UNIVERSITÉ DE ROUEN

Compilateur LLVM

Langage jouet Kawa

Document Technique - Analyse sémantique 0.1

3 avril 2015

Auteur(s): Pierre-Luc BLOT

Version	Date	Changelog
0.1	29/03/2015	Version initiale.
0.1.1	31/03/2015	Table des symboles : table de hachage + pile
0.1.2	02/04/2015	Interface c++ de la table des symboles

Table des matières

1 Table des symboles **2**
1.1 Fonctionnement de la table des symboles . . . 2
1.1.1 Comment fonctionne la pile avec la table de hachage ? . . . 2
1.2 Exemple d'analyse sur un programme simple . . . 3
1.2.1 Première étape . . . 3
1.2.2 Deuxième étape . . . 4
1.2.3 Troisième étape . . . 4
1.2.4 Quatrième étape . . . 5
1.2.5 Cinquième étape . . . 5

2 Interface de la table des symboles **6**
2.1 enterBloc . . . 6
2.2 exitBloc . . . 6

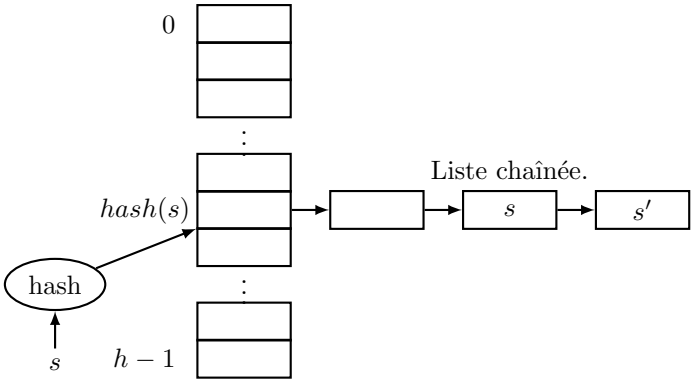
1 Table des symboles

La table des symboles permet de gérer les différents symboles qui seront analysés. Cette partie permet d'établir la logique de cette table des symboles. Afin d'aider à son implémentation, une interface est détaillée en fin de partie.

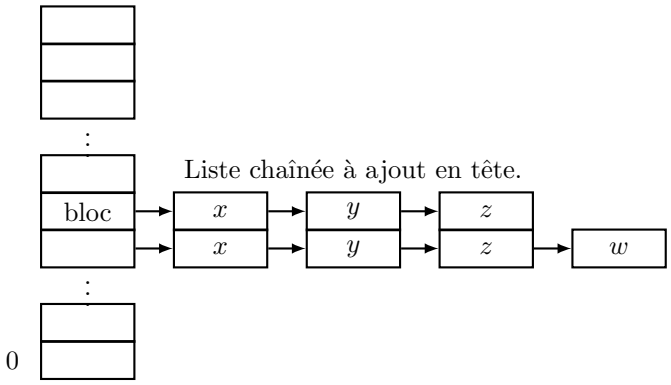
1.1 Fonctionnement de la table des symboles

Il est nécessaire lors de l'analyse sémantique de lire et d'écrire des symboles dans la table. Cette table permet aussi de répondre aux besoins qui sont soulevés par la notion d'accessibilité de certains symboles au sein d'un même scope.

Deux types de structures de données sont utilisées. Une table de hachage dont les valeurs sont des références vers les instances représentant des symboles. Le figure ci-dessous illustre cette structure de données. Dans cet exemple, on illustre l'insertion d'un symbole dont le nom est stocké dans une variable s. Le hash est calculé en fonction de s afin de déterminer sur quelle liste sera ajouté le symbol.



Une deuxième structure de données est utilisée afin de gérer la notion d'accès par scope de visibilité. Une liste de listes à ajout en tête est utilisée ici. Elle sera utilisée comme une pile. L'utilisation d'une pile n'est pas appropriée puisque dans de nombreux cas il sera intéressant de pouvoir parcourir la liste en profondeur. Or cela est coûteux avec une pile. Voici une illustration permettant de visualiser cette structure de données. (x, y, z et w représentent des noms de symbols donc de type String.)



1.1.1 Comment fonctionne la pile avec la table de hachage ?

Les blocs sont imbriqués les uns dans les autres. Lorsque des déclarations sont faîtes dans des blocs imbriqués il arrive que des symboles masquent des symboles déclararés dans des blocs parents.
Quand dans un bloc un symbol est utilisé il fait référence à la première déclaration qui en a été faite. C'est à ce niveau que la pile est importante.

Les niveaux de la pile représentent les niveaux d'imbrications. Dans cette pile, chacuns des niveaux ont

une liste de symboles. C'est symboles sont représentés par leurs noms. Ainsi, grâce à la table de hachage il est aisé de récupérer l'instance correspondant au symbol en question.

Lors de l'analyse, lorsqu'un bloc a été entièrement analysé, les déclarations de symboles qui y ont été réalisés deviennent obsolètes pour la suite de l'analyse. Il est nécessaire de faire du nettoyage dans la pile.

Cette étape consiste à dépiler le bloc courant de la pile. Pour ce faire, le symbol en tête de liste pointée par ce bloc (celui le plus haut dans la pile) est supprimé. Cela doit être répété afin de vider la liste. La suppression est rapide puisque qu'il suffit de supprimer la tete de liste à chaque fois.
Une fois la liste vide, le bloc peut être dépilé.

1.2 Exemple d'analyse sur un programme simple

```
{ // bloc 1
  int a = 2;
  int b = a + 1; // b=3
 { // bloc 2
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

Ce programme réalise différents appels et et différentes déclarations au sein de plusieurs blocs imbriqués. Nous remarquons également que certaines variables sont masquées. Les commentaires nous aides à comprendre l'effet que cela implique sur le programme puisque l'expression a + b n'a pas la même valeur en fonction du bloc dans lequel elle est évaluée. Car ils contiennent des déclarations de a et de b qui masquent les précédentes.

Déroulons pas à pas l'analyse des symboles.

1.2.1 Première étape

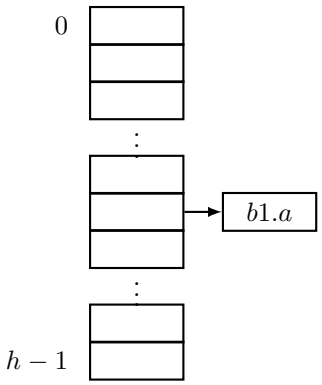
```
{ // bloc 1 <-----
  int a = 2;
  int b = a + 1; // b=3
 { // bloc 2
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

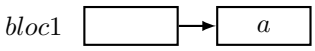
L'analyse commence par le bloc 1. La table de hachage est vide et la pile également.

1.2.2 Deuxième étape

```
{ // bloc 1
  int a = 2; <-----
  int b = a + 1; // b=3
 { // bloc 2
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

Bloc 1 :
Déclaration de variable : int a = 2 ;
La pile est vide donc on doit créer le symbol dans la table.

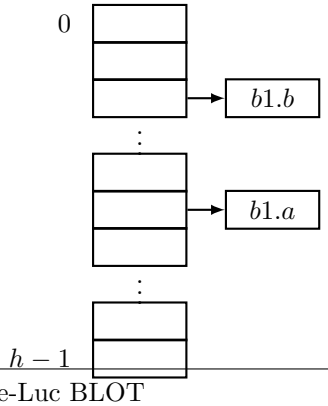


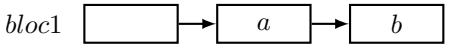


1.2.3 Troisième étape

```
{ // bloc 1
  int a = 2;
  int b = a + 1; // b=3 <-----
 { // bloc 2
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

Bloc 1 :
Déclaration de variable : int b = a + 1 ;
La pile n'est pas vide donc on cherche la définition du symbol a.
On commence par le haut de la pile et on parcours la liste.
Le symbol est trouvé donc il n'y a pas d'erreur de sémantique.
Le symbol b peut être créé.

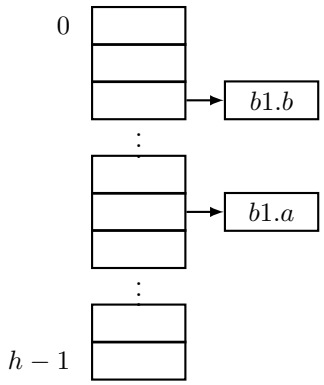


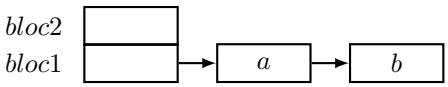


1.2.4 Quatrième étape

```
{ // bloc 1
  int a = 2;
  int b = a + 1; // b=3
 { // bloc 2 <-----
   int a = b + 2; // a=5
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

Bloc 2 : Création d'une liste vide sur la pile

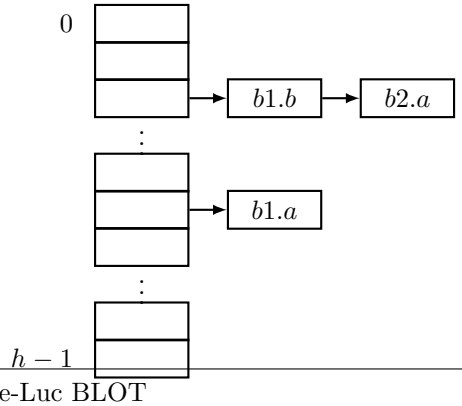


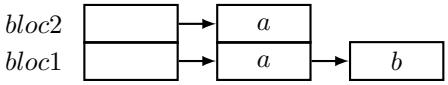


1.2.5 Cinquième étape

```
{ // bloc 1
  int a = 2;
  int b = a + 1; // b=3
 { // bloc 2
   int a = b + 2; // a=5 <-----
   { // bloc 3
     int b = a + 3; // b=8
     int c = a + b; // c=13
   }
   int c = a + b; // c=8
 }
 int c = a + b; // c=5
}
```

Bloc 2 :
Déclaration de variable : int a = b + 2 ;
La pile n'est pas vide donc on cherche la définition du symbol b.
On commence par le haut de la pile et on parcours la liste.
La liste est vide donc on recherche dans le bloc suivant immédiatement plus bas que le bloc courant.
Le symbol est trouvé donc il n'y a pas d'erreur de sémantique.
Le symbol a peut être créé.





2 Interface de la table des symboles

```
/**
 * Interface ITableOfSymbol.
 * Specifie les operations sur la table des symboles.
 */
class ITableOfSymbol{public:
  // Requete

  /**
   * Retourne l'instance representant le symbol en fonction de son nom.
   * @param sym Nom du symbol.
   */
  virtual SemanticPtr* getSymbol(String sym) = 0;

  // Commandes

  /**
   * Place le curseur interne sur un nouveau bloc.
   * Toutes les futures operations se feront sur ce bloc.
   */
  virtual void enterBloc() = 0;

  /**
   * Deplace le curseur interne sur le bloc precedent.
   * Toutes les futures operations se feront sur ce bloc.
   */
  virtual void exitBloc() = 0;

  /**
   * Ajoute une definition de symbol dans la table des symboles.
   * @param sym Nom du symbol
   * @param sptr pointeur sur l'instance qui represente le symbol.
   */
  virtual void addSymbol(String sym, SemanticPtr* sptr) = 0;
}
```

2.1 enterBloc

La fonction enterBloc doit être appellée lorsque l'analyseur sémantique entre dans un bloc. Ainsi un nouveau bloc sera empilé sur la pile. Les recherches de symboles se feront en prenant en compte les niveaux d'imbrications des blocs.

2.2 exitBloc

Réciproquement, lorsque l'analyseur sémantique quitte un bloc, la fonction exitBloc doit être appellée. Cela permet de dépiler un bloc de la pile afin de supprimer les déclarations qui sont obsolètes car innaccessibles depuis l'exterieur du bloc.