

TP Compilation (partie 2) :

Compilateur de VSL+

Il s'agit maintenant de greffer des actions sémantiques sur votre analyseur syntaxique de VSL+ de manière à effectuer le contrôle de type et la génération du code 3 adresses associé.

1. Fichiers fournis

Pour démarrer cette seconde partie, nous vous suggérons de créer un nouveau répertoire et d'y recopier le contenu du répertoire `/share/mlinfo/tpcomp/code3a`.

Vous disposez alors des fichiers (certains seront créés par la première exécution de `make`) :

- `scanner.l` : fichier source de l'analyseur lexical (complété par rapport à celui fourni pour la partie 1)
- `main.c` : programme principal qui lance l'analyseur syntaxique généré
- `makefile`
- `vcc_ref` : exécutable du compilateur VSL+ (fourni comme référence)
- un certain nombre de fichiers de suffixe `.o` et `.h` : les premiers correspondent aux exécutables de la mise en œuvre des types abstraits introduits pour réaliser ce compilateur ; les seconds aux fichiers en-tête correspondants.
- `vam` et `vas` pour permettre l'exécution du code (cf partie "découverte" du TP1)
- `lib` et `header` qui contiennent des données nécessaires à l'exécution (ne pas s'en soucier).

Sous ce même répertoire, recopiez votre fichier `parser.y` de la partie 1 qu'il va vous falloir compléter.

2. Commentaires sur les attributs utilisés et les types abstraits introduits

Les attributs introduits en cours pour la génération de code 3 adresses sont les deux attributs synthétisés `code` et `place`. Le premier, qui correspond au code 3 adresses généré, est associé à tous les non-terminaux. Le second est associé uniquement aux non-terminaux désignant une expression et correspond à la variable qui contiendra le résultat de cette expression lorsque le code 3 adresses associé à celle-ci sera exécuté.

L'attribut `code` sera mis en œuvre au travers du type abstrait **code** dont les primitives sont définies dans `code.h`. En ce qui concerne l'attribut `place`, il sera vu comme un champ du type structuré **expratt** puisque dans Yacc, un non-terminal possède un unique attribut ce qui oblige à regrouper dans une structure (ici **expratt**) les attributs associés au non-terminal `expression` par exemple.

Les attributs introduits pour le contrôle de type sont les attributs `type` et `tabsymb`. L'attribut `type` est synthétisé et sera mis en œuvre au travers du type abstrait **type** dont les primitives sont définies dans `type.h`. L'attribut `tabsymb` est hérité et doit par conséquent (puisque Yacc ne permet pas de mettre en œuvre des attributs hérités) être représenté par une variable globale de type **tab-symbole** manipulée grâce aux primitives définies dans `tabsymbole.h`.


```
code gen_code_print_exp( expratt expression)
{ code lecode = CodeAppend3(
    ExprattGetCode(expression),
    CodeNew(TAC_ARG, ExprattGetPlace(expression), NULL, NULL),
    CodeNew(TAC_CALL, NULL, library[LIB_PRINTN], NULL) );
    ExprattFree(expression) ;
    return lecode;
}
```

```
code gen_code_print_text (tokatt text)
{ code lecode = CodeAppend (
    CodeNew(TAC_ARG, TokattGetLabel(text), NULL, NULL),
    CodeNew(TAC_CALL, NULL, library[LIB_PRINTS], NULL) ) ;
    return lecode;
}
```

3.2. Les modifications

Pour débiter cette seconde partie, voici ce qu'il convient d'apporter comme modifications à votre fichier `parser.y`. Pour vous aider, vous pouvez vous reporter à l'exemple ci-dessus.

a) Au début du fichier `parser.y`, comme dans l'exemple ci-dessus, réaliser l'inclusion des fichiers en-tête nécessaires grâce à la directive `#include`. Le nom du fichier fourni en argument de cette directive est entouré de chevrons lorsque le fichier appartient à la bibliothèque C. Ces directives doivent figurer entre `%{` et `%}`.

b) Il est nécessaire de définir la liste des types associés à l'ensemble des symboles grâce à la commande `%union`.

Ainsi, dans le fichier `parser.y` ci-dessus, on trouve la commande suivante :

```
%union
{ int          scanner_integer ;
  char         *scanner_string ;
  code         yycode ;
  expratt      yyexpratt ;
}
```

c) En utilisant la commande `%type`, typer chacun de vos non-terminaux en utilisant l'une des variantes définies lors du `%union`. Ainsi, le non-terminal *expression* par exemple est de type *expratt* (type structuré regroupant les 3 attributs `code`, `place` et `type`) tandis que le non-terminal *print_statement* est de type *code* puisqu'il ne possède que l'attribut `code`. Attention, ce qu'il faut utiliser dans la commande `%type`, c'est le nom de la **variante** et pas le type !

On a ainsi :

```
%type <yyexpratt> expression
%type <yycode> instruction
```

Il faut également typer les tokens possédant des attributs lexicaux. Ainsi, pour l'exemple, on a :
`%token <scanner_integer> INTEGER`
car le token *INTEGER* possède un attribut lexical de la variante *scanner_integer* (et accessoirement de type *int*) qui correspond à la valeur de l'entier.

4. Comment procéder pour débiter le compilateur

Nous vous proposons de traiter les constructions du langage VSL+ dans un certain ordre et de garder tout ce qui concerne fonctions et prototypes pour la fin.

Vous devrez nous rendre le compilateur sans le traitement des fonctions et des prototypes la semaine suivant la séance de TP3 (**date limite : lundi 15 novembre**).

Les séances de TP 4, 5 et 6 seront consacrées aux fonctions et prototypes.

Pour cette première phase (compilateur sans les fonctions et prototypes), dans laquelle l'axiome devient *instruction* (*%start instruction*), nous vous suggérons de traiter dans l'ordre :

- expressions sans variable
- instructions de contrôle (if, while) et d'écriture
- blocs : déclarations, expressions avec variable, affectation, séquence, instruction de lecture.

Remarques :

- si l'un de vos non-terminaux a été typé (*yycode* par exemple) et que pour l'instant vous ne vous occupez pas de certaines productions où il apparaît en partie gauche, il vous suffit d'associer temporairement à ces productions des actions fictives (de la forme $$$ = \text{CodeEmpty}()$ par exemple).
L'action par défaut réalisée par Yacc est $$$ = \1 .
- vous pouvez utiliser la primitive *CodePrint(code c)* qui affiche le code 3 adresses *c*.
Une autre primitive qui pourra vous être utile est la primitive *SymbDumpTable()* qui réalise l'affichage de la table des symboles.

5. Un rappel des commandes utiles

- La commande **make** vous permet de générer votre compilateur (exécutable de nom *vcc*) , à partir de la suite de commandes figurant dans le fichier *makefile* (cf partie "découverte" du TP1).
- La commande *./vcc -3a <test* permet alors de lancer votre compilateur sur le fichier source *test* et de générer du code 3 adresses (l'option *-dst* permet l'affichage de la table des symboles, en cas d'erreur).
- Pour exécuter les programmes générés sur une machine virtuelle, il vous faut exécuter la suite de commandes suivantes (là-encore, vous pouvez vous reporter à la partie "découverte" du TP1) :

```
./vcc <test1 >test1.as  
./vas <test1.as >test1.o  
./vam test1.o
```