Polytech Nice Sophia
SI4 / 2018-2019
Erick Gallesio

Compilation

Feuille 5

Mini langage

Un frontend et trois backends

Pour réaliser de TD, prendre **l'archive** qui contient le squelette du compilateur à réaliser.

1 Présentation

Dans cette feuille, nous allons étendre la grammaire des expressions de la feuille précédente. En particulier, nous allons ajouter les mécanismes suivants:

- variables,
- instruction `if` ,
- instruction `while` ,
- blocs (entre ' `{` ' et ' `}` ') et
- une primitive d'impression (`print`).

Par rapport à la calculatrice que nous avions réalisée dans le TD précédent nous construirons, lors de l'analyse syntaxique, un arbre en mémoire. Ce dernier sera parcouru de trois façons différentes, pour produire trois types de résultats:

- évaluation de l'arbre (calculatrice "classique"),
- production de code pour une machine à pile (compilation),
- traduction de l'arbre vers un langage graphique (affichage).

Bien sûr, pour chacun de ces trois **back-ends**, seule la phase III du compilateur devra être modifiée. Cela implique donc que les phases I et II du compilateur ne doivent dépendre en aucune manière du traitement ultérieur qui sera fait sur l'arbre.

Pour travailler, vous téléchargerez sur le site du cours une archive qui contient un squelette du compilateur avec ses trois phases de production de code.

Le Makefile fourni dans cette archive permet de construire trois exécutables:

- **calc1**: cet exécutable interprète directement des expressions séparées par des `';'` . Les fichiers fournis dans le squelette permettent d'obtenir une version pratiquement complète de la calculatrice (on peut évaluer des expressions simples, mais les

instructions `if` et `while` ne sont pas du tout implémentées);
- **calc2**: cet exécutable produit du code pour une machine à pile fictive (à construire);
- **calc3**: cet exécutable produit un graphe au format `dot` (à construire).

Les fichiers fournis dans l'archive sont:

- `calc.c` : pour construire/détruire les nœuds de l'arbre.
- `calc.h` : définition des structures de données et macros pour la gestions des nœuds de l'arbre.
- `code.h` : ne contient que le prototype de la fonction `produce_code` (c'est cette fonction qui aura trois implémentations différentes).
- `code1.c` , `code2.c` , `code3.c` : les trois version du générateur de code.
- `lexical.l` : le source flex de la grammaire lexicale de la calculatrice.
- `syntax.y` : le source yacc de la grammaire de la calculatrice.

L'archive comprend aussi un répertoire `Resources` qui contient trois programmes de test et un interprète pour la machine à pile utilisée par `calc2` . Pour chaque programme de test (suffixé par `'.in'`), on a le code produit par calc2 (suffixé par `'.as'`) et le graphe du programme produit par calc3 (suffixé par `'.dot'`).

2 Extension de la calculatrice

L'analyseur fourni dans le squellete reconnaît une suite d'énoncés séparés par des `';'` . La grammaire des énoncés reconnus est donnée ci-dessous:

```
stmt      -> ';' | expr ';' | KPRINT expr ';' | var '=' expr ';' | '{' stmt_list '}'
stmt_list ->  stmt_list stmt | stmt
```

Étendez cette grammaire pour reconnaître les énoncés `if` (avec ou sans `else`) et `while` . Vous prendrez ici une syntaxe *à la* C.

1. Ajouter ce qu'il faut à la version fournie pour reconnaître la nouvelle grammaire.
2. Testez votre programme avec les fichiers `'.in'` du répertoire `Resources` .

3 Production de code pour une machine à pile

Le programme `calc2` doit produire du code pour une machine à pile. Quelques exemples de codes produits par ce programme sont donnés ci-dessous.

1. Si on entre l'énoncé
   ```
   print(3 * (5 + foo));
   ```
   le programme doit produire les instructions suivantes:
   ```
   push    3
   push    5
   load    foo
   add
   mul
   print
   ```
2. L'énoncé

```
if (x < y) z = 2; w = 100;
```

produira la séquence:

```
    load    x
    load    y
    cmplt
    jumpz   L000
    push    2
    store   z
L000:
    push    100
    store   w
```

3. Enfin, l'expression

```
while (i < 10) { print(i); i = i + 1; }
```

produira:

```
L001:
    load    i
    push    10
    cmplt
    jumpz   L002
    load    i
    print
    load    i
    push    1
    add
    store   i
    jump    L001
L002:
```

Pour tester le code produit par votre programme, vous pouvez utiliser le programme `interp` qui est dans le répertoire `Resources` :

```
$ ./calc2 Resources/fact.in > fact.as
$ ./interp fact.as
479001600.0
*stop*
$
```

Les instructions reconnues par l'interprète sont:

- manipulation de la pile et des variables: `push` , `load` , `store`
- affichage du sommet de la pile: `print`
- opérations unaires: `negate` (opposé)
- opération binaires: `add` , `sub` , `mul` , `div`
- comparaisons: `cmplt` , `cmple` , `cmpgt` , `cmpge` , `cmpeq` , `cmpne`
- sauts: `jumpz` , `jump` ,

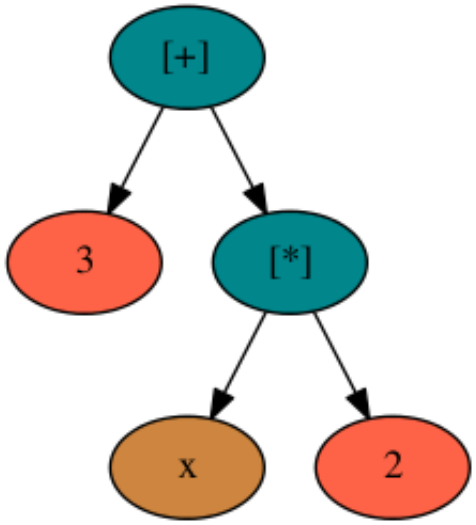
Les étiquettes sont de la forme `Lxxx` où `xxx` est un nombre. La définition d'une étiquette doit être placée en début de ligne alors qu'une instruction doit être précédée d'au moins un espace ou une tabulation.

4 Affichage de l'arbre d'analyse

Le programme `calc3` doit fournir l'arbre d'analyse dans le format **dot**. Ce format est assez simple à produire. Par exemple, l'expression `3+x*2` fournira la sortie suivante

```
digraph E{
    node [style="filled"];
    box1 [label="[+]", fillcolor="turquoise4"];
    box2 [label="3", fillcolor="tomato"];
        box1 -> box2;
    box3 [label="[*]", fillcolor="turquoise4"];
    box4 [label="x", fillcolor="peru"];
        box3 -> box4;
    box5 [label="2", fillcolor="tomato"];
        box3 -> box5;
        box1 -> box3;
}
```

Si on charge ce fichier dans l'outil `xdot`, le graphe suivant sera affiché:



5 Ajout de la primitive read

On veut ajouter la primitive read à notre langage. Un exemple de programme utilisant cette primitive est donné ci-dessous:

```
{
   read a;
   read b;
   print a + b
}
```

1. Quels sont les fichiers à modifier?
2. Dans la version interprétée, comme nous n'avons pas de chaîne, le nom de la variable sera affichée avant sa lecture. Pour le programme précédent, on obtient donc avec le fichier `read.in` qui vous est distribué:

- pour **calc1**:

```
$ ./calc1 read.in
a? 3
b? 2
5
Bye
```

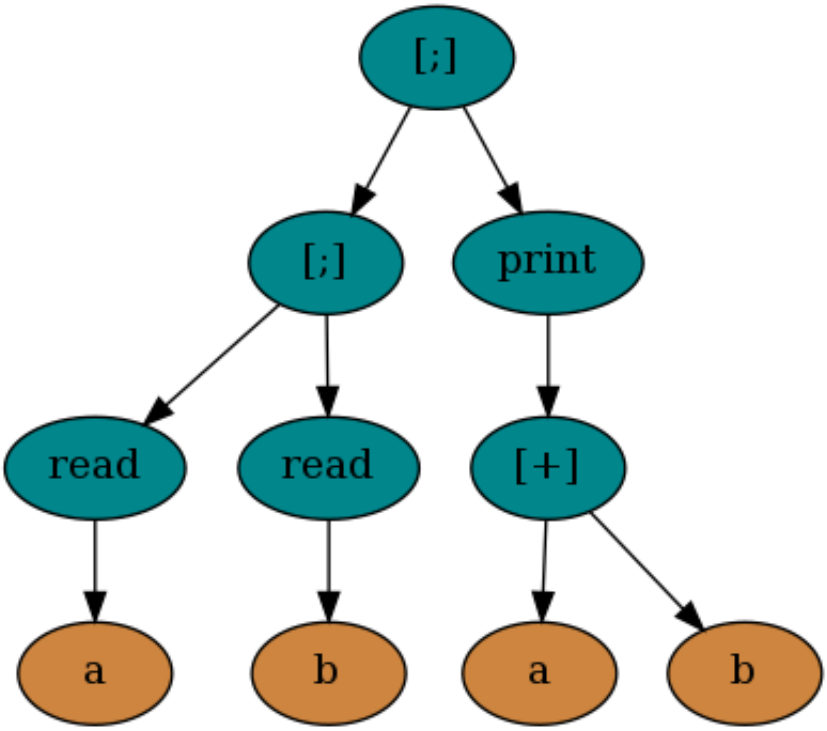
- pour **calc2**:

```
$ ./calc2 read.in > read.as
$ ./interp read.as
a? 10
b? 12
22
```

- et enfin pour **calc3**:

```
digraph E{
  node [style="filled"];
  box1 [label="[;]", fillcolor="turquoise4"];
  box2 [label="[;]", fillcolor="turquoise4"];
  box3 [label="read", fillcolor="turquoise4"];
  box4 [label="a", fillcolor="peru"];
       box3 -> box4;
       box2 -> box3;
  box5 [label="read", fillcolor="turquoise4"];
  box6 [label="b", fillcolor="peru"];
       box5 -> box6;
       box2 -> box5;
       box1 -> box2;
  box7 [label="print", fillcolor="turquoise4"];
  box8 [label="[+]", fillcolor="turquoise4"];
  box9 [label="a", fillcolor="peru"];
       box8 -> box9;
  box10 [label="b", fillcolor="peru"];
       box8 -> box10;
       box7 -> box8;
       box1 -> box7;
}
```

Ce fichier visualisé avec `xdot` afichera:


[;]
[;]
print
read
read
[+]
a
b
a
b