FacileFlexBison

March 31, 2019

Analyseur syntaxique et génération de code du langage facile. Le langage facile est la suite du langage abordé en cours et consiste à implémenter un langage simple dénué de fonctions

1 Avant propos

La structure des dossiers a été modifiée pour faciliter le développement et l'évaluation. A la racine du projet on peut trouver:

- `build.sh`: le script permettant de compiler le programme `facile` et puis distribuer le binaire dans `dist/`. Mode d'emploi `sh build.sh`
- `test.sh`: le script permettant d'effectuer un test à la fois les composants dans `/test`. Mode d'emploi `sh test.sh path/to/test/folder`

Le code CIL se trouve dans chaque sous-dossier dans `test`.
Le code source se trouve, naturellement, dans le dossier `src`.
Les instructions pour l'extensions (enrichissement de la grammaire) `if / while / foreach` sont rajoutées dans `(facile.y) instruction:`

```
instruction: affectation | print | read | if-stmt |
    while-stmt | foreach-stmt | loop-interuptor;
```

2 Instruction `IF`:

2.1 Analyse lexicale:

On souhaite définir les lexèmes pour `if`, `elseif`, `else`, `endif`, ainsi que les opérateurs logiques. On prend en compte également la précédence des opérateurs. Par exemple: `a && b >= c` ↔ `a && (b >= c)`

2.1.1 `facile.lex`

```
%{
/* Exercice 1: if elseif else */
%}

if    return TOK_IF;
then  return TOK_THEN;
```

```
end   return TOK_END;
endif return TOK_ENDIF;

elseif return TOK_ELSEIF;
else   return TOK_ELSE;

or    return TOK_BOOL_OR;
and   return TOK_BOOL_AND;
not   return TOK_BOOL_NOT;
false return TOK_BOOL_FALSE;
true  return TOK_BOOL_TRUE;
"="   return TOK_BOOL_EQ;
"#"   return TOK_BOOL_NEQ;
">="  return TOK_BOOL_GTE;
">"   return TOK_BOOL_GT;
"<="  return TOK_BOOL_LTE;
"<"   return TOK_BOOL_LT;
```

2.1.2 facile.y

```
/* Logical OP */
%token TOK_BOOL_OR;
%token TOK_BOOL_AND;
%token TOK_BOOL_EQ;
%token TOK_BOOL_NEQ;
%token TOK_BOOL_GTE;
%token TOK_BOOL_GT;
%token TOK_BOOL_LTE;
%token TOK_BOOL_LT;
%token TOK_BOOL_NOT;
%token TOK_BOOL_TRUE;
%token TOK_BOOL_FALSE;

/* Operator precedence */
%left TOK_BOOL_OR;
%left TOK_BOOL_AND;
%left TOK_BOOL_NOT;
%left TOK_BOOL_EQ TOK_BOOL_GT TOK_BOOL_GTE TOK_BOOL_LT TOK_BOOL_LTE TOK_BOOL_NEQ;

/* If ElseIf Else */
%token TOK_IF
%token TOK_THEN;
%token<string> TOK_END;
%token<string> TOK_ENDIF;

%token TOK_ELSEIF;
%token TOK_ELSE;
```

2.2 Analyse syntaxique:

En utilisant le diagramme de Conway, on peut traduire rapidement en règles grammatiques.

- Exemple pour `boolean_expr`

```
boolean_expr:
    TOK_OPEN_PARENTHESIS boolean_expr TOK_CLOSE_PARENTHESIS
    {
        $$ = g_node_new("boolexpr"); <- le nom sera util pour écrire le code CIL
        g_node_append($$, $2);
    }
|
    TOK_BOOL_TRUE
    {
        $$ = g_node_new("boolTrue"); <- Cas particulier
    }
|
    TOK_BOOL_FALSE
    {
        $$ = g_node_new("boolFalse"); <- Cas particulier
    }
|
...
```

- Exemple pour `if-statement`

```
if-stmt:
    TOK_IF boolean_expr TOK_THEN code elseif else endif
    {
        $$ = g_node_new("if");
    g_node_append($$, $2);
    g_node_append($$, $4);
        g_node_append($$, $5);
    g_node_append($$, $6);
    g_node_append($$, $7);
    }
;

endif: <- On peut aussi termier simplement avec 'end'
    TOK_END{$$ = g_node_new("endif");} | TOK_ENDIF{$$ = g_node_new("endif");}
;
```

- Exemple pour `elseif / else`

```
elseif:
    // On peut avoir autant de elseif qu'on veut
    elseif TOK_ELSEIF boolean_expr TOK_THEN code elseif
    {
```

```
        $$ = g_node_new("elseif");
        g_node_append($$, $1);
        g_node_append($$, $3);
        g_node_append($$, $5);
        g_node_append($$, $6);
    }
|
    %empty
    {
        $$ = g_node_new("");
    }
;

else:
    TOK_ELSE code
    {
        $$ = g_node_new("else");
        g_node_append($$, $2);
    }
|
    // On peut ne pas avoir else
    %empty
    {
        $$ = g_node_new("");
    }
;
```

2.3 Génération de code CIL

Le code généré n'est pas forcément le plus factorisé, mais il fait le travail correctement. Pour le débogage, j'ai utilisé SharpLab pour apprendre et tester les codes CIL.

2.3.1 Expression booléenne

Pour le plupart des cas, les instructions de CIL suffisent amplement. Je vais vous présenter les cas ou il faut chercher un peu plus loin.

```
In [ ]: /* Handle boolean_expr */
        else if (node->data == "boolTrue"){
            fprintf(stream, " ldc.i4.1\n");
        }

        else if (node->data == "boolFalse"){
            fprintf(stream, " ldc.i4.0\n");
        }

        else if(node->data == "not"){
            produce_code(g_node_nth_child(node, 0)); // 'a' la valeur dans le stack
            fprintf(stream, "    ldc.i4.0\nceq\n"); // 1 si a == 0  sinon 0
        }

        else if(node->data == "neq"){
            produce_code(g_node_nth_child(node, 0));
```

```
        produce_code(g_node_nth_child(node, 1));
        fprintf(stream, "    ceq\nldc.i4.0\nceq\n"); // litterally "equal + not"
    }

    else if(node->data == "gte"){
        produce_code(g_node_nth_child(node, 0));
        produce_code(g_node_nth_child(node, 1));
        fprintf(stream, "    clt\nldc.i4.0\nceq\n"); // litt. "not lesser than"
    }

    else if(node->data == "lte"){
        produce_code(g_node_nth_child(node, 0));
        produce_code(g_node_nth_child(node, 1));
        fprintf(stream, "    cgt\nldc.i4.0\nceq\n"); // litt. "not greater than"
    }
```

2.3.2 If / ElseIf / Else

On veut simplement sauter à un autre endroit (offset) si la condition n'est pas vérifier. L'instruction `brfalse.s <target>` permet de faire exactement ça. Pour aider, j'ai déclaré différents types de offset au début comme variable globale, qui sera incrémenté à chaque exécution de `produce_code()`.

En outre chaque instruction CIL est marqué avec un offset par défaut et qu'on peut le renommer:

```
IL_0000: <codeCIL>
JUMP_POINT_0: <codeCIL>
```

```
In [ ]: int yylex();
        int yyerror(char *msg);

        GHashTable *table;

        // Offset - Useful for branching
        unsigned int offset = 0; // +1 everytime
        unsigned int loop_offset = 0; // +1 when loop

        FILE *stream;
        char *module_name;
        unsigned int max_stack = 10;

        extern void begin_code();
        extern void end_code();
        extern void produce_code(GNode * node);
```

```
In [ ]: /* If statement */
        else if(node->data == "if"){
            produce_code(g_node_nth_child(node, 0)); // boolean_expr

            // Mark the jump address
            guint endSbl = offset;
            fprintf(stream, "   brfalse.s IF_%d\n\n", endSbl);

            fprintf(stream, "   nop\n");
            produce_code(g_node_nth_child(node, 1)); // code wrapped with nop for catching
            fprintf(stream, "   nop\n");

            fprintf(stream, "   nop\n");
            fprintf(stream, "   IF_%d:", endSbl); // end of code, mark jump point

            produce_code(g_node_nth_child(node, 2)); // elseif
            produce_code(g_node_nth_child(node, 3)); // else
            produce_code(g_node_nth_child(node, 4)); // endif
        }
```

`elseif` reprend la même code CIL que `if`

```
In [ ]: /*ElseIf statement*/
        else if(node->data == "elseif"){
            produce_code(g_node_nth_child(node, 0)); // elseif
            produce_code(g_node_nth_child(node, 1)); // boolean_expr

            // Mark the jump address
            guint endSbl = offset;
            fprintf(stream, "   brfalse.s ELSEIF_%d\n\n", endSbl);

            fprintf(stream, "   nop\n");
            produce_code(g_node_nth_child(node, 2)); // code
            fprintf(stream, "   nop\n");

            fprintf(stream, "   nop\n");
            fprintf(stream, "   br.s IL_LAST\n\n");
            fprintf(stream, "   ELSEIF_%d: ", endSbl); // end of code, mark jump point

            produce_code(g_node_nth_child(node, 3)); // elseif
        }

        /* Else */
        else if(node->data == "else"){
            fprintf(stream, "   nop\n");
            produce_code(g_node_nth_child(node, 0)); // code
            fprintf(stream, "   nop\n");
        }
```

3 Instruction `WHILE / FOREACH`

Dans cette partie, on va traiter les boucles. `foreach` n'est qu'un `while` spécial avec un compteur.

On gère également les interupteurs `continue` et `break`

3.1 Analyse lexicale:

On souhaite définir les lexèmes pour `while`, `foreach`.

3.1.1 `facile.lex`

```
%{
/* Exercice 2: While */
%}
while     return TOK_WHILE;
do        return TOK_DO;
endwhile  return TOK_ENDWHILE;
".."      return TOK_ARR_TO;
continue  return TOK_CONTINUE;
break     return TOK_BREAK;

%{
/* Exercice 3: Foreach */
%}
foreach     return TOK_FOREACH;
in          return TOK_IN;
endforeach  return TOK_ENDFOREACH;
```

3.1.2 facile.y

```
%type<node> while-stmt
%type<node> endwhile

%type<node> foreach-stmt
%type<node> endforeach
%type<node> loop-interuptor
```

3.2 Analyse syntaxique:

En utilisant le diagramme de Conway, on peut traduire rapidement en règles grammatiques.

- Exemple pour `loop-interuptor`

```
loop-interuptor:
    TOK_CONTINUE TOK_SEMICOLON
    {
        $$ = g_node_new("skipItr");
    }
|
    TOK_BREAK TOK_SEMICOLON
    {
        $$ = g_node_new("breakLoop");
    }
;
```

- Exemple pour `while-stmt`

```
while-stmt:
    TOK_WHILE boolean_expr TOK_DO code endwhile
    {
        $$ = g_node_new("while");
        g_node_append($$, $2);
        g_node_append($$, $4);
        g_node_append($$, $5);
    }
;

endwhile:
    TOK_END
    {
        $$ = g_node_new("endwhile");
    }
|
    TOK_ENDWHILE
    {
        $$ = g_node_new("endwhile");
    }
;
```

- Exemple pour `foreach-stmt`

```
foreach-stmt:
    TOK_FOREACH ident TOK_IN expr TOK_ARR_TO expr TOK_DO code endforeach
    {
        $$ = g_node_new("foreach");
        g_node_append($$, $4);
        g_node_append($$, $6);
        g_node_append($$, $8);
        g_node_append($$, $9);
    }
;

endforeach:
    TOK_END
    {
        $$ = g_node_new("endforeach");
    }
|
    TOK_ENDFOREACH
    {
        $$ = g_node_new("endforeach");
    }
;
```

Un interrupteur peut être `continue` ou `break`. Ce sont tous les deux des sauts vers un autre point dans le code. `continue` saute vers la prochaine itération, marquée mar une incrémentation du compteur puis une vérification de la condition. Au contraire, `break` saute vers la fin de la boucle.

```
In [ ]: /* Loop interuptor */
        else if(node->data == "skipItr"){
            guint endSbl = loop_offset;
            fprintf(stream, "   br.s LOOP_INCR_%d\n\n", endSbl);
        }

        else if(node->data == "breakLoop"){
            guint endSbl = loop_offset;
            fprintf(stream, "   br.s LOOP_END_%d\n\n", endSbl);
        }

In [ ]: /* While */
        else if(node->data == "while"){

            // Branch out
            guint endSbl = loop_offset;
            fprintf(stream, "   br.s LOOP_HEAD_%d\n", endSbl); // Init first iteration by
        jumping to head
            fprintf(stream, "   // Start loop (head: LOOP_HEAD_%d)\n", endSbl);
            fprintf(stream, "   LOOP_START_%d: nop\n", endSbl); // Mark beginning

            fprintf(stream, "   nop\n"); // un code sans nop c'est comme un jour sans le soleil
            produce_code(g_node_nth_child(node, 1)); // code
            fprintf(stream, "   nop\n");

            // Non existing counter
```

```
            fprintf(stream, "   LOOP_INCR_%d: nop\n", endSbl);

            fprintf(stream, "   LOOP_HEAD_%d: ", endSbl); // Mark head
            produce_code(g_node_nth_child(node, 0)); // boolean_expr
            fprintf(stream, "   brtrue.s LOOP_START_%d\n", endSbl); // jump to beginning of loop
        if cond

            produce_code(g_node_nth_child(node, 2)); // endwhile
            fprintf(stream, "   // End loop\n");
            fprintf(stream, "   LOOP_END_%d: nop\n", endSbl);

            loop_offset++;
        }
```

In [ ]:
```
        /* Foreach */
        else if(node->data == "foreach"){

            // Initialise counter variable
            fprintf(stream, "   nop\n");
            produce_code(g_node_nth_child(node, 0)); // expr
            fprintf(stream, "   stloc.0\n");

            // Branch off - while like
            guint endSbl = loop_offset;
            fprintf(stream, "   br.s LOOP_HEAD_%d\n", endSbl); // Jump to head
            fprintf(stream, "   // Start loop (head: LOOP_HEAD_%d)\n", endSbl);

            fprintf(stream, "   LOOP_START_%d: nop\n", endSbl); // Mark beginning

            // Procedure
            produce_code(g_node_nth_child(node, 2)); // code
            fprintf(stream, "   nop\n");

            // Increment counter
            fprintf(stream, "   LOOP_INCR_%d: ", endSbl);
            fprintf(stream, "   nop\n");
            fprintf(stream, "   ldloc.0\n");
            fprintf(stream, "   ldc.i4.1\n");
            fprintf(stream, "   add\n");
            fprintf(stream, "   stloc.0\n\n"); // Unload counter

            fprintf(stream, "   LOOP_HEAD_%d: ", endSbl); // Mark head
            fprintf(stream, "   ldloc.0\n"); // Reload counter
            produce_code(g_node_nth_child(node, 1)); // expr

            // Check condition
            fprintf(stream, "   cgt\n");
            fprintf(stream, "   ldc.i4.0\n");
            fprintf(stream, "   ceq\n");

            fprintf(stream, "   brtrue.s LOOP_START_%d\n", endSbl); // jump to beginning of loop
        if cond

            produce_code(g_node_nth_child(node, 3)); // endforeach
            fprintf(stream, "   // End loop\n");
            fprintf(stream, "   LOOP_END_%d: nop\n", endSbl);

            loop_offset++;
        }
```

4 Mieux gérer les erreurs de syntaxe

In [ ]: