Compilation

Construction d'un arbre syntaxique

SI4 — 2018-2019

Erick Gallesio

Contexte

On veut implémenter un mini langage (de type calculette)

On construit un arbre en mémoire lors de la phase d'analyse.

Ensuite, on peut avoir plusieurs "backends":

- évaluation de l'arbre (calculette)
- production de code pour une machine à pile (compilation)
- traduction de l'arbre vers un langage graphique (affichage)

Constat:

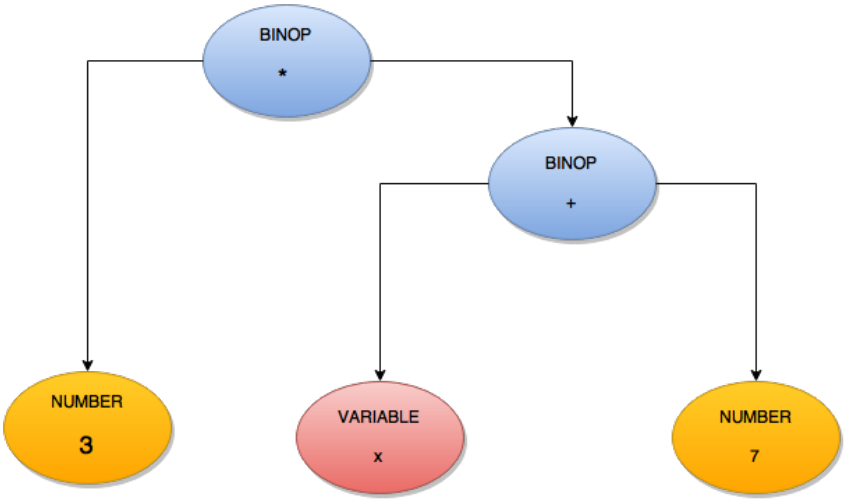
- les objets de l'arbre sont de types différents:
  - *entiers,*
  - *opérateurs,*
  - *variables,*
  - …

Problème:

- comment représenter l'arbre en mémoire?

Arbre syntaxique

Prenons l'expression **3 * (x + 7)**. L'arbre que l'on a en mémoire:



Trois types de nœuds avec des informations différentes:

- un entier pour un `NUMBER`
- une chaîne pour une `VARIABLE`
- Deux sous-arbres et l'operateur (une chaîne) pour un `BINOP`

Implémentation de l'arbre dans un LOO

On définit une classe `Ast` et une classe qui en hérite par type de nœud.

En C++ $\Longrightarrow$

```
class Ast {
public:
  virtual ~Ast() {}
  virtual int value() = 0;
};

class Binop : public Ast {
  char m_oper; Ast *m_left, *m_right;
public:
  Binop(char oper, Ast *left, Ast *right):
    m_oper(oper), m_left(left), m_right(right) {}

  virtual ~Binop() { delete m_left;  delete m_right; }
  virtual int value();
};

class Number : public Ast {
  int m_val;
public:
  Number(int val): m_val(val) {}
  virtual int value();
};
```

Fichier Yacc associé

```
%%
line : line  expr  '\n'  { if ($2) cout << $2->value()<< endl; delete $2; }
     | /* empty */       { ; }
     | error '\n'        { yyerrok; }
     ;

expr : expr '+' expr     { $$ = new Binop('+', $1, $3);  }
     | expr '-' expr     { $$ = new Binop('-', $1, $3); }
     | expr '*' expr     { $$ = new Binop('*', $1, $3); }
     | expr '/' expr     { $$ = new Binop('/', $1, $3); }
     | '(' expr ')'      { $$ = $2; }
     | IDENT             { $$ = new Ident(&vars[$1-'a']); }
     | NUMBER            { $$ = new Number($1); }
     | IDENT '=' expr    { vars[$1-'a'] = $3->value(); $$ = $3; }
     ;
```

- Lorsqu'on synthétise une règle
  - *allocation d'un nœud dont le type dépend de la règle*
  - *ce nœud constitue l'attribut synthétisé, décoration du nœud de l'arbre d'analyse.*
- Lorsqu'on reconnaît une expression complète:
  - *appel de la fonction value qui parcourt l'arbre construit pour produire le résultat (évaluation, production de code, …)*
  - *l'arbre de l'expression peut être détruit (**delete**)*

Evaluation de l'arbre

Pour évaluer l'arbre construit en mémoire, il suffit d'implémenter la méthode `value` :

```
// ---------- Binop value -----------------------------
int Binop::value() {
  switch (m_oper) {
    case '+': return m_left->value() + m_right->value();
    case '-': return m_left->value() - m_right->value();
    case '*': return m_left->value() * m_right->value();
    case '/': if (m_right->value())
                return m_left->value() / m_right->value();
              else {
                cerr << "Division by 0" << endl;
                break;
              }
    default: cerr << "Unknown operator" << endl;
  }
  return 0;
}

// ---------- Number value -----------------------------
int Number::value() {
  return m_val;
}

// ---------- Ident value -----------------------------
int Ident::value() {
  return *m_val;
}
```

Implémentation naïve de l'arbre en C

Pour implémenter l'arbre syntaxique en C, on peut passer par une **union**.

Une définition possible pour un nœud de l'arbre pourrait être:

```
typedef struct ast_node ast_node;

struct ast_node {
  int lineno;   // ligne source de l'unité lexicale (par exemple)
  enum { kident, knumber, kbinop } kind;    // type de nœud
  union {
    struct ident  { char value; } s_ident;
    struct number { int  value; } s_number;
    struct binop  {
      ast_node *left, *right;
      char operator;
    } s_binop;
  } info;
};
```

- Difficile à maintenir si on ajoute des types de nœuds
- peu efficace en mémoire
- forte localité des traitements (les fonctions travaillant sur l'arbre sont des gros `switch` sur tous les types de nœuds).
- $\Longrightarrow$ ne **passe pas** à l'échelle.

Implémentation de l'arbre en C (1/4)

Principe:

- Définir une structure de base `ast_node` qui contient ce qui est commun à tous les nœuds de l'arbre (ici, `lineno` et `type`)
- Utiliser des structures
  - *qui commencent par déclarer une structure de type `ast_node`*
  - *qui ne contiennent que ce qui est nécessaire pour le type de nœud*
- Utiliser **"à fond"**
  - *le mécanisme de cast*
  - *le pré-processeur C pour construire les accesseurs*

Le type `ast_node` peut être représenté par;

```
typedef struct ast_node ast_node;

struct ast_node {
  int lineno;   // ligne source de l'unité lexicale (par exemple)
  enum { kident, knumber, kbinop } kind;    // type de nœud
}

#define AST_LINENO(p)   (((ast_node *)(p))->lineno)
#define AST_KIND(p)     (((ast_node *)(p))->kind)
```

Implémentation de l'arbre en C (2/4)

Pour les nombres:

```
struct ast_number {
  ast_node_header;      // AST header
  int value;            // valeur de la constante
};

#define NUMBER_VALUE(p)    (((struct ast_number *) p)->value)
```

Pour les opération binaires:

```
struct ast_binop {
  ast_node_header;          // AST header
  ast_node *left, *right;   // les opérandes
  char operator;            // l'opérateur
};

#define BINOP_LEFT(p)     (((struct ast_binop *) p)->left)
#define BINOP_RIGHT(p)    (((struct ast_binop *) p)->right)
#define BINOP_OPERATOR(p) (((struct ast_binop *) p)->operator)
```

Pour les identificateurs: semblable aux nombres.

Implémentation de l'arbre en C (3/4)

Représentation mémoire.

Pourquoi ça marche?

Quelque soit le type de nœud que l'on manipule, les informations communes sont toujours au même endroit.

Pour accéder au champ `lineno` d'un `ast_binop` pointé par `p`, on peut faire:

- soit `p->header.lineno`
- soit `((ast_node *)p)->lineno` (c'est ce que fait la macro `AST_LINENO`).

Implémentation de l'arbre en C (4/4)

Allocation d'un opérateur binaire:

```
ast_node *make_binop(char oper, ast_node *left, ast_node *right) {
  ast_node *n = malloc(sizeof(stuct ast_binop)); // voir si NULL!!

  // Initialisation du header
  AST_LINENO(n)   = yylineno;
  AST_KIND(n)     = kbinop;

  // Initialisation des champs spécifiques
  BINOP_OPER(n)   = oper;
  BINOP_LEFT(n)   = left;
  BINOP_RIGHT(n)  = right;

  // renvoyer le nouveau nœud
  return new;
}
```

Dans le source *yacc*:

```
expr:      expr '+' expr { $$ = make_binop('+', $1, $3); }
    |      expr '-' expr { $$ = make_binop('-', $1, $3); }
    ...
```

Evaluation de cette solution

Par rapport à la solution avec des unions:

- l'ajout d'un nouveau type de nœud est assez simple
- l'utilisation mémoire est adaptée au type du nœud.
- la forte localité des traitements est toujours présente. En effet:

```
int value(ast_node *n) {
  if (!n)  return 0;      // raisonnable??
  switch (AST_KIND(n)) {
    case knumber: return NUMBER_VALUE(n);
    case kident:  return IDENT_NAME(n);
    case kbinop:  switch (BINOP_OPERATOR(n)) {
                    case '+': return BINOP_LEFT(n)+BINOP_RIGHT(n);
                    case '-': return BINOP_LEFT(n)-BINOP_RIGHT(n);
                    ......
                  }
  }
}
```

Ajout d'un nouveau type de nœud $\Longrightarrow$

- modification de a fonction `value`
- en fait, modification de **toutes** les fonctions de parcours de l'arbre.

Délocalisation des traitements (1/2)

Les pointeurs sur fonction vont permettre de délocaliser les traitements.

On redéfinit `ast_node` :

```
struct ast_node {
  int lineno;   // ligne source de l'unité lexicale (par exemple)
  enum { kident, knumber, kbinop } kind;    // type de nœud
  int (*value)(ast_node * n);               // valeur du nœud
}

#define AST_LINENO(p)   (((ast_node *)(p))->lineno)
#define AST_KIND(p)     (((ast_node *)(p))->kind)
#define AST_VALUE(p)    (((ast_node *)(p))->value)
```

Chaque type de nœud de l'arbre peut donc avoir sa procédure spécifique:

```
int binop_value(ast_node *node) {
  switch (BINOP_OPERATOR(n)) {
    case '+': return BINOP_LEFT(n)+BINOP_RIGHT(n);
    case '-': return BINOP_LEFT(n)-BINOP_RIGHT(n);
    ......
  }
}
```

Délocalisation des traitements (2/2)

La procédure d'allocation d'un nœud de l'arbre est à peine modifiée:

```
ast_node *make_binop(char oper, ast_node *left, ast_node *right) {
  ast_node *n = malloc(sizeof(stuct ast_binop)); // voir si NULL!!

  // Initialisation du header
  AST_LINENO(n)   = yylineno;
  AST_KIND(n)     = kbinop;
  AST_VALUE(n)    = binop_value;                          // ⟸   AJOUT ICI
  // Initialisation des champs spécifiques
  BINOP_OPER(n)   = oper;
  BINOP_LEFT(n)   = left;
  BINOP_RIGHT(n)  = right;

  // renvoyer le nouveau nœud
  return new;
}
```

Si on a beaucoup de fonctions de parcours d'arbre:

- cela complique l'écriture des allocateurs
- mais on peut faire produire le code par le pré-processeur C 😄

Parcours d'un arbre

Pour calculer la valeur d'un arbre:

```
int value(ast_node *node) {
  return node? AST_VALUE(node)(node) : 0;
}
```

Si on "embarque" un pointeur sur la fonction de libération d'un nœud:

```
void freenode(ast_node *node) {
  if (!node) return;
  AST_FREENODE(node)(node);
  free(node);
}
```

avec (par exemple) la fonction de libérarion mémoire suivante pour une opération binaire:

```
void freenode_binop(ast_node *n) {
  freenode(BINOP_LEFT(node));
  freenode(BINOP_RIGHT(node));
}
```

Parcours d'arbre dans un compilateur

Types de parcours d'arbre dans un compilateur

- typage des expressions (*e.g.* si un *float* intervient dans une expression, toute l'expression est de type *float*)
- analyse sémantique:
  - *vérifier qu'une variables est déclarée,*
  - *vérifier qu'il n'y a pas de double déclaration*
  - *vérifier q'un liste de paramètres effectifs est conforme au prototype*
  - …
- calcul des variables temporaires/registres nécessaires lors de la production de code
- calcul des expressions constantes à la compilation
- code refactoring (sortir les expressions invariantes des boucles)
- production de code objet

Analogie avec C++

La technique d'implémentation présentée ici est proche des premières implémentations C++ (les premiers compilateurs C++ produisaient du code C).

La definition du header de `ast_node` en tête de chaque type de nœud correspond à de **l'héritage.**

Les pointeurs sur fonction dans les structures correspondent, plus ou moins, à des **fonctions virtuelles**.

Le langage que l'on va implémenter par la suite dans ce cours est un langage objet à la Java. Il utilisera une technique d'implémentation des objets proche de la méthode vue ici.

Un exemple simple: une calculette

Pour illustrer ce cours nous étendrons en TD la grammaire des expressions du premier cours:

- ajout de certains mécanismes
  - *variables*
  - *while*
  - *blocs*
  - *primitive print*
  - *fonctions prédéfinies sin, cos, …*
- Une phase d'analyse
  - *vérification de type (e.g. les conditions sont booléennes)*
- Une phase de parcours de l'arbre avec 3 implémentations différentes
  - *évaluation de l'arbre (calculette)*
  - *production de code pour une machine à pile (compilation)*
  - *traduction de l'arbre vers un langage graphique (affichage)*

Bien sûr pour ces trois **back-ends**, seule la partie production de code devra être modifiée.
