Projet de Compilation – Premier devoir

Modalités pratiques

Ce projet complet de compilation correspond à 3 devoirs qui doivent être rendus dans les temps impartis indiqués par les enseignants.

Les trois devoirs correspondent à :

1. Analyse lexicale et syntaxique
2. Analyse sémantique
3. Génération et optimisation du code

Le but de ce projet est d'écrire un compilateur pour le langage procédural déclaratif présenté en annexe A. Ce langage est une ébauche d'un langage plus complet qui pourrait servir à des fins pédagogiques comme cela a été le cas du langage Pascal par exemple.

La présente section ne concerne que le premier devoir.

Première étape : création d'un dépot et développement de l'analyse lexicale et syntaxique

1. Créer un dépôt `svn` au Cremi.
2. Écrire un fichier `build.xml` pour avoir une compilation complète (cible `all`) grâce à la commande `ant`, un nettoyage à utiliser avant un archivage par exemple (cible `clean`), une phase de tests du projet sur des exemples (cible `test`)
3. Écrire les fichiers `Jflex`, `Cup` et `Main.java` pour un sous-ensemble du langage.

Le projet permettra d'intégrer très facilement les éléments de programmation qui suivront dans les devoirs 2 et 3. Pour cela, il est très important que le code soit organisé et très propre.

1 Étapes

1. Ecrire le code de l'analyseur lexical de sorte que le programme puisse analyser :
   — Les mots clef du langage de programmation,
   — Les commentaires de type Java sur une ou plusieurs lignes,
   — Les valeurs en nombre à virgule flottante (4.56e3),
   — Les chaînes de caractères entre guillemets comprenant les caractères d'échappements,
   — Les caractères simples entre apostrophes ou en valeur hexadécimale
2. Compléter le code de sorte que le programme affiche un message avec numéro de ligne et numéro de colonne en cas d'erreur de l'analyse lexicale ou de l'analyse syntaxique.

3. Faire en sorte que le programme puisse continuer l'analyse du code après une erreur lexicale ou syntaxique.
4. Faire en sorte que le programme puisse accepter des expressions complexes du langage. On ne demande pas d'implémenter l'évaluation des expressions ni leur type et encore moins la génération de code. Il faut simplement que les erreurs syntaxiques soient détectées.

Annexe A : spécifications du langage de programmation

Dans ce qui suit, seront notées entre chevrons les termes utilisés pour désigner des éléments du langage et non directement les termes du langage. Par exemple

`<identifier>: <type>;`

désigne une chaine de caractères où `<identifier>` et `<type>` ne sont pas littéralement écrits, mais substitués par leurs valeurs.

Commentaires

Même chose que Java.

Programme principal

La procédure `main` correspond au programme principal. Elle prend en argument un entier et une liste de chaines.
Exemple :

```
1 procedure main(argn: integer, argc: list of string)
2 {
3 }
```

Déclaration et implémentation des procédures et fonctions

Comme en C, la déclaration d'une fonction est suivie d'un point-virgule. On fait suivre la déclaration d'un **bloc** pour l'implémenter. Tous les arguments sur un mot sont passées par valeur, toutes les autres sont passées par pointeur.

Bloc

Un bloc est une suite d'instructions précédée d'une déclaration de variables locales au bloc.
Exemple :

```
1 {
2   aux: integer;
3   aux=x;
4   x=y;
5   y=aux;
6 }
```

Déclaration des variables

Toutes les variables doivent être déclarées avant d'être utilisées et doivent être statiquement typées. Le nom de la variable est suivi de deux points et de son type.
Exemples :

```
i: integer;
t: list of integer;
p: ^integer;
t2: list of structure {
  nom: string;
  date_de_naissance: integer;
};
```

Types simples

- `integer` (un mot)
- `character` (un mot)
- `float` (un mot)
- `boolean` (un mot)
- `string` (un pointeur sur un mot, la séquence est terminée par 0)
- `'<identifier>` (variable de type)

Types complexes (au sens où ils contiennent des types)

- `list of <type>` Désigne une liste de d'éléments de type `<type>`
- `^<type>` Désigne un pointeur sur un objet de type `<type>`
- `structure` {`<structure_content>`} Où `<structure_content>` désigne un ensemble de champs séparés par des points-virgules
- (**) `class` {`<class_content>`}, où `<class_content>` désigne un ensemble de variables et de fonctions d'instances séparés par des points-virgules.

Déclaration de type

Le programmeur peut lui-même définir ses propres types en utilisant le mot clef `type` et le signe `=`
Exemple :

```
type client = structure {
  nom: string;
  date_de_naissance: integer;
};
monTableau: list of client;
```

Expressions

Opérandes

- Variable désignée par un identificateur
- Constantes : `true`, `false`, `null`
- Nombres entiers
- Nombres à virgule flottante
- Caractère écrit entre deux `'`
- Chaine de caractères écrite entre deux `"`

Opérateurs

Les opérateurs suivants seront disponibles, avec leur sémantique habituelle (comme en C).

— Opérateurs arithmétiques `+, -, *, /, %, + unaire, - unaire, ++, --`
— Opérateurs d'affectation `=, (op)=`
— Opérateurs relationnels de comparaison `<, <=, >, >=, ==, !=`
— Opérateurs logiques `&&, ||, !`
— Opérateurs de manipulation de bits `&, |, ~, ^, <<, >>`
— Opérateurs d'accès à la mémoire `&, *, [], ., ->`
— Appel de procédures et fonctions `()`

Opérations complexes

Les listes peuvent être construites ainsi :

— Liste littérale `[<expr>, <expr>, ..., <expr>]`
Exemple :

```
1  t: list of integer;
2  t = [1, 3, 5, 7, 9];
3  foreach i in [0, 1, 2, 3, 4] {
4    afficher(t[i]);
5  }
```

— (**) Accès à un champ d'un objet. `<objet>.<champ>`
Exemple :

```
1  class Fiche {
2      //    private static uniqId: int;
3      static id: integer;
4      nom: string;
5      age: integer;
6
7      // définition d'un constructeur
8      fiche(){
9          nom := "";
10         age := 0;
11     }
12
13     // définition d'un destructeur
14     ~fiche(){
15     }
16
17     // fonction par défaut public et membre d'instance
18     function getNom(): string{
19         return nom;
20     }
21 }
```

Structures de contrôle

Les instructions sont les suivantes :

Instructions simples

- `<left_hand_side> = <expr>;` Affectation
- `<identifier>(<list_args>);` Appel de procédure, où `list_args` est une liste d'expressions séparées par des virgules
- `return <expr>;` Fin de fonction avec renvoi de valeur
- `stop;` Fin de procédure
- `break;` Fin de boucle

Instructions complexes

- `if <boolean_exp> <stm>`
- `if <boolean_exp> <stm> else <stm>`
- `foreach <identifier> in <list_or_set> <stm>`
- `while <boolean_exp> <stm>`
- `repeat <stm> while <boolean_exp>`