Compilation

Toy: un mini langage objet

SI4 — 2018-2019

Erick Gallesio

Déroulement de la fin du cours

Pour illustrer le cours *Techniques de compilation*, la fin du module repose sur l'étude d'un compilateur pour un petit langage objet, **Toy**, qui offre des traits provenant de C++ et de Java.

- Les TDs consistent à introduire des extensions à ce langage:
  - *nouvelles constructions syntaxiques*
  - *ajout de fonctions "runtime"*
  - *ajout de nouveaux types de données*
  - *modifications/améliorations sur le système objet*
- Le compilateur est écrit en C (+ flex / yacc) et produit du code C
- On étudiera:
  - *l'analyse lexicale et syntaxique*
  - *la construction d'une représentation de programme sous forme d'arbre abstrait*
  - *l'analyse sémantique*
  - *la production de code*

Toy: le langage de base

Tout d'abord, pour se familiariser avec le compilateur, on travaillera avec une version qui n'implémente que la partie non objet de *Toy* (*Toy-base*).

- Fourniture d'un compilateur fonctionnel pour le sous-ensemble non objet.
- Le langage "de base" ressemble à C:
  - *syntaxe identique à C*
  - *Types d'objets:* `int`, `char`, `bool`, `sting` et `void`
  - *les variables sont typées et doivent être déclarées*
  - *fonctions non imbriquées (comme en C)*
  - *une primitive* `print` *(pas une fonction à la différence de C)*
    - On peut faire print("x =", x, " nul?: ", (x == 0));
  - *le programme principal est dans la fonction* `main()`

Toy: le langage de base — exemple

```
int fact(int n) {
  return (n < 2) ?  1: n * fact(n-1);
}

int fib(int n) {
  if (n < 2)
    return n;
  else
    return fib(n-1) + fib(n-2);
}

int concise_fib(int n) {
  return (n < 2)? n:  concise_fib(n-1) + concise_fib(n-2);
}

int main() {
  print("10! = ", fact(10), "\n");
  print("fib(10) = ", fib(10), "\n");
  print("cfib(10) = ", concise_fib(10), "\n");

  print((concise_fib(12) == 144) ? "SUCCESS\n": "FAILURE\n");
  return 0;
}
```

Toy: le langage de base — extensions

En Td, vous implémenterez les extensions suivantes au langage de base:

- implémentation “propre” des comparaisons de chaînes de caractères.
- opérateurs `++` et `--` de C sur les variables entières
- ajout de l’énoncé `break`
- ajout de l’opérateur puissance (`'**'`) sur les entiers ($\Longrightarrow$ modification du runtime)
- ajout de l’énoncé `for` de C
- ajout du type `float`
- …

Toy: Un langage de classes

Cette version du compilateur propose des traits objets proches de ceux de Java.

```
class Point {
  int x, y;   // membres tjs publics, pas d'attibut statique ou cst

  void printobj() {
    // Fonction appelée lorsqu'on utilise print sur un point
    print("#<Point x=", this.x, " y=", this.y, ">");
  }

  Point init(int x, int y) {
    // Une sorte de constructeur
    this.x = x; this.y = y;
    return this;
 }
}

int main() {
  Point p1, p2;

  p1 = new Point;              // x et y initialisés à 0
  p2 = new Point.init(1, 2);  // Appel explicite à la fonction init

  print("p1 = ", p1, "\np2 = ", p2, "\n");
}
```

Toy: Héritage

```
class Circle extends Point {    // Pas d'interface, Héritage simple
  int r;

  void printobj() {
    print("#<Circle x=", this.x, " y=", this.y, " r =",this.r,">");
  }

  Circle init_from_Point(Point p, int r) {
    this.x = p.x; this.y = p.y; this.r = r;
    return this;
  }
}

int main() {
  Circle C;

  c = new Circle.init_from_Point(new Point, 7);
  print("c = ", c, "\n");
  return 0;
}
```

- `printobj` surcharge la version définie dans `Point`
- Affichage de `#<Circle x = 0 y = 0 r = 7>`

Toy: un vrai langage objet

```
class Animal {
  string talk() { return "??";  }
}

class Cat extends Animal {
  string talk() { return "Meow!"; }
}

class Dog extends Animal {
  string talk() { return "Woof!"; }
}

void hear(Animal a) {                  // Fonction hear
  print("hear: ", a.talk(), "\n");     // liaison dynamique
}

int main() {
  hear(new Cat);                       // polymorphisme a → Cat
  hear(new Dog);                       // polymorphisme a → Dog

  Animal a = new Dog;                  //
  print(a.typename());                 // ⟹    Dog ici
  return 0;
}
```

Le compilateur Toy

Compilateur: Analyse sémantique

Lors de l'analyse sémantique,

- Analyse des noms:
  - *quand un identificateur est utilisé, il faut savoir ce qu'il dénote, donc avoir mémorisé sa définition dans une table des symboles.*
  - *dans certains langages, un identificateur doit toujours être défini avant son utilisation (ou au moins être déclaré)* $\Longrightarrow$ *une compilation en une passe est possible*
  - *dans d'autres langages, les références en avant sont autorisées* $\Longrightarrow$ *une compilation en deux passes est nécessaire.*
- Analyse des types:
  - *vérifier la compatibilité*
    - du type des opérandes avec un opérateur dans une expression
    - des arguments d'un appel de fonction avec ses paramètres formels,
  - *résolution de la surcharge*
  - …

Compilateur: Production de code

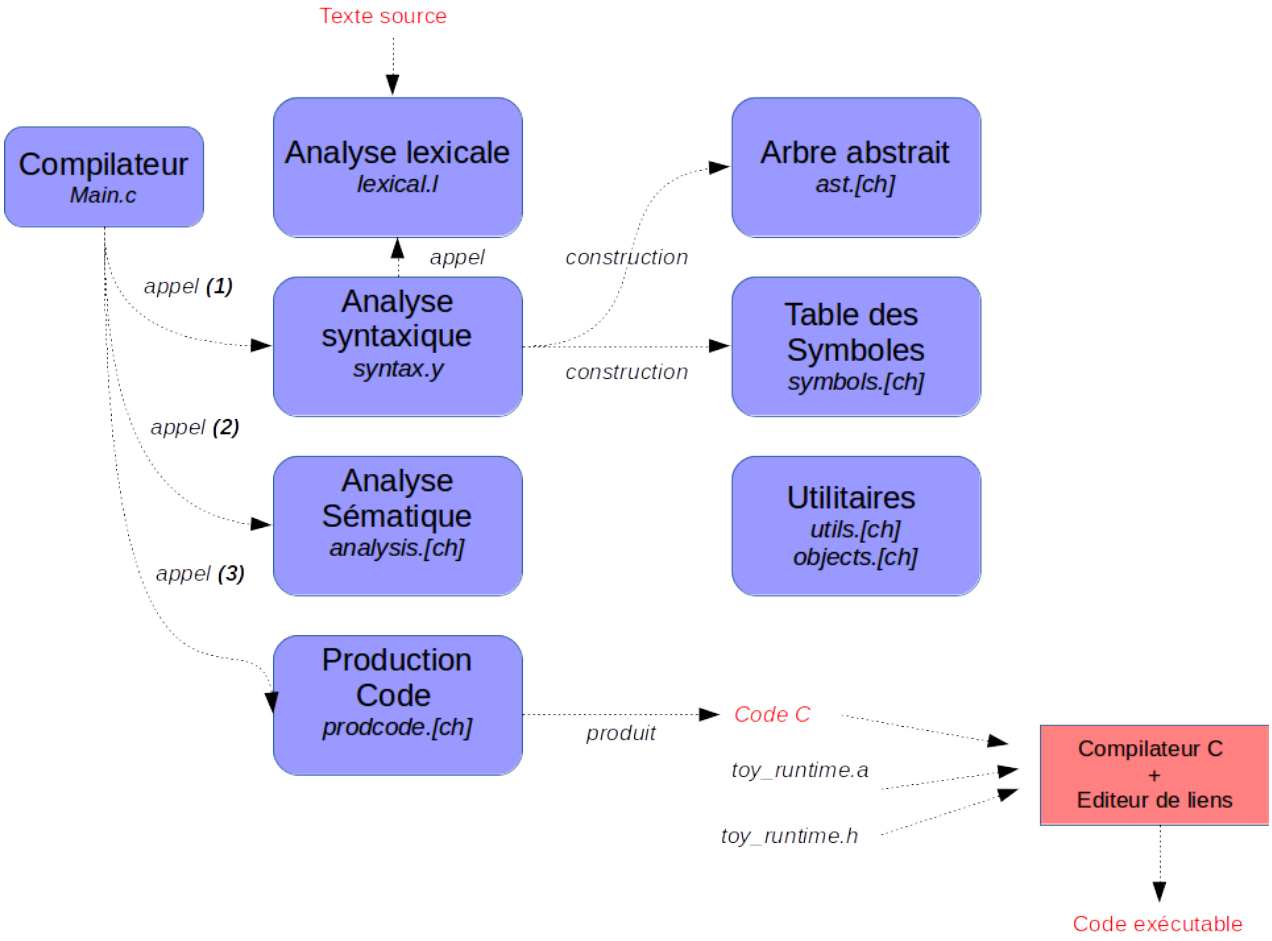
Ce que l'on va voir:

- les problèmes de traduction d'un langage de haut niveau vers un langage de niveau inférieur;
- les mécanismes d'implémentation d'un langages objet.

Ce que l'on ne verra pas (masqué par le fait qu'on produit du C):

- l'organisation de la mémoire;
- gestion de blocs;
- la sélection d'instructions;
- l'allocation des registres;
- les principales sources d'optimisation :
  - *optimisation des boucles,*
  - *calcul d'expression constantes,*
  - *optimisation des appels terminaux (tail calls),*
  - *analyse du flot de contrôle (code mort, déplacement des invariants de boucle, dérécursivation, …)*
  - …

Architecture du compilateur *Toy*



Toy: Analyse lexicale

L'analyse lexicale se trouve dans le fichier `lexical.l`

Extrait:

```
%option yylineno        /* lex gère les numéros de lignes */
%%

">="        return GE;

"and"       return KAND;
"while"     return KWHILE;
"if"        return KIF;
"print"     return KPRINT;

"int"       return TINT;
"bool"      return TBOOL;
"string"    return TSTRING;
"void"      return TVOID;

[a-zA-Z][_a-zA-Z0-9]*       /* pas de '_' au début */
                            { yylval.ident = strdup(yytext);
                              return IDENTIFIER; }

[ \t\n]+          ;       /* ignore whitespace */

%%
int yywrap(void) { return 1;  }   /* évite l'option -lfl */
```

Toy: Analyse syntaxique (expressions)

L'analyse syntaxique se trouve dans `syntax.y`. On a:

```
expr:
    INTEGER         {$$ = make_integer_constant($1); }
  | BOOLEAN         {$$ = make_boolean_constant($1); }
  | STRING          {$$ = make_string_constant($1);}
  | designator      {$$ = $1; }
  | var '=' expr    {$$ = make_expression("=",  assign, 2, $1,$3);}
  | expr '+' expr   {$$ = make_expression("+",  barith, 2, $1,$3);}
  | expr '>' expr   {$$ = make_expression(">",  comp,   2, $1,$3);}
  | expr KAND expr  {$$ = make_expression("&&", blogic, 2, $1,$3);}
  | KNOT expr       {$$ = make_expression("!",  ulogic, 1, $2); }
  | '('expr ')'     {$$ = make_expression("(", parenthesis,1,$2);}
  ...
  ;
```

Note: Pour les parenthèses, on aurait pu faire comme avec la calculatrice (càd `{ $$ = $2; }`).

- il faut davoir quand mettre des parenthèses (⇒ gérer les priorités dans `prodcode`), ce qui est inutile car on a les mêmes priorités que C.
- on met toujours des parenthèses (⇒ rend le code produit difficile à lire, mais utile si on produit du code pour un langage avec des priorités différentes de celles de C).

Toy: Analyse syntaxique (énoncés)

Extrait des règles relatives aux énoncés:

```
%%
stmt:
    ';'                             {$$=make_expr_statement(NULL);}
  | expr ';'                        {$$=make_expr_statement($1);}
  | KPRINT '(' eparam_list ')' ';'{$$=make_print_statement($3);}
  | '{' stmt_list '}'               {$$=make_block_statement($2);}
  | KWHILE '(' expr ')' stmt        {$$=make_while_statement($3,$5);}
  | KRETURN expr_opt ';'            {$$=make_return_statement($2);}
  | var_decl ';'                    {$$=$1;}
  | prototype ';'                   {$$=$1;}
  | error ';'                       {yyerrok; $$ = NULL;}
  ...
  ;

stmt_list:
    stmt_list stmt    { list_append($1, $2, free_node); $$ = $1; }
  | /* empty */       { $$ = list_create();  }
  ;
```

- `list_create`, `list_append` sont des utilitaires de gestion de listes d'objets de types quelconques;
- Quand le nœud sera libéré, `free_node` sera appelée sur celui-ci.

Toy: Analyse syntaxique (déclarations)

Les types sont représentés par des objets de type `s_type`.

Les types standard sont créés à l'initialisation du compilateur et sont accessibles au travers de variables globales.

Si un type n'est pas standard (`int`, `bool`, `string` ou `bool`), il doit correspondre à un nom de classe.

```
type:   TINT            { $$ = int_type; }
    |   TBOOL           { $$ = bool_type; }
    |   TSTRING         { $$ = string_type; }
    |   TVOID           { $$ = void_type; }
    |   IDENTIFIER      { $$ = make_type($1, "NULL"); }
    ;
```

Important:

- Au niveau syntaxique, on ne vérifie pas qu'un identificateur correspond à un nom de classe.
- C'est la phase d'analyse sémantique qui s'en chargera.

Analyse et production de code (1/3)

Le compilateur *Toy* produit du code dès qu'il a reconnu une déclaration

- de variable,
- de fonction,
- de classe.

```
%%
program:    declarations {return error_detected;} // var qui contient le nbre
  d'erreurs
            ;

declarations: declarations declaration  {analysis_and_code($2);}
            |     /* empty */
            ;

declaration:    var_decl ';'    { }
            |   func_decl       { }
            |   class_decl      { }
            ;
```

```
static void analysis_and_code(ast_node *node) {
  analysis(node);
  if (!error_detected) produce_code(node);
  free_node(node);
  symbol_table_free_unused_tables();
}
```

Analyse et production de code (2/3)

La fonction **analysis** travaille récursivement sur le nœud qui lui est passé en paramètre.

Elle est en charge

- de toutes les analyses sur les noms:
  - *variables correctement déclarées,*
  - *paramètres conformes au prototype,*
  - *gestion des portées sur les identificateurs.*
- de définir le type de toutes les expressions du programme.

Le code de cette fonction est simplement:

```
void analysis(ast_node *node) {
  if (!node) return;
  AST_ANALYSIS(node)(node);
}
```

On utililise ici les pointeurs de fonctions comme dans le cours précédent.

Analyse et production de code (3/3)

La fonction **produce_code** produit le code sur le nœud qui lui est passé en paramètre.

- Elle n'est pas appelée si on a détecté des erreurs précédemment.
- Elle est similaire à la procédure d'analyse (exécution de la fonction dont le pointeur est stocké dans le nœud).

```
void produce_code_return_statement(ast_node *node) {
  struct s_return_statement *n = (struct s_return_statement *)node;
  emit("return");
  if (n->expr) {
    emit(" "); code_expr_cast(n->expr); // ajoute évent. un cast
  }
  emit(";\n");
}

void produce_code_break_statement(ast_node *node) {
  emit("break;\n");
}
```

La phase d'analyse sémantique ayant au préalable vérifié que:

- **return** renvoie une expression dont le type est correct;
- **break** est bien utilisé dans une boucle.

Support d'exécution (runtime) (1/2)

Pour pouvoir exécuter le code produit par le compilateur, il faut

- la bibliothèque standard C;
- une bibliothèque spécifique au langage.

La bibliotèque peut être:

- chargée dynamiquement au moment de l'exécution (bibliothèque dynamique de type `'.so'` sur la plupart des Unix (dont Linux), DLL sous Windows).
- liée au moment de la compilation (bibliothèque statique généralement suffixée par `'.a'`

La déclaration des fonctions du *runtime* est dans le fichier `toy-runtime.h`.

Ce fichier est toujours inclus au début du programme C produit par le compilateur.

Support d'exécution (runtime) (2/2)

Que contiennent toy-runtime.[ah]?:

- des definitions de type (*e.g.* toy_string)
- des macros simplifiant le code produit
  - *allocation d'une instance*
  - *accès à un membre d'une instance*
- des fonctions utilisées par le code produit (*e.g.* `print_bool` qui permet d'afficher `false` ou `true` plutôt que 0 ou 1).

Certaines des extensions que l'on verra en TD nécessitent d'enrichir le runtime de *Toy*.
