Projet d'analyse syntaxique

Université Bordeaux 1 — 2011-2012

1 Modalités de réalisation

Deux sujets au choix sont proposés. Le langage d'implémentation est également au choix: soit C/flex/bison, soit Java/Jflex/CUPS ou BYACC/J.

Chaque membre d'un groupe doit réaliser une partie significative du travail de programmation. L'écriture seule du rapport, par exemple, est considérée comme insuffisante. Chaque membre d'un groupe doit connaître les options générales choisies par le groupe ainsi que les difficultés rencontrées. Chacun devra exposer individuellement sa contribution lors de la soutenance (**semaine du 21 mai**).

Le projet sera présenté dans un rapport synthétique (5–10 pages) illustré par des exemples, exposant les problèmes rencontrés et les choix effectués pour les résoudre. Remise du source et du rapport par mail (aux chargés de TD et de cours): **20 mai 2012 à midi** au plus tard.

Pour chacun des projets, un jeu de test sera fourni. La notation tiendra compte du degré de réalisation du sujet (combien de constructions sont couvertes dans le projet « compilateur », par exemple), de la qualité du code (portabilité, lisibilité, documentation), du nombre de tests traités avec succès, de la qualité du rapport, et de la présentation orale. La notation pourra varier à l'intérieur d'un groupe en cas de travail trop inégal, et tout plagiat détecté sera sévèrement sanctionné.

Sujet 1: Un générateur d'analyseur syntaxique LL(1)

À réaliser par groupe de 2 personnes, 3 avec les extensions proposées

Ce projet demande la réalisation d'un générateur d'analyseur syntaxique. Le générateur lit une spécification de grammaire dans le fichier qui lui est donné en argument et génère un analyseur LL(1) pour cette grammaire. La spécification de la forme du fichier décrivant la grammaire est la même que pour `bison`. Les trois sections devront être présentes. Dans la première partie, on permettra en particulier le bloc littéral que le générateur se contentera de copier dans le code de l'analyseur. Les directives `%union`, `%token`, `%type` devront pouvoir être traitées par le générateur.

Le fichier devra être compatible avec le format `bison`. En particulier, la forme d'entrée de chaque règle de grammaire sera du même type que celle employée par `bison` :

```
non-terminal:   membre droit de la règle 1 { action 1 }
              | membre droit de la règle 2 { action 2 }
              ...
              ;
```

chaque règle étant alors considérée comme distincte. Un membre droit peut par ailleurs être vide.

Le générateur d'analyseur syntaxique doit fournir :

1. la liste des ensembles `Premier(X)` et `Suivant(X)` pour chaque non-terminal `X`.
2. la table d'analyse LL(1), consultable sous forme texte.
3. le code d'un analyseur syntaxique prenant en entrée une chaîne de terminaux, construisant l'arbre de dérivation associé et indiquant les éventuelles erreurs.

Les lexèmes (*tokens*) seront définis par l'utilisateur, et pourront aussi être donnés sous forme littérale, représentés par un caractère. Si la grammaire n'est pas LL(1), l'analyseur doit l'indiquer. Les conflits devront pouvoir être levés à la main, le générateur demandant de façon interactive à l'utilisateur quelle règle appliquer dans un cas de conflit.

Enfin, les lexèmes doivent pouvoir porter un attribut. Comme dans `bison`, on pourra indiquer au niveau de la spécification de la grammaire des calculs d'attributs à effectuer. On se restreindra à des attributs synthétisés. Dans une action, les attributs seront notés `$$`, `$1`, `$2`,... avec la même signification qu'en `bison`. Le type des attributs sera de même donné dans une directive `%union`. Chaque action sera du type `$$=f($i,$j,...)`. L'initialisation des attributs au niveau des feuilles de l'arbre se fera via la variable `yylval`, qui devra être accessible à une fonction d'analyse lexicale.

Extension: transformation dune grammaire non LL(1)

Si cette extension est réalisée, un groupe peut comprendre 3 personnes.

Lorsque la grammaire n'est pas LL(1), l'analyseur appliquera des méthodes pour essayer de la rendre LL(1), tout en conservant le langage engendré: technique de factorisation, et suppression de la récursivité gauche. On demande dans ce cas d'écrire les algorithmes pour

- transformer la grammaire par ces techniques,
- réécrire la grammaire obtenue dans un fichier texte, au format bison,
- appliquer à nouveau l'algorithme LL(1) pour tester si la nouvelle grammaire est LL(1), et si oui, produire la table d'analyse au format texte et l'analyseur syntaxique en C ou Java.

Sujet 2: Un compilateur d'un sous-ensemble de C

À réaliser par groupe de 3 personnes

L'objectif est de construire un compilateur d'un petit langage de programmation correspondant à un sous-ensemble du langage C, vers l'assembleur X86 en syntaxe AT&T. Pour vérifier le fonctionnement du compilateur, on demande que l'entrée et la sortie soient compilables en exécutable par `gcc`.

Le langage source: éléments lexicaux

Les commentaires sont compris entre `/*` et `*/` et ne s'imbriquent pas. Les blancs sont interdits au milieu d'un mot-clé, ignorés ailleurs. Un `identificateur` est une suite de lettres, et une constante entière `nb_entier` est une suite de chiffres. Le lexème virgule `,` joue le rôle de séparateur d'identificateurs. Le point-virgule `;` joue le rôle de terminateur d'instruction. Les opérateurs relationnels sont `==`, `!=`, `<=`, `>=`, `<`, `>`. Les opérateurs logiques sont `||` (ou), `&&` (et), et `!` (non). L'affectation est notée `=`. Les opérateurs arithmétiques sont `+`, `-`, `*`, `%` et `/`. Le moins unaire est aussi noté `-`. Parenthèses `()`, crochets `[]`, accolades `{}` sont des lexèmes utilisés comme en C.

Le langage source: syntaxe

La syntaxe du langage source est décrite par la grammaire suivante, où les terminaux sont indiqués en `fonte courrier` et les non terminaux *<entre crochets>*. Il est permis de modifier la grammaire mais **pas** le langage engendré.

<programme> ⟶ *<liste de déclarations>* *<liste de fonctions>*
<liste de déclarations> ⟶ ε | *<déclaration>* `;` *<liste de déclarations>*
<déclaration> ⟶ `int identificateur` | `static int identificateur`
| `int identificateur[nb_entier]`
| `static int identificateur[nb_entier]`
| *<prototype>*
<prototype> ⟶ *<type>* `identificateur(` *<suite de types>* `)`
<type> ⟶ `void` | `int`
<suite de types> ⟶ `void` | *<liste de types>*
<liste de types> ⟶ `int` | `int,` *<liste de types>*
<liste de fonctions> ⟶ *<fonction>* | *<fonction>* *<liste de fonctions>*
<fonction> ⟶ *<en-tête>* `{` *<corps>* `}`
<en-tête> ⟶ *<type>* `identificateur(` *<suite de paramètres>* `)`
<suite de paramètres> ⟶ `void` | *<liste de paramètres>*
<liste de paramètres> ⟶ *<paramètre>* | *<paramètre>* `,` *<liste de paramètres>*
<paramètre> ⟶ `int identificateur`
| `int identificateur[nb_entier]`
<corps> ⟶ *<liste de déclarations>* *<liste d'instructions>*
<liste d'instructions> ⟶ ε | *<instruction>* `;` *<liste d'instructions>*
<instruction> ⟶ `identificateur =` *<expression>*
| `identificateur[` *<expression>* `] =` *<expression>*
| `exit(nb_entier)` | `return` *<expression>* | `return`
| `read_int(identificateur)`
| `if(` *<expression>* `)` *<instruction>* `else` *<instruction>*
| `if(` *<expression>* `)` *<instruction>*
| `while(` *<expression>* `)` *<instruction>*
| `{` *<liste d'instructions>* `}`
| *<expression>*
<expression> ⟶ *<expression>* *<opérateur binaire>* *<expression>*
| `-` *<expression>* | `!` *<expression>* | `(` *<expression>* `)`
| `identificateur` | `nb_entier`
| `identificateur(` *<suite d'arguments>* `)`
| `identificateur[` *<expression>* `]`
<suite d'arguments> ⟶ ε | *<liste d'arguments>*
<liste d'arguments> ⟶ *<expression>* | *<expression>* `,` *<liste d'arguments>*
<opérateur binaire> ⟶ `>` | `<` | `==` | `<=` | `>=` | `!=` | `+` | `-` | `*` | `/` | `%` | `||` | `&&`

Le langage source: sémantique

Les seuls types de base sont les types `void`, `int` et le type chaîne de caractères. Ce dernier n'est utilisé que sous forme de constante. L'instruction `printf("Hello world %d", 2012)` est donc légale et devra pouvoir être compilée. Le seul type nouveau que l'on peut construire est le type tableau d'entiers. Les fonctions peuvent être appelées récursivement, les paramètres étant passés par valeur. Bien remarquer que les tableaux peuvent être passés en argument.

La sémantique des différents opérateurs, ainsi que leur associativité et leur priorité est la même qu'en C. Les variables sont allouées dynamiquement sauf si elles sont globales ou déclarées statiques. Comme en C, le point d'entrée du programme est la fonction `main`, de prototype `int main(void)`.

Une fonction d'entrée `read_int` a été ajoutée. Elle permet de lire un entier sur l'entrée standard: `read_int(n)` est équivalente à `scanf("%d",&n)`. Cet ajout est nécessaire pour pouvoir tester le programme plus facilement, dans la mesure où les pointeurs (et donc la construction `&n`) ne sont pas gérés par le langage qu'on demande de compiler.

Le langage cible

Le langage cible est l'assembleur X86, syntaxe AT&T. Pour le jeu d'instructions, cf. App. B du livre `http://ftp.igh.cnrs.fr/pub/nongnu/pgubook/ProgrammingGroundUp-1-0-booksize.pdf` ainsi que `http://dept-info.labri.fr/ENSEIGNEMENT/archi/`.

Gestion des erreurs

Le compilateur ne devra pas arrêter l'analyse à la première erreur: on demande une resynchronisation, en utilisant les mécanismes fournis par `bison`.