UNIVERSITÉ TOULOUSE III PAUL SABATIER

TP 2 – Améliorer Autocell

Ce TP sont automatiquement évalués en se basant sur l'archive que vous aurez déposé, dans les temps, sur la page Moodle. Pour construire l'archive, il faut taper la commande :

```
> make archive
```

Cela produit un fichier nommé `archive-DATE.tgz` qu'il faudra déposer. Les TP de compilation sont organisé, à peu près, chaque 15 jours et la date limite de dépôt est habituellement placé le dimanché précédent la semaine de TP.

Ce TP a pour objectif de mettre en oeuvre une procédure permettant le compilateur de Cellang et de réaliser l'ajout des variables et des expressions arithmétiques.

1 Vue générale

Comme présenté durant le cours, le compilateur est divisé entre le frontal (qui analyse le programme source) et le back-end (qui génère le code machine). Le frontal est également divisé en 3 phases :

1. l'analyse lexicale générée par `ocamllex` à partir du fichier `lexer.mll`,
2. l'analyse syntaxique générée par `ocamlyacc` à partir du fichier `parser.mly`,
3. l'analyse sémantique écrite en *OCAML* et insérée dans les actions de `parser.mly`.

Remarquez que vous n'avez pas à appeler les différents outils vous-même mais juste à utiliser le `Makefile` fourni :

```
> make
```

1.1 L'analyseur syntaxique

Le programme principal est l'analyseur syntaxique. Le langage de programmation est décrit en utilisant une grammaire. L'exemple ci-dessous montre les règles pour une `expression` et pour désigner une `cellule` (extrait de `parser.mly`) :

```
expression:
        cell
                { CELL (0, fst $1, snd $1) }
|       INT
                { CST $1 }
;
```

```
cell:
        LBRACKET INT COMMA INT RBRACKET
                { ($2, 4) }
;
```

La règle commence par le nom du non-terminal, " :", suivi par plusieurs productions séparées par "|" et terminé par ";". Chaque production est une séquence symboles (terminaux en majuscule, non-terminaux en minuscule) terminée par une action entre "{...}". L'action est écrite en OCAML et *est exécutée quand la production est réduite durant l'analyse* $LALR(1)$. A l'exception de l'action, la syntaxe de la grammaire est très proche de celle présentée en cours et est donc traitée de la même manière.

Les terminaux (aussi appelés *tokens*) sont déclarés dans `ocamlyacc` mais sont produits par l'analyseur lexical qui traite le programme source. Dans `parser.mly`, la déclaration des tokens est réalisé par les lignes :

```
%token DIMENSIONS
%token OF
%token ASSIGN
%token COMMA
```

Les identificateur de terminaux doivent être exprimés en majuscule pour suivre les règles standards d'OCAML. Ils sont transformés en type union par `ocamlyacc` :

```
type token =
  | DIMENSIONS
  | OF
  | ASSIGN
  | COMMA
  | ...
```

A faire Examiner le fichier `parser.mly` pour localiser les éléments présentés ci-dessus.

1.2 L'analyse lexicale

Le travail de l'analyseur lexical est de traiter le texte source afin de reconnaître les terminaux et de renvoyer leurs valeurs à l'analyseyr syntaxique. Remarquez qu'il n'y a pas de lien direct entre le nom des terminaux et les mots qui leur correspondant. Si le terminal `DIMENSIONS` correspond au mot-clé `dimensions`, `ASSIGN` correspond au mot ":=". Ainsi, `ocamlyacc` n'a pas besoin de connaître la forme exacte des terminaux.

Ces terminaux dans `ocamllex` sont analysés en utilisant les règles suivantes (extraites de `lexer.mll`) :

```
rule token = parse
        ...
|       "dimensions"      { DIMENSIONS }
```

```
|       "of"                    { OF }
|       ":="                    { ASSIGN }
|       ','                             { COMMA }
...
```

Chaque règle commence avec un "|", suivi par une *expression régulière* (RE) et est terminée par une action entre "{...}". LE rôle de cette action est de renvoyer le terminal correspondant au mot analysés mais cela peut être n'importe quel code OCAML.

Le mot-clé `INT` est très intéressant : il ne correspond seulement à un seul mot mais à toute la famille de mots représentant une valeur entière (en décimal). D'un point de vue grammatical, il est suffisant de savoir qu'il s'agit d'un entier. Mais, à un certain point de la compilation (par exemple dans le back-end), le compilateur aura besoin de connaître la vraie valeur de l'entier. Une telle valeur est appelée *valeur sémantique* et est déclarée avec le terminal en utilisant la syntaxe (le type de la valeur sémantique est `int`) :

```
%token<int> INT
```

De son côté, `ocamllex` doit calculer cette valeur et la renvoyer :

```
|       dec     as n            { INT (int_of_string n) }
```

`dec` est une ER nommée reconnaissant un entier décimal. Le mot reconnu est une chaîne de caractère appelée `n` qui est convertie en entier et passée en paramètre au terminal `INT`.

L'ER `dec` déclarée par :

```
let digit = ['0'-'9']
let sign = ['+' '-']
let dec = sign? digit+
```

Qui doit être comprise comme :
- `dec` peut éventuellement démarre par un `sign`,
- puis il est composé d'un séquence non-vide de `digit`,
- un `sign` est soit un caractère '+' ou un caractère '-' (les apostrophes sont obligatoires),
- un `digit` est un caractère entre '0' et '9' (chiffres decimaux).

A faire Examiner le fichier `lexer.mll` pour localiser les éléments présentés jusque là.

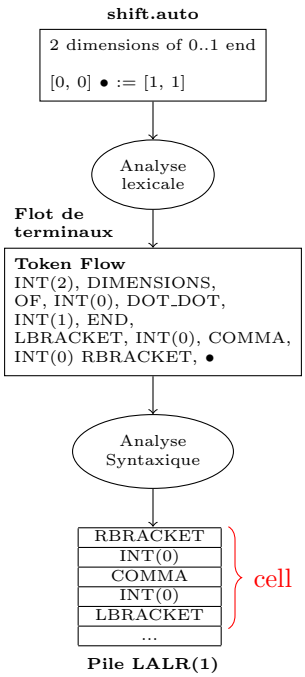
Note : les ER utilisés dans ocamlyacc peuvent être :
- "c_1c_2..." pour analyser la séquence de caractère C_1, c_2, ...,
- [*chars*] avec *chars* une séquence caractères seuls `'c'` ou d'intervalle de caract-re `'`c_1`'-'`c_2`'` (notez que seules les séquences de chiffres, de lettre majuscules ou minuscules sont acceptées),
- E`*` pour répéter E zéro ou plusieurs fois,
- E`+` pour répéter E une ou plusieurs fois,
- E_1 E_2 ... reconnaît la séquence E_1 puis E_2 puis...,
- E_1 | E_2 reconnaît E_1 ou E_2,
- `(` E `)` pour gérer la prioité dans les ER.
- _ (souligné) représente n'importe quel caractère $c \in \Sigma$.

1.3 Etendre notre compilateur

Pour résumer,

1. L'analyse lexicale lit les caractères correspondant aux terminaux, par exemple "123".
2. Ce texte est reconnue par le DFA corresponding à l'ER de `dec`.
3. L'action correspondante est appelée avec $n =$ "123".
4. Elle construit le terminal `INT 123` qui est transmis à l'analyseur syntaxique.
5. L'analyseur syntaxique utiliser le terminal pour progresser dans la table d'analyse LALR(1).
6. Quand une poignée de symbole a été accumulée dans la pile (provoquant une réduction), l'action correspondant au non-terminal est appelée. Par exemple, l'action de `cell` avec pour poignée `LBRACK INT COMMA INT RBRACKET`.

Le flot depuis le texte du fichier vers les terminaux et la pile d'analyse LALR(1) est montré ci-dessous :



Donc pour étendre le langage, il faut :

1. déclarer les terminaux manquants dans `parser.mly`
2. écrire les règles de grammaire à ajouter dans `parser.mly`
3. vérifier que le tout compile (avec la commande `make`)
4. ajouter les terminaux – déclarés précédemment – dans `lexer.mll`
5. vérifier que tout compile ensemble
6. tester les nouvelles règles avec des programmes qui les utilisent.

Par exemple, pour tester si la version courante du compilateur traite le programme en Autocell de la première session de TP, vous pouvez entrer :

```
> ../autocc autos/shift.auto
Assembly saved to autos/shift.s
```

Par contre, vous obtenez une erreur de syntax (pour une syntax non encore supportée) avec :

```
> ./autocc autos/vars.auto
ERROR:3:1: illegal char 'x'
```

A faire Tester les commandes ci-dessus.

2 Premiers pas

Dans l'état initial, `autocc` traite seulement des programmes de la forme :

```
2 dimensions of n..m end
[0, 0] := E
```

Avec E pouvant une cellule ou un entier et seul une affectation est supportée.

Cet exercice va permettre :

— d'ajouter les variables (utilisées ou affectées),

— de supporter plusieurs affectations.

Les variables dans Autocell : Une variable est définie par son identificateur qui peut être affecté ou utilisé dans une expression.

```
x := [1, 1]
y := x
[0, 0] := y
```

Dans Autocell, une variable n'a pas besoin d'être déclarée : elle est créée la première fois qu'elle est affectée. Cependant, une variable est identifiée par son nom, démarrant par lettre suivi par zéro ou plusieurs lettres, chiffres ou '_' (souligné).

1 On doit définir un terminal pour supporter la reconnaissances identifictaurs, nommons le `ID`. Est-ce qu'il nécessite une valeur sémantique ? Oui, car cette valeur est importante pour identifier la variable correspondante dans la mémoire. Quel est le type de cette valeur ? `string` évidemment ! Ainsi on ajoute la définition de ce terminal texttt-parser.mly dans la partie de déclaration des terminaux.

```
%token <string> ID
```

2 Où peut-on utiliser une variable dans le langage ? Deux endroits : dans une expression, ou en tant que destination d'une affectation. Ajoutez la ligne suivante dans les productions des `expressions` :

```
| ID
        { NONE }
```

L'action est mise à `NONE` car elle doit retourner l'arbre de dérivation pour les expressions en `NONE` représente l'expression vide. Pour l'instant, on va ignorer les actions mais il faut s'assurer que notre compilateur peut encore être compilé.

Lancer la compilation et corrigez de possibles erreurs :

```
> make
```

3 Le travail est fini au niveau syntaxique mais pas pour l'analyseur lexical. Ouvrez le fichier `lexer.mll` et ajoutez dans la partie dédiée à l'analyse des terminaux [1] :

1. Vous pourrez vous inspirer de ce qui a été fait pour `INT`.

— écrivez l'expression régulière pour un identificateur,
— écrivez l'action qui doit un `ID` (attention à la valeur sémantique).
Vérifiez que tout compile.

4 Testez votre compilateur avec les fichiers `autos/var1.auto`, `autos/var2.auto`, `autos/var3.auto` et `autos/var4.auto` [2].

5 Maintenant, on veut ajouter l'affectation d'une variable à notre langage. Observez dans les règles du non-terminal `statement` non-terminal comment est construire l'affectation pour une `cell`. Remarquez les différences entre `statement` et `expression` : une instruction (*statement* en anglais) peut changer la mémoire ou le flot d'execution alors qu'une expression est chargé de réaliser le calcul d'une valeur.

Ajoutez une production pour réaliser l'affectation de variable. Son action va renvoyer `NOP` (l'arbre de dérivation nul pour les instructions).

Testez votre compilateur avec `autos/varassign.auto`.

6 Le prochain problème à régler avec notre langage est qu'il ne supporte qu'une seule instruction : pour vérifier cela, testez `autocell` avec `autos/vars.auto`.

Modifiez l'analyseur syntaxique (`parser.mly`)pour supporter des programmes avec plusieurs instructions. Testez votre modification avec `autos/vars.auto`.

Résumé Pour étendre le langage Autocell utilisé dans le compilateur, il faut (a) créer de nouveaux terminaux dans `parser.mly` et dans `lexer.mll`, (b) écrire de nouvelles règles de grammaire dans `parser.mly` (avec une action nulle mais permettant la compilation) et (c) tester les modifications en utilisant des sources Autocell contenant ces nouvelles constructions. On va appliquer cette approche dans les questions suivantes.

Pour déboguer Si nécessaire, vous pouvez utiliser la fonction d'affichage d'OCAML pour vous aider à trouver les bogues : `print_string`, `print_int`, `printf`, etc. Mais rappelez vous que l'analyseur syntaxique utilise une approache $LALR(1)$ pour éviter toute incompréhension dans l'ordre d'affichage des messages.

3 Ajout des expressions arithmétiques

L'objectif de cet exercise est d'étendre les `expressions` d'Autocell avec les opérateurs arithmétiques :
— addition, soustraction
— parenthèses
— multiplication, division, modulo

La suite de ce TP propose un ordre d'implantation de ces nouvelles constructions syntaxiques et les fichiers de test `.auto` correspondants. Vous pouvez suivre ou non cet ordre mais nous pensons qu'il devrait être d'une grande aide pour les débutants avec

2. Ce dernier doit faire une erreur de compilation.

`ocamllex` et `ocamlyacc`. Quel que soit votre choix, mettez les actions des productions ajoutés à { `NONE` }. On les remplacera par des actions plus constructive dans les TPs suivants.

Le plan de travail est détaillé ci-dessous :

1. Ajoutez aux `expression`s l'opérateur d'addition dans sa forme la plus simple "$A_1 + A_2$" avec A_i une variable, une cellule ou un entier (les espaces ne sont pas significatifs). Testez avec `autos/add1.auto`.
2. On étend maintenant l'addition pour supporter l'associativité "$A_1+A_2+A_3+...A_n$" qui en fait peut être vu comme la combinaison de plusieurs opérateurs binaires : "$(...((A_1 + A_2) + A_3) + ... + A_n)$". Notez que dans Autocell (et dans la plupart des langages de programmation), l'addition est associative à gauche. Implémentez l'addition associative et testez la avec `autos/add2.auto`.

   Astuce : il n'est pas aisé de tester si vous avez réellement réalisé une associativité à gauche mais on peut le vérifier en utilisant des affichages dans les expressions. Ajoutez les actions d'affichage suivant dans les expressions :
   — `printf "%d\n" $1` pour `INT`,
   — `printf "%s\n" $1` pour `ID`,
   — `printf "[%d, %d]\n" (fst $1) (snd $1)` pour `ID`,
   — `printf "+\n"` pour l'addition.
   Comme l'analyse réalisée par `ocamlyacc` est ascendante, l'expression est exprimée en notation polonaise inversée. Par exemple, l'expression `1 + 2 + 3` produire la sortie `1 2 + 3 +` : d'abord on somme 1 et 2, le résultat est sommé avec 3 ce qui valide notre associativité à gauche.

   Si vous obtenez `1 2 3 + +`, on va d'abord sommer 2 et 3 et ensuite on va ajouter 1 au résultat ce qui réaliser en fait un associativité à droite.

   Une fois que vous avez validé votre implantation, vous pouvez commenter les appels à `printf`.
3. Etendez les `expression`s avec la soustraction "$A_1 - A_2$". La soustraction est aussi associative à gauche et a le même niveau de priorité que l'addition :
   — $A_1 - A_2 - A_3 \iff (A_1 - A_2) - A_3$
   — $A_1 + A_2 - A_3 \iff (A_1 + A_2) - A_3$
   — $A_1 - A_2 + A_3 \iff (A_1 - A_2) + A_3$
   Testez votre implantation avec `autos/addsub.auto`.
4. On peut ensuiter ajouter les `expression` parenthésées, "(any expression)". Elles sont utiles pour contrôler l'associativité : "$A_1 - (A_2 + A_3) \iff A_1 - A_2 - A_3$". Testez avec `autos/parent.auto`.
5. Maintenant nous pouvons ajouter la multiplication "$A_1 * A_2$". La multiplication est associative à gauche et a une priorité plus haute que l'addition ou la soustraction. Cela signifie que $A_1 + A_2 * A_3 \iff A_1 + (A_2 * A_3)$. Testez avec `autos/mult.auto`.
6. Ajoutez les opérateurs de division "A_1/A_2" et de modulo "$A_1\%A_2$" sont tous les deux associatifs à gauche et ont la même priorité que la multiplication. Testez avec `autos/divmod.auto`.

7. Finalement ajoutez le "+" et le "-" unaire. Remarquez que ces opérations ne sont pas associative (elle ne s'appliquent sur un seul opérande) et ont par conséquent la priorité la plus importante : elles s'appliquent directement sur l'opérande atomique (cellule, variable, entier, ...) qui les suient. Testez avec `autos/neg.auto`.