Chapitre 4 - Interprétation

INF7641 Compilation

Jean Privat

Université du Québec à Montréal

INF7641 Compilation
v251

Plan

1. Arbres syntaxiques et visiteurs

2. Calculatrice

3. MiniC

4. Interprétations naïve

Arbres syntaxiques et visiteurs

Arbres syntaxiques

Concrets (CST)

- Celui associé à la grammaire (arbre de dérivation)
- Contient tous les terminaux (feuilles de l'arbre)
- Contient tous les non-terminaux (même ceux pas très intéressants)

Abstrait (AST)

- On se concentre sur l'essentiel
- On simplifie et on élague
  - Suppression des jetons purement syntaxiques (mots clés, ponctuation, etc.)
  - Rééquilibrage de l'arbre (listes, opérations binaires, etc.)

Implémentation des AST

- L'AST sera utilisé par le reste du compilateur (de l'interpréteur)
- La plupart des traitements, analyses et transformations peuvent se faire directement dessus
- On veut de la performance, mais aussi s'aider soi-même
- Certains outils ne distinguent pas AST et CST

Détails de conceptions

- Implémentation d'arbres
- Stratégies et pratiques fortement variés
  - Dépendent parfois du langage de programmation
  - Mais souvent de choix de conception
  - Arbres mutables/immutables, homogènes/hétérogènes, parentés/non-parentés...
- Code de l'AST généré par l'outil
  - Oui : c'est parfois contraignant
  - Non : c'est souvent pénible

En SableCC + Java

Données = arbres hétérogènes

Tout est automatique

- Une classe par jeton
- Une classe abstraite par production
- Une sous-classe par alternative
- Un attribut par élément (jeton ou production)

Opérations

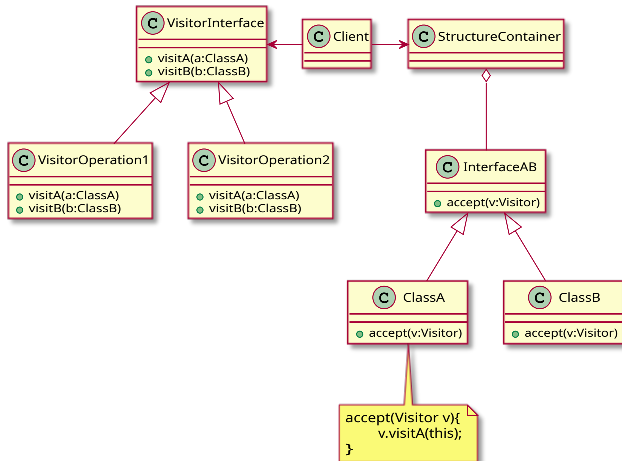
- Getters/setters des nœuds enfants
- Visiteur

Limites SableCC4beta2

- AST = CST
- Code généré parfois rigide

Patron de conception Visiteur (1994)

« Représente une opération à effectuer sur des éléments d'une structure d'objets. Visiteur permet de définir une nouvelle opération sans changer les classes des éléments sur lesquels elle opère. »



Visiteur en SableCC4

- Une classe abstraite `Walker` qui sert de visiteur
- Une méthode `void caseXyz(NXyz node)` par type de nœud
  - Par défaut, visite simplement les nœuds enfants
  - Redéfinissable par des visiteurs concrets
- L'état, si besoin, est à ajouter dans les visiteurs concrets

Limites

- On peut pas personnaliser la signature des méthodes `caseXyz`
- Mais on se débrouille:
  - Stocker arguments et valeurs de retour dans des attributs

Calculatrice

Langage des expressions arithmétiques (calc)

Vu la semaine dernière

```
Language calc;
Lexer
int = ('0'..'9')+;
Ignored #10, #32;

Parser
exp = {add:} [left:]exp '+' [right:]exp |
      {sub:} [left:]exp '-' [right:]exp |
      {mul:} [left:]exp '*' [right:]exp |
      {int:} int |
      {par:} '(' exp ')' ;
   Priority
      Left mul;
      Left add, min;
```

Exercice : implémenter un évaluateur

Conception possible

- Une sous-classe `Calc` de `Walker` sert d'évaluateur
- Un état « dernière valeur » dans l'évaluateur
  - Alternative : mémoriser la valeur de chaque sous-expression, mais c'est superflu ici
- Une méthode utilitaire `int visitExpr(Node n)`
  - Elle visite l'expression `n` et retourne la « dernière valeur »
- Redéfinir les méthodes `caseXyz` pertinentes
  - Utilise `visitExpr` pour chaque nœud enfant
  - Calcule et positionne la « dernière » valeur
- Laisser la récursivité et le polymorphisme faire le travail !

Moins de 100 lignes de Java !

Risque

- Oublier de redéfinir une méthode `case*` n'est pas signalé.
  Ça peut être difficile à déboguer

MiniC

MiniC

- Un vrai langage de programmation
  - D'entraînement
    - En mousse
- Pour pas (trop) se blesser

Intersection non vide avec le C

- On peut utiliser un vrai compilateur C pour se comparer
  - Sur les programmes valides en C et MiniC
- Plein de choses en C mais pas en MiniC
  - Parce que le C c'est gros et compliqué
- Certaines choses en MiniC mais pas en C
  - Pour expérimenter avec de la variété

Source: As Crônicas de Wesley, 2018

Syntaxe de MiniC

Fonctions

- Juste `int main() { ... }`, en dur pour l'instant

Instructions

- Déclaration d'une variable avec valeur initiale
  - Les variables sont réputées globales (pour l'instant)
- Affectation d'une variable
- Conditions `if` et `if/else`
  - Avec accolades obligatoires pour pas à avoir à gérer l'ambiguïté
- Boucles `while`
  - Avec accolades obligatoires pour être cohérent
- Affichage `printint(...)` et `println()`
  - Ce sont des mots clé en MiniC
  - Pas d'appel de fonctions pour l'instant

Syntaxe de MiniC (suite)

Commentaires

- `//`, `/* */` et `#` (`#` c'est une astuce)

Expressions

- `+`, `-`, `*`, entiers littéraux, parenthèses (comme Calc)
- `<`
- Lecture d'une variable

Types

- Juste `int` pour l'instant

Exemple de programme MiniC

```c
#include "minic.c"

int main() {
    int a = 0;
    int b = 1;
    int n = 0;
    while (n<15) {
        int t = a;
        a = b;
        b = b + t;
        n = n + 1;
    }
    printint(a);
    println();
}
```

Astuce C ∩ MiniC

En MiniC

- `#include "minic.c"` est un commentaire
- `printint` et `println` sont des mots clé

En C

- `printint` et `println` sont des appels de fonction
- `minic.c` est un fichier qui implémente ces fonctions
- Note: c'est **très sale** d'inclure un `.c`
  - Mais je vous donne une dérogation spéciale
  - Et puis ça simplifie la compilation
  - `gcc factorielle.c -o factorielle`

Exercice : développer un interpréteur MiniC

SableCC

- Écrire le fichier `minic.sablecc`
- Partez de `calc.sablecc`

Interpréteur

- Écrire l'interpréteur `MiniC.java`
- Partez de `Calc.java`

Conception possible

- Un `HashMap` qui associe nom de variable avec un `int`
- On implémente les `caseXyz` manquants
- C'est à peu près tout

Moins de 200 lignes de Java !

Interprétations naïve

Interprétations naïve

Découverte du programme au fur et à mesure de l'interprétation

- Pas très efficace, car on pourrait savoir d'avance ce qui s'en vient (prétraitement)
- Pas très robuste, car les *problèmes* sont découvert à l'exécution
- Sauf, bien sur, pour ce qui a déjà été traité et validé lors des analyses lexicales et syntaxiques

Interprétation récursive

- Ça rend le traitement de certaines fonctionnalités plus compliqué
- `return`, `break`, `continue`, exceptions, déboguage interactif, etc.

Analyses sémantiques

On préretraite le programme avant de l'interpréter (ou de le compiler)

- Plus d'efficacité
- Plus de robustesse

La semaine prochaine !