



ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET D'ANALYSE DES
SYSTÈMES - RABAT

Rapport de projet de compilation

Sujet : Développement d'un compilateur du langage
CashScript

Réalisé par :

Hajar DAMI :GL2
Obaydah BOUIFADENE :GL1
Amina CHAABANE :GL1
Yasser KARAMI :GL3
Ilyass MOUFID :ISEM

Encadré par :

- M. Youness TABII
- M. Rachid OULAD
HAJ THAMI

Année académique 2021/2022



Remerciements :

Ce projet est le fruit des conseils et des critiques bienveillantes d'un grand nombre de personnes. Nous tenons à les remercier ici et leur faire part de toutes nos gratitude, pour avoir été à l'écoute et toujours d'une aide précieuse.

Plus particulièrement, nous souhaitons adresser à travers ces courtes lignes nos remerciements les plus sincères à nos professeurs M. Youness Tabii et M. Rachid Oulad Haj Thami, qui ont eu le soin d'examiner et encadrer ce travail et ainsi nous offrir une véritable opportunité d'apprentissage.

Enfin, nos remerciements s'adressent aux professeurs et au corps administratif de l'Ecole Nationale Supérieure de l'Informatique et de l'Analyse des Systèmes ENSIAS.



Résumé

Le présent rapport résume le travail que nous avons réalisé dans le cadre de notre projet de compilation.

Ce projet consiste à élaborer le compilateur d'un langage de programmation basé sur CashScript. C'est un langage qui facilite d'une part les contrats intelligents sur Bitcoin Cash et d'une autre part il prend en considération tous les éléments d'un langage de programmation structuré.

Toutes les étapes ont été respectées, allons de l'analyseur lexical jusqu'à l'analyseur sémantique

Mots-clés : Langage de programmation, compilateur, contrats, Bitcoin, analyseur lexical, analyseur syntaxique

Abstract

This report summarizes and describes the work we done in our compilation project.

This project consists in developing the compiler of a CashScript programming language. It is a language that facilitates smart contracts on Bitcoin Cash and in the same time it takes into consideration all the elements of a structured programming language.

All the steps have been followed, from the lexical analyzer to the parser.

Key-words : Programming language ,compiler,contract,bitcoin, lexical analyzer ,parser

Table des matières

Introduction générale	1
1 Présentation du projet	2
1.1 Contexte général du projet	2
1.2 Notion général	2
1.2.1 Bitcoin	2
1.2.2 Bitcoin cash	3
1.2.3 Blockchain	3
1.3 Description de langage	3
1.3.1 Contract	3
1.3.2 Pragma	3
1.3.3 Types	3
1.3.4 Opération	4
1.3.5 Structure de contrôle	4
1.3.6 Les instruction de saisi/affichage de base :	5
1.3.7 Exemple de programme cashscript	5
2 Analyse	8
2.1 La liste des tokens :	8
2.2 Les règles de production	11
2.3 Les règles sémantiques	13
2.4 Les premiers et les suivants	13
2.5 Analyse lexicale	18
2.5.1 Structure	18
2.6 Analyse syntaxique	18
2.6.1 Structure	18
2.7 Analyse sémantique	20
3 Test de compilateur	21
3.1 1er Test (Analyseur lexical)	21
3.2 2 ème Test (Analyseur syntaxique)	22
3.3 3 ème Test (Analyseur sémantique)	25
3.4 4 ème test de compilateur	26
3.4.1 Programme	26
3.4.2 Exécution	26
Conclusion	28

Table des figures

1.1	exemple1	6
1.2	exemple2	7
3.1	Test1(Analyseur lexical)	21
3.2	exécution1 (Analyseur lexical)	21
3.3	correction Test1(Analyseur lexical)	22
3.4	correction exécution1 (Analyseur lexical)	22
3.5	Test2(Analyseur syntaxique)	23
3.6	exécution2 (Analyseur syntaxique)	23
3.7	correction Test2(Analyseur syntaxique)	24
3.8	correction exécution2 (Analyseur syntaxique)	25
3.9	Test 3(Analyseur sémantique)	25
3.10	exécution 3 (Analyseur sémantique)	25
3.11	Test4	26
3.12	exécution test4	26
3.13	exécution analyseur lexical	27
3.14	exécution analyseur lexical(suite)	27

Introduction générale

Ces dernières années, la technologie blockchain a suscité beaucoup d'intérêt. La communauté des développeurs s'est redoublée grâce à l'intégration de nouvelles utilisations de la blockchain dite : contrats intelligents.

La plus grande plateforme de contrats intelligents sur le marché est Ethereum, en utilisant principalement Solidity .il s'agit d'un langage de programmation orienté objet dédié à l'écriture de contrats intelligents. Il est utilisé pour implémenter des smartcontrat sur diverses blockchains, notamment Ethereum.

Dans ce sens et dans le cadre de notre projet compilation nous avons choisi de travailler sur la réalisation d'un compilateur d'un langage intitulé : cash script inspiré de Solidity ,qui permet au plus des fonctionnalité de base d'un langage de programmation, de rédiger des contrats mais de manière plus simplifier que Solidity .

Ce projet a été d'une part une occasion idéale de mettre en pratique tout ce qui a été assimilé pendant le semestre dans le cours de compilation et d'une autre part une opportunité pour voir de nouvelles notions et découvrir le monde de blockchain et Bitcoin.

Chapitre 1

Présentation du projet

Ce chapitre a pour objectif de situer le projet dans son contexte général, à savoir la problématique qui a inspiré la création de notre langage de programmation, la description du projet et les objectifs à atteindre.

1.1 Contexte général du projet

Ce projet s'inscrit dans le cadre de la synthèse des acquis du module de compilation de deuxième année à l'ENSIAS. Le travail qui nous a été assigné était de concevoir un compilateur. Nous avons choisi de travailler sur le compilateur d'un langage basé sur cashscript.

1.2 Notion général

Afin de mieux contextualiser notre sujet nous définissons quelques notions importantes

1.2.1 Bitcoin

Le bitcoin est une monnaie virtuelle créée en 2009 par une personne non identifiée dont le pseudonyme est Satoshi Nakamoto. Contrairement aux monnaies classiques (également appelées monnaie fiat), le bitcoin n'est pas émis et administré par une autorité bancaire. Il est émis sur le protocole blockchain du même nom.

1.2.2 Bitcoin cash

Le bitcoin cash est une devise virtuelle à part entière, créée en août 2017 suite à une scission du bitcoin. Bien que similaire au bitcoin à de nombreux égards, elle opère selon son propre ensemble de règles et avec sa propre blockchain (ou chaîne de blocs).

1.2.3 Blockchain

La blockchain (dont la traduction en français est chaîne de blocs) est une technologie qui permet de stocker et transmettre des informations de manière transparente, sécurisée et sans organe central de contrôle. Elle ressemble à une grande base de données qui contient l'historique de tous les échanges réalisés entre ses utilisateurs depuis sa création.

1.3 Description de langage

1.3.1 Contract

Un élément de base de CashScript est le contrat .les contacts sont similaires aux classes des langages orientés objet, mais avec une différence, en effet une fois qu'un contrat est instancié avec certains paramètres, ces valeurs ne peuvent pas changer. Ce contrat regroupe un ensemble de fonctions qui peut agir sur les valeurs du contrat pour pouvoir dépenser les fonds bloqués dans ce contrat.

1.3.2 Pragma

Un fichier de contrat peut commencer par une directive pragma pour indiquer la version de CashScript pour laquelle le contrat a été écrit. Cela garantit qu'un contrat n'est pas compilé avec une version de compilateur non prise en charge, ce qui pourrait entraîner des effets secondaires imprévus.(le langage en cours d'évolution)

1.3.3 Types

- Boolean :true ou false

- Integer
- Bytes :séquence de bits
- String
- datasig :Séquence de bytes représentant une signature de données
- sig : Séquence de bytes représentant une signature de transaction.
- pubkey :Séquence de bytes représentant une clé publique

1.3.4 Opération

Ce tableau donne la signification des opérations utilisées dans le langage

Opérateur	Signification
++, -	Postfix increment and decrement
-	Unary minus
!	Logical NOT
/, %	Division and modulo
+, -	Addition et soustraction
+	String / bytes concatenation
<, >, <=, >=	Numeric comparison
==, !=	Equality and inequality
&	Bitwise AND
^	Bitwise XOR
	Bitwise OR
&&	Logical AND
	Logical OR
=	Assignment

1.3.5 Structure de contrôle

Vous trouvez ci dessous les prototypes de quelque structure de contrôles

- Condition :
- Loop :

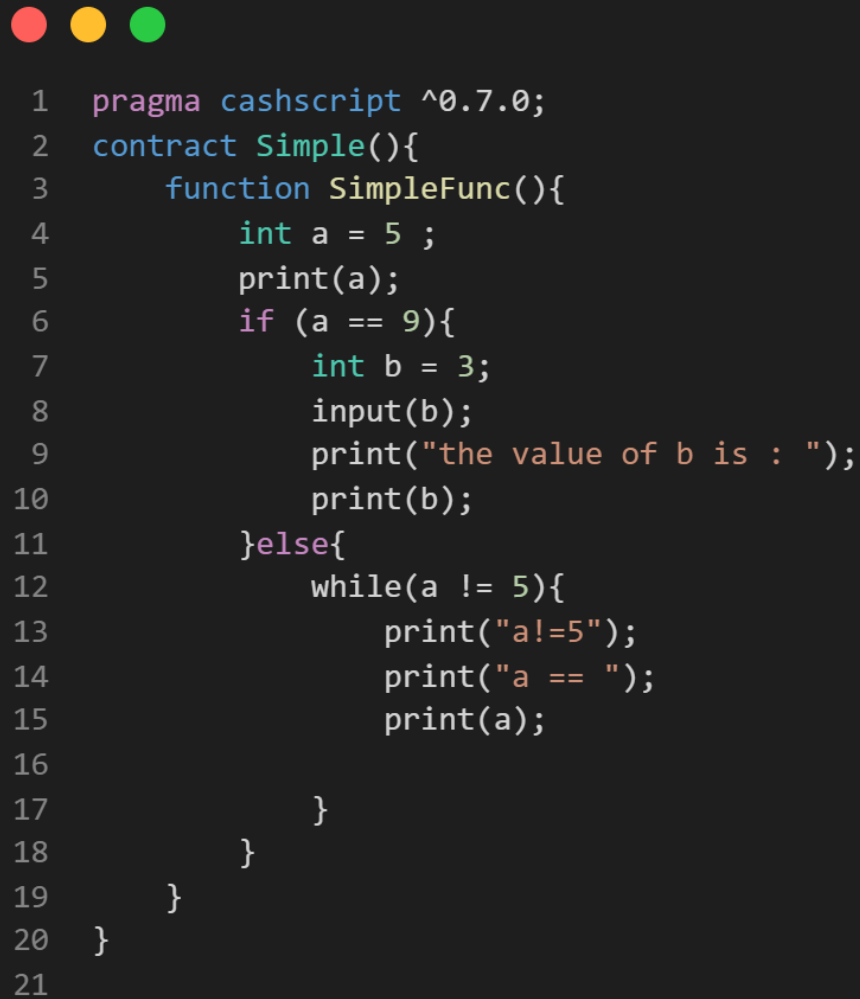
```
if(condition1) {  
  /*traitement*/  
}  
else {  
  /*traitement*/  
}  
  
while(condition1) {  
  /*Bloc*/  
}
```

1.3.6 Les instruction de saisi/affichage de base :

- Affichage : `print(valeur);`
- Saisi : `input(valeur);`

1.3.7 Exemple de programme cashscript

Vous trouvez ci-dessous quelques exemples de programmes cashscript



```
1  pragma cashscript ^0.7.0;
2  contract Simple(){
3      function SimpleFunc(){
4          int a = 5 ;
5          print(a);
6          if (a == 9){
7              int b = 3;
8              input(b);
9              print("the value of b is : ");
10             print(b);
11         }else{
12             while(a != 5){
13                 print("a!=5");
14                 print("a == ");
15                 print(a);
16             }
17         }
18     }
19 }
20 }
21
```

FIGURE 1.1 – exemple1

```

1  pragma cashscript ^0.6.0;
2
3  contract Mecenaz(bytes recipient, bytes funder, int pledge) {
4      // Allow the receiver to claim their monthly pledge amount
5      function receive(pubkey pk, sig s) {
6          // The transaction can be signed by anyone, because the money can only
7          // be sent to the correct address
8          require(checkSig(s, pk));
9
10         // Check that the UTXO is at least 30 days old
11         require(tx.age >= 30 days);
12
13         // Use a hardcoded miner fee
14         int minerFee = 1000;
15
16         // Retrieve the UTXO's value and cast it to an integer
17         int intValue = int(bytes(tx.value));
18
19         // Check if the UTXO's value is higher than the pledge amount + fee
20         if (intValue <= pledge + minerFee) {
21             // Create an Output that sends the remaining balance to the recipient
22             bytes out1 = new OutputP2PKH(bytes(intValue - minerFee), recipient);
23
24             // Enforce that this is the only output for the current transaction
25             require(hash256(out1) == tx.hashOutputs);
26         } else {
27             // Create an Output that sends the pledge amount to the recipient
28             bytes out1 = new OutputP2PKH(bytes(pledge), recipient);
29
30             // Create an Output that sends the remainder back to the contract
31             bytes remainder = bytes(intValue - pledge - minerFee);
32             bytes out2 = new OutputP2SH(remainder, hash160(tx.bytecode));
33             string tempString = "hello";
34             string tempString2 = "hello";
35             int a = 5;
36             int b = a + 6;
37             require_time(tempString == tempString2, 5);
38
39             // Enforce that these are the only outputs for the current transaction
40             require(hash256(out1 + out2) == tx.hashOutputs);
41         }
42         while (true){
43             int a = 5;
44             int b = a + 6;
45             require_time(tempString == tempString2, 5);
46         }
47         string alpha = "";
48         input(alpha);
49         print(alpha + "22");
50     }
51
52     // Allow the funder to reclaim their remaining pledges
53     function reclaim(pubkey pk, sig s) {
54         require(hash160(pk) == funder);
55         require(checkSig(s, pk));
56     }
57 }

```

FIGURE 1.2 – exemple2

Chapitre 2

Analyse

La réussite de tout projet dépend de la qualité de son départ. De ce fait, l'étape de l'analyse constitue la base de départ de notre travail, elle doit décrire sans ambiguïté le langage à développer. Pour assurer les objectifs attendus, il est essentiel que nous parvenions à une vue claire des différents besoins escomptés de notre projet. Au cours de ce chapitre, nous allons dégager les fonctionnalités attendues en définissant les différents analyseurs ainsi que la grammaire du langage.

2.1 La liste des tokens :

Comme il était demandé, nous avons essayé de traduire notre langage de programmation théorique vers une grammaire de type LL(1). La définition de cette grammaire sera démontré dans la partie suivante :

'while'	WHILE_TOKEN
'print'	ECRIRE_TOKEN
'input'	LIRE_TOKEN
'date('	DATE_TOKEN
'/*'	CO_TOKEN
*/'	CF_TOKEN
'/' ~[\r\n]*	COMMENTAIRE_LIGNE_TOKEN
'/*' .* ? '*/'	COMMENTAIRE_TOKEN
EOF	EOF_TOKEN

'true'	TRUE_TOKEN
'false'	FALSE_TOKEN
'satoshis'	SATOSHIS_TOKEN
'sats'	SATS_TOKEN
'finney'	FINNEY_TOKEN
'bits'	BITS_TOKEN
'bitcoin'	BITCOIN_TOKEN
'seconds'	SECONDS_TOKEN
'minutes'	MINUTES_TOKEN
'hours'	HOURS_TOKEN
'days'	DAYS_TOKEN
'weeks'	WEEKS_TOKEN
[-] ?[0-9]+ ([eE] [0-9]+)	NOMBRE_LITTERAL_TOKEN
'bytes'	BYTES_TOKEN
""	GUILLEMET_TOKEN
"\"	sGUILLEMET_SIMPLE_TOKEN
^[^']**\$	_VALEUR_TOKEN
'0' [xX] [0-9A-Fa-f]*	HEX_LITTERAL_TOKEN
'tx.age'	TX_AGE_TOKEN
'tx.time'	TX_TIME_TOKEN
'tx.version'	TX_VERSION_TOKEN
'tx.hashPrevouts'	TX_HASHPREVOUTS_TOKEN
'tx.hashSequence'	TX_HASHSEQUENCE_TOKEN
'tx.outpoint'	TX_OUTPOINT_TOKEN
'tx.bytecode'	TX_BYTECODE_TOKEN
'tx.value'	TX_VALUE_TOKEN
'tx.sequence'	TX_SEQUENCE_TOKEN
'tx.hashOutputs'	TX_HASHOUTPUTS_TOKEN
'tx.locktime'	TX_LOCKTIME_TOKEN
'tx.hashtype'	TX_HASHTYPE_TOKEN
'tx.preimage'	TX_PREIMAGE_TOKEN
[a-zA-Z] [a-zA-Z0-9_]*	IDENTIFIANT_TOKEN
[\t\r\n\u000C]	ESPACE_TOKEN
'pragma'	PRAGMA_TOKEN
';	POINT_VIRGULE_TOKEN
'.'	POINT_TOKEN
'cashscript'	CASHSCRIPT_TOKEN
'^'	OPERATEUR_BINAIRE_XOR_TOKEN
'~'	OPERATEUR_BINAIRE_NON_TOKEN

'>='	OPERATEUR_SUPEG_TOKEN
'>'	OPERATEUR_SUP_TOKEN
'<'	OPERATEUR_INF_TOKEN
'<='	OPERATEUR_INFEG_TOKEN
'='	OPERATEUR_EG_TOKEN
'contract'	CONTRAT_TOKEN
'{'	ACCOLADE_O_TOKEN
'}'	ACCOLADE_F_TOKEN
'function'	FONCTION_TOKEN
'('	PARENTHESE_O_TOKEN
','	VIRGULE_TOKEN
')'	PARENTHESE_F_TOKEN
'require'	REQUIRE_TOKEN
'require_time'	REQUIRE_TIME_TOKEN
'if'	SI_TOKEN
'else'	SINON_TOKEN
'new'	NOUVEAU_TOKEN
'['	CROCHET_O_TOKEN
']'	CROCHET_F_TOKEN
'reverse()'	INVERSER_TOKEN
'length'	TAILLE_TOKEN
'!'	OPERATEUR_NON_TOKEN
'_'	OPERATEUR_MOINS_TOKEN
'split'	SPLIT_TOKEN
'/'	OPERATEUR_DIVISER_TOKEN
'%'	OPERATEUR_MODULO_TOKEN
'+'	OPERATEUR_PLUS_TOKEN
'*'	OPERATEUR_FOIS_TOKEN
'=='	OPERATEUR_EGAL_TOKEN
'!'	OPERATEUR_DIFFERENT_TOKEN
'&'	OPERATEUR_BINAIRE_ET_TOKEN
' '	OPERATEUR_BINAIRE_OU_TOKEN
'&&'	OPERATEUR_ET_TOKEN
' '	OPERATEUR_OU_TOKEN
'int'	TYPE_ENTIER_TOKEN
'bool'	TYPE_BOOLEAN_TOKEN
'string'	TYPE_STRING_TOKEN
'pubkey'	TYPE_CLE_PUBLIQUE_TOKEN
'sig'	TYPE_SIGNATURE_TOKEN
'datasig'	TYPE_SIGNATURE_DONNEE_TOKEN

2.2 Les règles de production

Vous trouvez ci-dessous quelques règles de productions utilisées (vous trouvez la totalité dans le code source)

- **PROGRAMME** : DIRECTIVE_PRAGMA* DEFINITION_CONTRAT EOF_TOKEN;
- **DIRECTIVE_PRAGMA** : PRAGMA_TOKEN CASHSCRIPT_TOKEN VALEUR_PRAGMA POINT_VIRGULE_TOKEN;
- **VALEUR_PRAGMA** : CONTRAINTE_VERSION [CONTRAINTE_VERSION | eps];
- **CONTRAINTE_VERSION** : [OPERATEUR_VERSION | eps] VERSION_LITTERALE_TOKEN;
- **OPERATEUR_VERSION** : OPERATEUR_BINAIRE_XOR_TOKEN | OPERATEUR_BINAIRE_NON_TOKEN | OPERATEUR_SUPEG_TOKEN | OPERATEUR_SUP_TOKEN | OPERATEUR_INF_TOKEN | OPERATEUR_INFEG_TOKEN | OPERATEUR_EG_TOKEN;
- **DEFINITION_CONTRAT** : CONTRAT_TOKEN IDENTIFIANT_TOKEN LISTE_PARAMETRE ACCOLADE_O_TOKEN {DEFINITION_FONCTION} ACCOLADE_F_TOKEN;
- **DEFINITION_FONCTION** : FONCTION_TOKEN IDENTIFIANT_TOKEN LISTE_PARAMETRE ACCOLADE_O_TOKEN {DECLARATION} ACCOLADE_F_TOKEN;
- **LISTE_PARAMETRE** : PARENTHESE_O_TOKEN [(PARAMETRE (VIRGULE_TOKEN PARAMETRE)* [VIRGULE_TOKEN | eps])| eps] PARENTHESE_F_TOKEN;
- **PARAMETRE** : NOM_TYPE IDENTIFIANT_TOKEN;
- **BLOC** : ACCOLADE_O_TOKEN DECLARATION* ACCOLADE_F_TOKEN | DECLARATION;
- **DECLARATION** : DEFINITION_VARIABLE| AFFECTATION| TEMPS_DECLARATION | requireStatement| SI | WHILE | ECRIRE | LIRE;
- **DEFINITION_VARIABLE** : NOM_TYPE IDENTIFIANT_TOKEN OPERATEUR_EG_TOKEN EXPRESSION POINT_VIRGULE_TOKEN;
- **AFFECTATION** : IDENTIFIANT_TOKEN OPERATEUR_EG_TOKEN EXPRESSION POINT_VIRGULE_TOKEN;
- **TEMPS_DECLARATION** : REQUIRE_TIME_TOKEN PARENTHESE_O_TOKEN TX_VAR_TOKEN OPERATEUR_SUPEG_TOKEN EXPRESSION PARENTHESE_F_TOKEN POINT_VIRGULE_TOKEN;
- **requireStatement** : REQUIRE_TOKEN PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN POINT_VIRGULE_TOKEN;
- **SI** : SI_TOKEN PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN BLOC [(SINON_TOKEN BLOC)| eps];
- **LITTERAL** : BOOLEAN_LITTERAL_TOKEN | NOMBRE_LITTERAL | STRING_LITTERAL_TOKEN | DATE_LITTERAL | HEX_LITTERAL_TOKEN;

- **LISTE_EXPRESSIONS** : PARENTHESE_O_TOKEN
[(EXPRESSION (VIRGULE_TOKEN EXPRESSION)* [VIRGULE_TOKEN | eps]) | eps] PARENTHESE_F_TOKEN ;
- **EXPRESSION** : PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN
| NOM_TYPE PARENTHESE_O_TOKEN EXPRESSION [(VIRGULE_TOKEN EXPRESSION)
| eps] [VIRGULE_TOKEN | eps] PARENTHESE_F_TOKEN
| NOUVEAU_TOKEN IDENTIFIANT_TOKEN LISTE_EXPRESSIONS
| IDENTIFIANT_TOKEN [|LISTE_EXPRESSIONS | INVERSER_TOKEN | TAILLE_TOKEN
| (CROCHET_O_TOKEN NOMBRE_LITTERAL_TOKEN CROCHET_F_TOKEN)
| (SPLIT_TOKEN PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN)
| DEUXIEME_EXPRESSION_BINAIRE] | eps]
| LITTERAL [| INVERSER_TOKEN | TAILLE_TOKEN | (CROCHET_O_TOKEN
NOMBRE_LITTERAL_TOKEN CROCHET_F_TOKEN)
| (SPLIT_TOKEN PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN)
| DEUXIEME_EXPRESSION_BINAIRE] | eps]
| (OPERATEUR_NON_TOKEN | OPERATEUR_MOINS_TOKEN) EXPRESSION
| CROCHET_O_TOKEN [(EXPRESSION (VIRGULE_TOKEN EXPRESSION)* [VIRGULE_TOKEN | eps])
| eps] CROCHET_F_TOKEN
| CHAMP_AVANT_IMAGE_TOKEN -NOMBRE_LITTERAL : NOMBRE_LITTERAL_TOKEN [UNITE_TOKEN]
- **NOM_TYPE** : TYPE_ENTIER_TOKEN | TYPE_BOOLEAN_TOKEN
| TYPE_STRING_TOKEN | TYPE_CLE_PUBLIQUE_TOKEN
| TYPE_SIGNATURE_TOKEN | TYPE_SIGNATURE_DONNEE_TOKEN | BYTES ;
- **BYTES** : BYTES_TOKEN [|[1-9] [0-9]*] | eps] ;
- **DATE_LITTERAL** : DATE_TOKEN STRING_LITTERAL_TOKEN PARENTHESE_F_TOKEN ;
- **WHILE** : WHILE_TOKEN PARENTHESE_O_TOKEN EXPRESSION PARENTHESE_F_TOKEN BLOC ;
- **ECRIRE** : ECRIRE_TOKEN PARENTHESE_O_TOKEN EXPRESSION
PARENTHESE_F_TOKEN POINT_VIRGULE_TOKEN ;
- **LIRE** : LIRE_TOKEN IDENTIFIANT_TOKEN PARENTHESE_F_TOKEN POINT_VIRGULE_TOKEN ;
- **COMMENT** : CO_TOKEN .* CF_TOKEN ;
- **LINE_COMMENT** : COMMENTAIRE_LIGNE_TOKEN ~[\\r\\n]* ;
- **VERSION_LITTERALE_TOKEN** : NOMBRE_LITTERAL_TOKEN POINT_TOKEN
NOMBRE_LITTERAL_TOKEN POINT_TOKEN NOMBRE_LITTERAL_TOKEN ;
- **BOOLEAN_LITTERAL_TOKEN** : TRUE_TOKEN | FALSE_TOKEN ;
- **UNITE_TOKEN** : SATOSHIS_TOKEN | SATS_TOKEN | FINNEY_TOKEN
| BITS_TOKEN | BITCOIN_TOKEN | SECONDS_TOKEN | MINUTES_TOKEN
| HOURS_TOKEN | DAYS_TOKEN | WEEKS_TOKEN ;
- **STRING_LITTERAL_TOKEN** : [GUILLEMET_SIMPLE_TOKEN STRING_VALEUR_TOKEN
GUILLEMET_SIMPLE_TOKEN]
| [GUILLEMET_TOKEN STRING_VALEUR_TOKEN GUILLEMET_TOKEN] ;
- **TX_VAR_TOKEN** : TX_AGE_TOKEN | TX_TIME_TOKEN ;
- **CHAMP_AVANT_IMAGE_TOKEN** : TX_VERSION_TOKEN | TX_HASHPREVOUTS_TOKEN
| TX_HASHSEQUENCE_TOKEN | TX_OUTPOINT_TOKEN
| TX_BYTECODE_TOKEN | TX_VALUE_TOKEN | TX_SEQUENCE_TOKEN | TX_HASHOUTPUTS_TOKEN
| TX_LOCKTIME_TOKEN | TX_HASHTYPE_TOKEN | TX_PREIMAGE_TOKEN

2.3 Les règles sémantiques

Concernant les règles sémantiques, on a implémenté les règles suivantes dans notre mini-compilateur :

- PAS DE DOUBLE DÉCLARATIONS
- PAS D'OMBRE VARIABLES
- TOUTE VARIABLE DOIT ÊTRE Déclarée AVANT D'ÊTRE UTILISÉE
- LES OPÉRATEURS UTILISES ONT DES VARIABLES DE TYPE COMPATIBLE `.split`, `.reverse()`, `.length : string`
- LES COMMENTAIRES NE DOIVENT PAS ÊTRE INTERPRÉTÉS

2.4 Les premiers et les suivants

Vous trouvez ci-dessous la liste des premiers et des suivants des NT

	First	Follow
PROGRAMME	{ PRAGMA_TOKEN }	{ \$ }
DIRECTIVE_PRAGMA	{ PRAGMA_TOKEN }	{ CONTRAT_TOKEN }
VALEUR_PRAGMA	{ PRAGMA_TOKEN }	{ CONTRAT_TOKEN }
CONTRAINTE_VERSION	{ NOMBRE_LITTERAL_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_NON_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_EG_TOKEN }	{ NOMBRE_LITTERAL_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_NON_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_EG_TOKEN, POINT_VIRGULE_TOKEN }
OPERATEUR_VERSION	{ OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_NON_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_EG_TOKEN }	{ NOMBRE_LITTERAL_TOKEN }
DEFINITION_FONCTION	FONCTION_TOKEN	ACCOLADE_F_TOKEN
DEFINITION_CONTRAT	{ CONTRAT_TOKEN }	{ EOF_TOKEN }
LISTE_PARAMETRE	{ PARENTHESE_O_TOKEN }	{ ACCOLADE_O_TOKEN }
PARAMETRE	{ TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN, TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, BYTES_TOKEN }	VIRGULE_TOKEN
BLOC	{ ACCOLADE_O_TOKEN }	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
DECLARATION	{ TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN, TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, , BYTES_TOKEN, IDENTIFIANT_TOKEN, REQUIRE_TIME_TOKEN, REQUIRE_TOKEN, SI_TOKEN , WHILE_TOKEN, Ecrire_TOKEN , Lire_TOKEN }	{ SINON_TOKEN, ACCOLADE_F_TOKEN }

	First	Follow
DECLARATION	{ TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN, TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, BYTES_TOKEN, IDENTIFIANT_TOKEN, REQUIRE_TIME_TOKEN, REQUIRE_TOKEN, SI_TOKEN , WHILE_TOKEN, ECRIRE_TOKEN , LIRE_TOKEN}	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
DEFINITION_VARIABLE	{ TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN, TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, BYTES_TOKEN}	SINON_TOKEN, ACCOLADE_F_TOKEN }
AFFECTATION	{ IDENTIFIANT_TOKEN}	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
TEMPS_DECLARATION	{ REQUIRE_TIME_TOKEN}	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
requireStatement	{ REQUIRE_TOKEN}	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
SI	{ SI_TOKEN}	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
FONCTION	{ IDENTIFIANT_TOKEN}	{ OPERATEUR_EG_TOKEN, PARENTHESE_F_TOKEN, VIRGULE_TOKEN, PARENTHESE_O_TOKEN, VIRGULE_TOKEN, PARENTHESE_F_TOKEN, POINT_VIRGULE_TOKEN }
LISTE_EXPRESSIONS	{ PARENTHESE_O_TOKEN}	{ VIRGULE_TOKEN, PARENTHESE_F_TOKEN, POINT_VIRGULE_TOKEN}
EXPRESSION	{ PARENTHESE_O_TOKEN, TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN, TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, BYTES, NOUVEAU_TOKEN, IDENTIFIANT_TOKEN, TRUE_TOKEN, FALSE_TOKEN, NOMBRE_LITERAL_TOKEN, GUILLEMET_SIMPLE_TOKEN, GUILLEMET_TOKEN, DATE_TOKEN, HEX_LITERAL_TOKEN, OPERATEUR_NON_TOKEN, OPERATEUR_MOINS_TOKEN, CROCHET_O_TOKEN, TX_VERSION_TOKEN, TX_HASHPREVOUTS_TOKEN, TX_HASHSEQUENCE_TOKEN, TX_OUTPOINT_TOKEN, TX_BYTECODE_TOKEN, TX_VALUE_TOKEN, TX_SEQUENCE_TOKEN, TX_HASHOUTPUTS_TOKEN, TX_LOCKTIME_TOKEN, TX_HASHTYPE_TOKEN, TX_PREIMAGE_TOKEN}	{ VIRGULE_TOKEN, PARENTHESE_F_TOKEN, POINT_VIRGULE_TOKEN}

	First	Follow
DEUXIEME_EXPRESSION_BINAIRE	{ OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN }	{ VIRGULE_TOKEN, PARENTHESE_F_TOKEN, POINT_VIRGULE_TOKEN }
LITTERAL	{ TRUE_TOKEN, FALSE_TOKEN, NOMBRE_LITTERAL_TOKEN, DATE_TOKEN, HEX_LITTERAL_TOKEN }	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN }
NOMBRE_LITTERAL	{ NOMBRE_LITTERAL_TOKEN }	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN }
NOM_TYPE	{ TYPE_ENTIER_TOKEN, TYPE_BOOLEAN_TOKEN , TYPE_STRING_TOKEN, TYPE_CLE_PUBLIQUE_TOKEN, TYPE_SIGNATURE_TOKEN, TYPE_SIGNATURE_DONNEE_TOKEN, BYTES_TOKEN }	{ IDENTIFIANT_TOKEN, PARENTHESE_O_TOKEN }
DATE_LITTERAL	DATE_TOKEN	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN , OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN , OPERATEUR_OU_TOKEN }
WHILE	{ WHILE_TOKEN }	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
ECRIRE	{ ECRIRE_TOKEN }	{ SINON_TOKEN, ACCOLADE_F_TOKEN }
LIRE	{ LIRE_TOKEN }	{ SINON_TOKEN, ACCOLADE_F_TOKEN }

	First	Follow
VERSION_LITTERALE_TOKEN	{ NOMBRE_LITTERAL_TOKEN }	{ NOMBRE_LITTERAL_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_NON_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_EG_TOKEN, POINT_VIRGULE_TOKEN }
BOOLEAN_LITTERAL_TOKEN	{ TRUE_TOKEN, FALSE_TOKEN }	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN }
UNITE_TOKEN	{ SATOSHIS_TOKEN, SATS_TOKEN, FINNEY_TOKEN, BITS_TOKEN, BITCOIN_TOKEN, SECONDS_TOKEN, MINUTES_TOKEN, HOURS_TOKEN, DAYS_TOKEN, WEEKS_TOKEN }	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN }
STRING_LITTERAL_TOKEN	{ GUILLEMET_SIMPLE_TOKEN, GUILLEMET_TOKEN }	{ INVERSER_TOKEN, TAILLE_TOKEN, CROCHET_O_TOKEN, SPLIT_TOKEN, OPERATEUR_FOIS_TOKEN, OPERATEUR_DIVISER_TOKEN, OPERATEUR_MODULO_TOKEN, OPERATEUR_PLUS_TOKEN, OPERATEUR_MOINS_TOKEN, OPERATEUR_INF_TOKEN, OPERATEUR_INFEG_TOKEN, OPERATEUR_SUP_TOKEN, OPERATEUR_SUPEG_TOKEN, OPERATEUR_EGAL_TOKEN, OPERATEUR_DIFFERENT_TOKEN, OPERATEUR_BINAIRE_ET_TOKEN, OPERATEUR_BINAIRE_XOR_TOKEN, OPERATEUR_BINAIRE_OU_TOKEN, OPERATEUR_ET_TOKEN, OPERATEUR_OU_TOKEN, PARENTHESE_F_TOKEN }
TX_VAR_TOKEN	{ TX_AGE_TOKEN, TX_TIME_TOKEN }	{ OPERATEUR_SUPEG_TOKEN }
CHAMP_AVANT_IMAGE_TOKEN	{ TX_VERSION_TOKEN, TX_HASHPREVOUTS_TOKEN, TX_HASHSEQUENCE_TOKEN, TX_OUTPOINT_TOKEN, TX_BYTECODE_TOKEN, TX_VALUE_TOKEN, TX_SEQUENCE_TOKEN, TX_HASHOUTPUTS_TOKEN, TX_LOCKTIME_TOKEN, TX_HASHTYPE_TOKEN, TX_PREIMAGE_TOKEN }	{ VIRGULE_TOKEN, PARENTHESE_F_TOKEN, POINT_VIRGULE_TOKEN }

2.5 Analyse lexicale

L'analyse lexicale fait partie de la première phase de la compilation, elle est la conversion d'une chaîne de caractères (un texte) en une liste de symboles (tokens en anglais). Un programme réalisant une analyse lexicale est appelé un analyseur lexical, tokenizer ou lexer, notre analyseur lexical est codé en langage C.

2.5.1 Structure

- `analy_lex.h` : il contient la déclaration des fonctions principales utilisées lors de l'analyse lexicale :

```
void lex_get_next_char();
void next_sym();
void print_token(LEX_CODE curr_sym);
void skip_white_spaces();
void skip_comment();
void read_number();
void read_word();
LEX_CODE keyword_code(char* word);
void read_string();
void analy_lex(FILE *fp);
```

- `analy_lex.c` :il contient l'implémentation des fonctions ci-dessus

- `data.c`

- `data.h`

- `main.c`

2.6 Analyse syntaxique

L'analyse syntaxique est la deuxième étape après celle lexicale, après l'obtention des tokens générés par l'analyseur lexical, cette étape consiste à analyser et vérifier la syntaxe du code selon la grammaire citée en haut. Cet analyseur donne à la sortie une structure qui est une hiérarchie représentable par un arbre syntaxique.

2.6.1 Structure

- `analy_syn.h` :l contient la déclaration des fonctions principales utilisées lors de l'analyse syntaxique :


```

void PROGRAMME();
void DIRECTIVE_PRAGMA();
void VALEUR_PRAGMA();
void CONTRAINTE_VERSION();
void OPERATEUR_VERSION();
void DEFINITION_CONTRAT();
void DEFINITION_FONCTION();
void LISTE_PARAMETRE();
void PARAMETRE();
void BLOC();
void DECLARATION();
void DEFINITION_VARIABLE();
void AFFECTATION();
void TEMPS_DECLARATION();
void requireStatement();
void SI();
void FONCTION();
void LISTE_EXPRESSIONS();
void EXPRESSION();
void DEUXIEME_EXPRESSION_BINAIRE();
void LITTERAL();
void NOMBRE_LITTERAL();
void NOM_TYPE();
void BYTES();
void DATE_LITTERAL();
void WHILE();
void ECRIRE();
void LIRE();
void COMMENT();
void LINE_COMMENT();
void VERSION_LITTERALE_TOKEN();
void TX_VAR_TOKEN();
void BOOLEAN_LITTERAL_TOKEN();
void STRING_LITTERAL_TOKEN();
void UNITE_TOKEN();
void CHAMP_AVANT_IMAGE_TOKEN();
void parser();

```

□ `analy_syn.c` : il contient l'implémentation des fonctions ci-dessus

2.7 Analyse sémantique

Après avoir implémenté les analyseurs lexicaux et syntaxiques on a passé à faire l'analyse sémantique qui consiste à vérifier les règles sémantiques définies auparavant.

- `semantics.h` : contient la déclaration des fonctions principales utilisées lors de l'analyse syntaxique

```
bool identifierExists(LANGUAGE_KEYWORD identifier);  
bool stringOperationsAreValid(LANGUAGE_KEYWORD firstOperand);  
void printfDoubleDeclarationError(LANGUAGE_KEYWORD identifier, int line);
```

- `semantics.c` : il contient l'implémentation des fonctions ci-dessus

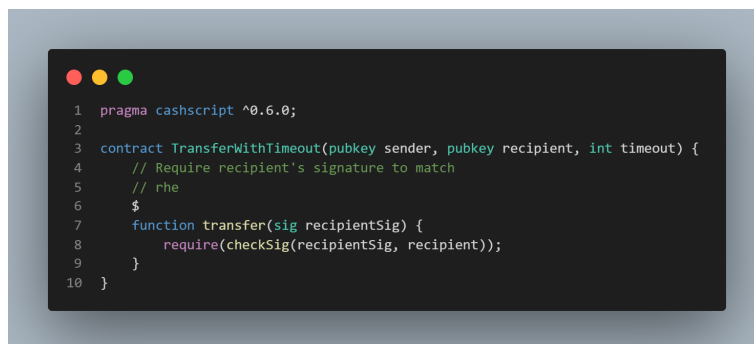
Chapitre 3

Test de compilateur

Dans cette partie nous allons tester notre 3 analyseurs

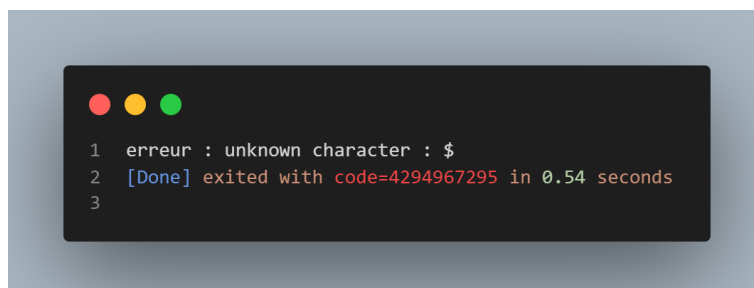
3.1 1er Test (Analyseur lexical)

L'analyseur lexical retourne une erreur dans l'exemple suivant



```
1 pragma cashscript ^0.6.0;
2
3 contract TransferWithTimeout(pubkey sender, pubkey recipient, int timeout) {
4     // Require recipient's signature to match
5     // rhe
6     $
7     function transfer(sig recipientSig) {
8         require(checkSig(recipientSig, recipient));
9     }
10 }
```

FIGURE 3.1 – Test1(Analyseur lexical)



```
1 erreur : unknown character : $
2 [Done] exited with code=4294967295 in 0.54 seconds
3
```

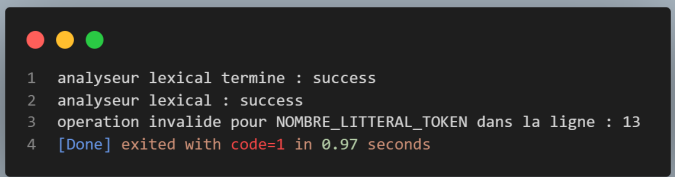
FIGURE 3.2 – exécution1 (Analyseur lexical)

Dans l'exemple suivant l'analyse lexicale est validé :



```
1 pragma cashscript ^0.6.0;
2
3 contract TransferWithTimeout(pubkey sender, pubkey recipient, int timeout) {
4     // Require recipient's signature to match
5     // rhe
6     function checkSig(sig tm1, pubkey rec){
7         int a = 0;
8     }
9     function transfer(sig recipientSig) {
10         require(checkSig(recipientSig, recipient));
11         int a = 5.length;
12     }
13 }
```

FIGURE 3.3 – correction Test1(Analyseur lexical)



```
1 analyseur lexical termine : success
2 analyseur lexical : success
3 operation invalide pour NOMBRE_LITTERAL_TOKEN dans la ligne : 13
4 [Done] exited with code=1 in 0.97 seconds
```

FIGURE 3.4 – correction exécution1 (Analyseur lexical)

3.2 2 ème Test (Analyseur syntaxique)

L'analyseur syntaxique retourne une erreur dans l'exemple suivant

```

1  pragma cashscript ^0.6.0;
2
3  contract Mecenaz(bytes recipient, bytes funder, int pledge) {
4      // Allow the receiver to claim their monthly pledge amount
5      function receive(pubkey pk, sig s) {
6          // The transaction can be signed by anyone, because the money can only
7          // be sent to the correct address
8          require(checkSig(s, pk));
9
10         // Check that the UTXO is at least 30 days old
11         require_time(tx_age >= 30 days);
12
13         // Use a hardcoded miner fee
14         int minerFee = 1000;
15
16         // Retrieve the UTXO's value and cast it to an integer
17         int intValue = int(bytes(tx_value));
18
19         // Check if the UTXO's value is higher than the pledge amount + fee
20         if (intValue <= pledge) {
21             // Create an Output that sends the remaining balance to the recipient
22             bytes out1 = new OutputP2PKH(bytes(intValue - minerFee), recipient);
23
24             // Enforce that this is the only output for the current transaction
25             string temp = hash256(out1);
26             string temp2 = tx_hashOutputs;
27             require( temp2 == temp);
28         } else {
29             // Create an Output that sends the pledge amount to the recipient
30             bytes out1 = new OutputP2PKH(bytes(pledge), recipient);
31
32             // Create an Output that sends the remainder back to the contract
33             int temp3 = intValue - pledge ;
34             int temp4 = temp3 - minerFee;
35             bytes remainder = bytes(temp3);
36             bytes out2 = new OutputP2SH(remainder, hash160(tx_bytecode));
37             string tempString = "hello";
38             string tempString2 = "hello";
39             int a = 5;
40             int b = a + 6;
41             require(tempString == tempString2);
42
43             // Enforce that these are the only outputs for the current transaction
44             string temp6 = hash256(out1 + out2);
45             string temp7 = tx_hashOutputs;
46             require( temp6 == temp7);
47         }
48         while (true){
49             int a = 5;
50             int b = a + 6;
51             require(tempString == tempString2);
52         }
53         string alpha = "";
54         input(alpha);
55         print(alpha + "22");
56     }
57
58     // Allow the funder to reclaim their remaining pledges
59     function reclaim(pubkey pk, sig s) {
60         temp6 = hash160(pk);
61         temp7 = funder;
62         require(temp6 == temp7);
63         require(checkSig(s, pk));
64     }
65 }

```

FIGURE 3.5 – Test2(Analyseur syntaxique)

```

1  PS C:\Users\oubay\OneDrive\Bureau\projects\cashScript-Compiler\cmake-build-debug> .\main.exe
2  analyseur lexical termine : success
3  Error in line 20: expected ')'

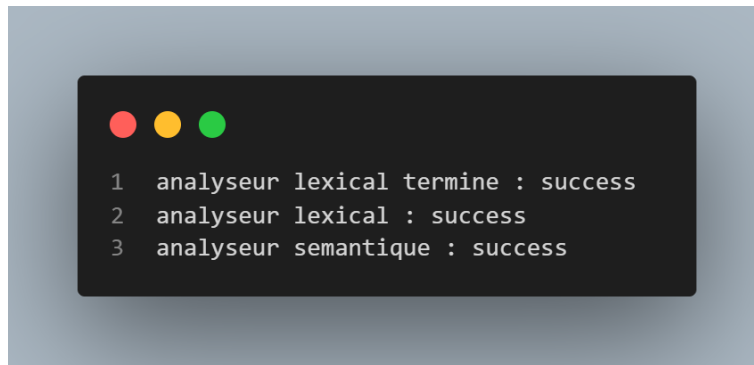
```

FIGURE 3.6 – exécution2 (Analyseur syntaxique)

Dans l'exemple suivant l'analyse lexicale est validé :

```
1  pragma cashscript ^0.6.0;
2
3  contract Mecenaz(bytes recipient, bytes funder, int pledge) {
4      function checkSig(sig sign, pubkey pubk){
5          print("nothing");
6      }
7      function OutputP2PKH(bytes by, bytes reci){
8          print("random OutputP2PKH");
9      }
10     function OutputP2SH(bytes by, bytes reci){
11         print("random OutputP2SH");
12     }
13     function hash256(bytes byt){
14         print("random hash256");
15     }
16     function hash160(bytes byt){
17         print("random hash160");
18     }
19     // Allow the receiver to claim their monthly pledge amount
20     function receive(pubkey pk, sig s) {
21         // The transaction can be signed by anyone, because the money can only
22         // be sent to the correct address
23         require(checkSig(s, pk));
24
25         // Check that the UTXO is at least 30 days old
26         require_time(tx_age >= 30 days);
27
28         // Use a hardcoded miner fee
29         int minerFee = 1000;
30
31         // Retrieve the UTXO's value and cast it to an integer
32         int intValue = int(bytes(tx_value));
33
34         // Check if the UTXO's value is higher than the pledge amount + fee
35         if (intValue <= pledge ) {
36             // Create an Output that sends the remaining balance to the recipient
37             bytes out1 = new OutputP2PKH(bytes(intValue - minerFee), recipient);
38
39             // Enforce that this is the only output for the current transaction
40             string temp = hash256(out1);
41             string temp2 = tx_hashOutputs;
42             require( temp2 == temp);
43         } else {
44             // Create an Output that sends the pledge amount to the recipient
45             bytes out1 = new OutputP2PKH(bytes(pledge), recipient);
46
47             // Create an Output that sends the remainder back to the contract
48             int temp3 = intValue - pledge ;
49             int temp4 = temp3 - minerFee;
50             bytes remainder = bytes(temp3);
51             bytes out2 = new OutputP2SH(remainder, hash160(tx_bytocode));
52             string tempString = "hello";
53             string tempString2 = "hello";
54             int a = 5;
55             int b = a + 6;
56             require(tempString == tempString2);
57
58             // Enforce that these are the only outputs for the current transaction
59             string temp6 = hash256(out1 + out2);
60             string temp7 = tx_hashOutputs;
61             require( temp6 == temp7);
62         }
63         while (true){
64             int a = 5;
65             int b = a + 6;
66             require(tempString == tempString2);
67         }
68         string alpha = "";
69         input(alpha);
70         print(alpha + "22");
71     }
72
73     // Allow the funder to reclaim their remaining pledges
74     function reclaim(pubkey pk, sig s) {
75         temp6 = hash160(pk);
76         temp7 = funder;
77         require(temp6 == temp7);
78         require(checkSig(s, pk));
79     }
80 }
```

FIGURE 3.7 – correction Test2(Analyseur syntaxique)



```
1 analyseur lexical termine : success
2 analyseur lexical : success
3 analyseur semantique : success
```

FIGURE 3.8 – correction execution2 (Analyseur syntaxique)

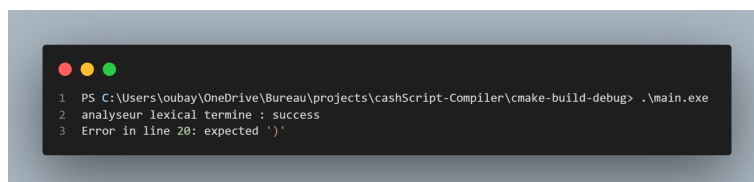
3.3 3 ème Test (Analyseur sémantique)

L'analyseur sémantique retourne une erreur dans l'exemple suivant



```
1 pragma cashscript ^0.6.0;
2
3 contract TransferWithTimeout(pubkey sender, pubkey recipient, int timeout) {
4     // Require recipient's signature to match
5     // rhe
6     function checkSig(sig tm1, pubkey rec){
7         int a = 0;
8     }
9     function transfer(sig recipientSig) {
10         require(checkSig(recipientSig, recipient));
11         int a = 5.length;
12     }
13 }
```

FIGURE 3.9 – Test 3(Analyseur sémantique)

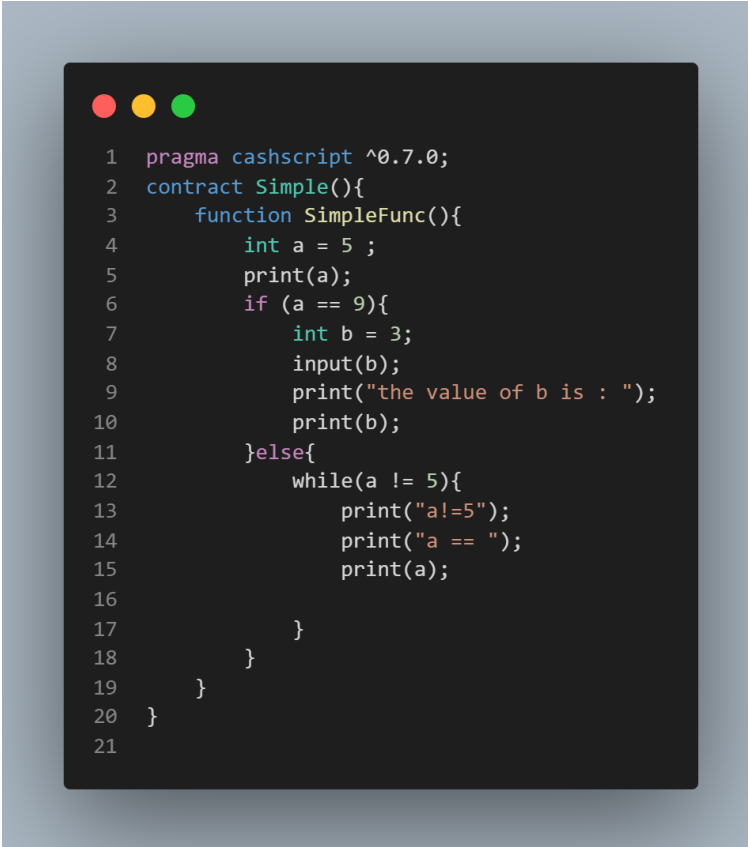


```
1 PS C:\Users\youbay\OneDrive\Bureau\projects\cashScript-Compiler\cmake-build-debug> .\main.exe
2 analyseur lexical termine : success
3 Error in line 20: expected ')'
```

FIGURE 3.10 – execution 3 (Analyseur sémantique)

3.4 4 ème test de compilateur

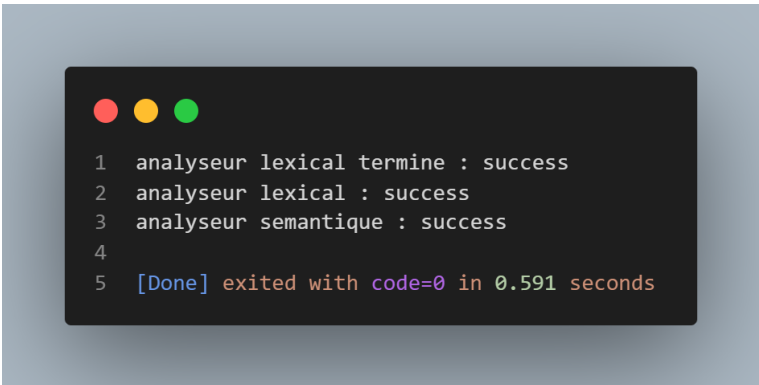
3.4.1 Programme



```
1  pragma cashscript ^0.7.0;
2  contract Simple(){
3      function SimpleFunc(){
4          int a = 5 ;
5          print(a);
6          if (a == 9){
7              int b = 3;
8              input(b);
9              print("the value of b is : ");
10             print(b);
11         }else{
12             while(a != 5){
13                 print("a!=5");
14                 print("a == ");
15                 print(a);
16             }
17         }
18     }
19 }
20 }
21
```

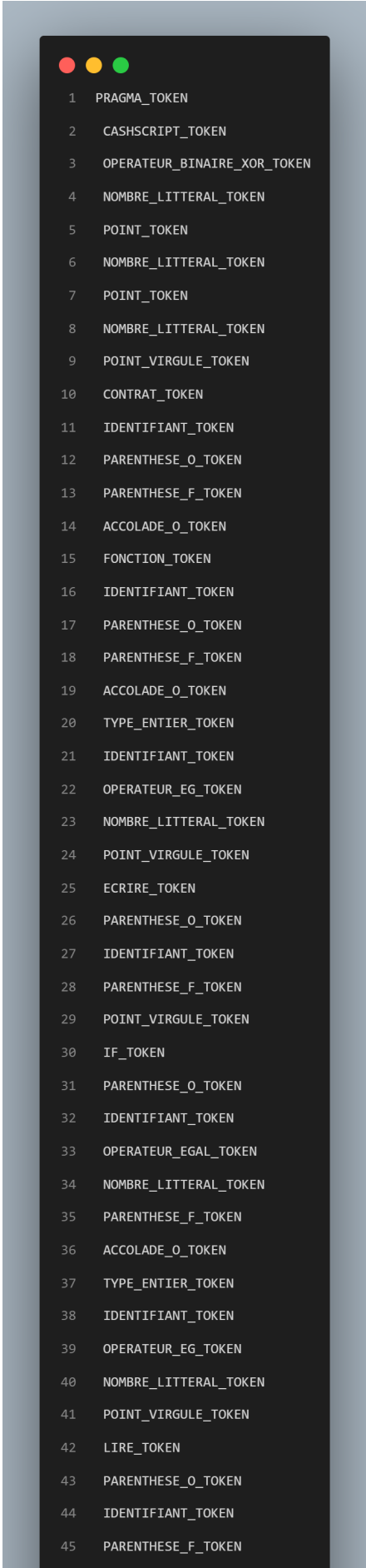
FIGURE 3.11 – Test4

3.4.2 Exécution



```
1  analyseur lexical termine : success
2  analyseur lexical : success
3  analyseur semantique : success
4
5  [Done] exited with code=0 in 0.591 seconds
```

FIGURE 3.12 – exécution test4



```
1 PRAGMA_TOKEN
2 CASHSCRIPT_TOKEN
3 OPERATEUR_BINAIRE_XOR_TOKEN
4 NOMBRE_LITTERAL_TOKEN
5 POINT_TOKEN
6 NOMBRE_LITTERAL_TOKEN
7 POINT_TOKEN
8 NOMBRE_LITTERAL_TOKEN
9 POINT_VIRGULE_TOKEN
10 CONTRAT_TOKEN
11 IDENTIFIANT_TOKEN
12 PARENTHESE_O_TOKEN
13 PARENTHESE_F_TOKEN
14 ACCOLADE_O_TOKEN
15 FONCTION_TOKEN
16 IDENTIFIANT_TOKEN
17 PARENTHESE_O_TOKEN
18 PARENTHESE_F_TOKEN
19 ACCOLADE_O_TOKEN
20 TYPE_ENTIER_TOKEN
21 IDENTIFIANT_TOKEN
22 OPERATEUR_EG_TOKEN
23 NOMBRE_LITTERAL_TOKEN
24 POINT_VIRGULE_TOKEN
25 ECRIRE_TOKEN
26 PARENTHESE_O_TOKEN
27 IDENTIFIANT_TOKEN
28 PARENTHESE_F_TOKEN
29 POINT_VIRGULE_TOKEN
30 IF_TOKEN
31 PARENTHESE_O_TOKEN
32 IDENTIFIANT_TOKEN
33 OPERATEUR_EGAL_TOKEN
34 NOMBRE_LITTERAL_TOKEN
35 PARENTHESE_F_TOKEN
36 ACCOLADE_O_TOKEN
37 TYPE_ENTIER_TOKEN
38 IDENTIFIANT_TOKEN
39 OPERATEUR_EG_TOKEN
40 NOMBRE_LITTERAL_TOKEN
41 POINT_VIRGULE_TOKEN
42 LIRE_TOKEN
43 PARENTHESE_O_TOKEN
44 IDENTIFIANT_TOKEN
45 PARENTHESE_F_TOKEN
```

FIGURE 3.13 – exécution analyseur lexical

```
46 POINT_VIRGULE_TOKEN
47 ECRIRE_TOKEN
48 PARENTHESE_O_TOKEN
49 GUILLEMET_TOKEN
50 STRING_VALEUR_TOKEN
51 GUILLEMET_TOKEN
52 PARENTHESE_F_TOKEN
53 POINT_VIRGULE_TOKEN
54 ECRIRE_TOKEN
55 PARENTHESE_O_TOKEN
56 IDENTIFIANT_TOKEN
57 PARENTHESE_F_TOKEN
58 POINT_VIRGULE_TOKEN
59 ACCOLADE_F_TOKEN
60 SINON
61 ACCOLADE_O_TOKEN
62 WHILE_TOKEN
63 PARENTHESE_O_TOKEN
64 IDENTIFIANT_TOKEN
65 OPERATEUR_DIFFERENT_TOKEN
66 NOMBRE_LITTERAL_TOKEN
67 PARENTHESE_F_TOKEN
68 ACCOLADE_O_TOKEN
69 ECRIRE_TOKEN
70 PARENTHESE_O_TOKEN
71 GUILLEMET_TOKEN
72 STRING_VALEUR_TOKEN
73 GUILLEMET_TOKEN
74 PARENTHESE_F_TOKEN
75 POINT_VIRGULE_TOKEN
76 ECRIRE_TOKEN
77 PARENTHESE_O_TOKEN
78 GUILLEMET_TOKEN
79 STRING_VALEUR_TOKEN
80 GUILLEMET_TOKEN
81 PARENTHESE_F_TOKEN
82 POINT_VIRGULE_TOKEN
83 ECRIRE_TOKEN
84 PARENTHESE_O_TOKEN
85 IDENTIFIANT_TOKEN
86 PARENTHESE_F_TOKEN
87 POINT_VIRGULE_TOKEN
88 ACCOLADE_F_TOKEN
89 ACCOLADE_F_TOKEN
90 ACCOLADE_F_TOKEN
91 ACCOLADE_F_TOKEN
92 EOF_TOKEN
93
```

FIGURE 3.14 – exécution analyseur lexical(suite)

Conclusion

Dans ce rapport, nous avons exposé les étapes de réalisation du compilateur du langage cashScript , on a balayé tous les étapes d'un compilteur LL(1) de l'analyse analytique en passant par l'analyse syntaxique et en fin l'analyse sémantique .

Ce projet nous a permis de maîtriser les différents concepts qu'on a étudié tout au long du module "Compilation".

Ainsi, c'était une véritable expérience de travail en collaboration, qui nous a permis de bien gérer la répartition des tâches et de renforcer l'esprit de partage de connaissances ainsi que la synchronisation de notre travail.