|  |
| --- |
| COMPTE RENDU |

### GL4/2

### -mohamed assil ben amor

### -ARSSLEN IDADI

PROJET

## COMPILATEUR

Cadre du projet

Le mini-projet est développé en C# 7 en utilisant l’IDE Microsoft Visual Studio 2017.

Ce Rapport va couvrir toutes les étapes réalisées en ordre croissant qui ont abouti à la version finale qui est déjà mis à la disposition des contributeurs world-wide sur GITHUB.

Analyse Lexicale (Scanner.cs)

En informatique, l'analyse lexicale, lexing, segmentation ou tokenization (à différencier du terme tokenization utilisé en sécurité informatique) fait partie de la première phase de la chaîne de compilation. Elle consiste à convertir une chaîne de caractères en une liste de symboles (tokens en anglais).

On a créé la classe Token

public class Token {

public int kind; // token kind

public int pos; // token position in bytes in the source text (starting at 0)

public int charPos; // token position in characters in the source text (starting at 0)

public int col; // token column (starting at 1)

public int line; // token line (starting at 1)

public string val; // token value

public Token next; // ML 2005-03-11 Tokens are kept in linked list

}

Le principe est simple, on charge le fichier dans un flot et après on lit charactère par charactère et on essaie de construire une liste de tokens en éliminant les charactères inutiles.

Lecture du Flot

void NextCh() {

if (oldEols > 0) { ch = EOL; oldEols--; }

else {

pos = buffer.Pos;

// buffer reads unicode chars, if UTF8 has been detected

ch = buffer.Read(); col++; charPos++;

// replace isolated '\r' by '\n' in order to make

// eol handling uniform across Windows, Unix and Mac

if (ch == '\r' && buffer.Peek() != '\n') ch = EOL;

if (ch == EOL) { line++; col = 0; }

}

}

La fonction NextToken (Voir le code dans le fichier Scanner.cs) fait l’affaire elle a pour but la détermination du token suivant en se basant sur la mémorisation de l’état courant et précèdent (Automate à état fini déterministe) .

La transformation du grammaire en LL(1)

On a transformé le grammaire donné en un grammaire LL(1) sans conflit et valide en éliminant les

Ci-dessous la grammaire finale LL(1) écrit en EBNF

programmes = prog ident semicolon declaration block\_stmt dot.

declaration = (variable\_declarations | ) method\_declarations.

variable\_declarations = variable\_declaration ( | variable\_declarations).

variable\_declaration = var ident\_list ':' int ';' .

ident\_list = ident ( | ',' ident\_list).

method\_declarations = method\_declaration ';' ( | method\_declarations).

method\_declaration = method\_header (variable\_declarations | ) block\_stmt.

method\_header = func ident ( '(' parameter\_list ')' | ) ':' int ';'

| proc ident ( '(' parameter\_list ')' | ) ';'

.

parameter\_list = parameter ( | ';' parameter\_list).

parameter = ident ':' int | var ident ':' int.

block\_stmt = '{' ( stmt\_list | ) '}'.

stmt\_list = stmt ( | ';' stmt\_list).

stmt = ident ( | '=' expr | '(' expr\_list')' )

| block\_stmt

| if expr then stmt else stmt

| while expr do stmt

.

expr\_list = expr (

| ',' expr\_list

).

expr = simple\_expr ( | oprel simple\_expr).

simple\_expr = signe terme

| terme (

| signe simple\_expr

| or simple\_expr

)

.

terme = facteur (opmul terme | ).

facteur = ident ( | '(' expr\_list')' )

| nb

| not facteur

| '(' expr ')'

.

signe = '+'

| '-'

.

opmul = '/' | '\*' | "div" | "mod" | "and".

oprel = "=="

| "<>"

| '>'

| '<'

| ">="

| "<="

.

Analyse Syntaxique (Parser.cs)

Après avoir obtenu la liste des symboles reconnu on doit voir s’ils ont été bien formés et respecte notre grammaire, pour ce faire on développera un analyseur syntaxique descendant (Recursive descent parser) dont chaque fonction va représenter une production du grammaire. Ainsi, la structure du programme résultant reflète étroitement celle de la grammaire qu'il reconnaît.

Exemple de productions :

void method\_declaration(ref MethodDeclaration m) {

method\_header(ref m);

Statement s = null; List<VariableDeclaration> vl = null;

if (la.kind == 4) {

variable\_declarations(ref vl);

} else if (la.kind == 29) {

vl = new List<VariableDeclaration>();

} else SynErr(45);

m.LocalVariables = vl;

block\_stmt(ref s);

m.Block = s;

}

void facteur(ref Expression ex) {

if (la.kind == 1) {

Get();

if (StartOf(8)) {

ex = new VariableExpression{ Name = t.val, Location = new Location(t.line, t.col, t.charPos) } ;

} else if (la.kind == 22) {

ex = new MethodInvocationExpression{ Name = t.val, Location = new Location(t.line, t.col, t.charPos) } ; List<Expression> el = null;

Get();

expr\_list(ref el);

(ex as MethodInvocationExpression).Arguments = el;

Expect(23);

} else SynErr(61);

} else if (la.kind == 2) {

Get();

ex = new IntegralExpression{ Value = int.Parse(t.val), Location = new Location(t.line, t.col, t.charPos) } ;

} else if (la.kind == 22) {

Expression target = null;

Get();

ex = new ParenthesisExpression{ Location = new Location(t.line, t.col, t.charPos) } ;

expr(ref target);

(ex as ParenthesisExpression).Target = target;

Expect(23);

} else if (la.kind == 17) {

Expression e = null;

Get();

ex = new UnaryOperationExpression{ Location = new Location(t.line, t.col, t.charPos), Operator = Operators.LogicalNot } ;

facteur(ref e);

(ex as UnaryOperationExpression).Expression = e;

} else SynErr(62);

}

Les fonctions de base :

void Get () {

for (;;) {

t = la;

la = scanner.Scan();

if (la.kind <= maxT) { ++errDist; break; }

la = t;

}

}

void Expect (int n) {

if (la.kind==n) Get(); else { SynErr(n); }

}

La fonction Get permet de récupérer le symbole suivant tandis que la fonction Expect comme son nom l’indique elle attend que le token avec Kind = n donné en paramètre est le suivant (la : Look Ahead Token).

Pour plus de détails veuillez voir le code source Parser.cs

Abstract Syntax Tree (AST)

AST est un arbre dont les nœuds internes sont marqués par des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs. Autrement dit, généralement, une feuille est une variable ou une constante.

En fait tous les nœuds implémentent l’interface **INode** et la classe **Node.**

public interface INode

{

Location Location { get; set; }

}

public abstract class Node : INode, IResolve

{

public Location Location { get; set; }

public virtual bool Resolve(ResolveContext rc)

{

rc.CurrentStatementState = ResolverState.Create(rc);

return true;

}

public virtual INode DoResolve(ResolveContext rc)

{

return this;

}

public virtual string CommentString()

{

return "";

}

}

L’interface IResolve

public interface IResolve

{

INode DoResolve(ResolveContext rc);

bool Resolve(ResolveContext rc);

}

Cette interface définit un contrat qui permet d’intégrer la résolution sémantique.

Analyse Sémantique

Cette partie est importante, en fait l’idée est de créer une classe nommée ResolveContext permet définir un contexte de résolution de symboles (Variables, Fonctions, Paramètres et variables locaux). Cette classe elle permet non seulement la résolution des noms mais elle sauvegarder l’état actuel de résolution dans l’arbre AST.

La classe Resolver représente un espace de stockage de nom.

La classe ResolverState permet de mémoriser l’état actuel du résolveur.

Les contrats de scopes :

public interface ILoop

{

ILoop ParentLoop { get; set; }

Label EnterLoop { get; set; }

Label ExitLoop { get; set; }

Label LoopCondition { get; set; }

}

public interface IConditional

{

IConditional ParentIf { get; set; }

Label Else { get; set; }

Label ExitIf { get; set; }

}

Les scopes de résolution

[Flags]

public enum ResolveScopes

{

Normal = 1 << 1,

Loop = 1,

If = 1 << 2,

}

Chaque classe va définir la fonction DoResolve(ResolveContext rc) qui va être déclenché d’une manière récursive en utilisant depuis la racine (CompilationUnit).

Le code est trop long veuillez voir le projet

Génération du code assembleur ASM16

Pour générer le code assembleur on ajoute l’implémentation des instructions assembleur (NCAsm), des optimiseurs (Assembleur) qui implémentent l’interface IOptimizer…

La classe qui va écrire le fichier assembleur s’appelle AssemblyWriter.

N.B : cette libraire a été développé complétement du zéro dans un autre projet appelé Vatu qui est dèja sur Github.

La classe EmitContext permet de définir un contexte d’émission de code (les instructions assembleurs).

Les nœuds implémentent l’interface IEmit, elle permet de définir un contrat d’émission de code

/// <summary>

/// Basic Emit for CodeGen

/// </summary>

public interface IEmit

{

/// <summary>

/// Emit code

/// </summary>

/// <returns>Success or fail</returns>

bool Emit(EmitContext ec);

}

Pour les expressions on doit ajouter plus que Emit on doit prendre en charge l’émission de l’expression à la pile. Donc on ajoute une interface qui s’appelle IEmitExpr

/// <summary>

/// Special Emit for expressions

/// </summary>

public interface IEmitExpr

{

bool EmitToStack(EmitContext ec);

bool EmitFromStack(EmitContext ec);

bool EmitToRegister(EmitContext ec, RegistersEnum rg);

bool EmitBranchable(EmitContext ec, Label truecase, bool val);

}

EmitBranchable est importante pour les operations booléennes

Exemple d’émission d’un code d’une fonction

proc test(s:int;i:int);

var v:int;

{

v = s + 1 \* i;

s = t

};

Code assembleur généré

test\_2:

; Method: Name = test

; create stackframe

push BP

mov BP, SP

; Parameters Definitions

; Parameter s @BP4

; Parameter i @BP6

; Block

; Push Parameter @BP 4

push word [BP + 4]

; Push Parameter @BP 6

; 1 \* i

mov word CX, [BP + 6]

mov word AX, 1

mul CX

; s + 1 \* i

pop word AX

add AX, CX

; v=s + 1 \* i

; Pop Var @BP-2

mov word [BP - 2], AX

; Push Field @t 0

push word [t]

; s=t

; Pop Parameter @BP 4

pop word [BP + 4]

; return label

test\_2\_ret:

; destroy stackframe

leave

ret 4

Exemple d’optimisation (Push/Pop)

Push 5

Pop Ax

Sera transformé en

Mov word ax,5

Génération du code binaire (NASM)

Cette étape est simple le fichier généré par notre compilateur sera fournit à l’outil NASM

nasm.exe test.s -o test.bin