Modif projet Compilation

1. Concat function

Pas de changements côtés lexical/syntaxique -> on utilise le même ‘+’ que pour une addition (barith)

Pas de changements main.c/symbols.c/toy.h/utils.c/ast.c

Fichier analysis.c :

Fonction analysis\_expression, case 2 / barith

suppression de la condition op1 & op2 doivent être des nombres.

On vérifie maintenant si op1&op1 sont des nombres, pour attribuer au nœud le bon type, si ce n’est pas des nombres le type du nœud sera « string » si l’opérateur est un +.

Fichier toy-runtime.h : déclaration de la nouvelle fonction concat

\_toy\_string \_toy\_concat\_string(\_toy\_string str1, \_toy\_string str2);

Fichier toy\_runtime.c: corps de la fonction concat

Prends en paramètre deux toy\_string (char\*) et retourne un toy\_string

Cas string null : return NULL

Fichier prodcode.c :

Dans le cas où c’est un nœud de type string, arité 2 et qu’il y a un ‘+’, on génère la fonction de concaténation.

Problème :

fail-concat.toy : print(« 1 » + « 2 »)  faux ? Print(1+2)  faux ?

2- Ajout des opérateurs +=, -=, \*= et /=

Lexical :

+= PE

-= ME

\*= TE

Syntaxic :

%right '=' PE ME TE DE

expr :

| var '=' expr { $$ = make\_expression("=", assign, 2, $1,$3);}  
| var PE expr { $$ = make\_expression("+=", assign, 2, $1,$3);}  
| var ME expr { $$ = make\_expression("-=", assign, 2, $1,$3);}  
| var TE expr { $$ = make\_expression("\*=", assign, 2, $1,$3);}  
| var DE expr { $$ = make\_expression("/=", assign, 2, $1,$3);}

Analysis.c

RAS

prodcode.c

RAS

3- Switch

lexical.l

"switch" return KSWITCH;  
"case" return KCASE;  
"default" return KDEFAULT;

syntax.y :

%token KWHILE KIF KPRINT KRETURN KBREAK KFOR KSWITCH KCASE KDEFAULT  
%type <lst> fparam\_list eparam\_list stmt\_list cond\_list  
%type <ast> defcond

KSWITCH '{' cond\_list defcond '} { $$ = make\_switch\_statement($3, $4);}

cond\_list: cond\_list KCASE expr ':' stmt { list\_append($1, $3, FREE\_NODE);  
 list\_append($1, $5, FREE\_NODE);  
 $$ = $1; }  
 | /\* empty \*/ { $$ = list\_create(); }  
 ;  
defcond : KDEFAULT ':' stmt { $$ = make\_expr\_statement($3); }  
 | /\* empty \*/ { $$ = make\_expr\_statement(NULL); }  
;

ast.h :

/\* ---- SWITCH ------------------------------------------------------------ \*/  
struct s\_switch\_statement {  
 ast\_node header; ///< AST header  
 List cases; ///< List of cases with statements  
 ast\_node \*caseDefault; ///< default statement (could be null)  
};  
  
  
ast\_node \*make\_switch\_statement(List cases, ast\_node \*defcond);

ast.c :

/\* ---- SWITCH ------------------------------------------------------------ \*/  
static void free\_switch\_statement(ast\_node \*node){  
 struct s\_switch\_statement \*n = (struct s\_switch\_statement \*)node;  
 list\_for\_each(n->cases, (list\_iterator) free\_node);  
 free\_node(n->caseDefault);  
 free(node);  
}  
  
ast\_node \*make\_switch\_statement(List cases, ast\_node \*defcond){  
 DEF\_AST(p, switch\_statement, NULL);  
 p->cases = cases;  
 p->caseDefault = defcond;  
 return (ast\_node \*) p;  
}

analysis.h :

void analysis\_switch\_statement(ast\_node \*node);

analysis.c

void analysis\_switch\_statement(ast\_node \*node){  
 struct s\_switch\_statement \*n = (struct s\_switch\_statement \*) node;  
 enter\_scope();  
 list\_for\_each(n->cases, (list\_iterator) analysis);  
 analysis(n->caseDefault);  
 leave\_scope();  
}

prodcode.h :

void produce\_code\_switch\_statement(ast\_node \*node);

prodcode.c :

void produce\_code\_switch\_statement(ast\_node \*node){  
 struct s\_switch\_statement \*n = (struct s\_switch\_statement \*) node;  
 int nbCases = list\_size(n->cases);  
 int i = 0;  
 FORLIST(p, n->cases) {  
 if(nbCases % 2 == 0) {  
 indent(0);  
 if(i == 0) {  
 emit("if( ");  
 i = 1;  
 } else {  
 emit("else if(");  
 }  
 }  
 else {  
 indent(+1);  
 }  
 code(list\_item\_data(p));  
 if(nbCases % 2 == 0) {  
 emit(")\n {\n");  
 }  
 else  
 emit("\n}");  
 nbCases --;  
 }  
 if(n->caseDefault){  
 indent(0);  
 if(i == 1) {  
 emit("else {\n");  
 indent(+1);  
 }  
 code(n->caseDefault);  
 if(i == 1) {  
 indent(-1);  
 emit("}\n");  
 }  
 }  
  
}

4- Exceptions

lexical.l

"try" return KTRY;  
"catch" return KCATCH;  
"finally" return KFINALLY

"throw" return KTHROW;

;

syntax.y :

%token KWHILE KIF KPRINT KRETURN KBREAK KFOR KSWITCH KCASE KDEFAULT KTRY KCATCH KFINALLY KTHROW  
%type <lst> fparam\_list eparam\_list stmt\_list cond\_list  
%type <ast> defcond  
%type <ast> catch finally

stmt : …

| KTRY stmt catch finally { $$ = make\_exception\_statement($2, $3, $4); }

| KTHROW ';' { $$ = make\_throw\_statement(); }

catch: KCATCH stmt { $$ = make\_expr\_statement($2); }  
 | /\* empty \*/ { $$ = make\_expr\_statement(NULL); }  
 ;  
  
finally: KFINALLY stmt { $$ = make\_expr\_statement($2); }  
 | /\* empty \*/ { $$ = make\_expr\_statement($2); }  
 ;

le token KTRY est obligatoire puis dans defexcep trois cas:

-le try est suivi d’un catch

-le try est suivi d’un finally (donc pas de catch)

- le try est seul (ni catch ni finally)

ast.h :

/\* ---- EXCEPTION ------------------------------------------------------------ \*/  
struct s\_exception\_statement {  
 ast\_node header; ///< AST header  
 ast\_node \*try;  
 ast\_node \*catch;  
 ast\_node \*finally;  
};  
  
ast\_node \*make\_exception\_statement(ast\_node \*try, ast\_node \*catch, ast\_node \*finally);

/\* ---- THROW ------------------------------------------------------------ \*/  
struct s\_throw\_statement {  
 ast\_node header; ///< AST header  
};  
  
ast\_node \*make\_throw\_statement();

ast.c :

/\* ---- EXCEPTION ------------------------------------------------------------ \*/  
static void free\_exception\_statement(ast\_node \*node){  
 struct s\_exception\_statement \*n = (struct s\_exception\_statement \*)node;  
 free\_node(n->try);  
 free\_node(n->catch);  
 free\_node(n->finally);  
 free(node);  
}  
  
ast\_node \*make\_exception\_statement(ast\_node\* try, ast\_node\* catch, ast\_node\* finally){  
 DEF\_AST(p, exception\_statement, NULL);  
 p->try = try;  
 p->catch = catch;  
 p->finally = finally;  
 return (ast\_node \*) p;  
}

/\* ---- THROW ------------------------------------------------------------ \*/  
static void free\_throw\_statement(ast\_node \*node){  
 free(node);  
}  
  
ast\_node \*make\_throw\_statement(void){  
 DEF\_AST(p, throw\_statement, NULL);  
 return (ast\_node \*) p;  
}

analysis.h :

void analysis\_exception\_statement(ast\_node \*node);

void analysis\_throw\_statement(ast\_node \*node);

analysis.c

void analysis\_exception\_statement(ast\_node \*node){  
 struct s\_exception\_statement \*n = (struct s\_exception\_statement \*) node;  
 enter\_scope();  
 analysis(n->try);  
 analysis(n->catch);  
 analysis(n->finally);  
 leave\_scope();  
}

void analysis\_throw\_statement(ast\_node \*node){  
   
}

prodcode.h :

void produce\_code\_exception\_statement(ast\_node \*node);

prodcode.c :