Génération de code

Rôles de la génération de code

Génération des instructions machine

- Sélectionner les bonnes instructions
- Sélectionner les bons modes d'adressage

Traduction des structures de contrôles (if, while, ...) en branchement (sauts)

Allocation mémoire pour les variables locales

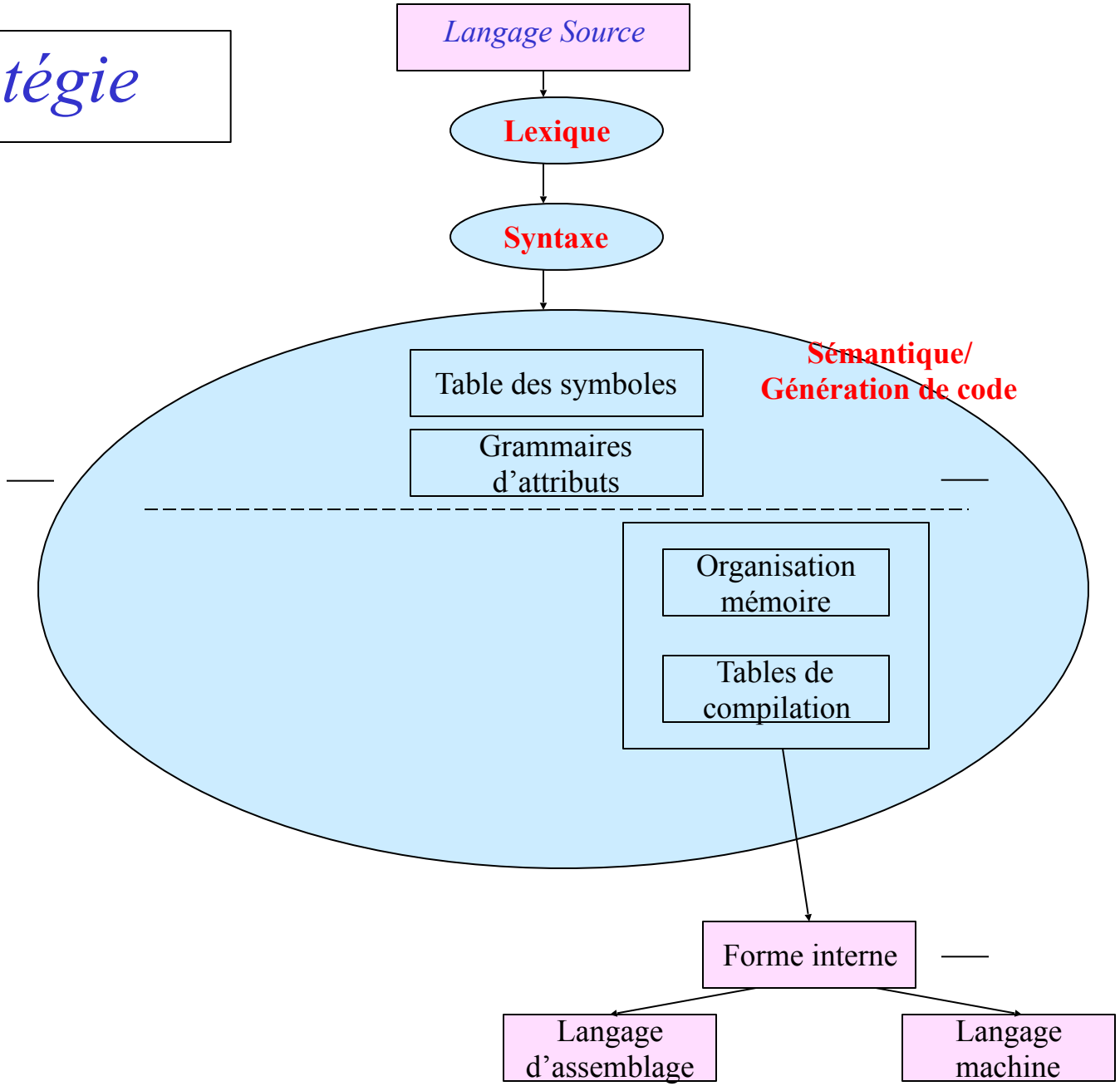
Faire quelques optimisations

Obtenir un fichier objet

Stratégie

1. **Étudier la machine cible**
   registres, formats des données, modes d'adressage, instructions, format des instructions,
2. **Concevoir les structures de données à l'exécution**
   - pile d'exécution pour la gestion des procédures
   - zone de données pour les variables globales, mémoire pour l'allocation dynamique
   - zone pour les constantes, …
3. **Implémenter le codage des instructions**
4. **Implémenter l'allocation des registres si nécessaire**
5. **Implémenter les routines de génération de code** (dans l'ordre suivant)
   - charger les valeurs et les adresses dans les registres (ou dans une pile)
   - traiter les formes x.y, a[i], ...
   - traduire les expressions
   - gérer les sauts et les étiquettes
   - traduire les instructions
   - traduire les méthodes(procédures, fonctions) et le passage des paramètres

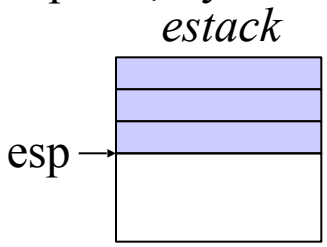
Stratégie


Langage Source
Lexique
Syntaxe
Sémantique/
Génération de code
Table des symboles
Grammaires
d'attributs
Organisation
mémoire
Tables de
compilation
Forme interne
Langage
d'assemblage
Langage
machine

Architecture du langage machine

Langage objet est une machine à pile

- pas de registres
- à la place,il y a une *pile d'expressions* (dans laquelle les valeurs sont chargées)



Taille maximale rangée dans les méta-données de chaque méthode

esp ... pointeur de la pile d'expressions

Comment fonctionne une machine à pile?

Exemple

instruction **i = i + j * 5;**

Suppose les valeurs suivantes de *i* et *j*

locals

0	3	i
1	4	j

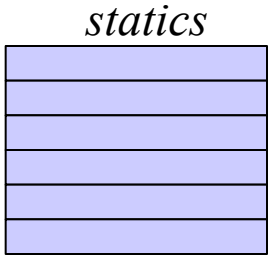
Simulation

instructions	*pile*	
ldloc.0	3	Charger la variable se trouvant à l'adresse 0 (c.a.d. *i*)
lldloc.1	3 4	Charger la variable se trouvant à l'adresse 1 (c.a.d. *j*)
ldc.i4.5	3 4 5	Charger la constante 5
mul	3 20	Multiplier les 2 éléments en sommet de pile
add	23	Ajouter les 2 éléments en sommet de pile
stloc.0		Ranger l'élément en sommet de pile à l'adresse 0

A la fin de chaque instruction la pile d'expression est vide!

Zones de données

Variables globales

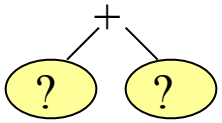


- Représentent les champs statiques de la classe ‘program’
- Les variables globales sont accessibles dans tout le programme
- Les variables globales sont adressées par des unités ‘medadata’

Les opérandes durant la génération de code

Exemple

Nous voulons ajouter 2 valeurs



Modèle de code désiré

load opérande 1
load opérande 2
add

Selon le type de l'opérande nous devons générer différentes instructions de chargement

Type d'opérande *instruction à générer*

Nous avons besoin d'un descripteur, qui nous donne toutes les informations nécessaires au sujet des opérandes.

Trois exemples

Saut conditionnel et inconditionnel

Saut inconditionnel

```
br offset
```

Saut conditionnel

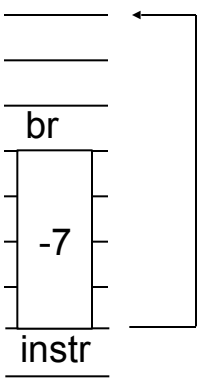
```
... load operand1 ...
... load operand2 ...
beq offset
```

si (opérande1 == opérande2) saut déplacement

beq	Saut si =
bge	Saut si >=
bgt	Saut si >
ble	Saut si <=
blt	Saut si <
bne.un	Saut si #

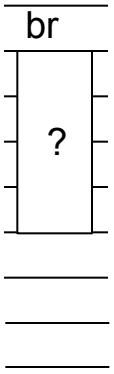
Sauts en avant et arrière

Saut arrière (backward)



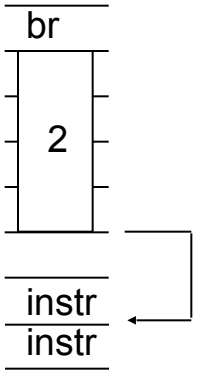
L'adresse cible (étiquette) est déjà connue
(car l'instruction à cet endroit a déjà été générée)

Saut en avant (Forward)



L'adresse cible (étiquette) n'est pas encore connue

- la laisser à vide
- la mettre à jour par la suite



La mise à jour est faite quand l'adresse devie
connue