La traduction arbre abstrait $\rightarrow$ code trois adresses se fait lors d'un parcours en profondeur de l'arbre abstrait.
- Nous utiliserons des grammaires attribuées fondées sur :
  - Une grammaire (ambiguë) correspondant à l'arbre abstrait [1]
  - Des attributs synthétisés
  - La fonction *newtemp()* qui génère un nouveau temporaire unique.
  - La fonction *newetiq()* qui génère une nouvelle étiquette unique.
  - La fonction *parcours(n)* qui provoque le parcours du nœud n.
  - Le code généré est représenté en bleu.

1. Un arbre de dérivation de cette grammaire est un arbre abstrait

Grammaire

1. $P \rightarrow LD\ LD$

2. $LD \rightarrow D\ LD$
3. $LD \rightarrow$ `null`

4. $D \rightarrow$ `fct id` $LD\ LD\ LI$
5. $D \rightarrow$ `var id`
6. $D \rightarrow$ `var id taille`

7. $V \rightarrow$ `id`
8. $V \rightarrow$ `id[` E`]`
9. $LI \rightarrow I\ LI$
10. $LI \rightarrow$ `null`

11. $I \rightarrow$ `aff` $V\ E$
12. $I \rightarrow$ `si` $E\ LI\ LI$
13. $I \rightarrow$ `tq` $E\ LI$
14. $I \rightarrow$ `app` APP
15. $I \rightarrow$ `ret` E
16. $I \rightarrow$ `ecr` E
17. $APP \rightarrow$ `id` LE
18. $LE \rightarrow E\ LE$
19. $LE \rightarrow$ `null`

20. $E \rightarrow E$ `op2` E
21. $E \rightarrow$ `op1` E
22. $E \rightarrow V$
23. $E \rightarrow$ `entier`
24. $E \rightarrow APP$
25. $E \rightarrow$ `lire`

Traduction des expressions I

- Principe général : à l'issue de l'exécution du code correspondant à une expression, le résultat de cette dernière doit se trouver dans une variable ou dans un temporaire
- Quatre cas :
  - *Constante* : la constante est renvoyée telle quelle
  - *Variable* : le pointeur de la variable dans la table des symboles est renvoyé
  - *Appel de fonction* : les paramètres de la fonction sont évalués, la fonction est appelée, et la valeur de retour de la fonction est mise dans un nouveau temporaire, qui est renvoyé
  - *Opération* : l'expression est décomposée et le résultat final est mis dans un temporaire, qui est renvoyé

Constantes et variables

Production			Action sémantique
E	$\rightarrow$	`nb`	$E.t = \texttt{nb}.val$
E	$\rightarrow$	`var`	$E.t = \texttt{var}.val$
E	$\rightarrow$	`var [` E_1 `]`	$E_1.t = parcours(E_1)$ $E.t = \texttt{var}[E_1.t]$
E	$\rightarrow$	`lire`	$E.t = newtemp()$ $E.t =$ `read`

Opérations arithmétiques

Production	Action sémantique
$E \rightarrow E_1 + E_2$	$E.t = newtemp()$
	$E_1.t = parcours(E_1)$
	$E_2.t = parcours(E_2)$
	$E.t = E_1.t + E_2.t$

Comparaisons et opérateurs logiques

- Pas d'opérateurs de comparaison ni logiques en code trois adresses.
- Le calcul est effectué par des sauts conditionnels.
- Les valeurs booléennes sont représentées par des entiers :
- FAUX $\rightarrow$ 0 , VRAI$\rightarrow$ 1

Comparaisons

Production	Action sémantique
$E \rightarrow E_1 < E_2$	$\texttt{vrai} = newetiq()$ $\texttt{suite} = newetiq()$ $E.t = newtemp()$ $E_1.t = parcours(E_1)$ $E_2.t = parcours(E_2)$ $\texttt{if } E_1.t < E_2.t \texttt{ goto vrai}$ $E.t = 0$ $\texttt{jump suite}$ $\texttt{vrai} : E.t = 1$ $\texttt{suite} :$

Opérations logiques

Production	Action sémantique

$E \rightarrow E_1$ & E_2	`faux` $= newetiq()$
	`suite` $= newetiq()$
	$E.t = newtemp()$
	$E_1.t = parcours(E_1)$
	`if` $E_1.t = 0$ `goto faux`
	$E_2.t = parcours(E_2)$
	`if` $E_2.t = 0$ `goto faux`
	$E.t = 1$
	`jump suite`
	`faux` : $E.t = 0$
	`suite` :

- Évaluation avec court-circuit :
  $\rightarrow E_2$ n'est pas évaluée si E_1 FAUX

Affectations

Production	Action sémantique
$I \rightarrow$ `var =` E	$\texttt{var} = E.t$
$I \rightarrow$ `var [` E_1 `] =` E_2	$E_1.t = parcours(E_1)$ $E_2.t = parcours(E_2)$ $\texttt{var}[E_1.t] = E_2.t$

Tant que

Production	Action sémantique

$I \rightarrow \texttt{tq}\ E\ LI$	$\texttt{test} = newetiq()$
	$\texttt{suite} = newetiq()$
	`test :`
	$E.t = parcours(E)$
	`if` $E.t = 0$ `goto suite`
	$parcours(LI)$
	`goto test`
	`suite :`

Si alors sinon

Production	Action sémantique

$I \rightarrow$ `si` $E\ LI_1\ LI_2$	`faux` $= newetiq()$
	`suite` $= newetiq()$
	$E.t = parcours(E)$
	`if` $E.t = 0$ `goto faux`
	$parcours(LI_1)$
	`goto suite`
	`faux :`
	$parcours(LI_2)$
	`suite :`

Entrées et sorties

Production	Action sémantique
$I \rightarrow$ `ecr` E	`write` $E.t$