Département IMA / 4A (S8)
Informatique Fondamentale

TP IF - Langage d'équations booléennes

Prérequis

Ce TP est basé sur le travail réalisé durant les TPs de CLO. Afin de pouvoir réaliser ce TP, il est donc nécessaire que les points suivants soient fonctionnels :

- Hiérarchie de classes des composants (CLO-TP2) ;
- Description des composants (méthode `String description()`, CLO-TP2) ;
- Évaluation des composants (méthode `boolean getEtat()`, CLO-TP3) ;
- Classe de circuits (CLO-TP4).

Présentation

L'objectif de ce tutorat est de réaliser un compilateur capable de traiter un petit langage d'équations booléennes. La représentation interne du langage (Abstract Syntax Tree, AST), ainsi que l'évaluation des équations, se basent sur le travail réalisé en TP de CLO. Le tutorat se concentre sur l'analyse syntaxique du langage. On donne ci-dessous un exemple de programme du langage à compiler :

Exemple 1

```
eq_circuit(in1,b,c) returns (out,o2)
  out=(in1 & b)|c;
  o2=(c&in1);
end

descr()
eval(true,true,false)
```

On définit informellement le langage comme suit :

- Un *programme* est constitué d'un *circuit équationnel* et d'une liste de *commandes* à appliquer sur ce circuit ;
- Un circuit équationnel est défini par :
  - Un ensemble de *ports* d'entrée nommés ;
  - Un ensemble de *ports* de sortie nommés ;
  - Un ensemble d'*équations*.
- Une équation définit la valeur d'un port à l'aide d'une *expression booléenne* ;
- Une expression booléenne est une combinaison des éléments suivants :
  - Un opérateur booléen de l'ensemble suivant : {`&`, `|`, `!`} ;
  - Un port du circuit ;
  - Une expression booléenne entourée de parenthèses.
- $descr$ est une commande permettant d'afficher la description du circuit ;
- $eval$ est une commande permettant d'effectuer l'évaluation du circuit (et d'en afficher le résultat) en affectant les valeurs constantes booléennes spécifiées lors de son "appel" aux entrées du circuit ;
- Pour le reste de la syntaxe, se référer à l'exemple.

1 Grammaire

1. Définissez sur papier la grammaire de ce langage. La structure de votre grammaire devrait suivre la description inductive du langage donnée précédemment. L'énoncé précise les non-terminaux principaux en italique. Notez bien qu'à cette étape la grammaire est totalement indépendante de l'AST que l'on va utiliser (`Circuit`, `Composant`, etc) ;
2. Implémentez la grammaire au format `AntLR4`. Pour cela, on définit d'abord le vocabulaire (mots clés, identifiants, constantes, opérateurs, ponctuation) utilisé dans le Lexer. Ensuite, les règles de grammaires, qui manipulent ce vocabulaire, sont définies dans le Parser. On n'associe pour l'instant aucune action aux règles du Parser ;
3. Validez votre grammaire sur des fichiers syntaxiquement corrects (au minimum celui de l'exemple), ainsi que sur des fichiers syntaxiquement incorrects.

2 AST

2.1 Classes Java

On commence tout d'abord par apporter les modifications suivantes à la hiérarchie de classes des composants :

1. Copiez depuis `~jforget/Public/IF` la classe `EquationCircuit`, qui permet de représenter un programme de notre langage. Prenez le temps de parcourir ce code. La principale nouveauté, par rapport à la classe `Circuit`, est de rajouter un ensemble d'entrées/sorties nommées et les méthodes associées ;
2. Pour que cette classe compile, ajoutez un champs `name` dans les classes `Interrupteur` et `Vanne`, ainsi qu'un constructeur adapté (cf constructeurs utilisés dans `EquationCircuit.setInputs/setOutputs`) ;
3. Définissez une méthode permettant d'ajouter un `Composant` dans un circuit existant.

2.2 Actions dans le Parser

On va maintenant réaliser la construction de l'AST dans le parser :

1. Comme chaque fichier de notre langage ne contient qu'un circuit, on déclare le circuit comme une variable membre de la grammaire (`@members`, cf S12 du cours), ce qui permet d'accéder au circuit depuis n'importe quelle règle de la grammaire ;
2. Lors de l'analyse de la *signature du circuit*, on initialise les listes de ports d'entrées/sorties du circuit (`setInputs/setOutputs`). Il est nécessaire de faire cette initialisation **avant** de traiter les équations, car au moment de traiter les équations on accède aux entrées/sorties du circuit. Pour exemple, ci-dessous `instr1` est exécutée après l'analyse de `symb1`, mais celle de `symb2` et donc avant l'exécution de l'instruction `instr2` :
   `regle : symb1 {instr1;} symb2 {instr2;}`
3. Lors de l'analyse d'une *équation* (`v=expr`), on récupère la sortie correspondant à `v` (via `getOutput`) et on connecte son entrée (`setIn`) au composant correspondant à `expr` ;
4. Une *expression* a un attribut (`returns`) de type composant. Lors de l'analyse de la règle, on crée ce composant, on le rajoute au circuit(`add` de `Circuit`), on connecte ses entrées (`setIn1/2`) aux composants correspondant à ses sous-expressions, et enfin on affecte le composant créé à l'attribut de la règle.

3 Analyse des commandes

Pour l'implémentation des commandes, vous pouvez directement appeler dans les actions du Parser les méthodes correspondantes de la classe `EquationCircuit` (méthodes `descr` et `eval`).

4 Extensions

Pour les plus rapides, on propose les extensions suivantes :

1. Les constantes `true` et `false` dans les expressions booléennes ;
2. Des variables intermédiaires/internes au circuit, qui ne font partie ni des entrées ni des sorties du circuit ;
3. Plusieurs circuits dans un même fichier. Adaptez `descr/eval` en conséquence ;
4. Permettez d'instancier un circuit dans un autre circuit, à la manière d'un appel de fonction (plus difficile).