STMicroelectronics

École Nationale d'Ingénieur de Sousse

Portage de la machine virtuelle Lua sur processeur ST40

Mohamed Abdessalem Hajri

Mémoire de projet de fin d'étude préparé pour l'obtention

du

Diplome National d'Ingénieur en Eléctronique Industrielle

Option : Conception des systèmes électroniques

Ecole Nationale D'Ingénieurs de Sousse

2011

Département Génie Industriel

Ecole Nationale D'Ingénieurs de Sousse

Abstract

Portage de la machine virtuelle Lua sur processeur ST40

Mohamed Abdessalem Hajri

REMERCIEMENTS

Je tiens à remercier les membres de mon Jury pour l'honneur qu'ils m'ont donnés de juger mon travail. Qu'ils trouvent ici ma reconnaissance et mon respect. Le mérite revient également à Mr. Ilyes Gouta , Ingénieur en STMicroelectronics de Tunis, pour m'avoir accueilli à ST Tunis et m'avoir fourni tous les éléments nécessaires pour la réalisation de ce projet.

Je remercie, également, mon encadrant de l'ENISo, Mr. Saber Jamalli, qui a bien voulu assurer la direction de ce travail. Je lui suis infiniment redevable pour sa patience, son assistance et ses précieuses recommandations.

Je suis redevable à L'administration de l'ENISo qui mon fournit tout le soutient nécessaire pour élaborer ce projet.

J'adresse enfin, mes vifs remerciements, à tous mes professeurs de l'ENISo qui ont contribué énormément à ma formation.

TABLE DES MATIERES

TABLE DES FIGURES

LISTE DES TABLES

ÉTAT DE L'ART

.

Chapitre 1

INTRODUCTION GÉNÉRALE

1.1 Introduction

De nos jours le monde est devenu de plus en plus dépendant de la technologie des systèmes embarqués à la fois dans les applications industriel, commercial et même domestique. L'innovation dans la technologie micro-électronique et l'invention des nouveaux concepts, comme les micro-systèmes et les circuits mixtes, permettent une évolution remarquable dans le domaine de l'électronique embarqué. Ces évolutions permettent une amélioration spectaculaire dans plusieurs domaines telle que le domaine d'accélération graphique.

Les systèmes embarqués sont des systèmes dédiés, qui ont pour rôle d'exécuté une application dédié de faible taille, composé d'une partie matériel et d'une partie logiciel qui ont pour but d'effectuer une ou plusieurs taches dédiés. La partie matériel est généralement composée d'une unité de traitement (processeur à faible consommation, micro-contrôleur, DSP, etc.), des unités de stockage (mémoire : RAM, Registre, EPROM, etc.) plus un ensemble des périphériques qui assure la communication entre l'application et les autres systèmes externes (Ethernet, HDMI, RS232, RGB, etc.). Pour assurer que l'application marche correctement, le système d'exploitation embarqué intervient pour établir la communication entre le matériel et l'application et fournis des services à l'application comme l'utilisation des ressources matérielles ou des services comme accès au réseau ou internet. À titre d'exemples, pour illustrer le rôle des systèmes d'exploitation embarqué dans l'évolution exponentiel des technologies, il est possible de gérer un avion à l'aide d'un système d'exploitation embarqué et même il facilite le contrôle de cette dernière.

C'est dans ce cadre que s'inscrit ce présent projet de fin d'études à l'École Nationale d'Ingénieur de Sousse proposer par STMicroelectronics, intitulé « Portage de la machine virtuel Lua sur processeur *ST40* ». Il consiste a effectué le portage[1] de Lua sur le *STB 7015DT2* basé sur le

[1] le faite de tourner l'application sur le plateforme embarqué *ST40/STLinux* (processur /OS)

processeur *ST40* d'architecture super H et de développer un interface entre *Lua*[2] et *DirectFB*[3] qui assure le contrôle de *DirectFB* à partir de Lua et ensuite effectuer le portage de cette interface sur le plateforme *ST40/STLinux*.

Ce rapport se subdivise en trois parties principaux qui se décomposes en plusieurs chapitres. La première partie présente l'état de l'art qui subdivise en trois chapitres.

- Un premièr chapitre introductif qui présente l'entreprise d'accueil et le cadre du projet.
- Un deuxième chapitre qui présente une étude personnelle des interpréteurs et ces techniques d'interprétation.
- Un troisième chapitre qui présente l'interpréteur Lua et les différentes méthode d'extension de cette dernière.

La deuxième partie dédié à la présentation de l'environnement matériel et la préparation de l'environnement de compilation croisé pour le système *STLinux* sur processeur *ST40*, chaque partie est présentée dans un chapitre à part.

Une troisième partie constituée de :

- Un chapitre qui présente le portage du Lua sur le système *ST40/STLinux* et les différente testes du langage.
- Un chapitre qui présente la différente topologie et méthode de développement de l'interface entre *Lua* et *DirectFB* et les différents testes effectués.

1.2 Présentation de l'entreprise d'acceuil

STMicroelectronics a été créé en 1987 par la fusion de deux sociétés de semi-conducteurs de longue date, SGS Microelettronica de l'Italie et Thomson Semiconducteurs de la France.

STMicroelectronics est l'un des leaders mondiaux de semiconducteurs avec des revenus nets de 10,35 milliards de dollars en 2010 et 2,53 milliards de dollars au 1er trimestre 2011. STMicroelectronics est une société indépendante globale qui œuvre dans le domaine des semi-conducteurs transnationale implantée dans 26 pays qui conçoit, développe, fabrique et lance sur le marché une large gamme de circuits intégrés à base de semi-conducteurs ainsi que des dispositifs discrets utilisés dans une grande variétée d'applications microélectroniques (systèmes de télécommunications, systèmes informatiques, produits de consommation, produits des véhicules à moteur et systèmes

[2]Lua est une langage de programmation extensible de taille minime, pour plus de détails consulter le chapitre 2.

[3]Bibliothèque d'accélération graphique, présenter dans le chapitre 5.

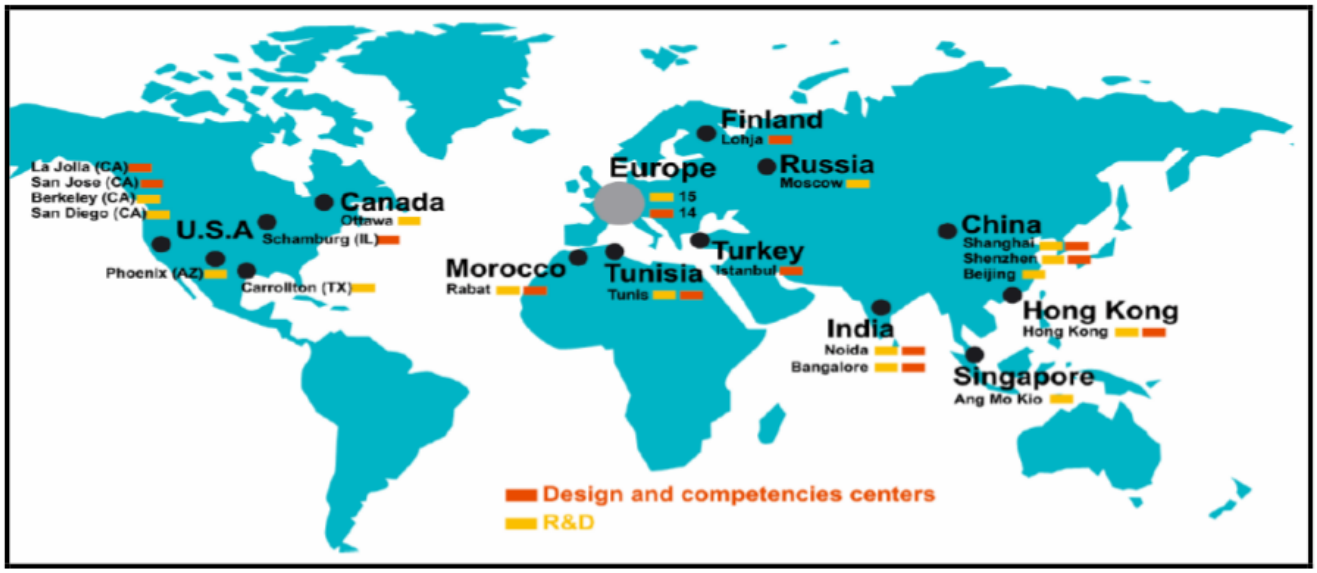

Figure 1.2.1: Répartition des centres de STMicroelectronics

de commande industriels). Le groupe comprend des centres de fabrication et de tri de circuits sur tranche et des centres d'assemblage et de tri des puces en boîtier. La répartition de ces centres est schématisée par la figure 1.2.1.

Le site de ST Tunis, créé en décembre 2001, développe des logiciels d'applications électroniques, des microcontrôleurs et des microprocesseurs informatiques et automobiles. Il contribue au développement d'un grand nombre de circuits intégrés destinés aux applications grand public, numérique, tels que les décodeurs, les lecteurs DVD et les appareils photos numériques. Le site de ST Tunis a commencé avec 9 ingénieurs et continue d'afficher une forte croissance avec 212 ingénieurs répartis dans plusieurs équipes collaborant avec des divisions dans des sites en France, en Italie et en Angleterre, en Inde, en Asie (Chine, Singapore, Japon), etc.

1.3 Présentation du projet

STMicroelectronics est parmi les leaders dans la conception et le développement des *Set-Top-Box* [4]. À travers un *STB* et sur l'écran d'une TV, nous pouvons faire tourner des jeux ainsi que plein d'autres services demandés par les clients. Dans le but d'intégrer ces nouveaux services et puisqu'il y a une concentration mondiale sur le domaine d'accélération graphique ainsi qu'une

[4]La boîte de placer-dessus de limite (Set-Top-Box en anglais) décrit un dispositif à la fois relié à une télévision et à une certaine source extérieure de signal, transforme le signal en contenu et l'affiche sur l'écran.

demande croissante des applications utilisant les technique d'interprétation, ST a, donc, intérêt à offrir aux clients la possibilité de faire tourner l'interpréteur Lua sur le plateforme *ST40/STLinux* et effectué un lien entre Lua et la bibliothèque d'accélération graphique *DirectFB* à travers un interface qui facilite énormément l'utilisation de cette bibliothèque pour profiter des services d'accélération graphique sur les *Set-Top-Box* de ST et de visualiser les outputs correspondants sur la TV. Le projet consiste a compilé le code source du *Lua* pour le système *ST40/STLinux* de STMicroelectronics et de l'exécuter sur un *Set-Top-Box* basé sur cette plateforme, ensuite de développer une interface entre *Lua* et *DirectFB* et le compiler pour le même système. Pour plus de détail voici la cahier de charge présenter par STMicroelectronics

Les besoins fonctionels principaux sont les suivants :

- La mise en place d'un environnement de compilation croisée pour l'environnement *STLinux* et processeur *ST40*.
- La compilation native et croisée de la bibliothèque open source *Lua*.
- La compilation native de la bibliothèque open source *DirectFB*.
- La conception et l'implémentation d'une nouvelle extensions pour interfacer *Lua* avec *DirectFB*.
- La nouvelle extension doit exposer les interfaces *IDirectFB* et *IDirectFBSurface*.
- Les fonctions de *IDirectFB* a implémenté sont : *SetCooperativeLevel()* et *CreateSurface()*.
- Les fonctions de *IDirectFBSurface* a implémenté sont les suivantes : *GetPosition()*, *GetSize()*, *Flip()*, *Clear()*, *SetClip()*, *GetClip()*, *SetColor()*, *SetPorterDuff()*, *Blit()*, *StretchBlit()*, *SetBlittingFlags()*, *SetDrawingFlags()*, *FillRectangle()*, *DrawRectangle()*, *DrawLine()*, *FillTriangle()* et *DrawString()*.

Les besoins fonctionnels complémentaires :

Les besoins fonctionnels optionnels sont les suivants :

- Exposer les interfaces *IDirectFBImageProvider* et *IDirectFBDisplayLayer* de *DirectFB*.
- Étudier le passage vers la machine virtuelle *LuaJIT*.
- Implémenter des émetteurs (de *LuaJIT*) d'instructions pour processur *ST40*.

Les besoins non-fontionnels sont comme suitte :

- Une étude de l'état de l'art en ce qui concerne les techniques d'interprétations de langages interprétés.
- Prévoir une bonne documentation du code source.

Contraintes : Ceci représente les contraintes ainsi que les conditions dont la solution doit prendre en compte :

- La solution pour l'extension *Lua/DirectFB* doit tenir compte de l'aspect orienté objet de *DirectFB*.
- La libération des objets *Lua/DirectFB* (tels que surfaces) doit être explicite via la méthode *release()*.

1.4 Conclusion

Dans le présent chapitre, nous avons essayés de cadrer notre projet dans son cadre, de présenter les différentes technologies existantes ainsi que l'intérêt de porter Lua sur l'architecture *SH4* et l'intérêt de porter l'interface entre *Lua* et *DirectFB* sur cette architecture. Dans le chapitre suivant nous allons présentés les interpréteurs ainsi qu'une étude et les différentes méthodes d'extensions du langage *Lua* .

Chapitre 2

LES TECHNIQUES D'INTERPRÉTATION

2.1 Introduction

Un programme et tout d'abord un texte et une séquence de symboles. Pour prendre son sens en tant que programme, ce texte doit être mis en présence d'un mécanisme capable de décoder et de produire un certain nombre de transformations, ce mécanisme appelé interpréteur et l'ensemble des symboles utilisés sont appelés grammaire. L'interpréteur ou interprète est un outil informatique ayant pour tâche d'analyser, de traduire et d'exécuter un programme écrit en langage dit langage interpréter. L'interpréteur lit le code source sous forme de script et dont exécuter les instructions après analyse lexicale, analyse syntaxique, analyse sémantique contenue pour générer une forme intermédiaire puis il effectue l'exécution d'une façon simultanée. Donc, l'interprète n'exécute pas le code source du programme, mais il exécute la forme intermédiaire. La génération de la forme intermédiaire consiste a analysé le code source des différents manières pour générer l'arbre abstrait décoré (section 2.3.3) qui sera par la suite interprété. La figure 2.1.1 donne une brève description de chaque module qui constitue un interpréteur.

Après la génération de la forme intermédiaire (arbre abstrait décoré), l'interpréteur passe à la phase d'exécution de la forme intermédiaire pour générer les résultats. Les sections suivantes présentent une description détaillée des différents modules d'un interpréteur.

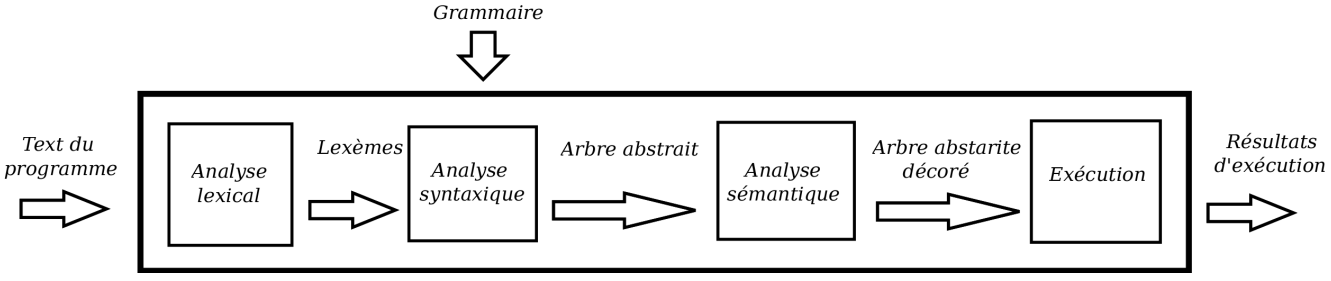

FIG. 2.1.1 – Structure d'interpréteur

2.2 Analyse lexical

2.2.1 Introduction

Après la phase de lecture du texte de programme, une suite des caractères est générée pour effectuer l'analyse lexicale afin de générer le flux lexèmes.

2.2.2 Principe

La tâche principale de l'analyseur lexical est de lire la suite des caractères du texte de programme, segmenter ce dernier en un ensemble des mots qu'on les appelles « tokens» (leur terme exact est « lexème », ce qui signifie unité lexicale). Ensuite fournir, d'une part la représentation du lexème (la chaîne de caractères), et d'autre part la classe du lexème (identificateur, nombre, opérateur, etc.). L'idée générale est, sachant où commence un symbole dans la chaîne de caractères du programme source, de rechercher où il se termine, en suivant les règles de définitions lexicales du langage, puis de trouver où commence le symbole suivant, etc. Il réalise également certaines tâches secondaires, une de ces tâches est l'élimination dans le programme source des commentaires et des espaces qui apparaissent sous forme de caractères blanc tabulation ou fin de ligne. Une autre tâche consiste à relier les messages d'erreur issus de l'interpréteur au programme source, par exemple un analyseur lexical peut associer un message d'erreur au numéro de ligne. Pour mieux comprendre le rôle de l'analyseur lexical prenons l'exemple suivant :
Prenant l'instruction suivante comme texte du programme a analysé :

$$a \ = \ 2 \ * \ (\ a + b) \ ; \tag{2.2.1}$$

La première phase consiste a éliminé les espaces, l'instruction devient :

$$a = 2 * (a + b) \tag{2.2.2}$$

Ensuite, l'analyseur lexical doit être capable de comprendre qu'il s'agit de la suite de lexèmes représentés dans le Tableau 2.1, ces lexèmes sont ensuite fournis en sortie de l'analyseur.

Si l'interpréteur trouve des erreurs au cours de l'exécution, l'analyseur lexical relie ces erreurs à la représentation du lexème. par exemple la variable b qui n'est pas déclarer dans le programme sera reliée à la ligne correspondante de l'erreur par l'analyseur lexical.

Classe du lexèmes	Représentation du lexèmes
Identificateur	a
Affectation	=
Nombre	2
Opération arithmétique	*
Caractère	(
Identificateur	a
Opération arithmétique	+
Identificateur	b
Caractère	)

TAB. 2.1 – Suite de lexèmes générer par l'analyseur lexical pour l'instruction 2.2.2

Erreur line1 : first declared here " b ".
L'exemple précédant, nous montre qu'après l'analyse de l'instruction, l'analyseur lexical lui effectue des transformations afin de générer une suite de lexèmes (classe et représentation, tableau 2.1) et les transmettre à l'analyseur syntaxique pour la suite des traitements. Cette opération n'est possible que si la structure lexicale du langage est correctement définie (c.-à-d. la structure du lexème).

2.2.3 Structure lexicale du langage

La structure lexicale peut être décrite d'une manière formelle dans le manuel du langage, par exemple : " Un identificateur est une suite de lettres, de chiffres et de soulignements qui commence par une lettre ; deux soulignements consécutifs sont interdits, ainsi qu'un soulignement final". Une telle description est satisfaisante pour l'utilisateur du langage, mais pour la construction d'un interpréteur, la structure lexicale est plus utilement décrite à l'aide des descriptions régulières qui sont composées par des expressions régulières. Donc pour définir la structure lexicale d'une langage donnée il faut définir les unités lexicaux (les lexèmes) ensuite définir les expressions régulières correspondantes.

Les lexèmes sont un ensemble des symboles (chaîne de caractères) qui sont la base de construction des règles de la structure lexicale du langage. Les commentaires et les espaces ne sont pas des

modèles élémentaires	***reconnaissant:***
x	le caractère x
.	tout caractère sauf la fin de ligne
[syz.....]	l'un des caractères x,y,z....
Opérateur de répétition:	
R?	une apparition R ou rien
R*	zéro ,une ou plusieur apparition de R
R+	une ou plusieur apparition de R
Opérateurs de composition:	
R1R2	une apparation de R1 suivie par une de R2
R1|R2	une apparition de R1 ou de R2
Groupement:	
(R)	R elle-meme

FIG. 2.2.1 – Modèles des expressions régulières

lexèmes, en ce sens l'analyseur syntaxique ne les consomme pas. Ils sont écartés par l'analyseur lexical, mais il est souvent utile de les préserver, afin de pouvoir montrer une partie du texte du programme entourant une erreur. Pour définir les lexèmes de notre langage, nous découpons l'information concernant les lexèmes en deux parties, la classe lexicale et la représentation. Les classes lexicaux des lexèmes sont utilisées pour spécifier les types de la représentation du lexème :

- Les littéraux : nombres (entier, réel, entier double...), chaîne, etc.
- Les symboles de ponctuation : (,), :=, =, [, [|, ; ... et
- Les identificateurs correspondant en fait à des mots réservés : if, then, int, etc...

Après la définition des lexèmes, il faut définir les expressions régulières qui définient la structure lexicale du langage. Une expression régulière est une formule qui décrit un ensemble des chaînes pouvant être infinie. L'expression régulière la plus élémentaire est celle qui reconnaît un seul caractère tant dis que la plus simple est celle qui spécifie ce caractère explicitement ; un exemple est le modèle a, qui reconnaît le caractère a. Il y a deux autres modèles élémentaires, l'un pour reconnaître un ensemble de caractères, l'autre pour reconnaître tous les caractères. Ces trois modèles élémentaires apparaissent dans la figure 2.2.1.

Un modèle élémentaire peut être suivi par un opérateur de répétition, par exemple : b ? pour

un b facultatif, b* pour une suite éventuellement vide ou non de b, b+ pour une suite non vide de b. Il y a deux opérateurs de composition l'un est l'opérateur invisible qui indique la concaténation, l'autre est l'opérateur |, qui sépare deux choix ; par exemple, ab*|cd ? reconnais ce qui est reconnu par ab* ou ce qui est connu par cd ?. Les opérateurs de répétition ont la plus forte priorité ; viens ensuite de concaténation ; l'opérateur de choix le plus faible priorité. Les parenthèses peuvent servir à grouper. Par exemple, l'expression sb*|cd ? est équivalente à (a(b*))|(c(d ?)).

En fin il suffit de regrouper les expressions régulières pour construire les descriptions régulières pour définir la structure lexicale d'un tel langage donné. L'exemple suivant montre une structure lexicale d'un réel dans le langage Lua :
FLOAT→ INT '.' INT ;
INT → ('0'..'9')+ ;

2.2.4 conclusion

La première étape du processus d'interprétation est l'analyse lexicale qui consiste a généré, à partir de la suite de caractères (texte du programme) et en utilisant sur la structure lexicale du langage comme référence pour l'analyse, une suite de lexèmes pour la suite de traitement. La phase suivante et l'analyse syntaxique qui utilise la suite de lexèmes générés par l'analyseur lexical.

2.3 Analyse syntaxique

2.3.1 Inroduction

Le résultat obtenu après analyse lexicale est une suite de lexèmes. Mais, toute une suite de lexèmes ne constitue pas un programme. Il faut, donc, vérifier la concordance de la suite avec la structure du langage du programme. L'analyseur syntaxique analyse les suites de lexèmes en utilisant la définition de la grammaire du langage, et vérifie si le programme est correctement écrit au point de vue syntaxique ensuite se charge de trouver les structures syntaxiques de programme et la représente par un arbre abstrait. L'interpréteur d'un langage donné ne peut pas effectuer l'analyse syntaxique si la grammaire du langage n'est pas définie correctement.

2.3.2 Grammaire

Les grammaires non contextuelles constituent le formalisme essentiel pour décrire la structure des programmes dans un langage de programmation. En principe, la grammaire d'un langage ne

décrit que la structure syntaxique, mais étant donné que la sémantique d'un langage est décrite en terme de syntaxe, la grammaire intervient également dans la définition sémantique. Une grammaire non contextuelle, on dit parfois grammaire BNF (pour Backus-Naur Form), est un quadruplet $G = (V_T, V_N, S_0, P)$ formé de:

- Un ensemble V_T de symboles terminaux,
- Un ensembe V_N de symboles non terminaux,
- un symbole $S_0 \in V_N$ particulier, appelé symbole de départ ou axiome,
- un ensemble P de règles de production[1], qui définit des construction syntaxique nommées, qui sont de la forme:

$$S \rightarrow S_1 S_2 S_3 ... S_k \qquad avec\ S \in V_N\, et\, S_i \in V_N \cup V_T$$

La partie gauche est le nom de la construction syntaxique; la partie droite donne la forme possible de la construction syntaxique. Voici un exemple de production:

$expression \rightarrow' (' expression\ opérateur\ expression ')'$

Nous pouvons expliquer ces éléments de la manière suivante :

1. Les symboles terminaux sont les symboles élémentaires qui constituent les chaînes du langage, Ce sont donc les lexèmes extraits du texte source par l'analyseur lexical.
2. Les symboles non terminaux désignant des ensembles des chaînes des symboles terminaux.
3. Le symbole de départ est un symbole non terminal particulier qui désigne la langage en son entier.
4. Les règles de productions peuvent être interprétées de deux manières :
   - Comme des règles permettant d'engendrer toutes les chaînes correctes. De ce point de vue, la production $S \rightarrow S_1 S_2 S_3 ... S_k$ se lit "pour produire un S correct [sous-entendu : de toutes les manières possibles] il fait produire un S_1 [de toutes les manières possibles] suivi d'un S_2 [de toutes les manières possibles] suivi d'un. . etc. suivi d'un S_k [de toutes les manières possibles] ".

[1] Appeler aussi production

- Comme des règles d'analyse, on dit aussi reconnaissance. La règle de production $S \rightarrow S_1S_2S_3...S_k$ se lit alors "pour reconnaitre un S, dans une suite de terminaux donnée, il faut reconnaitre un S_1 suivi d'un S_2 suivi d'un. etc. . suivi d'unS_k".

Si plusieurs règles de productions ont le même membre gauche:

$$S \rightarrow S_{1,1}S_{1,2}...S_{1,k}$$

$$S \rightarrow S_{2,1}S_{2,2}...S_{2,k}$$

$$\vdots$$

$$S \rightarrow S_{n,1}S_{n,2}...S_{n,k}$$

alors, on peut les noter simplement:

$$S \rightarrow S_{1,1}S_{1,2}...S_{1,k}|S_{2,1}S_{2,2}...S_{2,k}|.....|S_{n,1}S_{n,2}...S_{n,k}$$

La définition de la grammaire est l'énumération des règles de production. L'exemple suivant montre une définition d'une partie du langage Lua qui définit les nombres :

$number \rightarrow INT|FLOAT|EXP|HEX$;

Cette production décrit la structure syntaxique d'un nombre dans Lua. Un nombre dans Lua peut être un entier ou un réel ou exposant ou un nombre hexadécimal. INT, FLOAT, EXP et HEX sont des structures lexicales définies à l'aide des descriptions régulières.

La grammaire d'une langage est un outil indispensable pour l'analyseur syntaxique qui représente la suite de lexèmes en arbre abstrait appelé aussi arbre syntaxique. La section suivante définie l'arbre syntaxique est ça structure et l'arbre abstrait décore (qui est le résultat de l'analyse sémantique).

2.3.3 Arbre syntaxique, arbre abstrait, arbre abstraite décoré

L'arbre syntaxique du texte d'un programme est une structure des données qui montre avec précision comment les différents composants du texte peuvent être vus en termes de grammaire. On l'obtient grâce au processus d'analyse syntaxique qui consiste a structuréle texte selon la

grammaire donnée. La forme exacte de l'arbre syntaxique qu'implique la grammaire n'est en général la plus commode pour les traitements ultérieurs, aussi utilise-t-on une forme modifiée, appelée arbre syntaxique abstrait, ou plus simplement arbre abstrait. On peut attacher aux nœuds de cet arbre des informations détaillées sur la sémantique, sous la forme de décoration, qui sont conservées dans des champs supplémentaires des nœuds ; ce qui justifie le terme arbre abstrait décoré. La décoration concernant par exemple les informations de types (cette variable est de type réel). Les décorations attachées à un nœud sont aussi appelées attributs de ce nœud. C'est la tache de l'analyseur sémantique de construire et mettre en places les attributs.

Définition : Soit ω une chaine de symboles terminaux du langage $L(G)$; il existe donc une dérivation telle que $S \Longrightarrow^* \omega$. Cette dérivation peut étre représentée graphiquement par un arbre, appelé arbre syntaxique (on l'appelle encore arbre de dérivation, défini de la manière suivant:

- La racine de l'arbre est le symbole de dépard,
- Les nœuds intérieurs sont étiquetés par des symboles non terminaux,
- Si un nœuds intérieur η est étiqueté par le symbole S et si la production $S \rightarrow S_1S_2S_3...S_k$ a été utilisée pour dériver S alors les fils sont des noeuds étiquetés, de la gauche vers la droite, par $S_1S_2S_3...S_k$,

À titre d'exemple, voici la reconnaissance par un tel analyseur du texte "$10 * longeur + 500$" avec la grammaire G_1:

$expression \rightarrow expression\ "+"\ terme | terme$
$terme \rightarrow terme\ "*" facteur | facteur$
$facteur \rightarrow nombre | identificateur |\ "("expression")$

La chaîne d'entrée est donc:

$$(nombre" * "identificateur" + "nombre) \tag{2.3.1}$$

La figure 2.3.1 montre les différentes composants de l'arbre syntaxique de l'expression "$10 * longeur + 500$" construite avec la grammaire G_1.

La figure 2.3.2 montre l'arbre syntaxique de l'expression 10*longeur+500 construite avec la grammaire G1.

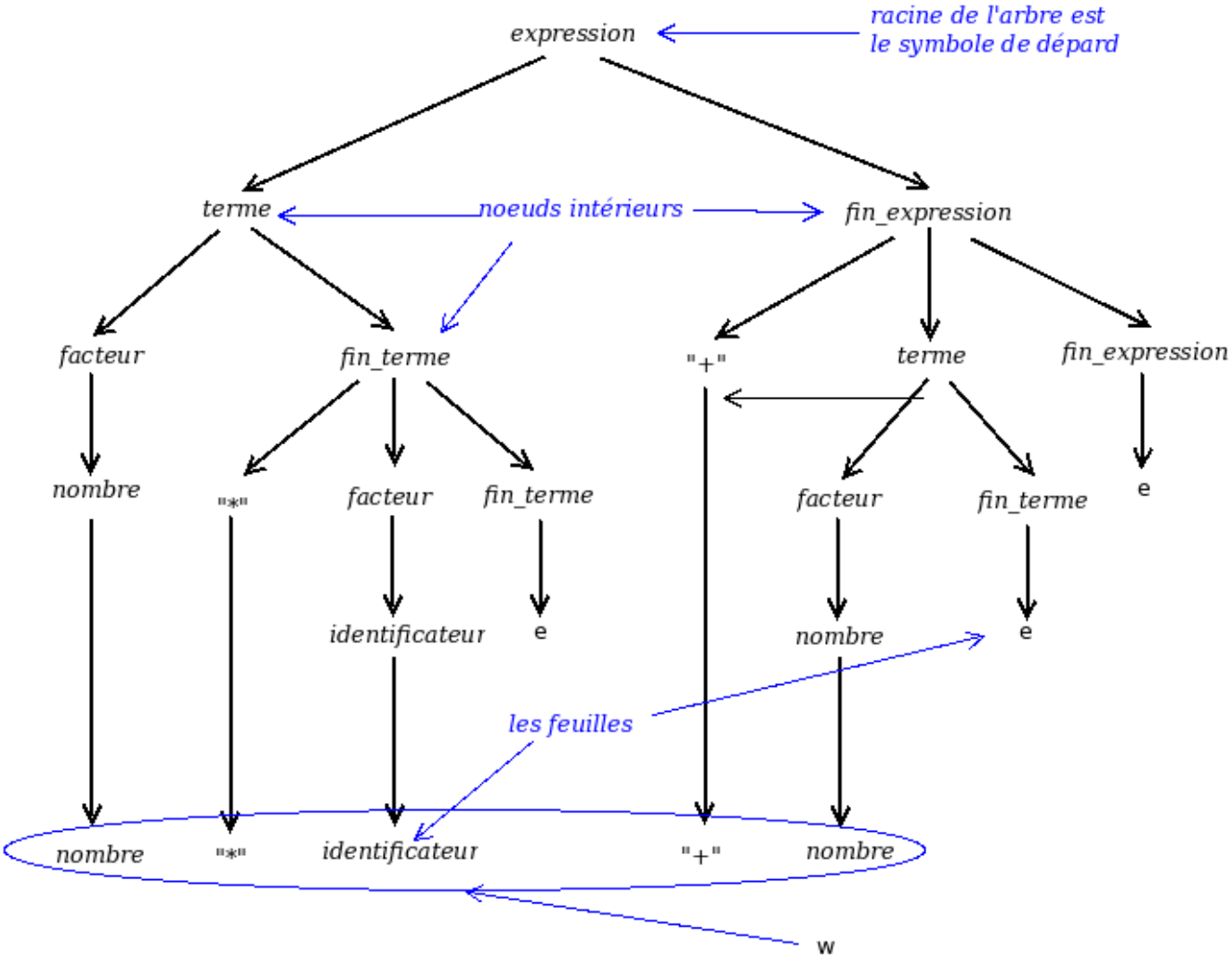

Figure 2.3.1: constitution de l'arbre syntaxique de l'expression 2.3.1 avec la grammaire G_1

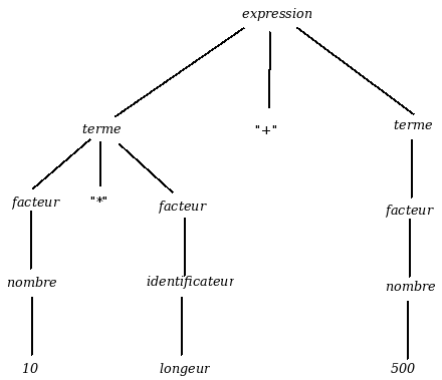

Figure 2.3.2: Arbre syntaxique de l'expression $10 * longeur + 500$

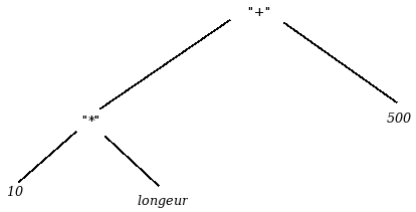

Figure 2.3.3: Arbre abstraite de l'expression $10 * longeur + 500$

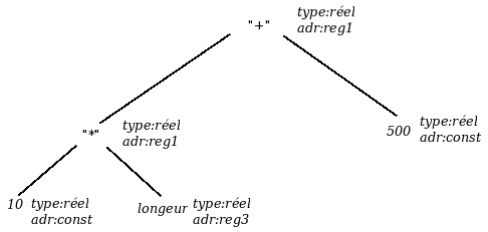

Figure 2.3.4: arbre abstrait décoré de l'expression 10*longeur+500

La figure 2.3.3 montre la même expression comme un arbre abstrait .

La figure 2.3.4 montre la même expression comme un arbre abstrait décoré dans la quel nous avons ajoutés des informations sur le type et l'emplacement.

Après la définition de la grammaire et l'arbre abstrait, on passe a expliqué les diffèrentes méthodes et les principes d'analyse syntaxique. Il y a deux manières d'effectuer l'analyse syntaxique : descendante et ascendante. Il existe deux méthodes bien connues et bien étudiées pour faire l'analyse : déterministe de gauche à droite et descendent (la méthode LL[2]), et déterministe de gauche à droite ascendante (la méthode LR[3]). De gauche à droite, signifie que les combinaisons de lexèmes sont traitées de gauche à droite, une lexème à la fois. Déterministe signifie qu'aucune recherche n'est nécessaire , chaque lexème amène l'analyseur un pas plus loin vers le but de construction de l'arbre syntaxique.

---

[2]Cela signifie qu'on peut en écrire des analyseurs :
- Lisant la chaîne source de la gauche vers la droite (gauche = left, c'est le premier L),
- Cherchant à construire une dérivation gauche (c'est le deuxième L).

[3]Cela signifie qu'on peut en écrire des analyseurs :
- Lisant la chaîne source de la gauche vers la droite (gauche = left, c'est le premier L),
- Cherchant à construire une dérivation droite (c'est le R, droite=right).

2.3.4 Analyseur descendant

Un analyseur descendant construit l'arbre syntaxique de la racine (le symbole de départ de la grammaire) vers les feuilles (la chaîne de terminaux). Pour en décrire schématiquement le fonctionnement nous allons donnés une fenêtre à symboles terminaux comme ci-dessus et une pile de symboles, c'est-à-dire une séquence de symboles terminaux et non terminaux à laquelle nous ajoutons et nous enlèvons des symboles par une même extrémité, en l'occurrence l'extrémité gauche (c'est une pile couchée à l'horizontale, qui se remplit de la droite vers la gauche).

Initialisation: Au départ, la pile contient le symbole de départ de la grammaire et la fenêtre montre le premier symbole terminal de la chaîne d'entrée.
Itération: Tant que la pile n'est pas vide, répéter les opérations suivantes:

- Si le symbole au sommet de la pile (c.-à-d. le plus à gauche) est un terminal α.
  - Si le terminal visible à la fenêtre est le même symbole α, alors :
    - dépiler le symbole au sommet de la pile et
    - faire avancer le terminal visible à la fenêtre,
  - Sinon, signaler une erreur (par exemple afficher "α attendu").
- Si non
  - S'il y a une seule production $S \rightarrow S_1S_2S_3...S_k$ ayant S pour membre gauche alors dépiler S et empiler $S_1S_2S_3...S_k$ à la place,
  - S'il y a plusieurs productions ayant S pour membre gauche, alors d'après le terminal visible à la fenêtre, sans faire avancer ce dernier, choisir l'unique production $S \rightarrow S_1S_2S_3...S_k$pouvant convenir, dépiler S et empiler $S_1S_2S_3...S_k$.

Terminaison: Lorsque la pile est vide:

- Si le terminal visible à la fenêtre est la marque qui indique la fin de la chaîne d'entrée alors l'analyse a réussi : la chaîne appartient au langage engendré par la grammaire,
- Sinon, signaler une erreur (par exemple, afficher caractères inattendus à la suite d'un texte correct).

A titre d'exemple, voici la reconnaissance par un tel analyseur du texte "$60 * vitesse + 200$" avec la grammaire G_1:

$expression \rightarrow terme\ fin\_expression$
$fin\_expression \rightarrow " + "\ terme\ fin\_expression | \varepsilon$
$terme \rightarrow facteur\ fin\_terme$
$fin\_terme \rightarrow " * "\ facteur\ fin\_terme | \varepsilon$
$facteur \rightarrow nombre | identificateur | "("expression")"$

La chaîne d'entrée est donc :

$$(nombre" * "identificateur" + "nombre) \tag{2.3.2}$$

Les états successifs de la pile et de la fenêtre sont représenté dans le tableau 2.2:

Lorsque la pile est vide, la fenêtre exhibe ϕ, la marque de fin de chaîne. La chaîne donnée appartient donc bien au langage considéré. L'analyseur syntaxique descendant construit l'arbre syntaxique en commencent par le nœud racine de l'arbre il construit ensuite les nœuds de l'arbre syntaxique en pré-ordre[4], ce qui signifie que la racine d'un sous-arbre est construite avant tous ses nœuds descendants. la figure 2.7 montre l'ordre et les différentes étape de construction de l'arbre syntaxique.

L'arbre syntaxique générer par l'analyseur syntaxique descendant est représenté dans la figure 2.8.

Après la présentation de l'analyseur descendant nous allons présentés l'analyseur ascendant.

2.3.5 Analyse ascendant

Le principe général des méthodes ascendantes est de maintenir une pile de symboles dans laquelle sont empilés (l'empilement s'appelle ici décalage) les terminaux au fur et à mesure qu'ils sont lus, tant que les symboles au sommet de la pile ne sont pas le membre droit d'une production de la grammaire. Si les k symboles du sommet de la pile constituent le membre droit d'une production alors ils peuvent être dépilés et remplacées par le membre gauche de cette production (cette opération s'appelle réduction). Lorsque dans la pile il n'y a plus que le symbole de départ de

[4]Quand on parcoure le nœud N en pré-ordre le processus de parcours visite le nœud N en premier puis, il parcours les sous-arbres de N de gauche à droite

N°	fenêtre	pile
étape 1	$nombre$	$expression$
étape 2	$nombre$	$terme\ fin\_expression$
étape 3	$nombre$	$facteur\ fin\_terme\ fin\_expression$
étape4	$nombre$	$nombre\ fin\_terme\ fin\_expression$
étape 5	$*$	$fin\_terme\ fin\_expression$
étape 6	$*$	$"*"\ facteur\ fin\_terme\ fin\_expression$
étape 7	$identificateur$	$facteur\ fin\_terme\ fin\_expression$
étape 8	$identificateur$	$identificateur\ fin\_terme\ fin\_expression$
étape 9	$"+"$	$fin\_terme\ fin\_expression$
étape 10	$"+"$	$\varepsilon\ fin\_expression$
étape 11	$"+"$	$fin\_expression$
étape 12	$"+"$	$"+"\ terme\ fin\_expression$
étape 13	$nombre$	$terme\ fin\_expression$
étape 14	$nombre$	$facteur\ fin\_expression$
étape 15	$nombre$	$nombre\ fin\_expression$
étape 16	ϕ	$fin\_expression$
étape 17	ϕ	ε
étape 18	lexèmes suivantes	Production suivante

Table 2.2: Étape d'analyse syntaxique descendant de la chaîne (2.3.2)

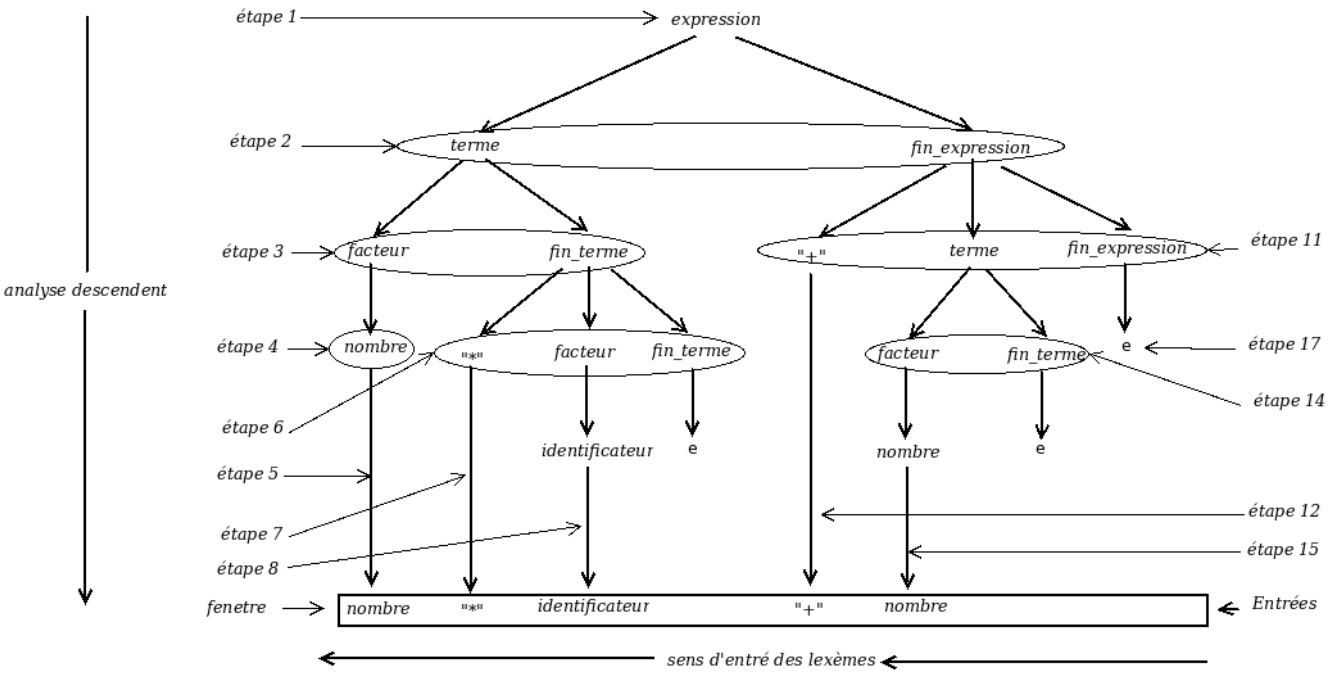

Figure 2.3.5: étape de génération de l'arbre syntaxique (analyse descendant) de la chaîne (2.3.2)

la grammaire, si tous les symboles de la chaîne d'entrée ont été lus, l'analyse a réussi. Le problème majeur de ces méthodes est de faire deux sortes de choix :

- Si les symboles au sommet de la pile constituent le membre droit de deux productions distinctes, laquelle utiliser pour exécuter la réduction ?
- Lorsque les symboles au sommet de la pile sont le membre droit d'une ou plusieurs productions, faut-il réduire tout de suite, ou bien continuer à décaler, afin de permettre ultérieurement une réduction plus juste ?

À titre d'exemple, voici la reconnaissance par un tel analyseur du texte "$60 * vitesse + 200$" avec la grammaire G_1:

$expression \rightarrow expression\ "+"\ terme | terme$

$terme \rightarrow terme\ "*"\ facteur | facteur$

$facteur \rightarrow nombre | identificateur |\ "("\ expression\ ")"$

La chaine d'entrée est donc:

$$(nombre\ "*"\ identificateur\ "+"\ nombre) \tag{2.3.3}$$

Les états successifs de la pile et de la fenêtre sont représenté dans le tableau 2.3.

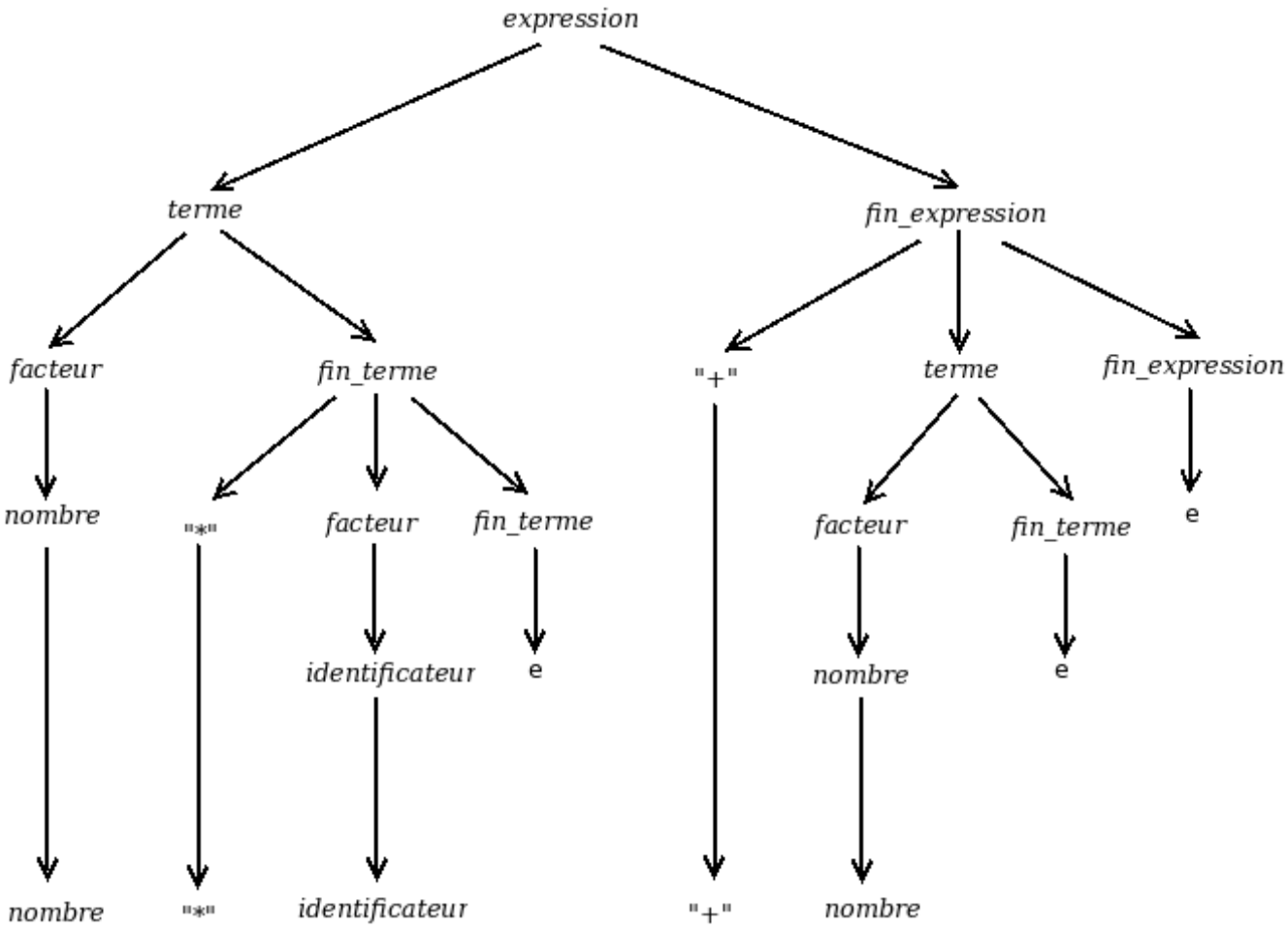

Figure 2.3.6: Arbre syntaxique (générer par l'analyseur descendant) de la chaîne(2.3.2)

étape	fenêtre	pile	action
étape1	$nombre$	_____	décalage
étape 2	" $*$ "	$nombre$	réduction
étape 3	" $*$ "	$facteur$	réduction
étape 4	" $*$ "	$terme$	décalage
étape 5	$identificateur$	$terme$ " $*$ "	décalage
étape 6	" $+$ "	$terme$ " $*$ " $identificateur$	réduction
étape 7	" $+$ "	$terme$ " $*$ " $facteur$	réduction
étape 8	" $+$ "	$terme$	réduction
étape 9	" $+$ "	$expression$	décalage
étape10	$nombre$	$expression$ " $+$ "	décalage
étape 11	ϕ	$expression$ " $+$ "$nombre$	réduction
étape 12	ϕ	$expression$ " $+$ "$facteur$	réduction
étape 13	ϕ	$expression$ " $+$ "$terme$	réduction
fin	ϕ	$expression$	succés

Table 2.3: étape d'analyse ascendant de la chaîne (2.3.3)

Les étapes de condtruction de l'arbre syntaxique par l'analyseur ascendant sont expliqué dans la figure 2.3.7:

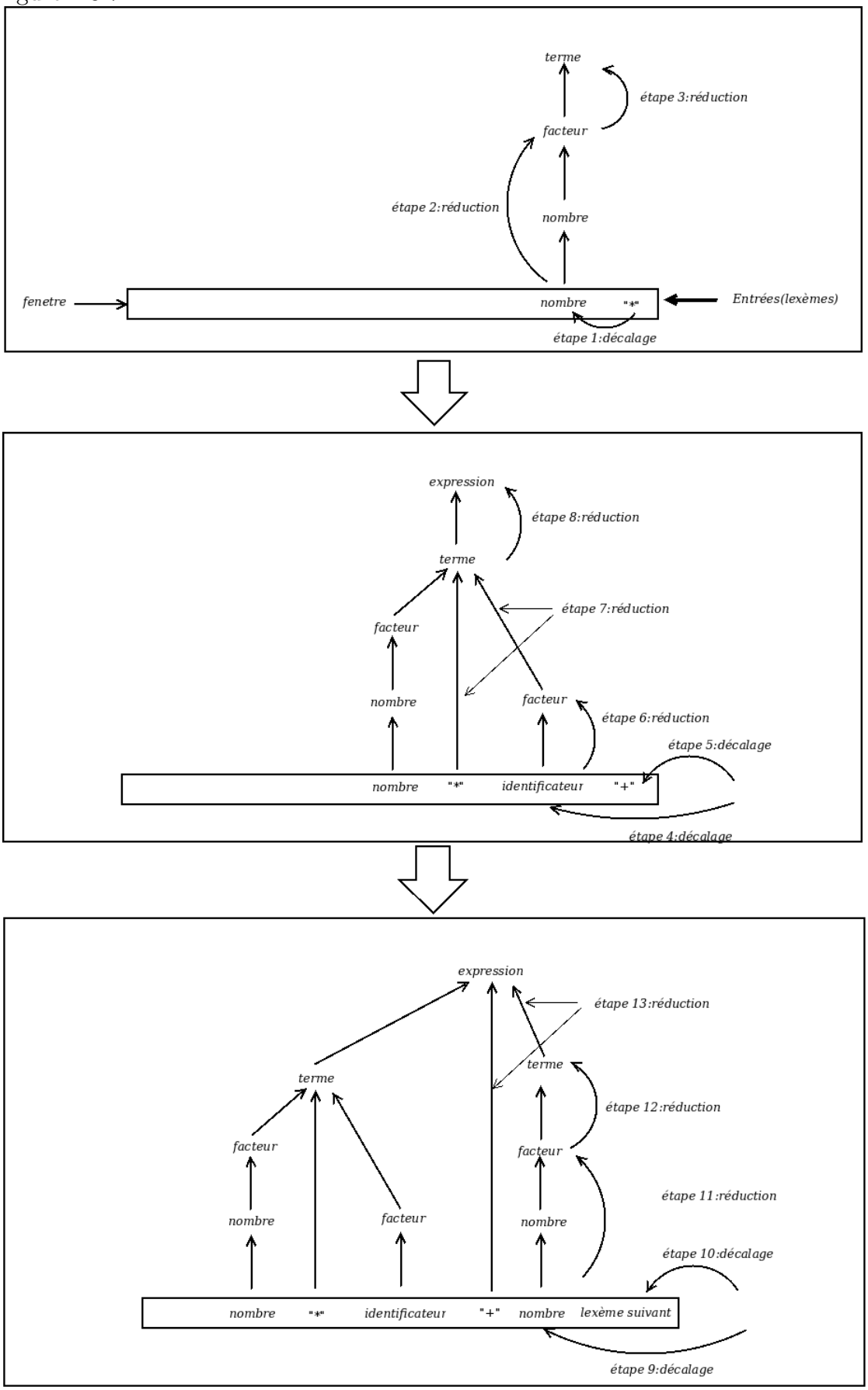

Figure 2.3.7: Étape de génération de l'arbre syntaxique de la chaîne (2.3.3)

On dit que les méthodes de ce type effectuent une analyse par décalage-réduction. Comme le montre le tableau 2.3, le point important est le choix entre la réduction et le décalage, chaque fois qu'une réduction est possible. Le principe est : les réductions pratiquées réalisent la construction inverse d'une dérivation droite. Par exemple, les réductions construisent à l'envers de la dérivation droite suivante :

$expression \Longrightarrow expression\ "+"\ terme$
$\Longrightarrow expression\ "+"\ facteur$
$\Longrightarrow expression\ "+"\ nombre$
$\Longrightarrow terme\ "+"\ nombre$
$\Longrightarrow terme\ "*"\ facteur\ "+"\ nombre$
$\Longrightarrow terme\ "*"identificateur\ "+"\ nombre$
$\Longrightarrow facteur\ "*"\ identificateur\ "+"nombre$
$\Longrightarrow nombre\ "*"\ identificateur\ "+"nombre$

2.3.6 Conclusion

La tâche principale de l'analyseur syntaxique et de transformer la suite de lexèmes générer par l'analyseur lexical en arbre abstrait en utilisant la structure syntaxique du langage définie par la grammaire non contextuelle. L'analyse syntaxique est effectué de deux manières, soit ascendant (type LL) soit descendant(type LR). L'analyseur syntaxique fournit l'arbre abstraite relative a une instruction aux module suivant de la chaîne d'interprétation qui va décoré l'arbre abstraite en ajoutant des informations sur la sémantique, c'est le module d'analyse sémantique.

2.4 Analyse sémantique

2.4.1 Introduction

Après la phase de l'analyse sémantique l'arbre abstrait est générer, maintenant il faut ajouter l'information sémantique pour générer par la suite l'arbre abstrait décorer. Le rôle de l'analyseur sémantique est d'effectuer les vérifications nécessaires à la sémantique du langage de programmation considéré et il ajoute des informations à l'arbre syntaxique abstrait et construit la table des symboles et de vérifier certaines contraintes liées au langage source comme :

- On ne peut pas utiliser dans une instruction une variable qui n'a pas été déclarée au préalable.
- On ne peut pas déclarer deux fois une même variable.

– On ne peut pas multiplier un réel avec une chaine de caractères.

⋮

Elle se fait généralement en même temps que l'analyseur syntaxique.

2.5 Phase d'exécution

À la fin de l'analyse sémantique, on à une représentation complète et vérifiée de la signification du programme sous forme d'un arbre syntaxique abstrait décoré. La façon la plus "simple" d'exécuter les actions décrites dans l'arbre abstrait est de les exécuter directement, un programme qui procède ainsi s'appelle un interprète ou interpréteur. Un interpréteur est un outil ayant pour tâche d'analyser, de traduire, et d'exécuter un programme écrit dans un langage informatique. Le cycle d'un interpréteur (ou le cycle d'exécution) est le suivant :

– Lire et analyser une instruction ;
– Si l'instruction est syntaxiquement correcte, l'exécuter ;
– Passer à l'instruction suivante.

Ainsi, l'interpréteur se charge aussi de l'exécution du programme, au fur et à mesure de son interprétation. L'exécution d'un programme interprété est généralement plus lente que le même programme compilé.

2.6 conclusion

L'intérêt des langages interprétés réside principalement dans la facilité de programmation et dans la portabilité. Les langages interprétés facilitent énormément la mise au point des programmes car ils évitent la phase de compilation, souvent longue, et limitent les possibilités de bogues. Ils permettent ainsi le développement rapide des applications ou de prototypes des applications. Ainsi, le langage BASIC fut la premiere langage interprétée a permet au grand public d'accéder à la programmation. La portabilité permet d'écrire un programme unique, pouvant être exécuté sur différentes plateformes sans changements, s'il existe un interpréteur spécifique à chaque plateforme matérielle.

Chapitre 3

ÉTUDE DU LANGAGE LUA

3.1 Introduction

Lua est une langage informatique de script créé en 1993. Lua[1] a été développé par Luiz Henrique de Figueiredo, Roberto Ierusalimschy et Waldemar Celes, membres du groupe de recherche TeCGraf, de l'université de Rio de Janeiro au Brésil. Lua est écrit en langage C ANSI strict, et de ce fait est compilable sur une grande variétée de systèmes. Les caractéristiques de Lua sont diverses : Performance (rapidité d'exécution), Portabilité (le langage peut être utilisé sous Unix, Windows et dans des terminaux portable ou encore des micro-processeurs), Embarquable (utilisable en plus d'un autre langage de programmation comme JAVA/C#/C/ADA...), Leger & gratuit (l'interpréteur Lua ne pèse qu'à l'entours de 200Ko et la langage est peut être utilisée dans des solutions commerciales sans frais).

Lua est conçue comme un langage extensible, nous pouvons l'étendre avec différents manières, soit par l'ajout des bibliothèques interne comme les bibliothèque standard du Lua, ou des bibliothèques dynamique programmer en Lua ou en C. Aussi Lua offre l'opportunité de communiquer avec C par l'intermédiaire d'une pile de type LIFO et en utilisant des API (Application Programme Interface) déjà définies en Lua qui facilite le transfère des arguments et des résultats a travers le stack entre Lua et C.

3.2 Installation, test et utilisation du langage Lua

3.2.1 Introduction

Avant l'étude de la structure de l'interpréteur Lua , nous allons présenter une section qui décrit la phase d'installation de lua sur la plateforme Linux.

---

[1]Qui signifie lune en portugais

3.2.2 Installation et test du langage Lua sur la plateforme Linux

Pour installer le langage Lua, il faut tout d'abord télécharger les fichiers sources, les configure ensuite on passe à la compilation enfin l'installation. La démarche à suivre pour installer Lua est le suivant :

1. Télécharger la version requise depuis le serveur FTP http ://www.lua.org/ftp/ , ensuite choisir la version. Généralement la dernière version lua-5.1.4.tar.gz.
2. Accéder au réparatoire du téléchargement :
   [hajri@localhost ˜]$ cd /home/hajri/Downloads/
3. *E*xtraire les fichiers source :
   [hajri@localhost Downloads]$ tar -zxf lua-5.1.4.tar.gz
4. Accéder au dossier des fichiers sources de Lua :
   [hajri@localhost Downloads]$ cd lua-5.1.4/src/
5. Compiler les fichiers sources pour la plateforme Linux :
   [hajri@localhost src]$ make linux
6. Retourner au réparatoire lua-5.1.4 et effectuer l'installation :
   [hajri@localhost src]$ cd ..
   [hajri@localhost lua-5.1.4]$ make install

Après la fin de l'installation, il suffit d'exécuter la ligne de commande suivante :
[hajri@localhost lua-5.1.4]$ lua
Lua 5.1.4 Copyright (C) 1994-2008 Lua.org, PUC-Rio
>

Maintenant, le langage Lua est installé sur notre système d'exploitation Linux (Fedora 14). Pour le test il suffit d'exécuter les scriptes dans le dossier test du répertoire de compilation. Pour nous assurer que Lua fonctionne correctement, nous allons exécuter un petit script écrit en Lua qui calcule la factorielle des nombres entiers de 1 à 16. L'algorithme suivant montre le code source du fichier fact.lua :

les résultats de test sont les suivant :

[hajri@localhost test]$ lua fact.lua
0 ! = 1
1 ! = 1

Algorithme 1 Code source du fichier fact.lua

```
Y = function (g)
local a = function (f)
return f(f)
end
return a
(function (f)
return g
(function (x)
local c=f(f) return c(x) end)
end)
end
F = function (f)
return function (n)
if n == 0 then
return 1
else return n*f(n-1)
end
end
end
factorial = Y(F)
function test(x)
io.write(x," ! = ",factorial(x),"\n")
end
for n=0,16 do
test(n)
end
```

2 ! = 2

3 ! = 6

4 ! = 24

5 ! = 120

6 ! = 720

7 ! = 5040

8 ! = 40320

9 ! = 362880

10 ! = 3628800

11 ! = 39916800

12 ! = 479001600

13 ! = 6227020800

14 ! = 87178291200

15 ! = 1307674368000

16 ! = 20922789888000

[hajri@localhost test]$

Nous allons passer à la présentation des bases d'écritures de script Lua.

3.2.3 Les bases d'écriture de script Lua

Afin de pourvoir écrire des scripts, il faut connaître un minimum des caractéristiques de ce langage. Lua utilise une structure de données flexible (les tables, à l'image de JavaScript), des variables globales peuvent être définies à travers les scripts et des librairies standards sont présentes (io, math, file, string...).

Les types de variables sont divers et communs aux autres langages :

- Nil
- Table,
- Nombre,
- Booléen,
- Chaînes de caractères, -
- Fonctions...

Les Tables peuvent être sous forme d'Array ou de List :

Algorithme 2 les structures répétitives de Lua

```
IF THEN ELSE
if type(a)== “table” then io.write(“a is table\n”)
else if type(a)==”number” then
io.write(“a is number\n”)
end
WHILE
a = { 1,2,3,4,5,6}
i = 1
while a[i] do
print(a[i]) i = i + 1
end
FOR
for a=1,10,2 do
print(a)
end
```

a = { 1,2,3,4 } ; print(a[1])

A = { [0]=1,2,3,4 } ; print(A[0])

Ou encore sous forme de Dictionnaire :

d = {i=1,j=2,k=3} ; print(d[”i”])

d = {i=4,j=5,k=6} ; print(d.i)

Il est important de noter que les éléments de ces tables peuvent être de n'importe quelle sorte. Tout est basé sur les tables dans le langage Lua. Lua possède les principaux fonctions algorithmiques des autres langages par exemple :

Nous pouvons écrire des scripts directement sur le Shell ou on peut écrire notre scripte Lua dans un fichier file.lua ensuite l'appeler à partir du Lua comme l'exemple de la factorielle présenter dans la section précédente.

3.2.4 Conclusion :

Nous avons présenter dans ce chapitre les méthodes d'écritures des scriptes Lua, d'une manière brève nous allons passer dans les sections suivantes a étudié l'interpréteur Lua en présentant la structure lexicale et syntaxique du langage ainsi le boucle d'interprétation.

3.3 Étude de l'interpréteur Lua :

3.3.1 Introduction :

Dans cette section nous allons présenter la structure lexicale et syntaxique du langage Lua.

3.3.2 La structure lexicale du langage Lua

Comme nous avons mentionner dans le chapitre 2 section d'analyse lexical que le processus d'analyse lexical ne peut pas être effectué sans la définition du structure lexicale du langage Lua. Pour y faire nous allons extraire les lexèmes du langage Lua ainsi que les descriptions régulières qui définient la structure lexical du langage Lua.

La définition du lexème a été aborder dans les section précèdentes, cette section consiste a identifié les classes lexicaux (Tokens) et les lexèmes (unité lexicale) du langage Lua. Pour identifier ces dernières, une analyse des fichier llex.c (analyseur lexical) et lparser.c (analyseur syntaxique du Lua) a étée effectuer.

Les classes lexicaux "Tokens" du Lua sons représentées par la structure (fichier llex.h, line :49) :

La classe lexicale INT : La classe lexicale INT, correspand aux valeurs entieres, est représenté par l'expression régulière suivante :

$$INT \ : \ ('0'..'9')+; \tag{3.3.1}$$

L'expression régulière 3.3.1 signifie qu'ne lexèmes qui appartient aux classe lexicale INT doit étre composé d'une ou plusieurs apparitions de l'un des chiffres compris entre 0 et 9.

La classe lexicale FLOAT : La classe lexicale FLOAT, correspand aux valeurs réeles, est représentée par l'expression régulière suivante :

$$FLOAT \ : \ INT \,'.'\, INT; \tag{3.3.2}$$

L'expression régulière 3.3.2 signifie qu'une lexème qui appartient au classe lexicale $FLOAT$ doit étre composée d'une apparition d'une lexème qui appartient au classe lexical INT suivie d'une apparition d'un point $'.'$ suivie d'une apparition d'une autre lexème qui appartient au classe lexical INT.

Algorithme 3 structure du lexèmes du Lua (fichier : llex.h, ligne :43).

```
typedef struct Token
{ int token;
SemInfo seminfo;
} Token;
/*le champ SemiInfo represante l'information sémantique et structure par l'union suivante*/
typedef union
{ lua_Number r;
TString *ts;
} SemInfo;
/*fichier lobject.c ligne :199*/
typedef union TString
{ L_Umaxalign dummy; /* ensures maximum alignment for strings */
struct
{CommonHeader;
lu_byte reserved;
unsigned int hash;
size_t len;
} tsv;
} TString;
```

La classe lexicale EXP : La classe lexicale EXP, correspand aux valeurs réeles ou entières représentées sous la forme d'un exposant (par exemple : $10.45\ E-4$ ou $10\,e2$), est représentée par l'expression régulière suivante :

$$EXP\ :\ (INT\,|\,FLOAT)\ ('E'|'e')\ ('-')?\ INT; \tag{3.3.3}$$

L'expression régulière signifie qu'une lexème qui appartient au classe lexicale EXP doit étre composée d'une apparition d'une lexème qui appartient aux classes lexicaux INT ou $FLOAT$ ($(INT\,|\,FLOAT)$), suivie d'une apparition d'une lexème qui appartient à {E,e}, suivie d'une apparition ou non du caractère - (('-') ?), suivie d'une apparition d'une lexème qui appartient au classe lexicale INT.

La classe lexicale HEX : La classe lexicale HEX, correspand aux valeurs hexadécimales, est représentée par l'expression régulière suivante :

$$HEX\ \ :\ '0x'\ ('0'..'9'|'a'..'f')+; \tag{3.3.4}$$

L'expression régulière 3.3.4 signifie qu'une lexème qui appartient au classe lexicale HEX doit étre composée d'une apparition de la chaine de caractàres $'0x'$, suivie d'un ou plusieurs apparitions de l'un des caractàres qui appartient à la plage comprises entre 0 et 9 ou a la plage comprises entre a et f ($('0'..'9'|'a'..'f')+$).

La classe lexicale NAME : La classe lexicale NAME correspand à l'identificateur qui peut être n'importe quelle caractère, chiffre, ou "underscores _ " est représentée par l'expression régulière suivante :

$$name\ :\ ('a'..'z'|'A'..'Z'|'\_'|'0'..'9')*; \tag{3.3.5}$$

L'expression régulière 3.3.5 signifie qu'une lexème qui appartient au classe lexicale $NAME$ doit étre composée de zéro, une ou plusieur apparitions d'un caractére alphabétique minuscule ou majuscule ou une "underscores" ou un chiffre. Les mots clés suivant sont reservés et ne peuvent pas être identifiés ou considérés commes des lexèmes du classe lesicale $NAME$:

and break do else elseif end false for
function if in local nil not or repeat

Algorithme 4 défintion du tableau qui contient la définition des classes lexicaux du LUA(fichier llex.c, ligne :37).

```
const char *const luaX_tokens [] =
{ "and", "break", "do", "else", "elseif",
"end", "false", "for", "function", "if",
"in", "local", "nil", "not", "or", "repeat",
"return", "then", "true", "until", "while"
, "..", "...", "==", ">=", "<=", "~=", "
<number>", "<name>", "<string>", "<eof>",
NULL };
```

return then true until while

Les caractères suivant appartient aussi au lexèmes du lua :

\+ - * / % ^ # == ~= <= >= < > = ( ) { } [ ]; : , ...

Certain de ces caractères et les mots clé du langage sont représentés par des symboles dans le code du LUA (llex.h, llex.c) seront utiliser par la suite dans le code du LUA. Pour être plus claire, le programmeur qui a écrit le langage LUA, utilise les symboles représenter dans l'algorithme 5 au lieu d'utiliser directement les symboles représenter dans l'algorithme 4.

Les symboles enumerés dans l'algorithme 5 correspand en ordre respective aux lexèmes (ou classes lexicaux) definient dans l'algoritme 4.

À partir des expressions régulières qui représentent une partie des classes lexicaux (Tokens) du Lua :

$number : INT|FLOAT|EXP|HEX;$

$string$: je;cherche encore a l'identifier.....

$name : ('a'..'z'|'A'..'Z'|'underscors'|'0'..'9')*$

$EXP : (INT|FLOAT)('E'|'e')('-')INT;$

$FLOAT : INT'.'INT;$

$INT : ('0'..'9')+;$

$HEX :' 0x'('0'..'9'|'a'..'f')+;$

Le but de cette section est d'identifier les lexèmes du LUA.La figure 3.3.1 décrit tous les

Algorithme 5 Symboles utiliser par le programmeur du LUA pour définir une partie des Tokens (fichier : llex.h, ligne : 24).

```
enum RESERVED {
/* terminal symbols denoted by reserved words */
TK_AND = FIRST_RESERVED, TK_BREAK,
TK_DO, TK_ELSE, TK_ELSEIF, TK_END,
TK_FALSE, TK_FOR, TK_FUNCTION,
TK_IF, TK_IN, TK_LOCAL, TK_NIL, TK_NOT
, TK_OR, TK_REPEAT, TK_RETURN, TK_THEN,
TK_TRUE, TK_UNTIL, TK_WHILE,
/* other terminal symbols */
TK_CONCAT, TK_DOTS, TK_EQ, TK_GE
, TK_LE, TK_NE, TK_NUMBER,
TK_NAME, TK_STRING, TK_EOS
};
```

TK_AND, TK_BREAK, TK_DO, TK_ELSE, TK_ELSEIF, TK_END, TK_FALSE, TK_FOR, TK_FUNCTION, TK_IF, TK_IN, TK_LOCAL, TK_NIL, TK_NOT, TK_OR, TK_REPEAT, TK_RETURN, TK_THEN, TK_TRUE, TK_UNTIL, TK_WHILE,TK_CONCAT, TK_DOTS, TK_EQ, TK_GE, TK_LE, TK_NE, TK_NUMBER, TK_NAME, TK_STRING, TK_EOS,+, -, *, /, %, ˆ, #, <, >, =, (,), {, }, [,],; , :,.,

FIG. 3.3.1 – Lexèmes du LUA.

lexèmes utilisées pour effectuer l'analyse lexicale :

Après la présentation de la structure lexicale du langage Lua nous allons présenter dans la section suivante la structure syntaxique de cette dernière.

3.3.3 La structure syntaxique du langage Lua

Nous avons vue dans le chapitre 2 dans la section Analyse syntaxique que l'analyseur syntaxique d'un tel langage interprétée ne peut pas fonctionner sans la définition de la structure syntaxique de ce langage. La définition de la structure syntaxique consiste à définir la grammaire de ce langage.

La définition de la grammaire a étée aborder dans la section précédante, nous présentons ainsi la grammaire du langage Lua. La grammaire du langage Lua, extraite du fichier lparser.c, est composée d'un ensemble de productions définies à l'aide des descriptions régulières présenter dans l'algorithme 6.

Les règles de productions qui définient la grammaire de Lua sont écrites à la base des descriptions régulières (définie dans le chapitre 1). Nous pouvons expliquer la grammaire de Lua de la manière suivante : un script Lua est composé par un block ou chunk qui est composé de zéro ou une ou plusieurs apparitions de stat suivies par une apparition ou rien de ' ;' suivi d'une apparition ou rien de laststat suivie par une apparition ou rien de « ; ». Ce qui montre les deux règles de productions suivantes :

block : chunk ;

chunk : (stat (' ;') ?)* (laststat (' ;') ?) ? ;

Un stat est composé par :

- Liste des expressions : varlist1'='explist1 par exemple, a=11,
- Une appelle fonction,
- Les boucles répétitives et itératives (do ,while, for , if else),
- Une description d'une fonction : 'function' funcname funcbody,
- Une fonction ou liste des variables locales voici la règle de production qui définie stat :

stat : varlist1 '=' explist1 |

functioncall |

'do' block 'end' |

'while' exp 'do' block 'end' |

'repeat' block 'until' exp |

'if' exp 'then' block ('elseif' exp 'then' block)* ('else' block) ? 'end' |

'for' NAME '=' exp ',' exp (',' exp) ? 'do' block 'end' |

'for' namelist 'in' explist1 'do' block 'end' |

'function' funcname funcbody |

'local' 'function' NAME funcbody |

'local' namelist ('=' explist1) ? ;

Le symbole non terminal « laststat » est défini par la règle de production suivante :

laststat : 'return' (explist1) ? | 'break' ;

Algorithme 6 Grammaire du langage Lua

```
chunk : (stat (';') ?)* (laststat (';') ?) ? ;
block : chunk ;
stat : varlist1 '=' explist1 |
functioncall |
'do' block 'end' |
'while' exp 'do' block 'end' |
'repeat' block 'until' exp |
'if' exp 'then' block ('elseif' exp 'then' block)* ('else' block) ? 'end' |
'for' NAME '=' exp ',' exp (',' exp) ? 'do' block 'end' |
'for' namelist 'in' explist1 'do' block 'end' |
'function' funcname funcbody |
'local' 'function' NAME funcbody |
'local' namelist ('=' explist1) ? ;
laststat : 'return' (explist1) ? | 'break' ;
funcname : NAME ('.' NAME)* (' :' NAME) ? ;
varlist1 : var (',' var)* ;
namelist : NAME (',' NAME)* ;
explist1 : (exp ',')* exp ;
exp : ('nil' | 'false' | 'true' | number | string | '...' | function | prefixexp | tableconstructor | unop exp) (binop exp)* ;
var : (NAME | '(' exp ')' varSuffix) varSuffix* ;
prefixexp : varOrExp nameAndArgs* ;
functioncall : varOrExp nameAndArgs+ ;
/*
var : NAME | prefixexp '[' exp ']' | prefixexp '.' NAME ;
prefixexp : var | functioncall | '(' exp ')' ;
functioncall : prefixexp args | prefixexp ' :' NAME args ;
*/
varOrExp : var | '(' exp ')' ;
nameAndArgs : (' :' NAME) ? args ;
varSuffix : nameAndArgs* ('[' exp ']' | '.' NAME) ;
args : '(' (explist1) ? ')' | tableconstructor | string ;
function : 'function' funcbody ;
funcbody : '(' (parlist1) ? ')' block 'end' ;
parlist1 : namelist (',' '...') ? | '...' ;
tableconstructor : '{' (fieldlist) ? '}' ;
fieldlist : field (fieldsep field)* (fieldsep) ? ;
field : '[' exp ']' '=' exp | NAME '=' exp | exp ;
fieldsep : ',' | ' ;' ;
binop : '+' | '-' | '*' | '/' | '^' | '%' | '..' |
'<' | '<=' | '>' | '>=' | '==' | '~=' |
'and' | 'or' ;
unop : '-' | 'not' | '#' ;
number : INT | FLOAT | EXP | HEX ;
string : NORMALSTRING | CHARSTRING | LONGSTRING ;
```

L'algorithme 6 définie toute la grammaire du langage Lua, il suffit de connaître la signification des descriptions régulières définies dans le chapitre 2. Le reste de la grammaire du Lua définit les symboles non terminaux qui constituent les règles de production des symboles non terminaux « stat » et « laststat ».

3.4 Méthodes d'extention du langage Lua

3.4.1 Introduction :

Comme nous avons introduire dans les chapitres précédant, Lua est une langage extensible, c'est-à-dire, nous pouvant étendre la langage Lua pour qu'elle support des nouveaux fonction définie dans la bibliothèque. Autrement dit, le fait d'étendre Lua correspond à ajouter des nouveaux lexèmes au langage. Pour le faire, on peut utiliser plusieurs méthodes. Les sections suivantes présentent en détaille les différentes méthodes pour étendre Lua.

3.4.2 Étendre Lua avec une bibliothèque interne

Cette méthode consiste a étendre Lua avec une bibliothèque interne comme les bibliothèques standard du Lua. Cette bibliothèque doit être écrite en suivant le démarche spécifique. Pour mieux comprendre le principe d'extension en suivant cette méthode nous allons traités un exemple illustratifs.

L'exemple consiste à ajouter au Lua une bibliothèque appeler *ldirectfb* qui contient une fonction *directfb_open()* qui affiche le message « testlib library works ». Pour effectuer cette extension il faut suivre l'algorithme suivant :

- Créer la nouvelle bibliothèque testlib.c,
- Ajouter la bibliothèque au lualib.c,
- Ajouter la bibliothèque au fichier d'initialisation linit.c,
- Changer le Makefile de Lua en ajoutant la bibliothèque testlib.c,
- Compiler tous les codes sources et installer la nouvelle Lua et effectuer un test.

Créer la nouvelle bibliothèque

La première étape consiste a créé la nouvelle bibliothèque ldirectfb.c. Tout d'abords nous allons définir la fonction qui charge la bibliothèque et enregistre ces fonctions, cette fonction

Algorithme 7 code source de ldirectfb.c

```
#include <stdio.h>
#define ldirectfblib_c /* Définir la bibliothèque*/
#define LUA_LIB
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"
/*la fonction C directfb_open */
static int directfb_open (lua_State *L){
printf("directfblib working ");
return 1; };
//Les nouvelles lexème lier au C function, pour appeler la fonction (directfb_open) de Lua nous devant
exécutés l'instruction suivante : >directfb.open() //
static const luaL_Reg directfblib[] = {
{"open", directfb_open},
{NULL, NULL} };
/*cette function ouvre directfb_lib et enregistre ces fonctions */
LUALIB_API int luaopen_directfb (lua_State *L){
luaL_openlib(L, LUA_DIRECTFBLIBNAME, directfblib, 0);
return 1; };
```

est *LUALIB_API int luaopen_directfb (lua_State *L)*. Cette dernière est une API[2] du Lua qui sert a ouvrir la bibliothèque ldirectfb.c. Ensuite, nous devons définir la bibliothèque au début de fichier ldirectfb.c (*#define ldirectfblib_c*). Après nous devons devellopper les fonctions C de la bibliothèque et on les ajoutent au tableau *static const luaL_Reg directfblib[]* afin de les enregistrer dans Lua lors de l'ouverture de la bibliothèque ldirectfb. L'algorithme 7 montre la structure de fichier source ldirectfb.c.

Pour ajouter des nouveaux fonctions au ldirectfblib, il suffit de les définir dans le code source et les ajouter au tableau *static const luaL_Reg directfblib[]* afin d'être enregistrer. Si on ne les ajoutent pas au tableau, ces fonctions ne seront pas reconnu par Lua.

[2]C'est l'abriviation en anglais du Application Programme Interface

Ajouter la bibliothèque au lualib. c

Après le développement de la bibliothèque, pour que Lua reconnaît cette detnière comme une bibliothèque interne, on doit l'ajouter au lualib.h. L'algorithme 8 montre la méthode d'ajout de la bibliothèque au lualib.h :

Ajouter la bibliothèque au fichier d'initialisation linit.c

Lors de démarrage du Lua, le chargement des bibliothèques internes ce fait avec le fichier d'initialisation linit.c. Donc pour que Lua charge notre nouvelle bibliothèque lors de son démarrage il faut l'ajouter au fichier d'initialisation du Lua comme nous montre l'algorithme 9.

Il faut indiquer le non de la bibliothèque avec la constante définie ultérieurement dans lualib.h (*LUA_DIRECTFBLIBNAME*) et non pas par le nom de la bibliothèque. Il faut aussi indiquer le non de la fonction qui ouvre et charge la bibliothèque (*luaopen_directfb*). Après cette action, au démarrage du Lua, Lua cherche le nom de la bibliothèque qui est déjà définie dans lualib.h, une foie trouver elle utilise la fonction définie dans linit.c pour ouvrir la bibliothèque. Maintenant, nous avons terminer l'écriture des code source nous allons passer aux phases de compilation et de test après le changement du Makefile du Lua.

Changement de Makefile,compilation, installation et test du nouvelle Lua

Une fois tous les codes sources sont écrite nous allons changer le Makefile du Lua afin de pouvoir compiler la nouvelle bibliothèque avec Lua. Le changement effectuer dans le Makefile consiste a ajouté les règles de compilation de la bibliothèque ldirectfb.c qui sont les fichiers de dépendance de cette dernière et nous devant l'ajouter au LIB_O (qui définie les bibliothèque interne du Lua dans la Makefile) comme suit :

ldirectfblib.o :ldirectfblib.c lualib.h lua.h lauxlib.h

LIB_O= lauxlib.o lbaselib.o ldblib.o liolib.o lmathlib.o loslib.o ltablib.o \
lstrlib.o loadlib.o linit.o ldirectfblib.o

Il suffit maintenant de compiler tous les codes source avec la ligne de commande shell suivant :

[Hajri@STHajri src]$ make linux

Le figure 3.4.1 montre le résultat de compilation.

Après la compilation nous devant réinstaller Lua avec la ligne de commande shell suivant :

[Hajri@STHajri lua-5.1.4+directfb lib]$ make install

Algorithme 8 Code source de lualib.h après l'ajout de ldirectfb.

```
** $Id : lualib.h,v 1.36.1.1 2007/12/27 13 :02 :25 roberto Exp $
** Lua standard libraries
** See Copyright Notice in lua.h */
#ifndef lualib_h
#define lualib_h
#include "lua.h"
/* Key to file-handle type */
#define LUA_FILEHANDLE "FILE*"
#define LUA_COLIBNAME "coroutine"
LUALIB_API int (luaopen_base) (lua_State *L);
#define LUA_TABLIBNAME "table"
LUALIB_API int (luaopen_table) (lua_State *L);
#define LUA_IOLIBNAME "io"
LUALIB_API int (luaopen_io) (lua_State *L);
#define LUA_OSLIBNAME "os"
LUALIB_API int (luaopen_os) (lua_State *L);
#define LUA_STRLIBNAME "string"
LUALIB_API int (luaopen_string) (lua_State *L);
#define LUA_MATHLIBNAME "math"
LUALIB_API int (luaopen_math) (lua_State *L);
#define LUA_DBLIBNAME "debug"
LUALIB_API int (luaopen_debug) (lua_State *L);
#define LUA_LOADLIBNAME "package"
LUALIB_API int (luaopen_package) (lua_State *L);
#define LUA_DIRECTFBLIBNAME "directfb"
LUALIB_API int (luaopen_directfb) (lua_State *L);
/* open all previous libraries */
LUALIB_API void (luaL_openlibs) (lua_State *L);
#ifndef lua_assert
:
```

Algorithme 9 Code source de linit.c après l'ajout de la bibbliothèque ldirectfb

```
/* ** $Id : linit.c,v 1.14.1.1 2007/12/27 13 :02 :25 roberto Exp $
** Initialization of libraries for lua.c
** See Copyright Notice in lua.h */
#define linit_c
#define LUA_LIB
#include "lua.h"
#include "lualib.h"
#include "lauxlib.h"
static const luaL_Reg lualibs[] = {
{"", luaopen_base},
{LUA_LOADLIBNAME, luaopen_package},
{LUA_TABLIBNAME, luaopen_table},
{LUA_IOLIBNAME, luaopen_io},
{LUA_OSLIBNAME, luaopen_os},
{LUA_STRLIBNAME, luaopen_string},
{LUA_MATHLIBNAME, luaopen_math},
{LUA_DBLIBNAME, luaopen_debug},
{LUA_DIRECTFBLIBNAME, luaopen_directfb},
{NULL, NULL} };
LUALIB_API void luaL_openlibs (lua_State *L) {
const luaL_Reg *lib = lualibs;
for (; lib->func; lib++) {
lua_pushcfunction(L, lib->func);
lua_pushstring(L, lib->name);
lua_call(L, 1, 0); } }
```

```
[Hajri@STHajri src]$ make linux
make all MYCFLAGS=-DLUA_USE_LINUX MYLIBS="-Wl,-E -ldl -lreadline -lhistory -lncurses"
make[1]: Entering directory `/home/Hajri/Desktop/PFE-STMicroelectronic/Lua_task/Extand lua/lua-5.1.4+directfb lib/src'
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o lbaselib.o lbaselib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o ldblib.o ldblib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o liolib.o liolib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o lmathlib.o lmathlib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o loslib.o loslib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o ltablib.o ltablib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o lstrlib.o lstrlib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o loadlib.o loadlib.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o linit.o linit.c
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o ldirectfblib.o ldirectfblib.c
ar rcu liblua.a lbaselib.o ldblib.o liolib.o lmathlib.o loslib.o ltablib.o lstrlib.o loadlib.o linit.o ldirectfblib.o
ranlib liblua.a
gcc -O2 -Wall -DLUA_USE_LINUX   -c -o lua.o lua.c
gcc -o lua  lua.o liblua.a -lm -Wl,-E -ldl -lreadline -lhistory -lncurses
gcc -o luac  luac.o print.o liblua.a -lm -Wl,-E -ldl -lreadline -lhistory -lncurses
make[1]: Leaving directory `/home/Hajri/Desktop/PFE-STMicroelectronic/Lua_task/Extand lua/lua-5.1.4+directfb lib/src'
[Hajri@STHajri src]$
```

FIG. 3.4.1 – Compilation du code sourc Lua et ldirectfb

```
[root@STHajri lua-5.1.4+directfb lib]# make install
cd src && mkdir -p /usr/local/bin /usr/local/include /usr/local/lib /usr/local/man/man1 /usr/local/share/lua/5.1 /usr/local/l
ib/lua/5.1
cd src && install -p -m 0755 lua luac /usr/local/bin
cd src && install -p -m 0644 lua.h luaconf.h lualib.h lauxlib.h ../etc/lua.hpp /usr/local/include
cd src && install -p -m 0644 liblua.a /usr/local/lib
cd doc && install -p -m 0644 lua.1 luac.1 /usr/local/man/man1
[root@STHajri lua-5.1.4+directfb lib]#
```

FIG. 3.4.2 – Installation du nouvelle Lua

```
[Hajri@STHajri ~]$ lua
Lua 5.1.4  Copyright (C) 1994-2008 Lua.org, PUC-Rio
> directfb.open()
directfblib working > ^C
```

FIG. 3.4.3 – Démarrage et test du Lua

Le figure 3.4.2 montre l'installation du nouvelle Lua qui contient la bibliothèque ldirectfb.

Finallement on démarre Lua et on effectue le test, le figure 3.4.3 monter le résultats d'exécution du test du nouvelle Lua.

La ligne de commande *directfb.open()* dans l'interpréteur Lua signifie que : l'ouverture de ldirectfb (avec la fonction :*LUALIB_API int luaopen_directfb (lua_State *L)*) qui contient le non de la nouvelle lexème open qui pointe vers la fonction C suivante définie dans la bibliothèque ldirectfb.c :

*static int directfb_open (lua_State *L){*

printf("directfblib working ") ;

return 1 ; } ;

qu'elle doit afficher "directfblib working" est effectué avec succès et le message afficher "*directfblib working*" montre que notre extension de Lua marche correctement.

Cette méthode d'extension consiste a ajouté au Lua des bibliothèques interne qu'on doit les compiler avec les codes sources du Lua. La section suivante nous montre une autre méthode d'extension avec les bibliothèques externe lier dynamiquement avec Lua.

3.4.3 Étendre Lua avec une bibliothèque dynamique

Comme nous avons vue dans les deux sections précédentes, l'action d'extension de langage Lua peut être avec une bibliothèque interne qui sera compiler avec le code source de Lua, soit avec une bibliothèque externe Lua qui sera appeler à partir du Lua. Nous présentons dans cette section la dernière méthode d'extension du Lua en utilisant les bibliothèques lier dynamiquement avec l'application.

La base de cette méthode est la fameuse fonction require. Require est une fonction du langage Lua qui charge et exécute les bibliothèques lier dynamiquement lors de son appelle. Pour mieux comprendre voici l'exemple suivant :

Exemple : Soit la bibliothèque dynamique luadirectfb.so. Pour l'appeler a partir du Lua et l'exécuter on utilisera le scripte suivant :

package.cpath= package.cpath.. ' .\ ?.so'

require "luadirectfb "

Expliquant un peut plus. Lorsque nous appelons require avec le nom du module (non de la bibliothèque), require utilise un tableau qui s'appelle package.loaders qui contient un ensemble des fonctions de recherche. Ces fonctions cherchent la bibliothèque en utilisant le chemin enregistrer dans package.cpath. Require utilise le chemin de la manière suivante :

Elle change le point d'interrogation par le nom du module, après il effectue la recherche. Les fonctions de recherche de package.loaders ont le rôle de chercher les modules dans le chemin enregistrer dans package.cpath. On distingue quartes types des fonctions de recherche :

- La première fonction cherche le module dans package.preloaded, c.-à-d. elle vérifie que le module appeler est déjà charger dans Lua ou non. Si oui elle utilise le module déjà charger, si non elle informe la fonction require que le module appeler n'existe pas dans les modules déjà charger.
- La deuxième fonction, elle cherche le module comme étant une bibliothèque Lua en utilisant

le chemin enregistrer dans package.path qui est définie comme suit :
package.path=package.path. ; ';\ ?.lua ; ;\ ?.lc ;/usr/local/include/ ?/init.lua'

Package.path est un chemin d'accèr utiliser pour la recherche des module écrite en Lua.

- La troisième fonction cherche le module comme une bibliothèque écrite en C, appeler bibliothèque C, en utilisant le chemin d'accès enregistrer dans package.cpath comme suit :
  package.cpath =package.cpath.. '.\ ?.so ;.\ ?..dll ;/usr/local/include/ ?/init.so'
- La quatrième fonction s'appele all_in_one loader qui effectue les mêmes opérations que la troisième fonction mais, elle prend en charge le sous-module.

Une fois la bibliothèque trouver, la fonction de recherche lies dynamiquement la bibliothèque avec l'application, ensuite elle ouvre la bibliothèque et se met a la recherche d'une fonction C spéciale pour l'utiliser au chargement de la bibliothèque. Le nom de cette fonction doit être sous la forme suivante : luaopen_ suivie du nom du module. C'est-à-dire pour ouvrir le module luadirectfb, la fonction doit être sous la forme :

LUALIB_API luaopen_luadirectfb(lua_State L)*

Et pour charger les sous-module, la fonction doit être sous la forme luaopen_ suivie du nom du module, suivie du nom du sous-module. Prenons l'exemples suivant :

Le nom du module mére est luadirectfb et le nom du sous-module est IDirectFB. Pour que Lua comprend que IDirectFB est un sous-module du luadirectfb, il faut que le nom du sous-module soit sous la forme luadirectfb.IDirectFB. Pour charger la bibliothèque toute entière, il faut que la bibliothèque contient deux fonctions de chargement. L'une pour charger le module mère (luaopen_luadirectfb) et l'autre pour charger le sous-module (luaopen_luadirectfb_IDirectFB). Le nom du module mère et le sous-module doivent être séparer par un tirés bas « _ ».

Finalement require exécute la bibliothèque et enregistre cette dernière en retournant au Lua un tableau qui est nommée par le nom du module mère et les sous-module comme des champs de ce tableau.

Donc pour étendre Lua avec une bibliothéque C dynamique, il faut écrire les codes sources des modules et les sous-modules en C. Chaque module ou sous-module doit avoire un fichier source à part qui contient la fonction de chargement (luaopen_ suivie du nom du module ou du sous-module), les fonctions et les méthodes de cette bibliothèque et les nouveaux lexèmes par l'intermédiare du tableau *luaL_Reg*. En suite, nous devons écrire une *Makefile* qui sert a compilée les codes sources et génére une bibliothèque dynamique *luadirectfb.so* qui sera appeler ultérieurement dans Lua par

l'intermédiare du fonction require en utilisant le chemin enregistrer dans package.cpath.

3.5 Conclusion

Lua est un langage embarquée extensible, c'est-à-dire on peut l'étendre pour qu'elle supporte des nouveaux fonctions selon nos besoins. On peut l'étendre soit par des bibliothèques interne compiler avec le code source du Lua, soit avec des bibliothèques externe Lua, soit par des bibliothèques externe C dynamique.

PRÉSENTATION DU PLATEFORME DE TRAVAIL

Chapitre 4

PRÉSENTATION DE L'ENVIRONNEMENT DE TRAVAIL

4.1 Introduction

Dans la partie précédente nous avons présenter le cadre du projet et l'interpréteur Lua ainsi que les différentes méthodes d'extension de ce langage. Dans cette partie, nous allons présenter l'environnement de travail matériel et logiciel ainsi que la mise en place de l'environnement de compilation croisée pour le système *STLinux* sur processeur *ST40*.

4.2 Environnement matèriel

4.2.1 Inroduction

L'environnement matériel qui comprend le *Set-Top-Box ST7015DT2* (cible), le *STMicroConnect*, L'ordinateur bureautique Fedora (hôte) et l'installation du réseau local qui lie l'ordinateur avec le *ST7015DT2* et le *STMicroConnect*.

4.2.2 La Set-Top-Box 7105DT2

STi7015 contient une unité centrale de calcul générique *RISC ST40-300* (la nouvelle gamme de *ST40*) à 450Mhz et deux processeurs *ST231* à 450Mhz. Le processeur central *ST40* accéde à toute la mémoire de la puce qui est une mémoire DDR2. La communication entre processeurs se fait en utilisant un autre bloc de la mémoire qui est partagée et liée à un contrôleur d'interruptions. Au niveau le plus bas, la communication entre processeurs passe toujours par le processeur principal ST40. La plate-forme *STi7015* dispose d'un port JTAG à travers duquel il est possible de déployer directement les différents processeurs. Ce port s'utilise à l'aide d'un boîtier spécifique, appelé *STMC2*.

4.2.3 Processeur ST40

ST40 est le processeur central du système sur puce (SoC) *7105DT2*. C'est un processeur de 32-bits de l'architecture SH4 superH[1]. Ce qui permet au SoC de supporter les systèmes d'exploitation des STB. La famille de ST40-300 est caractérisée par sa haute performance. ST40 est choisi maintenant comme processeur pour les courantes et futures solutions embarquées des produits ST et il est supporté par une large gamme d'applications et d'outils de développement. Voici les caractéristiques principaux du processeur *ST40-300* :

- Les instructions et les données sont configurées entre 4K octets et 64K octets ce qui indique l'excellence du système et une bonne utilisation des codes à hautes densités.
- Les systèmes d'exploitation existants pour le ST40 sont OS21, Linux et WinCE.
- L'architecture de ST40 est désignée pour supporter le développement avec le langage C.
- Ce processeur est d'une vitesse 450 MHZ avec une architecture RISC (Reduced Instruction Set Computer)

4.2.4 STMicro-connect

STMicroConnect est une interface intelligente entre hôte et cible qui permet à une hôte, avec une configuration bien spécifique, de se connecter à n'importe quelle plateforme de développement ayant un support de débogage.

L'avantage d'utiliser le STMicroconnect relié à l'Ethernet est qu'il permet au système cible d'être géré à distance et d'être facilement mis en commun entre des utilisateurs. Le STMicroconnect supporte le protocole TCP/IP qui se connecte directement à la plateforme a utilisé et cela à travers un câble JTAG.

4.3 Environnement logiciel

4.3.1 Introduction :

Après la présentation de l'environnement matériel nous allons passer à présenter l'environnement logiciel et les différents outils de base nécessaires pour notre projet.

---

[1]SH-4 est une architecture de microcontrôleur RISC qui est conçue pour une utilisation dans les applications graphiques 3D.

4.3.2 Linux :

Linux désigne une famille de système d'exploitation "Unix-Like", type UNIX utilisant le noyau Linux. La première version du noyau Linux fut publiée en 1991 par Linus Torsvald, père et principal développeur du noyau Linux. Le développement de Linux est l'un des exemples les plus marquants de la collaboration logicielle, ce projet compte des milliers de développeurs (amateurs ou professionnels) partout dans le monde. Le système Linux est un système multi-utilisateurs et multi-tâches. En tant que système d'exploitation, son rôle principal est, donc, d'assurer aux différentes tâches et aux différents utilisateurs une bonne répartition des ressources de l'ordinateur (mémoire, processeur(s), espace disque, imprimante(s), programmes utilitaires...) et cela sans intervention des utilisateurs ; il prend totalement en charge ces utilisateurs et lorsque les demandes sont trop importantes pour être satisfaites rapidement, l'utilisateur le ressent par un certain ralentissement (qui peut être effectivement important, voire insupportable...), mais le système (en principe) ne se bloque pas.

Linux est par ailleurs un système de développement et les utilisateurs y ont à leur disposition un très grand nombre d'outils, pour la plupart assez simples à utiliser, leur permettant d'écrire, de mettre au point et de documenter leurs programmes (éditeurs, compilateurs, débogueurs, système de traitement de textes...).

Le système Linux est composé de :

- Un noyau assurant la gestion de la mémoire et des entrées-sorties de bas niveau et l'enchaînement des différentes tâches ;
- Un (ou plusieurs) interpréteur(s) de langage de commandes ; il existe en effet différents langages de commandes nommés Shell, le plus connu étant le Bourne Shell (du nom de son auteur), un autre étant le C-Shell développé à l'université de Berkeley et le plus répandu actuellement étant le Bash.
- Un grand nombre de programmes utilitaires dont évidemment un compilateur de langage C, des éditeurs, des outils de traitement de textes, des logiciels de communication avec d'autres systèmes Linux (ou autres), des générateurs d'analyseurs lexicaux et syntaxiques...

Quelques caractéristiques de Linux :

- Linux est fiable : L'écran bleu de Windows n'existe pas sous Linux. Les systèmes Linux et Unix peuvent fonctionner pendant des années sans échec (Windows aussi mais tout dépend aussi l'utilisation que vous en faites). Cette fiabilité est due à son noyau qui, d'après plusieurs

études, contient bien moins de bogues que ses concurrents propriétaires.

- Linux fonctionne partout : L'efficacité de Linux et de tous les Unix-oïdes peut être utilisée sur pratiquement tous les ordinateurs même les plus vieux.
- Linux est gratuit : les distributions Linux sont faitent en open source.
- Le support de Linux : Le support est gratuit (pour les distributions non professionnelles).

Nous avant présenter en bref le système Linux ainsi que ces composantes et quelque caractéristiques de ce derniers. La distribution que nous allons utilisé dans notre projet est Fedora 14.

4.3.3 STLinux :

La distribution STLinux est un environnement de développement qui fournit tout le nécessaire pour construire des systèmes basés sur Linux pour les plates-formes à base de ST40. Le noyau peut être porté sur les cartes (plateformes) personnalisées ou utilisées directement sur les plates-formes de référence ST. STLinux est open source, offert sous ces deux formes source et binaire, ce qui rendre facile à étendre et améliorer les capacités de votre plate-forme spécifique. Nous pouvons ajouter des codes en espace noyau (tels que les pilotes de périphériques supplémentaires). Les applications en espace utilisateur, écrites en ANSI C ou C++, peuvent profiter de l'API riche au niveau des applications fournies par Linux. Il contient un puissant ensemble des outils de développement croisé rendre le développement facile. Au cours du développement, STLinux utilise STMicroConnect pour télécharger le noyau, et utilise le serveur NFS pour monter les fichiers racines du système.

Caractéristiques du système STLinux :

• Système d'exploitation Linux open source, les outils et l'environnement de développement basé sur la technologie du noyau Linux 2.6, porté et optimisé pour platesformes à base de *ST40.*

• Un ensemble complet des pilotes pour les périphériques de base du système.

• Un ensemble des outils C et C + +, basé sur la technologie de compilateur GNU pour la compilation croisée et la compilation native.

• Le gestionnaire de mise à jour réseau maintient votre installation à jour avec les dernières versions et mises à jour STLinux.

• Tous les logiciels STLinux peuvent être téléchargés sous forme binaire et source gratuitement à partir du site Web *www.stlinux.com* STLinux STLinux a pour rôle d'établir un lien entre les applications et la plateforme.

L'environnement de développement STLinux est basé sur l'architecture x86 PC Linux comme hôte de développement.

La distribution et les outils sont pris en charge sur :

- Station de travail Red Hat Enterprise Linux 3.
- Distributions Red Hat Fedora libre.

Dans la pratique, des nombreuses distributions Linux sont utilisables.

Une grande variétée de plates-formes cibles de référence STMicroelectronics est prise en charge par le noyau. Par exemple :

- STi7100/7109
- STi7109-REF
- STi7109-REF
- STi7200-MBOARD
- STi7111-MBOARD
- STi7105-MBOARD
- STi7141-MBOARD
- STi5202-MBOARD
- STB7109 HD

4.3.4 DirectFB :

Introduction :

DirectFB est une bibliothèque graphique se plaçant au-dessus du périphérique framebuffer de Linux. Il offre des meilleures performances du matériel avec un minimum d'utilisation des ressources. DirectFB prend en charge les opérations graphiques suivantes :

- remplissage / dessin de Rectangle
- remplissage / dessin de Triangle
- Dessin
- (tendue) blitting
- mélange avec une alphachannel
- Le mélange avec un facteur alpha Neuf différentes fonctions de mélange, respectivement, pour la source et de destination, si toutes les règles Porter /Duff sont pris en charge.
- Coloriser (modulation de la couleur alias)

- saisie de couleur Source
- saisie de couleur de destination

DirectFB a son propre gestionnaire des ressources pour la mémoire vidéo. Les ressources comme les couches d'affichage ou les périphériques d'entrée peuvent être verrouillé pour un accer exclusif, par exemple pour les jeux en plein écran, DirectFB fournit une abstraction pour les objectifs différents graphiques comme des couches d'affichage, des fenêtres et toutes les surfaces à usage général.

Architecture des API de DirectFB :

L'API de DirectFB est structuré en utilisant des interfaces. Une interface est une structure C qui contient les pointeurs de fonction. Ces pointeurs poindrent aux fonctions de base de DirectFB en fonction de l'implémentation de l'interface de DirectFB. Par exemple les pointeurs de l'interface IDirectFBDisplayLayer poindre vers les fonctions de base différentes au fonction pointé par les pointeurs de l'interface IDirectFBSurface.

IDirectFB est Super l'interface de DirectFB, qui est la seule qui peut être créer par la fonction global (DirectFBCreate ()). Toutes les autres interfaces sont accessibles via cette interface. Les nouvelles interfaces peuvent être créer en appelant une fonction de l'interface IDirectFB (par exemple IDirectFBSurface).

IDirectFB : La figure 4.3.1 montre l'architecture des interfaces de la bibliothèque *DirectFB*. *IDirectFB* est le superinterface de la bibliothèque directfb. Il représente l'intermédiaire entre l'application et la bibliothèque, c. à-d. Pour accéder et utiliser, par exemple, les fonctions d'*IDirectFBSurface* il faut tout d'abord créer un superinterface avec la fonction *DirectFBCreate()* ensuite créer une surface à fin d'utiliser ces fonctions pour la manipulation des surfaces.

IDirectFBSurface : C'est une interface de la bibliothèque directfb qui permet, après sa création, de créer des surfaces. Après la création de cette surface, cette interface nous donne l'opportunité d'utiliser des fonctions qui effectuent des manipulations sur cette surface. Ces fonctions effectuent des opérations de dessin et de remplissage. Aussi elles agitent sur les couleurs et effectuent des opérations de translation sur des parties de la surface ou sur la surface tout entière.

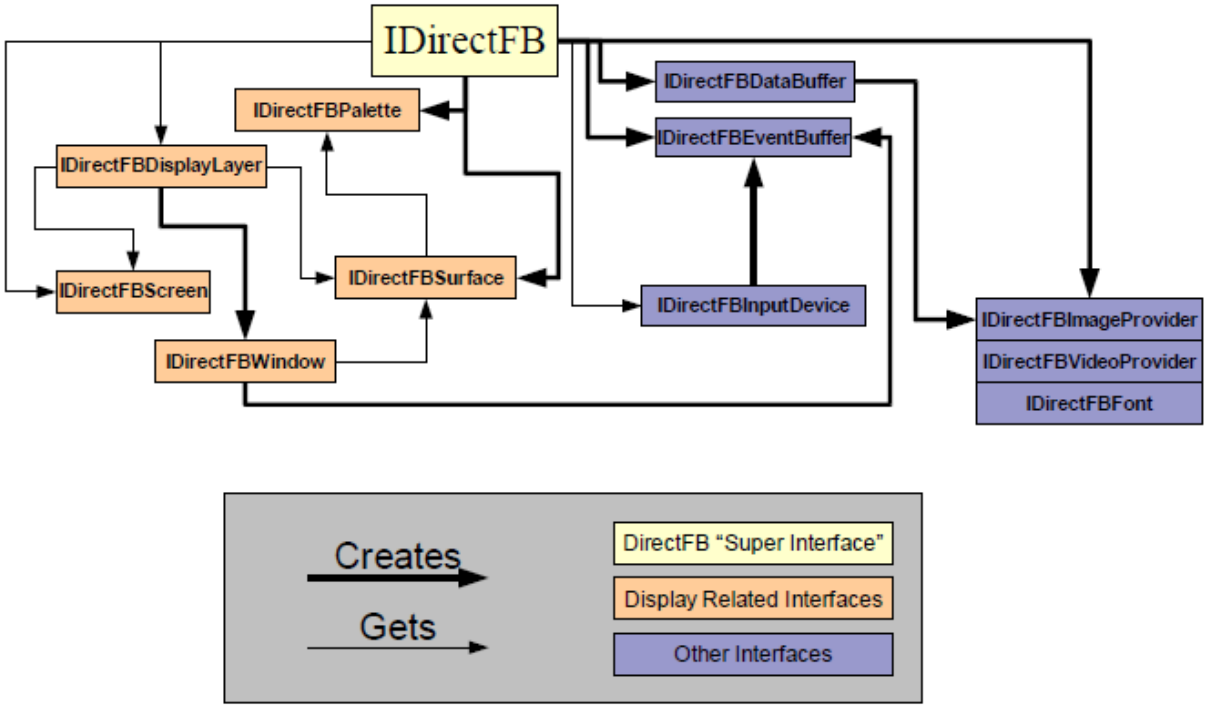

Fig. 4.3.1 – Architecture des API de DirectFB

IDirectFBImageProvider : C'est une interface de la bibliothèque directfb qui offre l'opportunité d'utiliser des images. Il nous permet de charger des images de différentes extensions à partir d'un emplacement spécifique. Après le chargement de l'image, nous pouvons effectuer sur elle des opérations d'agrandissement ou de translation.

IDirectFBDisplayLayer : L'écran d'affichage, pour la bibliothèque directfb, est divisé sur plusieurs couches. Les fonctions de cette interface, nous permettons d'effectuer des opérations sur les couches de l'écran, d'identification changement de l'ordre, la transparence, etc.

4.4 Conclusion :

Après la présentation de l'environnement matériel nous allons passer à présenter l'environnement logiciel et les différents outils de base nécessaires pour notre projet

Chapitre 5

MISE EN PLACE DE L'ENVIRONNEMENT DE COMPILATION CROISÉ

5.1 Introduction

Après la présentation de l'environnement matériel et logiciel, la première tache à effectué et la mise en place d'un environnement de compilation croisée pour le système STLinux sur processeur ST40. Cette tâche est nécessaire pour la suite du projet, elle se résume dans l'installation du STLinux et ces outils sous la machine host Fedora 14, ensuite la configuration du noyau du STLinux pour le faire tourner sur la carte 7015DT2. Après la configuration, nous allons procéder à la compilation et enfin en démarre le système sur la carte (target) 7015DT2. Ce chapitre présente les différentes étapes et outils pour la mise en place de cette environnement de développement.

5.2 Installation du STLinux

La première étape est l'installation du STLinux et ces outils sous le host machine Fedora 14. Pour effectuer cette tâche, on utilise l'internet pour l'installation à travers le serveur ftp de STLinux pour télécharger une image du CD su STLinux. Pour télécharger une image iso du système STLinux, nous allons connecter au serveur ftp en utilisant l'adresse *ftp// :ftp.stlinux.com/pub/stlinux*. En suite nous allons sélectionner la version du système qu'on doit télécharger (par exemple pour télécharger STLinux-2.3 nous allons choisir le dossier 2.3). Pour télécharger l'image iso il suffit de sélectionner le dossier iso ensuite sélectionner la version requis (par exemple STLinux-2.3-sh4-uclibc-03-11-07.iso). En fin, nous allons enregistrer cette copie sur le disque dur. Après la fin du téléchargement il suffit de graver cette image sur une DVD en suite, il suffit d'accéder au DVD et effectuer l'installation avec la commande Shell install -all-sh4-glibc.
Après l'installation de STLinux et ces outils (sh4-linux-gcc, sh4-linux-g++,etc...), nous allons passer à la phase de configuration des fichiers source du noyau du *STLinux* pour le target *7015DT2*.

5.3 Configuration et compilation du noyau de système STLinux

La deuxièmes étape du préparation de l'environnement de compilation croisé est la configuration du noyau du STLinux pour le target 7015DT2 avant la compilation. Nous allons commencer

par la copie des fichiers sources du noyau du STLinux en utilisant le script suivant :

```
[root@STHajri ~]# cd /opt/STM/STLinux-2.3/devkit/sources/kernel/
[root@STHajri kernel]# cp -r linux-sh4-2.6.23.17_stm23_0125/ /home/Hajri/Desktop/STLINUX/
```

Ensuite nous allons configurés le noyau avec le script suivant, mais on doit changer le Makefile dans le shell en utilisant l'outil " vi ". Les erreurs dans le Makefile sont deux :

- La première erreur est dans la ligne 417, on doit changer le script suivant :

  ```
  config %config : scripts_basic outputmakefile FORCE
  $(Q)mkdir -p include/linux include/config
  $(Q)$(MAKE) $(build)=scripts/kconfig $@
  ```

  par le script suivant :

  ```
  config : scripts_basic outputmakefile FORCE
  $(Q)mkdir -p include/linux include/config
  $(Q)$(MAKE) $(build)=scripts/kconfig $@
  %config : scripts_basic outputmakefile FORCE
  $(Q)mkdir -p include/linux include/config
  $(Q)$(MAKE) $(build)=scripts/kconfig $@
  ```

- La deuxième erreur est dans la ligne 1493 , on doit changer le script suivant :

  ```
  / %/ : prepare scripts FORCE
  $(Q)$(MAKE) KBUILD_MODULES=$(if $(CONFIG_MODULES),1)
  \ $(build)=$(build-dir)
  ```

  par le script suivant :

  ```
  / : prepare scripts FORCE
  $(Q)$(MAKE) KBUILD_MODULES=$(if $(CONFIG_MODULES),1)
  \ $(build)=$(build-dir)
  %/ : prepare scripts FORCE
  $(Q)$(MAKE) KBUILD_MODULES=$(if $(CONFIG_MODULES),1)
  \ $(build)=$(build-dir)
  ```

Comme le Makefile est protéger contre l'écriture, on doit effectuer les changement à partir du shell en utilisant l'outil vi de la manière suivant :

```
[root@STHajri STLINUX]# vi Makefile
```

Après la modification du Makefile on utilisera les commandes suivant pour lancer la configuration du noyau :

```
[root@STHajri STLINUX]# export PATH=/opt/STM/STLinux-2.3/devkit/sh4/bin/ :$PATH
[root@STHajri STLINUX]# make ARCH=sh CROSS_COMPILE=sh4-linux- menuconfig
```

Le résultat est une fenêtre de configuration du noyau présenté dans la figure 5.3.1.

Puisque la carte, qu'on y doit tourner notre système, est dédié, nous devons effectuer deux patchs pour ajouter la description du carte au configuration du noyau STLinux. Pour effectuer

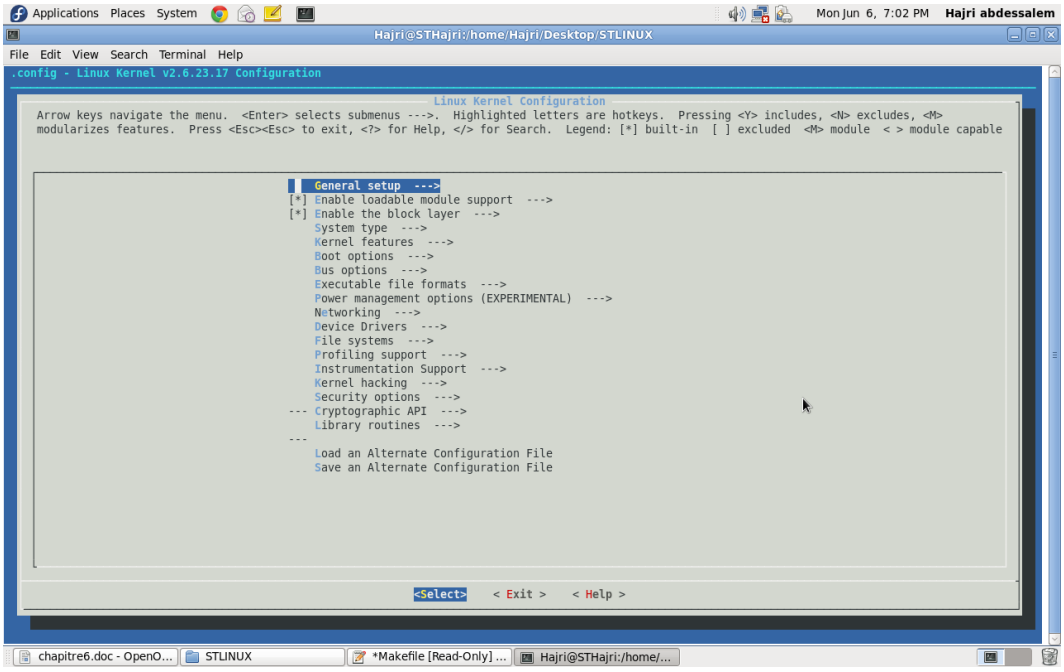

FIG. 5.3.1 – Fenêtre de configuration du noyau STLinux

ces deux patchs, on doit les copier dans le répertoire du compilation ensuite les exécuter de la manière suivante :

[root@STHajri STLINUX]# cp /home/Hajri/Desktop/PFE-STMicroelectronic/STlinux/STlinuxkernel_sh4_build/0001-7105dt2-kernel-configuration.patch /home/Hajri/Desktop/STLINUX/

[root@STHajri STLINUX]# cp /home/Hajri/Desktop/PFE-STMicroelectronic/STlinux/STlinuxkernel_sh4_build/0002-7105dt2-platform-startup-code.patch /home/Hajri/Desktop/STLINUX/

[root@STHajri STLINUX]# patch -p1 <0001-7105dt2-kernel-configuration.patch

patching file arch/sh/Kconfig

patching file arch/sh/Makefile

patching file arch/sh/configs/7105dt2_defconfig

[root@STHajri STLINUX]# patch -p1 <0002-7105dt2-platform-startup-code.patch

patching file arch/sh/boards/st/7105dt2/Makefile

patching file arch/sh/boards/st/7105dt2/setup.c

Maintenant la configuration de notre carte est ajouté au fichiers de configuration par défaut des cartes (target) de STMicroelectronics. Donc, il suffit de chercher la fichier de configuration et l'appliquer.

[root@STHajri STLINUX]# make ARCH=sh CROSS_COMPILE=sh4-linux- help |grep dt2

7105dt2_defconfig - Build for 7105dt2

[root@STHajri STLINUX]#

[root@STHajri STLINUX]# make ARCH=sh CROSS_COMPILE=sh4-linux- 7015dt2_defconfig

Après cette étape les fichier sources du noyau du STLinux sont configurés avec succès pour notre carte 7015DT2. Donc, il nous reste que la compilation des fichiers sources pour générer une

image des fichiers système, pour cela nous allons utiliser la commande suivante :

```
[root@STHajri STLINUX]# make ARCH=sh CROSS_COMPILE=sh4-linux- vmlinux
```

Notre image du noyau du système STLinux est prêt pour le démarrage mais avant qu'on puisse le démarrer il faut préparer le script de chargement d'image du noyau sur la carte et démarrer le services nfs et installer l'outil Putty. Les fichiers sources du noyau du systèmes STLinux, pour la carte 7015DT2, sont partager entre le système host Fedora 14 et le système embarqué STLinux tournant sur la carte a travers le réseau. Le service nfs contrôle l'accès à ces fichiers, c'est pour cela qu'on doit le démarrer avant le démarrage du système embarqué.

5.4 Démarrage du système embarqué STLinux

Après la configuration et la compilation du noyau système STLinux pour la carte 7015DT2, et avant de procéder au démarrage du noyau, nous allons écrire le script de chargement. Puisque la communication entre la carte 7015DT2 (cible) et l'ordinateur (host machine Fedora) est établie a travers un réseau local, le script de démarrage doit spécifier les adresses IP du host machine, du target, du STMicroconnect, le getway et le masque du réseau. Il doit aussi spécifier le chemin d'accès des fichiers sources du système pour la carte 7015DT2 partager et le chemin d'accès de l'image du noyau du système STLinux compilé ultérieurement. Enfin le script de chargement doit spécifié les règles de chargement de l'image du noyau du système STLinux.

l'algorithme 10 montre le fichier shell du loadscript :

Maintenant pour charger l'image du noyau du système STLinux il suffit d'exécuter le load-script.sh par la commande suivante :

[Hajri@STHajri STlinuxkernel_sh4_build]$ sh loadscript.sh

Après le chargement et le démarrage du noyau STLinux sur la carte 7015DT2, nous allons démarrer le shell du système embarqué sous la machine host à travers le réseau local. Pour cela, nous allons utiliser l'outil Putty pour recevoir les données de la part du système embarqué STLi-nux. Mais avant de démarrer le shell il faut tout d'abord démarrer le service nfs, cette action doit être exécuté comme super utilisateur (root) par la commande suivant :

```
[root@STHajri STlinuxkernel_sh4_build]# /etc/init.d/nfs restart
Shutting down NFS mountd:                                  [FAILED]
Shutting down NFS daemon:                                  [FAILED]
Shutting down NFS quotas:                                  [FAILED]
Starting NFS services:                                     [  OK  ]
Starting NFS quotas:                                       [  OK  ]
Starting NFS daemon:                                       [  OK  ]
Starting NFS mountd:                                       [  OK  ]
[root@STHajri STlinuxkernel_sh4_build]#
```

Algorithme 10 Fichier shell du script du démarrage

```
#!/bin/sh
JEI=192.168.2.123
TARGETIP=192.168.2.124
SERVERIP=192.168.2.120
GWIP=192.168.1.1
NETMASK=255.255.255.0
NAME=Soussa
AUTOCONF=off
# Root of target's file system
SERVERDIR=/opt/STM/STLinux-2.3/devkit/sh4/target
# Kernel image
KERNEL=/home/lab1/linux-sh4-2.6.23.17_stm23_0125-7105/vmlinux
/opt/STM/STLinux-2.3/devkit/sh4/bin/st40load_gdb \
        -t $JEI :iptv7105 :st40,seuc=1,silent=1,no_convertor_abort=1 \
        -b $KERNEL \
        console=ttyAS1,115200 \
        root=/dev/nfs \
        nfsroot=$SERVERIP :$SERVERDIR,nfsvers=2,rsize=4096,wsize=8192,tcp \
        ip=$TARGETIP : :$GWIP :$NETMASK :$NAME : :$AUTOCONF ipconfdelay=3 \
        nwhwconf=device :eth0,hwaddr :00 :80 :e3 :12 :7f :7d \
        bigphysarea=2000
```

Après le démarrage du service nfs, il suffit d'exécuter le script de chargement représenter dans l'algorithme 10.

Le logiciel Putty, nous permet d'établir une voie de communication entre les deux systèmes (target système, host système). Pour le faire, il suffit de lancer le logiciel Putty,après quelque minute, ensuite ouvrir une nouvelle session avec l'adresse IP du TARGET. Esuite, le shell du noyau du système embarqué STLinux sur la carte 7015DT2 démarre .

Après le démarrage du shell du noyau, il nous reste que d'entrer le mot de passe : root.

5.5 Conclusion

La préparation de l'environnement de compilation croisé du système STLinux pour le processeur ST40 se résume dans l'installation du système sous le host machine Fedora, ensuite la configuration du noyau et la compilation des fichiers sources du noyau et la génération de l'image du noyau pour le charger ensuite sur la carte (target) 7015DT2. Enfin on charge et on démarre l'image du noyau du système. Nous allons passer ensuite à la compilation native du langage Lua pour le processeur ST40 et le développement de l'interface entre la bibliothèque DirectFB et Lua.

CONCEPTION DE L'INTERFACE LUADIRECTFB

Chapitre 6

LA COMPILATION NATIVE DE LUA POUR PROCESSEUR ST40

6.1 Inroduction

Dans les section précédente, nous avons vue que Lua peut être compiler pour plusieurs platform comme linux, MacOS, etc. Pour l'installation du Lua sous Linux nous avons utiliser le Makefile prédéfinie, mais cette dernière ne définie pas les règles de compilation du code source du Lua pour sh4 du processeur ST40. La compilation native du Lua pour le processeur ST40 est la compilation des codes sources en utilisant le compilateur du linux pour l'architecture SH4 qui est sh4-linux-gcc

6.2 Préparation du Makefile et compilation

Make est un logiciel traditionnel d'UNIX. C'est un « moteur de production » : il sert à appeler des commandes créant des fichiers. Make exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes. Il sert principalement à faciliter la compilation et l'édition de liens puisque dans ce processus le résultat final dépend des opérations précédentes. Pour ce faire, make utilise un fichier de configuration appelé makefile qui porte souvent le nom de Makefile. Ce dernier décrit des cibles (qui sont souvent des fichiers, mais pas toujours), de quelles autres cibles elles dépendent, et par quelles actions (des commandes) y parvenir. Les règles de dépendance peuvent être explicites (noms de fichiers donnés explicitement) ou implicites (via des motifs de fichiers ; par exemple, fichier.o dépend de fichier.c, si celui-ci existe, via une recompilation).
Pour que nous puissions compiler Lua pour l'architecture SH4 du processeur ST40, nous allons préparer une Makefile qui définit les règles de compilation de ce langage. Comme nous avons vue dans le chapitre 3 la partie qui montre les différentes étapes d'installation du Lua (compilation du code source pour l'architecture du processeur i686 et installation), nous pouvons installer Lua sur plusieurs plateformes comme Linux MacOS...

L'idée est d'ajouter au Makefile les règles de compilation d'une nouvelle plateforme que nous le nome STLinux .

PLATS= aix ansi bsd freebsd STLinux linux macosx mingw posix solaris

En suite il suffit d'activer les flags (drapeau qui indique les options supplémentaires de compilation) DLUA_USE_DLOPEN et ldl pour activer le chargement des bibliothèques dynamiques. Voici la règle de compilation de Lua pour la nouvelle plateforme STLinux : STLinux :

```
$(MAKE) all MYCFLAGS="-DLUA_USE_DLOPEN" MYLIBS="-ldl -Wl,-E"
```

Après l'ajout de ces règles de compilation il suffit d'exécuter la ligne de commande suivant :
[Hajri@STHajri src]$ make STLinux
À cette étape le fichier exécutable Lua pour l'architecture sh4 du processeur ST40, il nous reste maintenant, que d'effectuer le portage sur la cible 7015DT2 ensuite effectuer une série des testes pour s'assurer que Lua fonctionne correctement .

6.3 Portage et test du Lua sur processeur ST40

La compilation du code source du Lua, pour la génération d'un fichier exécutable pour l'architecture sh4 du processeur ST40, est terminée. Maintenant nous allons passer au portage qui consiste tout simplement à copier le fichier exécutable générer précédemment dans le dossier des fichiers système, tournant sur la cible, root, voici le chemin de ce dossier :

/opt/STM/STLinux-2.3/devkit/sh4/target/root

ensuite, accéder au Shell de la cible 7015DT2 (le chapitre 6 montre comment effectue cette étape) et exécuter Lua :
./lua
Le test du Lua sur le processeur ST40 sera effectuer dans le chapitre suivant.

6.4 Conclusion

L'action de portage du langage Lua sur le processeur ST40 se résume en deux étapes. La première étape consiste a configuré le Makefile de compilation des fichiers sources du Lua pour l'architecture ST40. La deuxième étape et la copie du fichier exécutable générer dans le dossier root des fichiers systèmes STLinux.

Chapitre 7

DÉVELOPPEMENT DE L'INTERFACE LUA ET DIRECTFB

7.1 Introduction

Dans les chapitres précédents, nous avons présenter les différentes outils nécessaire pour le développement de l'interface entre Lua et DirectFB. Ce chapitre consiste a présenté l'application et les différentes techniques utilisées dans le processus de développement.

7.2 Présentation de l'application

L'idée est de développer une bibliothèque dynamique codée en C qui sert a effectué des opérations graphique en utilisant la bibliothèque DirectFB et ces interfaces à partir du Lua. L'objectif est d'exécuter les fonctions suivantes de l'interface IDirectFBSurface, IDirectFBImageProvider et IDirectFBDisplayLayer de DirectFB à partir du Lua.

Pour le faire nous allons développer une bibliothèque dynamique "luadirectfb.so" qui vas nous permettre d'exécuter ces fonctions à partir de Lua. À cause de l'architecture des Interfaces de DirectFB, présenter ultérieurement dans le chapitre 4, nous allons utiliser la notion de module du Lua. Nous allons définir luadirectfb comme le module mère de la bibliothèque, et IDirectFB, IDirectFBSurface, IDirectFBImageProvider et IDirectFBDisplayLayer comme des sous-modules du module mère.

les fonctions spécifiées ultérieurement seront définies dans les sous-modules, IDirectFB, IDirectFBSurface, IDirectFBImageProvider et IDirectFBDisplayLayer. La technique de développement va être présenter dans la section suivante.

7.3 Technique de développement

Nous avons vue dans le chapitre 4 (les méthodes d'extension du langage Lua) les différentes techniques qui nous permettons d'étendre Lua. L'un de ces technique est les bibliothèques dynamiques codé en C, pour cela nous allons utiliser la fonction require. Comme nous avons vue dans la section étendre lua avec une bibliothèque dynamique du chapitre 4, pour utiliser une bibliothèque externe dans un scripte Lua il faut que Lua la charge et l'exécute en utilisant la fonction "require"

comme suite : *require"luadirectfb"*
La fonction require vas utiliser "package.Loaders[1] " pour chercher la bibliothèque luadirectfb.so dans le chemin décrit dans package.cpath qui est sous la forme "./. ?so". La fonction require utilise la fonction de recherche "all-in-one" du tableau package.loaders. Puis la fonction require change le point d'interrogation par le nom de la bibliothèque ensuite efféctue la recherche. Une fois trouver, require charge et lies dynamiquement la bibliothèque avec Lua, ensuite elle cherche les fonctions luaopen_luadirectfb,
lua_open_luadirectfb_IDirectFB,
luaopen_luadirectfb_IDirectFBSurface,
luaopen_luadirectfb_IDirectFBImageProvider et
luaopen_luadirectfb_IDirectFBDisplayLayer, dans la bibliothèque pour charger le module luadirectfb, luadirectfb.IDirectFB, luadirectfb.IDirectFBSurface,
luadirectfb.IDirectFBImageProvider et luadirectfb.IDirectFBDisplayLayer. En fin require retourne au Lua un tableau avec luadirectfb comme nom et les sous-modules comme des champs de ce tableau. La procédure de chargement consiste a enregistrée les modules, leurs fonctions, ces méthodes et les fonctions qui crées les objets de directFB dans Lua.
Nous allons commencer tout d'abord par enregistrer la module mère luadirectfb ensuite enregistrer les sous-modules IDirectFB IDirectFBSurface, IDirectFBDisplayLayer et IDirectFBImageProvider à travers le module mére. Comme nous avons vu dans le chapitre 3 (Étude du langage Lua) lua utilise le "stack" pour communiquer avec C, les arguments et les résultats sont passés de lua vers C à travers le "stack" ou dans le sens inverse. A cause de l'incompatibilité des types entre Lua et DirectFB nous allons utiliser le type userdata pour passer les arguments et les résultats de et vers lua à traver le stack.

7.3.1 Le module luadirectfb

Le module luadirectfb est spécifier dans le fichier luadirectfb.c. Comme nous avons décrit ultérieurement, la procédure de chargement et l'enregistrement du module luadirectfb avec ses fonctions ce fait à traver la fonction luaopen_luadirectfb qui doit créer le tableau retourner au Lua. Ensuite,la fonction luaL_register enregistre les fonctions de luadirectfb en utilisant luaL_Reg dfbfunctions (ces tableaux utilisés pour énumérer les fonctions de luadirectfb). L'algorithme 11

[1]C'est un tableau du lua qui contient des fonction de recherche des module

Algorithme 11 Fonction d'enregistrement de module luadirectfb

```
static const luaL_Reg dfbfunctions[]={
{"init",lua_dfbinit},
{NULL,NULL}
};
LUALIB_API int luaopen_luadirectfb(lua_State* L){
lua_newtable(L);
luaL_register(L,"luadirectfb",dfbfunctions);
}
```

décrit la fonction qui enregistre le module luadirectfb et le tableau des fonctions :

Alors pour initialiser DirectFB il faut écrire la fonction *lua_init(),* ensuite l'énumérer dans le tableau *luaL_Reg dfbfunctions []* en spécifions la chaine de caractère utiliser par Lua pour appeler la fonction lua_init(), par exemple nous allons utiliser la chaine « init» pour appeler *lua_init().* Donc pour exécuter lua_init à partir du Lua, il faut utiliser le script :

> *luadirectf b.init()*

Pour initialiser directfb il faut tout d'abord appeler *DirectFBInit(NULL,NULL)* pour créer une superinterface à l'aide de la fonction *DirectFBCreate(dfb).* Donc, la fonction d'initialisation doit retourner le pointeur qui pointe vers la superinterface créer. Pour que nous puisions créer la superinterface dfb il faut appeler le sous-module luadirectfb.IDirectFB pour cela, nous avons définir une fonction qui sert a appeler tous les sous-modules du module luadirectfb. Pour créer un objet de DirectFB il faut recevoir les arguments de Lua , réserver un espace mémoire pour l'objet, créer cet objet et retourner un pointeur qui pointe vers l'objet créer. Donc pour créer un objet DirectFB il faut appeler le sous-module associée (c.-à-d., si l'objet et une surface il faut appeler luadirectfb.IDirectFBSurface), ensuite l'enregistrer dans Lua, enfin appeler la fonction qui crée l'objet. Lua nous offre un type des fonctions spécifique pour ce genre de traitement, c'est le type lua_CFunction. Pour enregistrer un sous-module, on a besoin de trois fonctions :

- Une fonction qui sert a appelée le module et créer les objets. C'est la fonction lua_ClasseConsructor qui prend comme argument le nom de sous-module.
- Une fonction de types lua_CFunction qui crée les objets. C'est la fonction luadirectfb_Cfunc.
- Une fonction qui enregistre le sous-module. C'est luaopen_module qui prend comme argument le nom de sous-module, le tableau des fonctions à enregistrer, le tableau des méthodes à enregistrer et la fonction luadirectfbCfunc à appeler.

Algorithme 12 Fonction d'initialisation de DirectFB

```
int lua_dfbinit (lua_State* L){
DirectFBInit(NULL,NULL);
/*construction de la superinterface IDirectFB*/
lua_classeconstructor(L,"luadirectfb.IDirectFB");
/*transformation de l'objet IDirectFB super interface en userdata type*/
IDirectFB** dfb = (IDirectFB**) lua_touserdata(L, -1);
/*retourner le superinterface*/
lua_pushvalue(L,-1);
return 1;
}
```

L'algorithme 12 défini la fonction luaopen_luadirectfb, la fonction d'initialisation de DirectFB lua_init() et le tableau luaL_Reg dfbfunctions

L'algorithme 13 décrit la fonction luaopen_module qui enregistre les sous-modules de notre bibliothèque.

Le cahier de charge indique que notre bibliothèque doit garder l'aspect orienté objet de directfb.C'est pour cela que nous avons utilisé .

Le module luadirectfb sert aussi à définir les fonctions de conversion des types. Ces dernières servent à convertir les arguments passés de Lua vers directFB ou dans le sens contraire. Comme les objets de directFB (les surfaces, les layers,etc.) sont définis par des structures et Lua ne les connait pas, nous avons choisir de passer les arguments de Lua vers DirectFB sous forme de tableau ensuite effectuer des opérations de conversion à l'aide des fonctions de conversion. L'algorithme 14 présente l'une de ces fonctions qui effectue la conversion du tableau vers *DFBRectangle*.

Le principe de conversion est simple, nous allons extraire les arguments un par un du Stack, ensuite les affecter aux champs correspond de la structure DFBRectangle. Toutes les fonctions fonctionnent de la même manière.

Dans cette section, nous avons présenter le module luadirectfb et les différents concepts de développement pour nous assurer que le chargement des sous-modules et ces différentes fonctions et méthodes sont enregistrés correctement. Nous allons passer à la présentation des différents sous-modules de notre bibliothèque (luadirectfb. IDirectFB, luadirectfb.IDirectFBSurface, luadirectfb.IDirectFBImageProvider, luadirectfb.IDirectFBDisplayLayer) et de présenter les différentes techniques utilisées pour le développement de ces interfaces.

Algorithme 13 Fonction d'enregistrement des sous-modules du luadirectfb

```
LUALIB_API int luaopen_module(lua_State*L,const luaL_Reg* dfbmethods,const luaL_Reg* dfbfunctions,
lua_CFunction lua_directfbCfunc){
const char* module = luaL_checkstring(L, -1) ;
luaL_newmetatable(L,module) ;
lua_pushvalue(L, -1) ;
/*enregistrement des méthode du module*/
lua_setfield(L, -2, "__index") ;
luaL_register(L, NULL, dfbmethods) ;
lua_pop(L,1) ;
/*enregistregistrement des fonctions du module*/
luaL_register(L,module,dfbfunctions) ;
lua_newtable(L) ;
/*enregistrer la fonction C qui crées l'objet */
lua_pushcfunction(L,lua_directfbCfunc) ;
lua_setfield(L,-2,"__call") ;
lua_setmetatable(L,-2) ;
lua_remove(L,-2) ;
return 1 ;
}
```

Algorithme 14 La fonction StackvaluetoDFBRectangle

```
/*fonction de transformation du tableau vers la structure DFBRectangle*/
void StackvaluetoDFBRectangle(lua_State* L, DFBRectangle* rect)
{
int index= lua_gettop(L) ;
lua_getfield(L, index, "x") ;
rect->x=luaL_checknumber(L, -1) ;
lua_getfield(L, index, "y") ;
rect->y=luaL_checknumber(L, -1) ;
lua_getfield(L, index, "w") ;
rect->w=luaL_checknumber(L, -1) ;
lua_getfield(L, index, "h") ;
rect->h=luaL_checknumber(L, -1) ;
lua_settop(L, index) ;
}
```

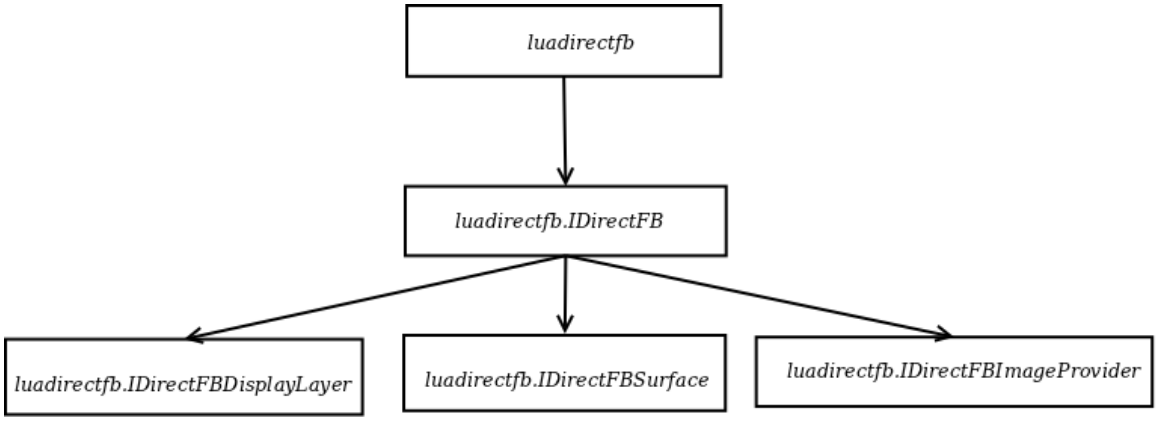

FIG. 7.3.1 – Architecture de luadirectfb

7.3.2 Le sous-module IDirectFB

Nous avons vue dans le chapitre 5 que la bibliothèque DirectFB possède des interfaces entre la bibliothèque est les applications qui sont accessibles à travers l'interface DirectFB. DirectFB est la superinterface de la bibliothèque directfb qui à partir de lui en accède aux interfaces IDirectFB-Surface, IDirectFBImageProvider, IDirectFBDisplayLayer. Alors, le fichier source qui décrit le sous-module DirectFB doit contenir les fonctions qui créent ces sous-modules. La figure 7.3.1 montre l'architecture de notre bibliothèque et le lien entre ces différentes composantes.

L'algorithme 15 montre le code source des fonctions d'enregistrement des sous-modules IDirectFBSurface, IDirectFBImageProvider, IDirectFBDisplayLayer :

Pour la création de la superinterface IDirectFB, nous allons utiliser la fonction DirectFBCreate de directfb. Comme nous avons expliquer dans la section précédente, la fonction lua_directfbCfunc de type lua_CFunction qui crée les interfaces de directfb. Mais avant la création de l'objet, il faut lui réserver de l'espace mémoire, ensuite l'associer à son metatable. L'algorithme 16 définit la fonction lua_directfbCfunc de l'interface IDirectFB.

Les méthodes à enregistrer dans le sous-module IDirectFB sont les fonctions de directfb qui permettent de créer les interfaces IDirectFBSurface, IDirectFBImageProvider et IDirectFBDisplayLayer. Pour créer ces dernières, nous devons développer des fonctions qui font appelle à la fonction *lua_ClasseConstructor* qui ouvre le sous-module correspondant aux interfaces voulues et enregistre ces méthodes et ces fonctions. L'algorithme 16 montre ces fonctions.

Après le développement de ces fonctions, nous devons leur associer des nouveaux lexèmes pour que nous puisions les appeler à partir du Lua. Pour le faire, nous allons définir ces fonctions

Algorithme 15 Fonctions d'enregistrement des sous-modules (IDirectFB.c)

```
/*création du sous-module IDirectFBImageProvider*/
LUALIB_API int luaopen_luadirectfb_IDirectFBImageProvider(lua_State*L);
static int lua_CreateImageProvider(lua_State*L){
lua_classeconstructor(L,"luadirectfb.IDirectFBImageProvider");
return 1;
}
/*création du submodule IDirectFBSurface*/
LUALIB_API int luaopen_luadirectfb_IDirectFBSurface(lua_State*L);
static int lua_CreateSurface(lua_State*L){
lua_classeconstructor(L,"luadirectfb.IDirectFBSurface");
return 1;
}
/*création du sous-module IDirectFBDisplayLayer*/
LUALIB_API int luaopen_luadirectfb_IDirectFBDisplayLayer(lua_State*L);
static int lua_GetDisplayLayer(lua_State*L){
lua_classeconstructor(L,"luadirectfb.IDirectFBDisplayLayer");
return 1;
}
```

Algorithme 16 La fonction lua_directfbCfunc d'IDirectFB

```
/*Création de l'objet directfb superinterface dfb*/
static int lua_directfbCfunc(lua_State* L){
/*allocation du mémoire*/
IDirectFB** dfb=(IDirectFB**) lua_newuserdata(L,sizeof(IDirectFB*));
/*association du metatable du submodule IDirecFB à l'objet dfb*/
luaL_getmetatable(L,"luadirectfb.IDirectFB");
lua_setmetatable(L, -2);
/*création du super interface*/
DFBCHECK(DirectFBCreate (dfb));
return 1;
}
```

Algorithme 17 Fonctions d'enregistrement du sous-module IDirectFB avec ces fonctions et ces méthodes.

```
/*association des nouveaux lexèmes aux méthodes du sous-module IDirectFB*/
static const luaL_Reg dfbmethods[]={
{ "Release", lua_Release},
{ "CreateImageProvider",lua_CreateImageProvider},
{ "GetDisplayLayer", lua_GetDisplayLayer},
{ "SetCooperativeLevel",lua_SetCooperativeLevel},
{ "CreateSurface", lua_CreateSurface},
{ NULL, NULL}
};
static const luaL_Reg dfbfunctions[] = {
{NULL, NULL}
};
/*ouverture et enregistrement du sous-module IDirectFB et de ces fonctions */
LUALIB_API int luaopen_luadirectfb_IDirectFB(lua_State*L) {
luaopen_module(L,dfbmethods,dfbfunctions,lua_directfbCfunc);
return 1;
}
```

comme des méthodes d'IDirectFB pour garder l'aspect orienté objet de la bibliothèque directfb. Ces nouveaux lexèmes sont les même utilisées par la bibliothèque directfb. Maintenant, il nous reste que de définir la fonction d'enregistrement du sous-module IDirectFB. Pour que l'enregistrement de sous-module, avec ces méthodes et ces fonctions, soit correctement effectué la fonction *luaopen_luadirectfb_IDirectFB* doit faire appelle à la fonction luaopen_module. L'algorithme 17 montre ces fonctions.

Après le développement de sous-module IDirectFB, nous allons passer aux sous-modules IDirectFBSurface. Cet ordre est précisé par le cahier de charge du projet.

7.3.3 Le sous-module IDirectFBSurface

IDirectFBSurface est une interface de la bibliothèque directfb. Comme le sous-module luadirectfb.IDirectFB nous allons développer cette interface

(luadirectfb.IDirectFBSurface) de la même manière. Le fichier du code source de cette interface est IDirectFBSurface.c. Nous allons définir dans ce fichier la fonction de chargement du sous-

module luadirectfb. IDirectFBSurface, la fonction qui créer l'objet de type IDirectFBSurface et l'affecte au metatable correspondant. Nous allons ensuite développer les fonctions de manipulation des surfaces précisées ultérieurement dans le cahier de charge. La liste de ces fonctions, ainsi que les nouveaux lexèmes utilisées pour appeler ces fonctions à partir du Lua et la fonction d'enregistrement du sous-module sont présentés dans l'algorithme 18.

L'algorithme 18 présente les fonctions à développer pour notre bibliothèque, nous allons expliquer ces fonctions dans la section suivante.

Ensuite, nous allons développer la fonction qui crée les objets de type IDirectFBSurface. Pour le faire nous allons définir la fonction lua_directfbCfunc qui réserve de l'espace mémoire pour l'objet, ensuite crée l'objet et l'affecte à son metatable. Mais pour les sous-modules IDirectFBSurface, IDirectFBDisplayLayer, IDirectFBImageProvider nous allons créer ces objets par l'intermédiaire des clés registres du langage Lua. Cette solution est utilisée parce que si on ne l'utilise pas et on crée ces objets comme IDirectFB, les méthodes de ces derniers ne fonctionnent pas. C'est un problème d'allocation mémoire parce que Lua ne peut pas indexer les méthodes de ces sous-modules si on n'utilise pas cette solution. L'algorithme 19 décrit la fonction *luadiretfbCfunc* du sous-module IDirectFBSurface.

Nous avons présenter dans cette section les différentes techniques de développement de l'interface luadirectfb.IDirectFBSurface. Cette interface est équivalente à l'interface IDirectFBSurface de la bibliothèque directfb. Nous allons passer maintenant au développement de l'interface luadirectfb.ImageProvide dans la section suivante.

7.3.4 Le sous-module IDirectFBImageProvider :

Le développement de cette interface est optionnelle, d'après le cahier de charge de notre projet. Ce sous-module est équivalant à l'interface IDirectFBImageProvider de la bibliothèque directfb. Comme indique sont nom cette interface nous permet de charger des images à partir des emplacements précis. Il nous permet aussi d'effectuer des opérations sur ces derniers en utilisant des fonctions prédéfinies de la bibliothèque directfb. Les objets créer à partir de cette interface sont liées à l'image charger par ce dernier, c.-à-d. Nous devons préciser le chemin d'accès de l'image à charger l'hors de la création des objets de cette interface. La fonction *lua_directfbCfunc*, qui crée l'interface, est définie de la même manière que la fonction de création de l'interface IDirectFBSurface. Cette fonction est présentée dans l'algorithme 20.

Algorithme 18 Les fonctions à implémenter de IDirectFBSurface

```
static const luaL_Reg dfbmethods[] = {
{ "Release", lua_Release},
{ "GetSize", lua_GetSize },
{ "Flip", lua_Flip },
{ "Clear", lua_Clear },
{ "SetClip", lua_SetClip },
{ "GetClip", lua_GetClip },
{ "SetColor", lua_SetColor},
//{ "SetSrcBlendFunction",lua_SetSrcBlendFunction},
//{ "SetDstBlendFunction",lua_SetDstBlendFunction},
{ "SetPorterDuff", lua_SetPorterDuff},
{ "SetSrcColorKey", lua_SetSrcColorKey},
{ "SetDstColorKey", lua_SetDstColorKey},
{ "SetBlittingFlags", lua_SetBlittingFlags},
{ "Blit", lua_Blit},
//{ "TileBlit", lua_TileBlit},
//opt{ "BatchBlit", lua_BatchBlit},
{ "StretchBlit", lua_StretchBlit},
//{ "TextureTriangles", lua_TextureTriangles},
{ "SetDrawingFlags", lua_SetDrawingFlags},
{ "FillRectangle", lua_FillRectangle },
{ "DrawRectangle", lua_DrawRectangle },
{ "DrawLine", lua_DrawLine },
{ "DrawLines", lua_DrawLines},
{ "FillTriangle", lua_FillTriangle},
//opt{ "FillRectangles", lua_FillRectangles},
{ "FillSpans", lua_FillSpans},
//opt{ "FillTriangles", lua_FillTriangles},
{ "DrawString", lua_DrawString},
{ NULL,NULL} };
static const luaL_Reg dfbfunctions[] = {
{ NULL,NULL} };
LUALIB_API int luaopen_luadirectfb_IDirectFBSurface(lua_State*L){
luaopen_module(L,dfbmethods,dfbfunctions,lua_directfbCfunc);
return 1;

}
```

Algorithme 19 La fonction luadirectfbCfunc du sous-module IDirectFBSurface

```
static int lua_directfbCfunc(lua_State *L) {
IDirectFB* dfb=* (IDirectFB**) luaL_checkudata(L, 2, "luadirectfb.IDirectFB");
DFBSurfaceDescription description;
StackvaluetoDFBSurfaceDescription(L, &description);
IDirectFBSurface* surface;
DFBCHECK(dfb->CreateSurface(dfb, &description, &surface));
lua_pushlightuserdata(L,surface);
IDirectFBSurface** _surface = (IDirectFBSurface**) lua_newuserdata(L, sizeof(IDirectFBSurface*));
*_surface = surface;
luaL_getmetatable(L, "luadirectfb.IDirectFBSurface");
lua_setmetatable(L, -2);
return 1;
}
```

Algorithme 20 La fonction luadirectfbCfunc du sous-module IDirectFBImageProvider.

```
/*Création de l'objet directfb image provider imageprovider*/
static int lua_directfbCfunc(lua_State *L) {
IDirectFB* dfb=* (IDirectFB**) luaL_checkudata(L, 2, "luadirectfb.IDirectFB");
IDirectFBImageProvider* imageprovider; const char* path=luaL_checkstring(L, 3);
DFBCHECK(dfb->CreateImageProvider (dfb,path, &imageprovider));
lua_pushlightuserdata(L,imageprovider);
IDirectFBImageProvider** _imageprovider = (IDirectFBImageProvider**) lua_newuserdata(L, si-
zeof(IDirectFBImageProvider*));
*_imageprovider = imageprovider;
luaL_getmetatable(L, "luadirectfb.IDirectFBImageProvider");
lua_setmetatable(L, -2);
return 1;
}
```

Comme le cahier de charge ne spécifi pas les fonctions à implémenter, nous allons nous limitons par les deux fonctions RenderTo et GetSurfaceDescription pour qu'on puisse charger des images et les afficher sur le moniteur d'affichage. La fonction *GetSurfaceDescription*, nous permettons d'avoir la description d'une surface. La fonction RenderTo() prend comme arguments l'objet crées de types IDirectFBImageProvider la surface de l'opération et le rectangle de destination, pour agrandir l'image de sa taille principale vers la taille du rectangle de destination. La fonction de chargement de luadirectfb.IDirectFBImageProvider ce fait à travers la fonction *LUALIB_API int luaopen_luadirectfb_IDirectFBImageProvider* qui est défini de la même manière que les sous-modules luadirectfb.IDirectFBSurface.

Nous allons présenter dans cette section le sous-module luadirectfb.IDirectFBImageProvider. C'est un sous-module optionnel de notre bibliothèque *luadirectfb.so*. Nous avons dévelloper le sous-module luadirectfb.IDirectFBDisplayLayer. Maintenant, nous allons passer à la phase de compilation et de test de notre bibliothèque.

7.4 Compilation et test du luadirectfb

Après le développement de tous les codes sources des fichiers qui représentent les différentes interfaces de la bibliothèque directfb, nous allons procéder à la compilation de ces derniers pour l'architecture SH4 du processeur ST40. Pour compiler les fichiers sources et générer notre bibliothèque, nous allons développer un Makefile qui défini les règles de compilation. Comme l'architecture du processeur est SH4, le compilateur que nous allons utiliser le sh4-linux-gcc. Nous allons compiler les codes sources un par un ensuite effectuer le lien entre eux pour générer notre bibliothèque dynamique luadirectfb.so. Le Makefile est présenté dans l'algorithme 21

La première étape consiste à définir le compilateur :

CC=/opt/STM/STLinux-2.3/devkit/sh4/bin/sh4-linux-gcc

Dans la conception de notre bibliothèque, nous avons inclus les fichiers directfb.h, lua.h -*I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src* Ensuite, nous allons générer les fichiers « *. o » par la compilation enfin effectuer le lien entre ces fichiers pour générer luadirectfb.so.

Les options d'édition de lien à ajouter sont :

-shared -fpic -o -ldirectfb

Maintenant, il suffit d'accéder au dossier contenant les codes sources et le Makefile à traver le

Algorithme 21 Makefile de la bibliothèque luadirectfb

```
CC=/opt/STM/STLinux-2.3/devkit/sh4/bin/sh4-linux-gcc
all : $(CC) -I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src -g -fpic -c luadirectfb.c
$(CC) -I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src -g -fpic -c IDirectFB.c
$(CC) -I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src -g -fpic -c IDirectFBSurface.c
$(CC) -I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src -g -fpic -c IDirectFBDisplayLayer.c
$(CC) -I/usr/local/include/directfb -I/home/lab2/Desktop/lua-5.1.4/src -g -fpic -c IDirectFBImageProvider.c
$(CC) -shared -fpic -o luadirectfb.so luadirectfb.o -ldirectfb \
IDirectFB.o \
IDirectFBSurface.o\
IDirectFBImageProvider.o\
IDirectFBDisplayLayer.o\
clean :
rm *.o
rm *.so
```

Shell ensuite exécuter make comme suit :

[Hajri@STHajri ~]$ cd /media/STHAJRI/codesource/ [Hajri@STHajri codesource]$ make

Après la fin de compilation, notre bibliothèque est générée sous le nom spécifié dans le Makefile luadirectfb.so.

Pour tester notre bibliothèque, nous allons effectuer le portage de cette dernière sur le processeur ST40. De la même manière que Lua, il suffit de copier luadirectfb.so dans le dossier root des fichiers systèmes de la cible de STLinux. Voici le chemin d'accès à ce dossier :

/opt/STM/STLinux-2.3/devkit/sh4/target/root

Notre bibliothèque est compilée correctement. Nous allons passer à la phase de test. Nous avons choisi d'effectuer un test de toutes les interfaces et les fonctions énumérées dans le cahier de charge dans un seul fichier de test. L'algorithme décrit le fichier de test de notre bibliothèque.

Pour exécuter ce test il suffit :

./lua all_test.lua

Les résultats de test sont conformes aux résultats attendues. Notre bibliothéque effectue les mêmes opérations d'accélération graphique que la bibliothèque directFB.

Algorithme 22 Fichier all_test.lua de test du bibliothèque luadirectfb.so

```
package.cpath = package.cpath .. './?.so'
require 'luadirectfb'
description = { flags = luadirectfb.DSDESC_CAPS;
caps= luadirectfb.DSCAPS_PRIMARY;
widht=1;
hight=1;
pixelformat=DSPF_RGB24}
dfb = luadirectfb.init()
dfb :SetCooperativeLevel()
primary=dfb :CreateSurface(description)
a,b = primary :GetSize()
c=math.min(a,b)
for i=1,c do
primary :SetColor(255,i,0,0)--red
primary :FillRectangle(i, i, a/2, b/2)
primary :SetColor(i,255,0,0)--green
primary :SetColor(10,128,120,0)--green
primary :DrawRectangle( a/4, b/4,i,b-i)
end
primary :Clear(0,0,0,0)
for i=1 ,1000 do
primary :SetColor(255-i,i,255,0)
primary :DrawLine( i, i,a-i,b-i)
primary :SetColor(0,i,255,0)
primary :DrawLine( 0, i,a,i)
primary :SetColor(i,0,i,0)
primary :DrawLine( i, 0,i,b)
primary :FillTriangle(i,i,a-i,0,0,b-i)
end
primary :Clear(0,0,0,0)
for i=1,100 do
rectangle={x=i;y=i;w=a/12;h=b/12}
rectangle.x = i; rectangle.w = a/12;
primary :SetColor(0,0,i,0)
primary :FillRectangles(rectangle)
primary :SetColor(128,128,128,0)
primary :FillRectangles(rectangle)
lines={x1=i,y1=a,x2=a/2,y2=b/2}
primary :SetColor(i,0,255,0)
primary :DrawLines(lines)
triangle={x1=i;y1=i;x2=a-i;y2=0;x3=0;y3=b-i}
primary :SetColor(255-i,i,255,0)
primary :FillTriangles(triangle)
span={x=i;w=b-i}
primary :FillSpans(10,span)
end
primary1=dfb :CreateSurface(description)
layer =dfb :GetDisplayLayer(00)
imageprovider=dfb :CreateImageProvider ("ENISo.png" )
primary = dfb :CreateSurface(description)
rectangle={x=0;y=0;w=a;h=b}
imageprovider :RenderTo(primary1,rectangle)
imageprovider=dfb :CreateImageProvider ( "images.jpg")
logo = dfb :CreateSurface(description)
a,b = primary :GetSize()
for i=1,16 do
rectangle={x=i;y=i;w=3*a/8;h=b/4}
imageprovider :RenderTo(logo,rectangle)
end
imageprovider :Release(imageprovider)
primary :Clear(0,0,0,0)
primary :Release(primary)
layer :Release(layer)
dfb :Release(dfb)

io.read()
```

7.5 Conclusion :

Dans ce chapitre, nous avons développer notre interface entre Lua et directFB. Nous avons utiliser le principe d'extension du langage Lua avec des bibliothèques liées dynamiquement. Notre méthodologie de développement est de garder la même architecture des interfaces de la bibliothèque directFB et le même concept de programmation (aspect orienté objet). Pour cela, nous avons utilisé la notion des modules et des sous-modules pour concevoir l'architecture de notre bibliothèque. Ensuite nous avons compilé les codes sources pour générer la bibliothèque luadirectfb.so. Après la génération du luadirectfb.so, nous avons effectuer le portage sur la cible 7015DT2. En fin nous avons tester toutes les composantes de la bibliothèque pour nous assurer que notre bibliothèque répond au besoin principal et complémentaire du cahier de charge.

CONCLUSION GÉNÉRALE

Notre projet de fin d'études a été réalisé au sein de l'ENISo proposer par la société STMicroelectronics durant environ quatre mois. Nous avons pu faire le portage du langage Lua sur l'architecture SH4 des processeurs ST40. La finalité de ce projet est d'exécuter avec succès l'interface entre Lua est DirectFB. Dans le premier partie de ce travail, nous avons présenté l'organisme d'accueil ainsi que le cadre du projet et la problématique du sujet et nous avons présenter les différents techniques d'interprétation.

Ensuite nous avons efféctuer un étude du langage Lua et nous avant extraire la structure lexicale et syntaxique les différentes méthodes d'extension du Lua. Le deuxième partie a décrit l'environnement matériel et logiciel qui a été adopté au cour du projet. La troixième partie consiste a présenté les différente solution pour élaborer l'interface entre lua et directFB.

Ce projet nous a permis de découvrir un nouveau système d'exploitation embarqué et de découvrir l'environnement de la programmation dans l'embarqué avec ses propres outils et langages. Sur le plan personnel, l'apport de ce travail a été d'une importance considérable, puisqu'il a représenté une occasion exceptionnelle pour collaborer avec les membres de la société STMicrelectronics et pour découvrir l'importance de la communication pour le bon déroulement d'un projet. La prochaine étape de ce projet consiste à effectuer le portage de la machine virtuelle Lua sur le processeur ST40.