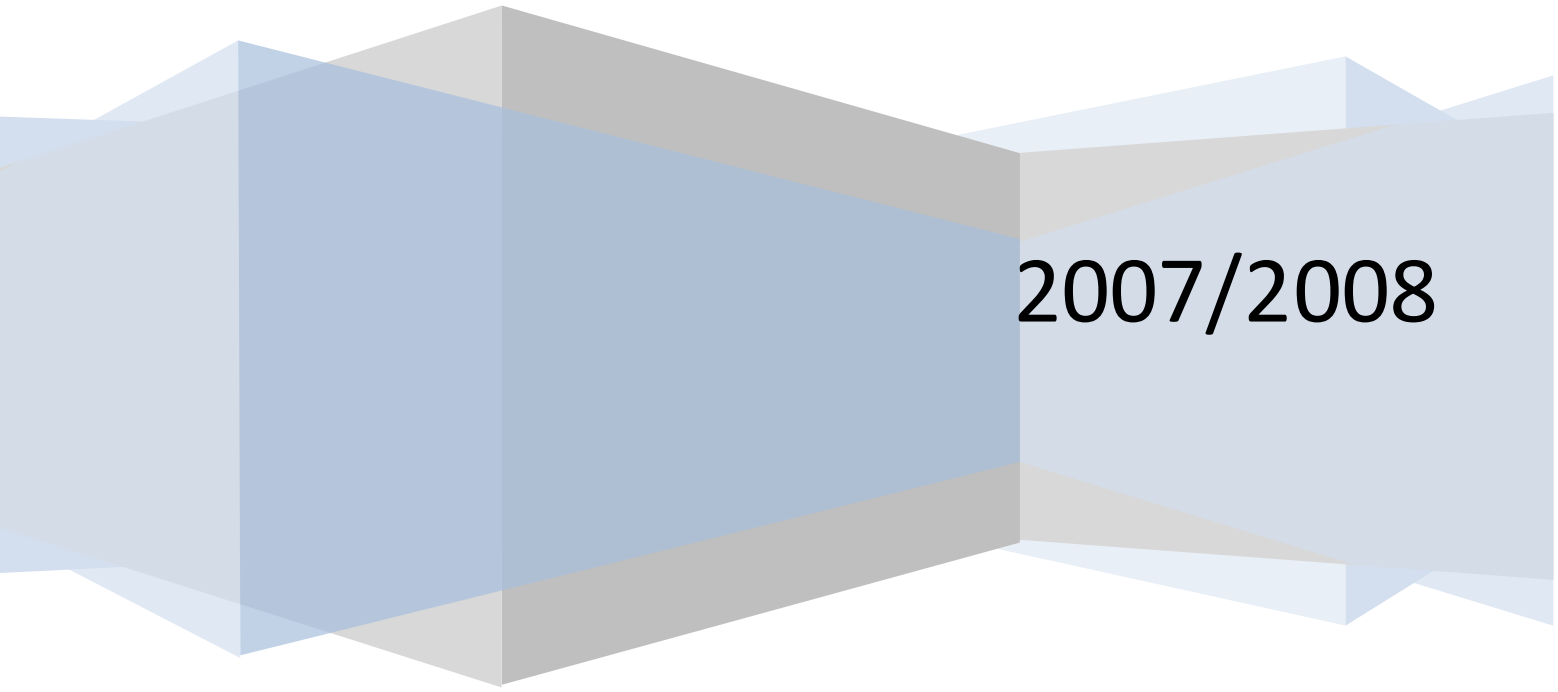
Pascal MIETLICKI – Jihed OTHMANI
4 GI
GROUPE C

Rapport Projet Systèmes informatiques



2007/2008

Table des matières

Introduction générale

Dans le cadre de notre formation en 4ème année de génie informatique, nous avons été amené à réaliser un projet complet englobant, à la fois, la théorie des langages ainsi que l'architecture matérielle.
Ce projet consistait en première lieu à la réalisation d'un compilateur pour un pseudo langage ***C*** avec un interpréteur afin de générer les instructions assembleurs correspondantes à l'aide des outils ***lex*** et ***yacc*** puis en deuxième et dernière partie à la conception d'un microprocesseur à ***architecture RISC*** avec pipe-lines.
Ce projet intéressant nous a permis de mettre en application et approfondir les enseignements théoriques que nous avions suivi au préalable.

REALISATION DU COMPILATEUR

Introduction

Dans le sens le plus usuel du terme, la compilation est une transformation que l'on fait subir à un programme écrit dans un langage évolué pour le rendre exécutable. Fondamentalement, c'est une traduction : un texte écrit en Pascal, C, Java, etc., exprime un algorithme et il s'agit de produire un autre texte, spécifiant le même algorithme dans le langage d'une machine que nous cherchons à programmer.

Structure de principe d'un compilateur

La nature de ce qui sort d'un compilateur est très variable. Cela peut être un programme exécutable pour un processeur physique, comme un Pentium III ou un G4, ou un fichier de code pour une machine virtuelle, comme la machine Java, ou un code abstrait destiné à un outil qui en fera ultérieurement du code exécutable, ou encore le codage d'un arbre représentant la structure logique d'un programme, etc.
En entrée , on trouve toujours la même chose : une suite de caractères, appelée texte source. Voici les phases dans lesquelles se décompose le travail d'un compilateur, du moins d'un point de vue logique :

- Analyse lexicale : Dans cette phase, les caractères isolés qui constituent le texte source sont regroupés pour former des unités lexicales, qui sont les mots du langage. L'analyse lexicale opère sous le contrôle de l'analyse syntaxique ; elle apparaît comme une sorte de fonction de lecture améliorée, qui fournit un mot lors de chaque appel.
- Analyse syntaxique Alors que l'analyse lexicale reconnaît les mots du langage, l'analyse syntaxique en reconnaît les phrases. Le rôle principal de cette phase est de dire si le texte source appartient au langage considéré, c'est-`a-dire s'il est correct relativement à la grammaire de ce dernier.
- Analyse sémantique La structure du texte source étant correcte, il s'agit ici de vérifier certaines propriétés sémantiques, c'est-à-dire relatives à la signification de la phrase et de ses constituants :
  - les identificateurs apparaissant dans les expressions ont-ils été declarés ?
  - les opérandes ont-ils les types requis par les opérateurs ?
  - les opérandes sont-ils compatibles ? N'y a-t-il pas des conversions à insérer ?
  - les arguments des appels de fonctions ont-ils le nombre et le type requis ?

– Génération de code intermédiaire : Après les phases d'analyse, certains compilateurs ne produisent pas directement le code attendu en sortie, mais une représentation intermédiaire, une sorte de code pour une machine abstraite. Cela permet de concevoir indépendamment les premières phases du compilateur qui ne dépendent que du langage source compilé et les dernières phases qui ne dépendent que du langage cible.

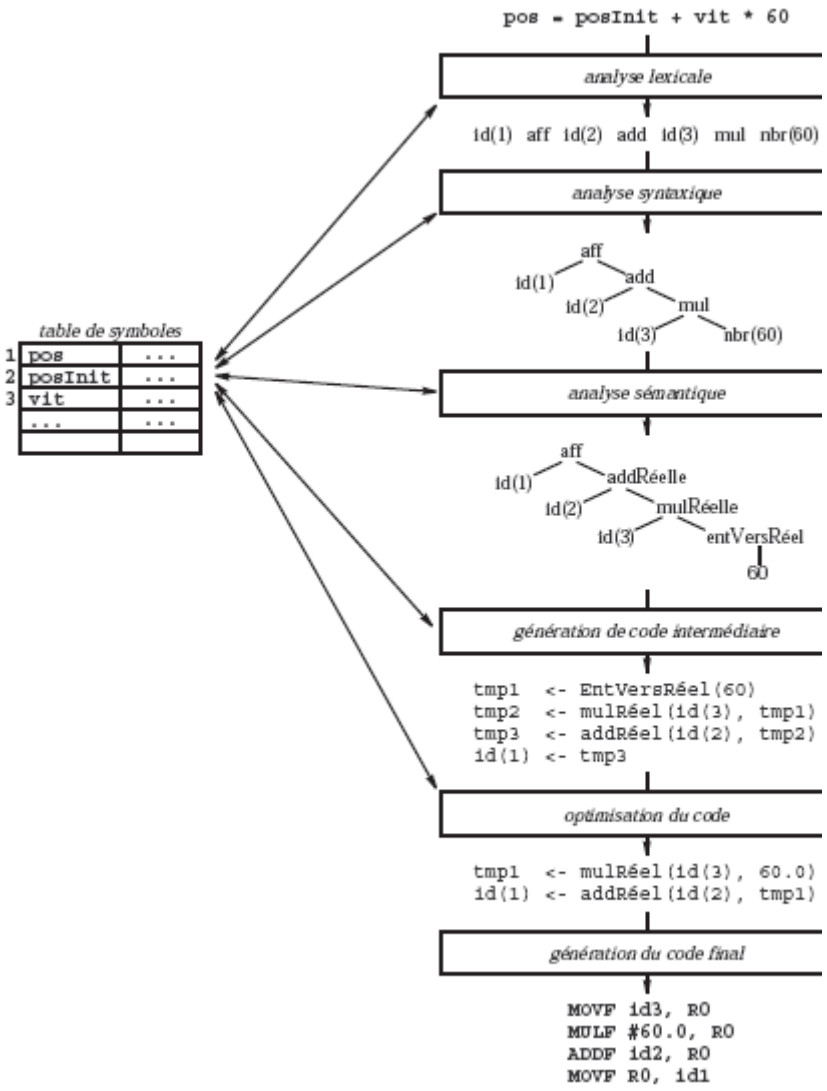


FIG 1: Phases logiques de la compilation d'une instruction

– Optimisation du code : Il s'agit géénéralement de transformer le code afin que le programme résultant s'exécute plus rapidement. Par exemple :
  – détecter l'inutilité de recalculer des expressions dont la valeur est déjà connue,
  – transporter à l'extérieur des boucles des expressions et sous-expressions dont les opérandes ont la même valeur à toutes les itérations
  – détecter, et supprimer, les expressions inutiles

Analyse lexicale

L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se

présente comme un flot de caractères, l'analyse lexicale reconnaît des *unités lexicales*, qui sont les mots avec lesquels les phrases sont formées, et les présente à la phase suivante, l'analyse syntaxique.
Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

- les caractères spéciaux simples : +, =, etc.
- les caractères spéciaux doubles : <, *, etc.
- les mots-clés : while, if, etc.
- les constantes littérales : 123, -5, etc.
- et les identificateurs : vitesse_du_vent, i, etc.

A propos d'une unité lexicale reconnue dans le texte source on doit distinguer quatre notions importantes :
– l'unité lexicale, représentée généralement par un code conventionnel ; pour nos dix exemples +, =, <, *, if, while, 123, -5, i et vitesse_du_vent, ce pourrait être, respectivement : t_plus, t_affec, t_inf, t_mult, t_if, t_while, t_entier, t_entier, t_variable, t_variable.
– le lexème, qui est la chaîne de caractères correspondante. Pour les dix exemples précédents, les lexèmes correspondants sont : "+", "=", "<", "*", "if", "while", "123", "-5", "i" et "vitesse_du_vent"
– éventuellement, un attribut, qui dépend de l'unité lexicale en question, et qui la complète. Seules les dernières des dix unités précédentes ont un attribut ; pour un nombre, il s'agit de sa valeur (123, −5) ; pour un identificateur, il s'agit d'un renvoi à une table dans laquelle sont placés tous les identificateurs rencontrés : la table des symboles.
– le modèle qui sert à spécifier l'unité lexicale comme :
– pour les caractères spéciaux simples et doubles et les mots réservés, le lexème et le modèle coïncident,
– le modèle d'un nombre est une suite de chiffres, éventuellement précédée d'un signe ,
– le modèle d'un identificateur est une suite de lettres, de chiffres et du caractère ' ', commençant par une lettre.

Outre la reconnaissance des unités lexicales, les analyseurs lexicaux assurent certaines tâches mineures comme la suppression des caractères de décoration (blancs, tabulations, fins de ligne, etc.) et celle des commentaires (généralement considérés comme ayant la même valeur qu'un blanc), l'interface avec les fonctions de lecture de caractères, à travers lesquelles le texte source est acquis, la gestion des fichiers et l'affichage des erreurs, etc.

Remarque. La frontière entre l'analyse lexicale et l'analyse syntaxique n'est pas fixe. D'ailleurs, l'analyse lexicale n'est pas une obligation, on peut concevoir des compilateurs dans lesquels la syntaxe est définie à partir des caractères individuels. Mais les analyseurs syntaxiques qu'il faut alors écrire sont bien plus complexes que ceux qu'on obtient en utilisant des analyseurs lexicaux pour reconnaître les mots du langage. Simplicité et efficacité sont les raisons d'être des analyseurs lexicaux. Les techniques pour reconnaître les unités lexicales sont bien plus simples et efficaces que les techniques pour vérifier la syntaxe.

Expressions régulières

Les expressions régulières sont une importante notation pour spécifier formellement des modèles. Pour les définir correctement il nous faut connaitre les concepts suivants :

- Un alphabet est un ensemble de symboles. Exemples : {0, 1}, {A, C, G, T }, l'ensemble de toutes les lettres, l'ensemble des chiffres, le code ASCII, etc. On notera que les caractères blancs (c'est-à-dire les espaces, les tabulations et les marques de fin de ligne) ne font généralement pas partie des alphabets.

- Une chaîne (on dit aussi mot) sur un alphabet est une séquence finie de symboles. Exemples : 00011011, ACCAGTTGAAGTGGACCTTT, Bonjour, 2001. On note ε la chaîne vide, ne comportant aucun caractère.

- Un langage sur un alphabet est un ensemble de chaînes construites. Exemples : ∅, le langage vide, {ε}, le langage réduit à l'unique chaîne vide. L'ensemble des nombres en notation binaire, l'ensemble des chaînes ADN, l'ensemble des mots de la langue francaise, etc.

Les opérations sur les langages suivantes nous serviront à définir les expressions régulières. Soient L et M deux langages, on définit :

dénomination	notation	définition
l'union de L et M	$L \cup M$	$\{ x \mid x \in L \text{ ou } x \in M\}$
la concaténation de L et M	LM	$\{ xy \mid x \in L \text{ et } y \in M\}$
la fermeture de Kleene de L	L^*	$\{ x_1x_2...x_n \mid x_i \in L, n \in \mathbb{N} \text{ et } n \geq 0\}$
la fermeture positive de L	L^+	$\{ x_1x_2...x_n \mid x_i \in L, n \in \mathbb{N} \text{ et } n > 0\}$

Soit Σ un alphabet. Une expression régulière r sur Σ est une formule qui définit un langage L(r) sur Σ, de la manière suivante :

1. ε est une expression régulière qui définit le langage {ε}
2. Si a ∈ Σ, alors a est une expression régulière qui définit le langage {a}
3. Soient x et y deux expressions régulières, définissant les langages L(x) et L(y). Alors

- (x)|(y) est une expression régulière définissant le langage L(x) ∪ L(y)
- (x)(y) est une expression régulière définissant le langage L(x)L(y)
- $(x)^*$ est une expression régulière définissant le langage $(L(x))^*$
- (x) est une expression régulière définissant le langage L(x)

La dernière règle ci-dessus signifie qu'on peut encadrer une expression régulière par des parenthèses sans changer le langage défini. D'autre part, les parenthèses apparaissant dans les règles précédentes peuvent souvent être omises, en fonction des opérateurs en présence : il suffit de savoir que les opérateurs $*$, concaténation et | sont associatifs à gauche, et vérifient :

priorité ($*$) > priorité (concaténation) > priorité (|)

Ainsi, on peut écrire l'expression régulière oui au lieu de (o)(u)(i) et oui|non au lieu de (oui)|(non), mais on ne doit pas écrire oui^* au lieu de $(oui)^*$.

Voici quelques définitions régulières, et notamment celles de identificateur et nombre :

lettre → A | B | . . . | Z | a | b | . . . | z
chiffre → 0 | 1 | . . . | 9
identificateur → lettre (lettre | chiffre)*
chiffres → chiffre chiffre *
fraction-opt → . chiffres | ε
exposant-opt → (E (+ | - | ε) chiffres) | ε

nombre → chiffres fraction-opt exposant-opt

Les définitions de lettre et chiffre données ci-dessus peuvent se réécrire en notation abrégée :
lettre → [A–Za–z]
chiffre → [0–9]

Ce que les expressions régulières ne savent pas faire

Les expressions régulières sont un outil puissant et pratique pour définir les unités lexicales, c'est-à-dire les constituants élémentaires des programmes. Mais elles se prêtent beaucoup moins bien à la spécification de constructions de niveau plus élevé, car elles deviennent rapidement d'une trop grande complexité.
De plus, il y a des chaînes qu'on ne peut pas décrire par des expressions régulières. Par exemple, le langage suivant (supposé infini) : { a, (a), ((a)), (((a))), . . . } ne peut pas être défini par des expressions régulières, car ces dernières ne permettent pas d'assurer qu'il y a dans une expression de la forme ((. . . ((a)) . . .)) autant de parenthèses ouvrantes que de parenthèses fermantes. On dit que les expressions régulières ne savent pas compter.
Pour spécifier ces structures équilibrées, si importantes dans les langages de programmation (parenthèses dans les expressions arithmétiques, les crochets dans les tableaux, begin...end, {...}, if...then..., etc.) nous ferons appel aux grammaires non contextuelles.

Reconnaissance des unités lexicales

Nous avons vu comment spécifier les unités lexicales ; notre problème maintenant est d'écrire un programme qui les reconnaît dans le texte source. Un tel programme s'appelle un analyseur lexical.
Dans un compilateur, le principal client de l'analyseur lexical est l'analyseur syntaxique. L'interface entre ces deux analyseurs est une fonction *int uniteSuivante(void)* , qui renvoie à chaque appel l'unité lexicale suivante trouvée dans le texte source.
Cela suppose que l'analyseur lexical et l'analyseur syntaxique partagent les définitions des constantes conventionnelles définissant les unités lexicales. Si on programme en C, cela veut dire que dans les fichiers sources des deux analyseurs on a inclus un fichier d'entête (fichier .h) comportant une série de définitions comme :

```
#define  IDENTIF     1
#define  NOMBRE    2
#define  SI     3
#define  ALORS       4
#define  SINON       5
etc.
```

Cela suppose aussi que l'analyseur lexical et l'analyseur syntaxique partagent également une variable globale contenant le lexème correspondant à la dernière unité lexicale reconnue, ainsi qu'une variable globale contenant le (ou les) attribut(s) de l'unité lexicale courante, lorsque cela est pertinent, et notamment lorsque l'unité lexicale est NOMBRE ou IDENTIF.

Diagrammes de transition

Les diagrammes de transition sont une étape préparatoire pour la réalisation d'un analyseur lexical. Au fur et à mesure qu'il reconnaît une unité lexicale, l'analyseur lexical passe par divers états. Ces états sont numérotés et représentés dans le diagramme par des cercles.

Par exemple, la figure 2 montre les diagrammes traduisant la reconnaissance des unités lexicales INFEG, DIFF, INF, EGAL, SUPEG, SUP et IDENTIF respectivement définie par les expressions régulières <=, <>, <, =, >=, > et lettre (lettre |chiffre)$*$, lettre et chiffre ayant été définis précedemment.

Un diagramme de transition est dit non déterministe lorsqu'il existe plusieurs flèches issues d'un même état et étiquetées par le même caractère, ou bien lorsqu'il existe des flèches étiquetées par la chaîne vide ε. Dans le cas contraire, le diagramme est dit déterministe.

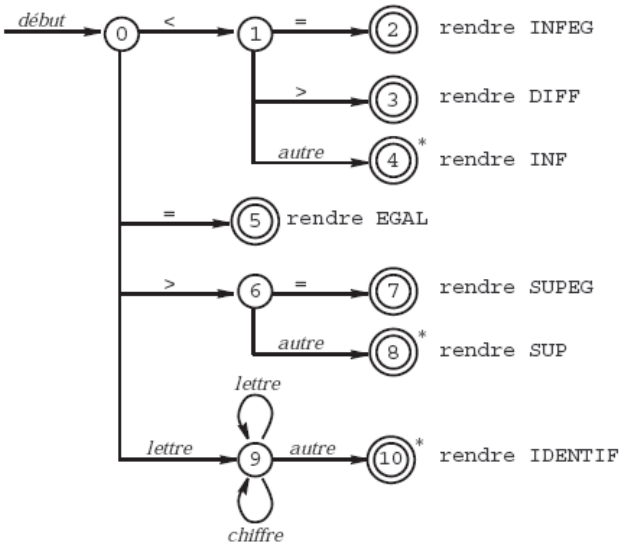


FIG 2: Diagramme de transition pour les opérateurs de comparaison et les identificateurs

Les diagrammes de transition sont une aide importante pour l'écriture d'analyseurs lexicaux. On peut très vite, à partir de ceux ci, obtenir un analyseur lexical.

```
int  uniteSuivante(void)  {
.       char  c;
.       c  =  lireCar();.       /*  etat  =  0  */
.       while  (estBlanc(c))
.       c  =  lireCar();
.       if  (c  ==  '<')  {
.       .       c  =  lireCar();.       /*  etat  =  1  */
.       .       if  (c  ==  '=')
.       .       return  INFEG;. /*  etat  =  2  */
.       .       else..
.       .       if  (c  ==  '>')
.       .       .       return  DIFF;.  /*  etat  =  3  */
.       .       else  {
.       .       .       delireCar(c);.  /*  etat  =  4  */
.       .       .       return  INF;
.       .       }
.       }
.       else  if  (c  ==  '=')
.       .       return  EGAL;.  /*  etat  =  5  */
.       else  if  (c  ==  '>')  {
.       .       c  =  lireCar();.       /*  etat  =  6  */
.       .       if  (c  ==  '=')
.       .       .       return  SUPEG;. /*  etat  =  7  */
.       .       else  {
.       .       .       delireCar(c);.  /*  etat  =  8  */
.       .       .       return  SUP;
.       .       }
.       }
....
```

FIG. 3: Début du programme pour l'analyse lexicale

Problème des mots réservés

Les mots réservés appartiennent au langage défini par l'expression régulière lettre (lettre |chiffre)∗ , tout comme les identificateurs. Leur reconnaissance peut donc se traiter de deux manières :

– soit on incorpore les mots réservés au diagrammes de transition, ce qui permet d'obtenir un analyseur très efficace, mais au prix d'un travail de programmation plus important, car les diagrammes de transition deviennent très volumineux,

– soit on laisse l'analyseur traiter de la même manière les mots réservés et les identificateurs puis, quand la reconnaissance d'un identificateur ou mot réservé est terminée, on recherche le lexème dans une table pour déterminer s'il s'agit d'un identificateur ou d'un mot réservé.

Dans les analyseurs lexicaux, on utilise souvent la deuxième méthode, plus facile à programmer. On se donne donc une table de mots réservés :

```
struct {
char *lexeme;
int uniteLexicale;
} motRes[] = {
{ "si", SI },
{ "alors", ALORS },
{ "sinon", SINON },
...
```

};
int nbMotRes = sizeof motRes / sizeof motRes[0];
puis on modifie en conséquence l'analyseur syntaxique pour prendre en compte cette table.

Automates finis

Un automate fini est défini par :
– un ensemble fini d'états E,
– un ensemble fini de symboles (ou alphabet) d'entrée Σ,
– une fonction de transition, transit : E × Σ → E,
– un état ε0 distingué, appelé état initial,
– un ensemble d'états F , appelés états d'acceptation ou états finaux.

Un automate peut être représenté graphiquement par un graphe. Un tel graphe est exactement ce que nous avons appelé diagramme de transition. Si on en reparle ici c'est qu'on peut en déduire un autre style d'analyseur lexical.
On dit qu'un automate fini accepte une chaîne d'entrée si et seulement si il existe dans le graphe de transition un chemin joignant l'état initial à un certain état final. Pour transformer un automate fini en un analyseur lexical il suffira donc d'associer une unité lexicale à chaque état final et de faire en sorte que l'acceptation d'une chaîne produise comme résultat l'unité lexicale associée à l'état final en question.

On obtiendra un analyseur peut être plus encombrant que dans la première manière mais beaucoup plus rapide puisque l'essentiel du travail de l'analyseur se réduira à répéter l'action.

Lex et Yacc

Lex et yacc sont des outils très populaires de génération d'analyseurs lexicaux (Lex) et syntaxiques (Yacc) en langage C.

LEX, un générateur d'analyseurs lexicaux

Les analyseurs lexicaux basés sur des tables de transitions sont les plus efficaces... une fois la table de transition construite. Or, la construction de cette table est une opération longue et délicate. Le programme lex fait cette construction automatiquement : il prend en entrée un ensemble d'expressions régulières et produit en sortie le texte source d'un programme C qui, une fois compilé, est l'analyseur lexical correspondant au langage défini par les expressions régulières en question.
Lex permet de réaliser de façon semi-automatique l'analyse lexicale. Pour rappel, le but principal de l'analyse lexicale est de transformer une suite de symboles en terminaux (un terminal peut être par exemple un nombre, un signe '+', un identificateur, etc...). Une fois cette transformation effectuée, la main est repassée à l'analyseur syntaxique. Le but de l'analyseur lexical est donc de 'consommer' des symboles et de les fournir à l'analyseur syntaxique.

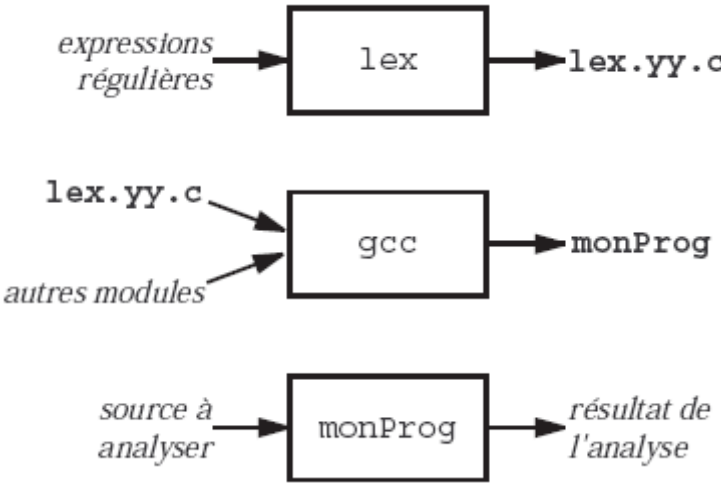


FIG 4: Utilisation de lex

Plus précisément, lex produit un fichier source C, nommé lex.yy.c, contenant la définition de la fonction int yytext(void), un programme appelle cette fonction et elle renvoie une unité lexicale reconnue dans le texte source.

Lex est un standard pour la génération d'analyseurs lexicaux sur la plupart des systèmes Unix. Lex lit un fichier d'entrée spécifiant l'analyseur lexical et produit du code source implémentant cet analyseur en langage C.

Structure d'un fichier source pour Lex

Lex écrit un fichier source C. Ce fichier est composé :
– des tables de valeurs calculées à partir des expressions régulières fournies,
– des morceaux de code C invariable, et notamment le moteur de l'automate, c'est-à-dire la boucle qui répète inlassablement etat ← transit (etat, caractere),
- des morceaux de code C, trouvés dans le fichier source lex et recopiés tels quels, à l'endroit voulu, dans le fichier produit.

Un fichier source pour lex doit avoir un nom qui se termine par ".l" . Il est fait de trois sections, délimitées par deux lignes réduites au symbole %% :
%{
déclarations pour le compilateur C
%}
définitions régulières
%%
règles
%%
fonctions C supplémentaires

La partie "déclarations pour le compilateur C" et les symboles %{ et %} qui l'encadrent peuvent être omis. Quand elle est présente, cette partie se compose de déclarations qui seront simplement recopiées au début du fichier produit. En plus d'autres choses, on trouve souvent ici une directive *#include* qui produit l'inclusion du fichier ".h" contenant les définitions des codes conventionnels des unités lexicales.

La troisième section “fonctions C supplémentaires” peut être absente également (le symbole %% qui la sépare de la deuxième section peut alors être omis). Cette section se compose de fonctions C qui seront simplement recopiées à la fin du fichier produit.

Les définitions régulières sont de la forme :
identificateur expression Régulière
où *identificateur* est écrit au début de la ligne (pas de blancs avant) et séparé de *expression Régulière* par des blancs. Exemples de notre fichier source :
lettre [A-Za-z]
chiffre [0-9]
Les identificateurs ainsi définis peuvent être utilisés dans les règles et dans les définitions subséquentes ; il faut alors les encadrer par des accolades. Exemples :

```
lettre              [a-zA-Z]
chiffre             [0-9]
separateurs         [ \t]+
alphanum            {lettre}|{chiffre}
variable            {lettre}{alphanum}*
nombre              {chiffre}+("."{chiffre}+)?
nb_signe            [+-]?{nombre}
expo                [Ee][-+]?{nombre}

%%

"//".*      ;     /* gestion des commentaires */
"/*"((.*)|(.[^*/]*\n.[^*/]*)*)"*/"              { comptelignes(yytext);
}

{variable}      {
                        ECHO;
                        yylval.carac=strdup(yytext);
                        return t_variable;
                }
```

Les règles sont de la forme :
expression Régulière { action }
où *expression Régulière* est écrit au début de la ligne (pas de blancs avant) ; action est un morceau de code source C, qui sera recopié tel quel, au bon endroit, dans la fonction yylex.
Les règles signifient “à la fin de la reconnaissance d’une chaîne du langage défini par *expression Régulière* exécutez *action”*.
Le traitement par lex d’une telle règle consiste donc à recopier l’action indiquée à un certain endroit de la fonction yylex. Dans les exemples ci-dessus, les actions étant toutes de la forme “return unite”, leur signification est claire : quand une chaîne du texte source est reconnue, la fonction yylex se termine en rendant comme résultat l’unité lexicale reconnue. Il faudra appeler de nouveau cette fonction pour que l’analyse du texte source reprenne.

A la fin de la reconnaissance d’une unité lexicale la chaîne acceptée est la valeur de la variable yytext, de type chaîne de caractères. Un caractère nul indique la fin de cette chaîne ; de plus, la variable entière yylen donne le nombre de ses caractères. Par exemple, la règle suivante reconnaît les nombres entiers et en calcule la valeur dans une variable yylval :

```
{nb_signe}|
{nombre}{expo}    {
                        ECHO;
```

```
                        yylval.nb = atoi(yytext); /* yytext contient
l'expression trouvée */
                        return t_nombre;
                }
```

Les expressions régulières acceptées par lex sont une extension de celles précédemment définies.
Les méta-caractères précédemment introduits, c'est-à-dire (,), |, ∗ , +, ?, [,] et − à l'intérieur des crochets, sont légitimes dans *lex* et ont le même sens. En outre, on dispose de ceci (liste non exhaustive) :
– un point . signifie un caractère quelconque, différent de la marque de fin de ligne,
– on peut encadrer par des guillemets un caractère ou une chaîne, pour éviter que les méta-caractères qui s'y trouvent soient interprétés comme tels. Par exemple, "." signifie le caractère . (et non pas un caractère quelconque), " " signifie un blanc, "[a-z]" signifie la chaîne [a-z], etc.,
D'autre part, on peut sans inconvénient encadrer par des guillemets un caractère ou une chaîne qui n'en avaient pas besoin,
– l'expression [^*caractères*] signifie : tout caractère n'appartenant pas à l'ensemble défini par [caractères],
– l'expression "^*expression Régulière*" signifie : toute chaîne reconnue par expression Régulière à la condition qu'elle soit au début d'une ligne,
– l'expression "*expression Régulière$*" signifie : toute chaîne reconnue par *expression Régulière* à la condition qu'elle soit à la fin d'une ligne.

L'analyseur lexical produit par lex prend son texte source sur l'entrée standard et l'écrit, avec certaines modifications, sur la sortie standard. Plus précisement :
– tous les caractères qui ne font partie d'aucune chaîne reconnue sont recopiés sur la sortie standard (ils traversent l'analyseur lexical sans en être affectés),
– une chaîne acceptée au titre d'une expression régulière n'est pas recopiée sur la sortie standard.
Bien entendu, pour avoir les chaînes acceptées dans le texte écrit par l'analyseur il suffit de le prévoir dans l'action correspondante. Par exemple, la règle suivante reconnaît les identificateurs et fait en sorte qu'ils figurent dans le texte sorti :
[A-Za-z][A-Za-z0-9]* { ECHO; **return** t_variable; }

La fonction yywrap qui apparaît dans le fichier source pour lex est appelée lorsque l'analyseur
rencontre la fin du fichier à analyser. Outre d'´éventuelles actions utiles dans telle ou telle application particulière, cette fonction doit rendre une valeur non nulle pour indiquer que le flot d'entrée est définitivement épuisé, ou bien ouvrir un autre flot d'entrée.

Analyse syntaxique

Grammaires non contextuelles

Les langages de programmation sont souvent définis par des règles récursives, comme : “on a une expression en écrivant successivement un terme, ’+’ et une expression” ou “on obtient une instruction en écrivant à la suite si, une expression, alors, une instruction et, éventuellement, sinon et une instruction”. Les grammaires non contextuelles sont un formalisme particulièrement bien adapté à la description de telles règles.

Rappel

Une grammaire non contextuelle, on dit parfois grammaire BNF (pour Backus-Naur form), est un quadruplet G = (VT , VN , S0 , P) formé de
– un ensemble VT de symboles terminaux,
– un ensemble VN de symboles non terminaux,
– un symbole S0 ∈ VN particulier, appelé symbole de départ ou axiome,
– un ensemble P de productions

Nous pouvons expliquer ces éléments de la manière suivante :
1. Les symboles terminaux sont les symboles élémentaires qui constituent les chaînes du langage, les phrases. Ce sont donc les unités lexicales, extraites du texte source par l’analyseur lexical (il faut se rappeler que l’analyseur syntaxique ne connaît pas les caractères dont le texte source est fait, il ne voit ce dernier que comme une suite d’unités lexicales).
2. Les symboles non terminaux sont des variables syntaxiques désignant des ensembles de chaînes de symboles terminaux.
3. Le symbole de départ est un symbole non terminal particulier qui désigne le langage en son entier.
4. Les productions peuvent être interprétées de deux manières :
   – comme des règles d’écriture (on dit plutôt de réécriture), permettant d’engendrer toutes les chaînes correctes.
   – comme des règles d’analyse, on dit aussi reconnaissance.

La définition d’une grammaire devrait donc commencer par l’énumération des ensembles VT et VN. En pratique on se limite à donner la liste des productions, avec une convention typographique pour distinguer les symboles terminaux des symboles non terminaux.

En exemple, voici la grammaire G1 définissant le langage dont les chaînes sont les expressions arithmétiques formées avec des nombres, des identificateurs et les deux opérateurs + et *, comme “60 * vitesse + 200”. Suivant notre convention, les symboles non terminaux sont expression, terme et facteur; le symbole de départ est expression :
expression → expression "+" terme | terme
terme → terme "*" facteur | facteur
facteur → nombre | identificateur | "(" expression ")"

Dérivation et arbres de dérivation

Le processus par lequel une grammaire définit un langage s’appelle dérivation.

La dérivation gauche est entièrement composée de dérivations en une étape dans lesquelles à chaque fois c'est le non-terminal le plus à gauche qui est réécrit. On peut définir de même une dérivation droite, où à chaque étape c'est le non-terminal le plus à droite qui est réécrit.

Cette dérivation peut être représenté graphiquement par un arbre :

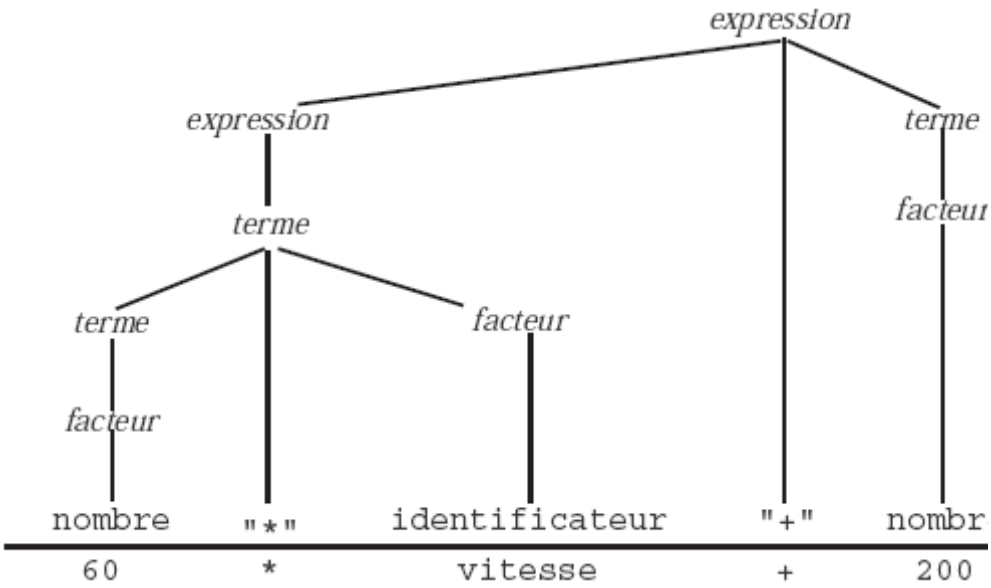


FIG 5: Arbre de dérivation

- la racine de l'arbre est le symbole de départ,
- les nœuds intérieurs sont étiquetés par des symboles non terminaux,
- les feuilles sont étiquetées par des symboles terminaux et, si on allonge verticalement les branches de l'arbre (sans les croiser) de telle manière que les feuilles soient toutes à la même hauteur, alors, lues de la gauche vers la droite, elles constituent la chaîne initiale.

Qualités des grammaires en vue des analyseurs

Etant donnée une grammaire G = {VT , VN , S0 , P }, faire l'analyse syntaxique d'une chaîne w ∈ VT c'est répondre à la question "w appartient-elle au langage L(G) ?". Un analyseur syntaxique est donc un programme qui n'extrait aucune information de la chaîne analysée, il ne fait qu'accepter (par défaut) ou rejeter (en annonçant une erreur de syntaxe) cette chaîne.

En réalité on ne peut pas empêcher les analyseurs d'en faire un peu plus car, pour prouver que w ∈ L(G) il faut exhiber une dérivation, c'est-à-dire construire un arbre de dérivation dont la liste des feuilles est w. Or, cet arbre de derivation est déjà une première information extraite de la chaîne source, un début de compréhension de ce que le texte signifie.

Nous examinons ici les qualités qu'une grammaire doit avoir et des défauts dont elle doit être exempte pour que la construction de l'arbre de dérivation de toute chaîne du langage soit possible et utile. Nous avons utilisé cette méthodologie afin d'écrire nos différentes règles de grammaire.

Une grammaire est ambiguë s'il existe plusieurs dérivations gauches différentes pour une même chaîne de terminaux. Par exemple, la grammaire suivante est ambiguë :

expression → expression "+" expression | expression "*" expression | facteur

facteur → nombre | identificateur | "(" expression ")"

En effet, la figure 6 montre deux arbres de dérivation gauche distincts pour la chaîne "2 * 3 + 10".

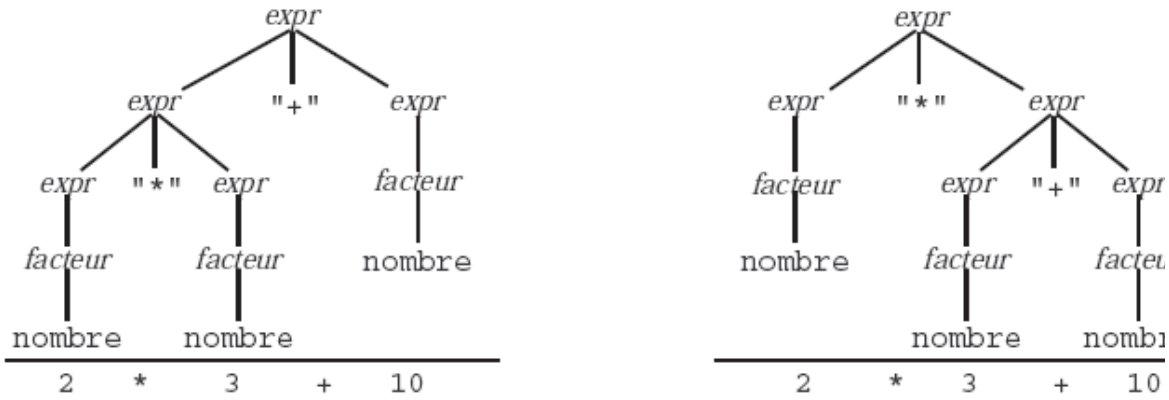


FIG 6: Deux arbres de dérivation pour la même chaine

Deux grammaires sont dites équivalentes si elles engendrent le même langage. Il est souvent possible de remplacer une grammaire ambiguë par une grammaire non ambiguë équivalente, mais il n'y a pas une méthode générale pour cela.

Factorisation à gauche

Nous cherchons à écrire des analyseurs prédictifs. Cela veut dire qu'à tout moment le choix entre productions qui ont le même membre gauche doit pouvoir se faire, sans risque d'erreur, en comparant le symbole courant de la chaîne à analyser avec les symboles susceptibles de commencer les dérivations des membres droits des productions en compétition.

Une grammaire contenant des productions comme :

$A \rightarrow \alpha\beta1 \mid \alpha\beta2$

viole ce principe car lorsqu'il faut choisir entre les productions $A \rightarrow \alpha\beta1$ et $A \rightarrow \alpha\beta2$ le symbole courant est un de ceux qui peuvent commencer une dérivation de α, et on ne peut pas choisir à coup sûr entre $\alpha\beta1$ et $\alpha\beta2$.

Exemple classique.

Les grammaires de la plupart des langages de programmation définissent ainsi l'instruction conditionnelle :

instr si → si expr alors instr | si expr alors instr sinon instr

Pour avoir un analyseur prédictif il faudra opérer une factorisation à gauche :

instr si → si expr alors instr fin instr si fin instr si → sinon instr | ε

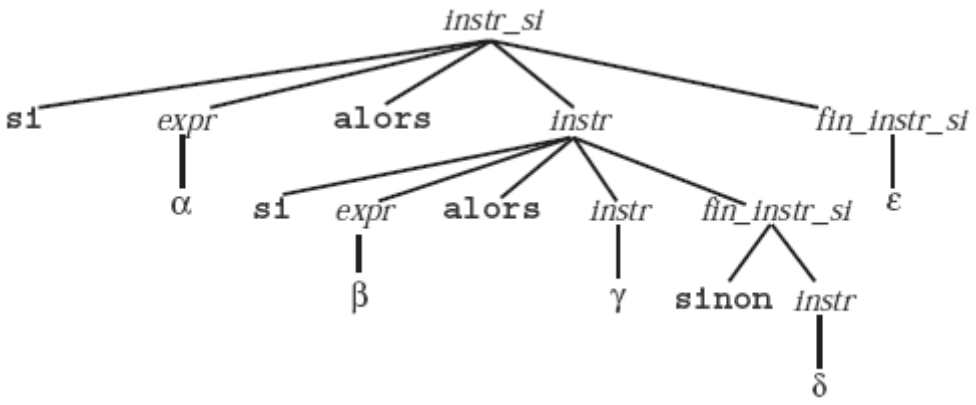


FIG 7: Arbre de dérivation

Ce que les grammaires non contextuelles ne savent pas faire

Les grammaires non contextuelles sont un outil puissant, en tout cas plus puissant que les expressions régulières, mais il existe des langages (pratiquement tous les langages de programmation) qu'elles ne peuvent pas décrire complètement.

On démontre par exemple que le langage $\{L=wcw|w\in(a|b)*\}$, ou` a, b et c sont des terminaux, ne peut pas être décrit par une grammaire non contextuelle. L est fait de phrases comportant deux chaînes de a et b identiques, séparées par un c, comme ababcabab. L'importance de cet exemple provient du fait que L modélise l'obligation, qu'ont la plupart des langages, de vérifier que les identificateurs apparaissant dans les instructions ont bien été préalablement déclarés (la première occurrence de w dans wcw correspond à la déclaration d'un identificateur, la deuxième occurrence de w à l'utilisation de ce dernier).

Autrement dit, l'analyse syntaxique ne permet pas de vérifier que les identificateurs utilisés dans les programmes font l'objet de déclarations préalables. Ce problème doit nécessairement être remis à une phase ultérieure d'analyse sémantique.

Les différents types d'analyseurs

Les grammaires des langages que nous cherchons à analyser ont un ensemble de propriétés qu'on résume en disant que ce sont des grammaires LL(1). 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),
– dans lesquels un seul symbole de la chaîne source est accessible à chaque instant et permet de choisir, lorsque c'est nécessaire, une production parmi plusieurs candidates (c'est le 1 de LL(1)).

Pour réfléchir au fonctionnement de nos analyseurs il est utile d'imaginer que la chaîne source est écrite sur un ruban défilant derrière une fenêtre, de telle manière qu'un seul symbole est visible à la fois. Un mécanisme permet de faire avancer –jamais reculer– le ruban, pour rendre visible le symbole suivant.

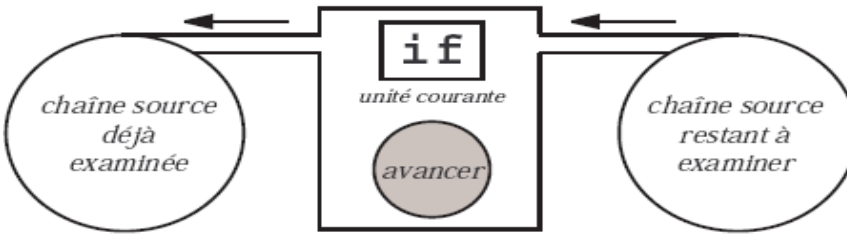


FIG 8: Fenêtre à symboles terminaux

Analyseur descendant

Un analyseur descendant construit l'arbre de dérivation de la racine (le symbole de départ de la grammaire) vers les feuilles (la chaîne de terminaux).

fenêtre	pile
nombre	*expression*
nombre	*terme fin_expression*
nombre	*facteur fin_terme fin_expression*
nombre	nombre *fin_terme fin_expression*
"*"	*fin_terme fin_expression*
"*"	"*" *facteur fin_terme fin_expression*
identificateur	*facteur fin_terme fin_expression*
identificateur	identificateur *fin_terme fin_expression*
"+"	*fin_terme fin_expression*
"+"	ε *fin_expression*
"+"	*fin_expression*
"+"	"+" *terme fin_expression*
nombre	*terme fin_expression*
nombre	*facteur fin_expression*
nombre	nombre *fin_expression*
¶	*fin_expression*
¶	ε
¶	

FIG 9: Exemple d'analyse descendante

Il existe aussi des analyseurs descendants non récursif, des analyses par descente récursives (étroitement lié à la grammaire analysée) et enfin, ce qui nous intéresse ici : les analyseurs ascendants.

Analyseurs ascendants

Comme nous l'avons dit, le but de l'analyse syntaxique est la construction d'un arbre de dérivation qui prouve $w \in L(G)$. Les méthodes descendantes construisent cet arbre en partant du symbole de départ de la grammaire et en allant vers la chaîne de terminaux. Les méthodes ascendantes, au contraire, partent des terminaux qui constituent la chaîne d'entrée et vont vers le symbole de départ.

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 remplacés 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 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 effectuer 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 ?

A titre d'exemple, avec la grammaire G1 :

expression → expression "+" terme | terme

terme → terme "*" facteur | facteur (G1)

facteur → nombre | identificateur | "(" expression ")"

voici la reconnaissance par un analyseur ascendant du texte d'entrée "60 * vitesse + 200", c'est-à-dire la chaîne de terminaux (nombre "*" identificateur "+" nombre) :

fenêtre	pile	action
nombre		décalage
"*"	nombre	réduction
"*"	*facteur*	réduction
"*"	*terme*	décalage
identificateur	*terme* "*"	décalage
"+"	*terme* "*" identificateur	réduction
"+"	*terme* "*" *facteur*	réduction
"+"	*terme*	réduction
"+"	*expression*	décalage
nombre	*expression* "+"	décalage
¶	*expression* "+" nombre	réduction
¶	*expression* "+" *facteur*	réduction
¶	*expression* "+" *terme*	réduction
¶	*expression*	succès

FIG 10: Analyse ascendante de G1

On dit que les méthodes de ce type effectuent une analyse par *décalage-réduction*. Comme le montre le tableau ci-dessus, le point important est le choix entre réduction et 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 faites dans l'analyse précédente construisent, à l'envers, la dérivation droite suivante :

expression ⇒ expression "+" terme

⇒ expression "+" facteur

⇒ expression "+" nombre

⇒ terme "+" nombre

⇒ terme "*" facteur "+" nombre

⇒ terme "*" identificateur "+" nombre

⇒ facteur "*" identificateur "+" nombre

⇒ nombre "*" identificateur "+" nombre

Analyse LR(k)

Il est possible, malgré les apparences, de construire des analyseurs ascendants plus efficaces que les analyseurs descendants, et acceptant une classe de langages plus large que la classe des langages traités par ces derniers.

Le principal inconvénient de ces analyseurs est qu'ils n´ecessitent des tables qu'il est extrêmement difficile de construire à la main. Heureusement, des outils existent pour les construire automatiquement, à partir de la grammaire du langage ; tel que yacc sans doute l'un des plus connus.

Les analyseurs LR(k) lisent la chaîne d'entrée de la gauche vers la droite (d'où le L), en construisant l'inverse d'une dérivation droite (d'où le R) avec une vue sur la chaîne d'entrée large de k symboles ; lorsqu'on dit simplement LR on sous-entend k = 1, c'est le cas le plus fréquent.

Etant donnée une grammaire G = (VT , VN , S0 , P), un analyseur LR est constitué par la donnée d'un ensemble d''etats E, d'une fenêtre à symboles terminaux (c'est-à-dire un analyseur lexical), d'une pile de doublets (s, e) où s ∈ E et e ∈ VT et de deux tables Action et Suivant, qui représentent des fonctions :

Action : E × VT → ({ decaler } × E) ∪ ({reduire} × P) ∪ { succes, erreur }

Suivant : E × VN → E

Un analyseur LR comporte enfin un programme, indépendant du langage analysé, qui exécute les opérations d'initialisation et d'itérations afin d'enchaîner les actions.

Yacc, un générateur d'analyseurs syntaxiques

Comme nous l'avons indiqué, les tables Action et Suivant d'un analyseur LR sont difficiles à construire manuellement, mais il existe des outils pour les déduire automatiquement des productions de la grammaire considérée.

Le programme yacc (Yet Another Compiler Compiler) est un générateur d'analyseurs syntaxiques. Il prend en entrée un fichier source constitué essentiellement des productions d'une grammaure non contextuelle G et sort à titre de résultat un programme C qui, une fois compilé, est un analyseur syntaxique pour le langage L(G).

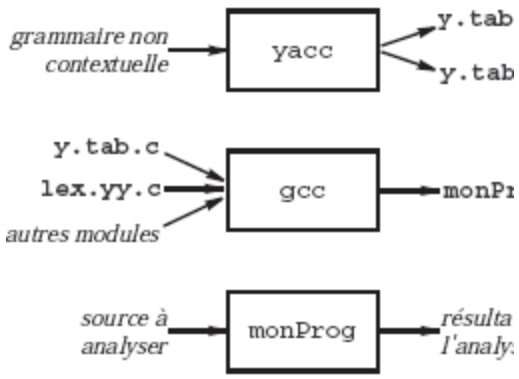


FIG 11: Utilisation de yacc

Dans la description de la grammaire donnée à yacc on peut associer des actions sémantiques aux productions ; ce sont des bouts de code source C que yacc place aux bons endroits de

l'analyseur construit. Ce dernier peut ainsi exécuter des actions ou produire des informations déduites du texte source, c'est-à-dire devenir un compilateur.

Un analyseur syntaxique requiert pour travailler un analyseur lexical qui lui délivre le flot d'entrée sous forme d'unités lexicales successives. Par défaut, yacc suppose que l'analyseur lexical disponible à été fabriqué par lex. Autrement dit, sans qu'il faille de déclaration spéciale pour cela, le programme produit par yacc comporte des appels de la fonction yylex aux endroits où l'acquisition d'une unité lexicale est nécessaire.

Structure d'un fichier source pour yacc

Un fichier source pour yacc doit avoir un nom terminé par ".y". Il est fait de trois sections, délimitées par deux lignes réduites au symbole %% :

%{

déclarations pour le compilateur C

%}

déclarations pour yacc

%%

règles (productions + actions sémantiques)

%%

fonctions C supplémentaires

Les parties "déclarations pour le compilateur C" et "fonctions C supplémentaires" sont simplement recopiées dans le fichier produit, respectivement au début et à la fin de ce fichier. Chacune de ces deux parties peut être absente.

Dans la partie "déclarations pour yacc" on rencontre souvent les déclarations des unités lexicales, sous une forme qui laisse yacc se charger d'inventer des valeurs conventionnelles :

```
%token t_if t_else
%token t_main

%token t_while
%token t_acc_ouv t_acc_fer
%token t_par_ouv t_par_fer
```

Ces déclarations d'unités lexicales intéressent yacc, qui les utilise, mais aussi lex, qui les manipule en tant que résultats de la fonction yylex. Pour cette raison, yacc produit un fichier supplémentaire, nommé "y.tab.h3", destiné à être inclus dans le source lex. Par exemple, le fichier produit pour les déclarations ci-dessus ressemble à ceci :

```
t_main = 258,
t_if = 259,
t_else = 260,
t_while = 261,
t_acc_ouv = 262,
t_acc_fer = 263,
t_par_ouv = 264,
t_par_fer = 265,...
```

Notez que yacc considère que tout caractère est susceptible de jouer le rôle d'unité lexicale; pour cette raison, ces constantes sont numérotées à partir de 258.

Spécification de VT , VN et S0

Dans un fichier source yacc :

– les caractères simples, encadrés par des apostrophes comme dans les programmes C, et les identificateurs mentionnés dans les déclarations %token sont tenus pour des symboles terminaux,

– tous les autres identificateurs apparaissant dans les productions sont considérés comme des symboles non terminaux,

– par défaut, le symbole de départ est le membre gauche de la première règle.

Règles de traduction.

Une règle de traduction est un ensemble de productions ayant le même membre gauche, chacune associé à une action sémantique.

Une action sémantique est un morceau de code source C encadré par des accolades. C'est un code que l'analyseur exécutera lorsque la production correspondante aura été utilisée dans une réduction.

Si on écrit un analyseur pur, c'est-à-dire un analyseur qui ne fait qu'accepter ou rejeter la chaîne d'entrée, alors il n'y a pas d'actions sémantiques et les règles de traduction sont simplement les productions de la grammaire.

Dans les règles de traduction, le méta-symbole → est indiqué par deux points ":" et chaque règle (c'est- à-dire chaque groupe de productions avec le même membre gauche) est terminée par un point-virgule ";". La barre verticale "|" a la même signification que dans la notation des grammaires.

L'analyseur syntaxique se présente comme une fonction int yyparse(void), qui rend 0 lorsque la chaîne d'entrée est acceptée, une valeur non nulle dans le cas contraire.

Il faut aussi écrire la fonction appelée en cas d'erreur. C'est une fonction de prototype void yyerror(char *message), elle est appelée par l'analyseur avec un message d'erreur.

Par exemple :

```
yyerror()
    {
            affichErr("Erreur de syntaxe");
    }
```

Actions sémantiques et valeurs des attributs

Une action sémantique est une séquence d'instructions C écrite, entre accolades, à droite d'une production. Cette séquence est recopiée par yacc dans l'analyseur produit, de telle manière qu'elle sera exécutée, pendant l'analyse, lorsque la production correspondante aura été employée pour faire une réduction.

Attributs

Un symbole, terminal ou non terminal, peut avoir un attribut, dont la valeur contribue à la caractérisation du symbole. Par exemple, la reconnaissance du lexème "2001" donne lieu à l'unité lexicale NOMBRE avec l'attribut 2001.

Un analyseur lexical produit par lex transmet les attributs des unités lexicales à un analyseur syntaxique produit par yacc à travers une variable yylval qui, par défaut, est de type int. Dans le fichier ".tab.h" fabriqué par yacc et destiné à être inclus dans l'analyseur lexical contient, outre les définitions des codes des unités lexicales, les déclarations :

#define YYSTYPE int

...

extern YYSTYPE yylval;

Dans les actions sémantiques de Yacc, on peut mettre certains symboles que yacc remplace par des expressions C correctes. Ainsi, $1, $2, $3, etc. désignent les valeurs des attributs des symboles constituant le membre droit de la production concernée, tandis que $$ désigne la valeur de l'attribut du symbole qui est le membre gauche de cette production.

L'action sémantique { $$ = $1 ; } est implicite et il n'y a pas besoin de l'écrire.

Conflits et ambiguïtés

```
expression  : expression '+' expression  { $$ = $1 + $3; }
            | expression '-' expression  { $$ = $1 - $3; }
            | expression '*' expression  { $$ = $1 * $3; }
            | expression '/' expression  { $$ = $1 / $3; }
            | '(' expression ')'         { $$ = $2; }
            | nombre
            ;
```

Cette grammaire est ambiguë ; elle provoquera donc des conflits. Lorsqu'il rencontre un conflit, yacc applique une règle de résolution par défaut et continue son travail ; à la fin de ce dernier, il indique le nombre total de conflits rencontrés et arbitrairement résolus. Il est impératif de comprendre la cause de ces conflits et dans la mesure du possible, les supprimer (par exemple en transformant la grammaire). Les conflits possibles sont :

1. Décaler ou réduire (shift/reduce conflict)

Ce conflit se produit lorsque l'algorithme de yacc n'arrive pas à choisir entre décalage et réduction, car les deux actions sont possibles et n'amènent pas l'analyse à une impasse. Un exemple typique de ce conflit a pour origine la grammaire usuelle de l'instruction conditionnelle.

Résolution par défaut : l'analyseur fait le décalage (c'est un comportement glouton).

2. Comment réduire (reduce/reduce conflict)

Ce conflit se produit lorsque l'algorithme ne peut pas choisir entre deux productions distinctes dont les membres droits permettent tous deux de réduire les symboles au sommet de la pile.

La résolution par défaut est : dans l'ordre où les règles sont écrites dans le fichier source pour yacc, on préfère la première production.

Les grammaires d'opérateurs

Pour rendre ces grammaires non ambiguë, il suffit de préciser le sens de l'associativité et la priorité de chaque opérateur.

En yacc, cela se fait par des déclarations %left et %right qui spécifient le sens d'associativité des opérateurs, l'ordre de ces déclarations donnant leur priorité : à chaque nouvelle déclaration les opérateurs déclarés sont plus prioritaires que les précédents.

Ainsi, on a ajouté des déclarations indiquant que +, −, ∗ et / sont associatifs à gauche et que la priorité de ∗ et / est supérieure à celle de + et −.

```
%left t_mult t_div
%left t_mult t_div
```

Analyse sémantique

Après l'analyse lexicale et l'analyse syntaxique, l'étape suivante dans la conception d'un compilateur est l'analyse sémantique dont la partie la plus visible est le contrôle de type. Des exemples de tâches liées au contrôle de type sont :

– construire et mémoriser des représentations des types définis par l'utilisateur, lorsque le langage le permet,

– traiter les déclarations de variables et fonctions et mémoriser les types qui leur sont appliqués,

– vérifier que toute variable référencée et toute fonction appelée ont bien été préalablement déclarées,

– vérifier que les paramètres des fonctions ont les types requis,

– contrôler les types des opérandes des opérations arithmétiques et en déduire le type du résultat,

– au besoin, insérer dans les expressions les conversions imposées par certaines règles de compatibilité,

– etc.

Représentation et reconnaissance des types

Une partie importante du travail sémantique qu'un compilateur fait sur un programme est :

– pendant la compilation des déclarations, construire des représentations des types déclarés dans le programme,

– pendant la compilation des instructions, reconnaître les types des objets intervenant dans les expressions.

La principale difficulté de ce travail est la complexité des structures à construire et à manipuler. En effet, dans les langages modernes les types sont définis par des procédés récursifs qu'on peut composer à volonté. Par exemple, en C on peut avoir des entiers, des adresses (ou pointeurs) d'entiers, des fonctions rendant des adresses d'entiers, des adresses de fonctions rendant des adresses d'entiers, etc.

Ainsi le compilateur que nous avons effectué dans le cadre de ce projet ne traite que les types entiers.

Dictionnaires (tables de symboles)

Dans les langages de programmation modernes, les variables et les fonctions doivent être déclarées avant d'être utilisées dans les instructions. Quelque soit le degré de complexité des

types supportés par un compilateur, celui-ci doit gérer une table de symboles, appelée aussi dictionnaire, dans laquelle se trouveront les identificateurs couramment déclarés, chacun associé à certains attributs, comme son type, son adresse et d'autres informations.

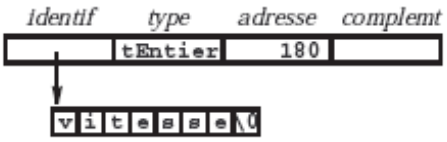


FIG 12: Une entrée dans le dictionnaire

Grosso modo le dictionnaire fonctionne ainsi :

– quand le compilateur trouve un identificateur dans une déclaration, il le cherche dans le dictionnaire en espérant ne pas le trouver (sinon c'est l'erreur identificateur déjà déclaré), puis il l'ajoute au dictionnaire avec le type que la déclaration spécifie,

- quand le compilateur trouve un identificateur dans la partie exécutable d'un programme, il le cherche dans le dictionnaire avec l'espoir de le trouver (sinon c'est l'erreur identificateur non déclaré), ensuite il utilise les informations que le dictionnaire associe à l'identificateur.

Malheureusement c'est normalement un peu plus compliquée que cela à cause de la diffrénce fondamentale entre les variables globales et les variables locales.

Dictionnaire global & dictionnaire local

Par manque de temps, nous n'avons malheureusement pas implémenté la gestion des variables locales mais nous expliquons le principe ici.
Un programme est essentiellement une collection de fonctions, entre lesquelles se trouvent des déclarations de variables. A l'intérieur des fonctions se trouvent également des déclarations de variables.

Les variables déclarées entre les fonctions et les fonctions elles-mêmes sont des objets globaux. Un objet global est visible depuis sa déclaration jusqu'à la fin du texte source, sauf aux endroits où un objet local de même nom le masque.

Les variables déclarées à l'intérieur des fonctions sont des objets locaux. Un objet local est visible dans la fonction où il est déclaré, depuis sa déclaration jusqu'à la fin de cette fonction ; il n'est pas visible depuis les autres fonctions. En tout point où il est visible, un objet local masque tout éventuel objet global qui aurait le même nom.

En définitive, quand le compilateur se trouve dans une fonction il faut posséder deux dictionnaires : un dictionnaire global, contenant les noms des objets globaux couramment déclarés, et un dictionnaire local dans lequel se trouvent les noms des objets locaux couramment déclarés (qui, parfois, masquent des objets dont les noms se trouvant dans le dictionnaire global).

Dans ces conditions, l'utilisation des dictionnaires que fait le compilateur se précise :

– Lorsque le compilateur traite la déclaration d'un identificateur i qui se trouve à l'intérieur d'une fonction, i est recherché dans le dictionnaire local exclusivement ; normalement, il ne s'y trouve pas (sinon, erreur : identificateur déjà déclaré). Suite à cette déclaration, i est ajouté

au dictionnaire local. Il n'y a strictement aucun intérêt à savoir si i figure à ce moment là dans le dictionnaire global.

– Lorsque le compilateur traite la déclaration d'un identificateur i en dehors de toute fonction, i est recherché dans le dictionnaire global, qui est le seul dictionnaire existant en ce point ; normalement, il ne s'y trouve pas (sinon, erreur : identificateur déjà déclaré). Suite à cette déclaration, i est ajouté au dictionnaire global.

– Lorsque le compilateur compile une instruction exécutable, forcément à l'intérieur d'une fonction, chaque identificateur i rencontré est recherché d'abord dans le dictionnaire local ; s'il ne s'y trouve pas, il est recherché ensuite dans le dictionnaire global (si les deux recherches échouent, erreur : identificateur non déclaré). En procédant ainsi on assure le masquage des objets globaux par les objets locaux.

- Lorsque le compilateur quitte une fonction, le dictionnaire local en cours d'utilisation est détruit, puisque les objets locaux ne sont pas visibles à l'extérieur de la fonction. Un dictionnaire local nouveau, vide, est créé lorsque le compilateur entre dans une fonction.

Implémentation du tableau pour la gestion des variables

L'implémentation la plus simple des dictionnaires consiste en un tableau dans lequel les identificateurs sont placés dans l'ordre où leurs déclarations ont été trouvées dans le texte source. Dans ce tableau, les recherches sont séquentielles. Lorsqu'il existe, le dictionnaire local se trouve au-dessus du dictionnaire global.

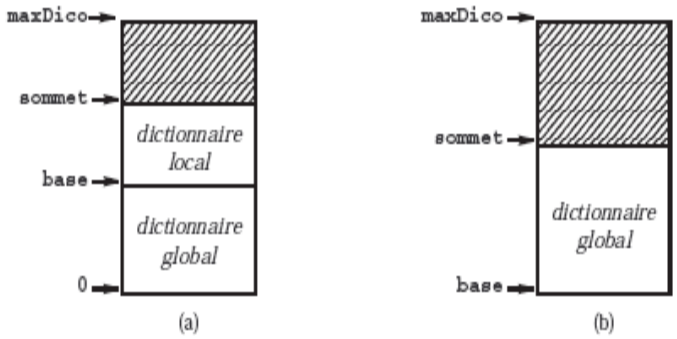


FIG 13: Dictionnaire quand on est à l'intéreur (a) et à l'extérieur des fonctions (b)

Dans notre programme, nous n'avons crée qu'un seul dictionnaire, celui gérant les variables globales (b).

Trois variables sont essentielles dans la gestion du dictionnaire :
max est le nombre maximum d'entrées possibles, sommet est le nombre d'entrées valides dans le dictionnaire ; on doit avoir sommet ≤ maxDico, base est le premier élément du dictionnaire du dessus (c'est-à-dire le dictionnaire local quand il y en a deux, le dictionnaire global quand il n'y en a qu'un).
Avec tout cela, la manipulation du dictionnaire devient simple. Les opérations nécessaires sont :
1. Recherche d'un identificateur pendant le traitement d'une déclaration (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction)

2. Recherche d'un identificateur pendant le traitement d'une expression exécutable : rechercher l'identificateur en parcourant dans le sens des indices décroissants

3. Ajout d'une entrée dans le dictionnaire (que ce soit à l'intérieur d'une fonction ou à l'extérieur de toute fonction) : après avoir vérifié que sommet < maxDico, placer la nouvelle entrée
4. Creation d'un dictionnaire local, au moment de l'entrée dans une fonction : faire base ← sommet,
5. Destruction du dictionnaire local, à la sortie d'une fonction : faire sommet ← base puis base ← 0.

Augmentation de la taille du dictionnaire

Une question technique assez agaçante qu'il faut régler lors de l'implémentation d'un dictionnaire par un tableau est le choix de la taille à donner à ce tableau, étant entendu qu'on ne connaît pas à l'avance la grosseur (en nombre de déclarations) des programmes que notre compilateur devra traiter.
La bibliothèque C offre un moyen pratique pour résoudre ce problème, la fonction realloc qui permet d'augmenter la taille d'un espace alloué dynamiquement tout en préservant le contenu de cet espace. Voici, à titre d'exemple, la déclaration et les fonctions de gestion réalisées; chaque fois que la place manque, le tableau est agrandi d'autant qu'il faut pour loger 25 nouveaux éléments :

```
%{ #include <stdio.h>
   #include <string.h>
   #include <stdlib.h>

typedef struct {
   char nom[30];
   int type; /* 0 pour variable et 1 pour constante */
   } variable;

#define PILE 256
#define INCREMENT 25

variable *dico;
/* pointeurs memoire */
   int deb = 0;
   int i = PILE/2 + 1;

void creerDico(void) {
   dico = calloc(PILE, sizeof(variable));
   if (dico == NULL)
           affichErr("Creation dico : pas assez de memoire");
   }

/* Pour agrandir la table des symboles */
   void agrandirDico(void) {
           maxDico = maxDico + INCREMENT;
           i = i + INCREMENT;
           dico = realloc(dico, maxDico);
           if (dico == NULL)
                   affichErr("Agrandir dico : pas assez de memoire");
   }

int affichErr(char * erreur)
   {
           fprintf(stderr, "Ligne %d : Erreur %s\n",nb_lignes,erreur);
           nb_erreurs++;
   }
```

Pour montrer une utilisation de tout cela, voici la fonction qui ajoute une entrée au dictionnaire :

```
void declaration(char * nom)
   {
            if (indiceVar(nom) == -1) {
                     if (deb>=(maxDico/2))
                     /* On agrandit la table des Symboles */
                              agrandirDico();

                     strcpy(dico[deb].nom,nom);
                     dico[deb].type = 0; /* Variable */
                     deb++;
            }
            else printf("Ligne %d : Attention variable %s deja declaree
\n",nb_lignes,nom);
   }
```

Nous avons choisi cette implémentation qui est la plus simple possible mais, malheureusement, pas la plus performante (la complexité des recherches est en moyenne de l'ordre de O(n); les insertions se font en temps constants).
Dans la pratique on recherche des implémentations plus efficaces, car un compilateur passe beaucoup de temps à rechercher des identificateurs dans les dictionnaires.

Une première amélioration aurait consisté à maintenir des tableaux ordonnés, permettant des recherches par dichotomie (la complexité d'une recherche devient ainsi O(log2 n), ce qui est beaucoup mieux).
On peut encore améliorer le procédé en utilisant des arbres binaires de recherche (plus rapide pour l'insertion) ou un addressage dispersé mais, faute de temps et d'analyse préalable, nous avons privilégié la solution la plus simple.

Les objets et leurs adresses

Classes d'objets

Les programmes manipulent trois classes d'objets :
1. Les objets statiques existent pendant toute la durée de l'exécution d'un programme ; on peut considérer que l'espace qu'ils occupent est alloué par le compilateur pendant la compilation.
Les objets statiques sont les fonctions, les constantes et les variables globales. Ils sont garnis de valeurs initiales : pour une fonction, son code, pour une constante, sa valeur et pour une variable globale, une valeur initiale explicitée par l'auteur du programme (dans les langages qui le permettent) ou bien une valeur initiale implicite, souvent zéro.
Les objets statiques peuvent être en lecture seule ou en lecture-écriture. Les fonctions et les constantes sont des objets statiques en lecture seule. Les variables globales sont des objets statiques en lecture-écriture.
On appelle espace statique l'espace mémoire dans lequel sont logés les objets statiques. Il est généralement constitué de deux zones : la zone du code, où sont les fonctions et les constantes, et l'espace global, où sont les variables globales.
L'adresse d'un objet statique est un nombre entier qui indique la première (parfois l'unique) cellule de la mémoire occupée par l'objet. Elle est presque toujours exprimée comme un décalage par rapport au d´ebut de la zone contenant l'objet en question.

2. Les objets automatiques sont les variables locales des fonctions, ce qui comprend :
– les variables déclarées à l'intérieur des fonctions,

– les arguments formels de ces dernières.
Ces variables occupent un espace qui n'existe pas pendant toute la durée de l'exécution du programme, mais uniquement lorsqu'il est utile. Plus précisément, l'activation d'une fonction commence par l'allocation d'un espace, appelé espace local de la fonction, de taille suffisante pour contenir ses arguments et ses variables locales.
Un espace local nouveau est alloué chaque fois qu'une fonction est appelée, même si cette fonction était déjà active et donc qu'un espace local pour elle existait déjà (c'est le cas d'une fonction qui s'appelle elle-même, directement ou indirectement). Lorsque l'activation d'une fonction se termine son espace local est détruit.

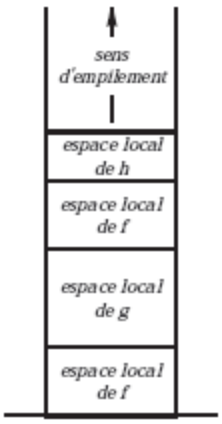


FIG 14: Empilement d'espaces locaux

(f a appelé g qui a appelé f qui a appelé h)

3. Les objets dynamiques sont alloués lorsque le programme le demande explicitement (par exemple à travers la fonction malloc de C ou l'opérateur new de Java et C++). Si leur destruction n'est pas explicitement demandée ces objets existent jusqu''a la terminaison du programme.

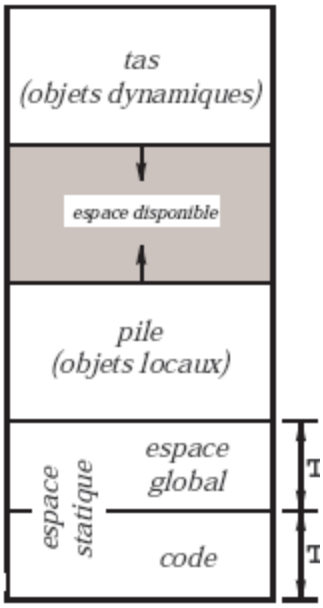


FIG 15: Organisation de la mémoire

– le code (espace statique en lecture seule), contenant le programme et les constantes,
– l'espace global (espace statique en lecture-écriture), contenant les variables globales,
– la pile (stack), contenant variables locales,
– le tas (heap), contenant les variables allouées dynamiquement.

Compilation séparée et édition de liens

Tout identificateur apparaissant dans une partie exécutable d'un programme doit avoir été préalablement déclaré. La d´eclaration d'un identificateur i, que ce soit le nom d'une variable locale, d'une variable globale ou d'une fonction, produit son introduction dans le dictionnaire adéquat, associé à une adresse, notons-la adri . Par la suite, le compilateur remplace chaque occurrence de i dans une expression par le nombre adri .

On peut donc penser que dans le code qui sort d'un compilateur les identificateurs qui se trouvaient dans le texte source ont disparu et, de fait, tel peut être le cas dans les langages qui obligent à mettre tout le programme dans un seul fichier.

Mais les choses sont plus compliquées dans les langages, comme C ou Java, où le texte d'un programme peut se trouver éclaté dans plusieurs fichiers sources destinés à être compilés indépendamment les uns des autres.

En effet, dans ces langages il doit être possible qu'une variable ou une fonction déclarée dans un fichier soit mentionnée dans un autre. Cela implique qu'à la fin de la compilation il y a dans le fichier produit quelque trace des noms des variables et fonctions mentionnées dans le fichier source.

Notez que cette question ne concerne que les objets globaux. Les objets locaux, qui ne sont déjà pas visibles en dehors de la fonction dans laquelle ils sont déclarés, ne risquent pas d'être visibles dans un autre fichier.

Le principal intéressé par cette affaire n'est pas le compilateur, mais un outil qui lui est associé, l'éditeur de liens (ou linker) dont le rôle est de concaténer plusieurs fichiers objets, résultats de compilations séparées, pour en faire un unique programme exécutable, en

vérifiant que les objets référencés mais non définis dans certains fichiers sont bien définis dans d'autres fichiers, et en complétant de telles références insatisfaites par les adresses des objets correspondants.

Faute de temps, le langage dont nous écrivons le compilateur ne supportera pas la compilation séparée. Nous n'avons donc pas besoin d'éditeur de liens dans notre système.

Machine à registre

Les langages évolués permettent l'écriture d'expressions en utilisant la notation algébrique comme X = Y + Z , cette formule signifiant ajoutez le contenu de Y à celui de Z et rangez le résultat dans X (X, Y et Z correspondent à des emplacements dans la mémoire de l'ordinateur).
Une telle expression est trop compliquée pour le processeur, il faut la décomposer en des instructions plus simples. La nature de ces instructions plus simples dépend du type de machine dont on dispose. Relativement à la manière dont les opérations sont exprimées.

Les machines à registres possèdent un certain nombre de registres, notés ici R1, R2, etc., qui sont les seuls composants susceptibles d'intervenir dans une opérations (autre qu'un transfert de mémoire à registre ou réciproquement) à titre d'opérandes ou de résultats. Inversement, n'importe quel registre peut intervenir dans une opération arithmétique ou autre ; par conséquent, les instructions qui expriment ces opérations doivent spécifier leurs opérandes. L'affectation X = Y + Z devra être traduite en quelque chose comme (notant X, Y et Z les adresses des variables X , Y et Z) :

```
MOVE  Y,R1    // déplace la valeur de Y dans R1
MOVE  Z,R2    // déplace la valeur de Z dans R2
ADD   R1,R2   // ajoute R1 à R2
MOVE  R2,X    // déplace la valeur de R2 dans X
```

Gestion du if et du while

En fonction de la machine à registre précédemment définie, nous avons crée les règles de grammaires non ambigus suivantes :

```
If :
.       Entete_If  Corps_gen {gerer_ifelse($1);} Corps_Else {gerer_else($1+1);}
;

Entete_If :
.       t_if t_par_ouv Condition t_par_fer        {$$ = gerer_if($3);}
;

Corps_gen :
.       Corps_fct
.       | Inst
;

Corps_Else :
.       t_else Corps_fct
.       | t_else Inst
;

While :
.        Entete_While Corps_gen {fin_while($1+1);}
;

Entete_While :
 .      t_while t_par_ouv {cond_while();} Condition t_par_fer {$$=corps_while($4
;
```

FIG 16: Règles de grammaires du if et du while

Afin de pouvoir mémoriser l'instruction à laquelle il faut aller au cas où la condition serait vraie, nous avons aussi créer une nouvelle structure :

```
/* Les struct */

typedef struct{
.       int num_inst
} entree;
```

FIG 17: Structure pour la gestion des adresses de sauts

Nous avons donc implémenté les fonctions permettant la gestion de ces instructions un peu différentes car on va devoir effectuer ou non un saut en fonction de l'évaluation de la condition du if ou du while :

```
void cond_while()
{
.	etiquette[p+1].num_inst = num_inst;.
.	 /*On connait l'adresse du saut de fin */
.	p = p+2;
}

int corps_while(int adr)
{
.	fprintf(f,"JMF %d %d\n",adr,p-2); /* si adr faux, jump a la fin du while */
.	num_inst++;
.	return p-2;
}

void fin_while(int id)
{
.	fprintf(f,"JMP %d\n",id);
.	num_inst++;
.	etiquette[id-1].num_inst = num_inst;.
}

int gerer_if(int adr)
{
.	fprintf(f,"JMF %d %d\n",adr,p); /* si adr faux, jump au debut du else */
.	num_inst++;
.	p = p+2; /*pour le jump du else*/
.	return p-2;
}

void gerer_ifelse(int id)
{
.	fprintf(f,"JMP %d\n",id+1); /*saut a la fin du else */
.	num_inst++;
.	etiquette[id].num_inst = num_inst;
}

void gerer_else(int id)
{
.	etiquette[id].num_inst = num_inst;
}
```

Cependant, afin de connaître le numéro d'instructions à laquelle on doit effectuer le saut, il faut d'abord avoir effectuer l'analyse syntaxique de l'ensemble du programme dans le but de connaître le nombre d'instructions, c'est pourquoi, nous avons eu besoin de créer une fonction qui va générer le code assembleur final en analysant les instructions jump présente dans ce même code :

```
void generer_code()
{
.	char ligne[30];
.	char inter[30];
.	char param[8];

.	int num = 0;
.	int c;

.	fseek(f,0,0);
.	fic = fopen("asm.tmp","w");
.
.	while (num < num_inst-1) {
.	.	num++;
.	.	fgets(ligne,29,f);
.	.	if(strncmp("JMF",ligne,3)==0){
.	.	.	for (c=8; c<30; c++) inter[c-8] = ligne[c];
.	.	.	strncpy(param,ligne,7);
.	.	.	param[7] = '\0';
.	.	.	fprintf(fic,"%s %d\n",param,etiquette[atoi(inter)].num_inst);
.	.	}
.	.	else if (strncmp("JMP",ligne,3)==0){
.	.	.	for(c=4;c<30;c++) inter[c-4] = ligne[c];
.	.	.	fprintf(fic,"JMP %d\n",etiquette[atoi(inter)].num_inst);
.	.	}
.	.	else fprintf(fic,"%s",ligne);
.	}
.	rename("asm.tmp","asm.txt");
}
```

C'est pourquoi nous avons du gérer les instructions if et while en deux temps. Dans un premier temps, nous effectuons tout simplement l'analyse syntaxique et écrivons l'instruction jump dans le code associé mais sans connaitre, à ce moment là, l'instruction vers laquelle aller puis, dans un deuxième temps, après analyse complète du fichier nous reprenons le code assembleur pour initialiser les jump avec la bonne valeur de saut.

Interpréteur

L'interpréteur va permettre l'analyse des instructions assembleurs générés. Après avoir compris le fonctionnement de lex et yacc, il a été plutôt triviale de réaliser son implémentation et cela ne nous a pas prise beaucoup de temps.

```
    #include "interpreteur.yacc.h"
entier_decimal                  . [0-9]
%%

AFC     .       .       return t_afc;
COP     .       .       return t_cop;
ADD     .       .       return t_add;
SOU     .       .       return t_sou;
MUL     .       .       return t_mul;
DIV     .       .       return t_div;
PRI     .       .       return t_pri;
JMP     .       .       return t_jmp;
JMF     .       .       return t_jmf;
INF     .       .       return t_inf;
SUP     .       .       return t_sup;
EQU     .       .       return t_equ;

" "             .       ;
\n              .       return t_fin_ins
```

FIG 18: Définition régulière pour l'interprétation du code assembleur (lex)

Ensuite, dans la partie sur yacc, nous avons réalisé les règles de grammaires suivantes :

```
Lignes: Ligne Lignes
      |
;

Ligne: instr t_fin_inst
;

instr:
      t_add t_entier t_entier t_entier      {place(ADD,$2,$3,$
      | t_mul t_entier t_entier t_entier    {place(MUL,$2,$3,$
      | t_sou t_entier t_entier t_entier    {place(SOU,$2,$3,$
      | t_div t_entier t_entier t_entier    {place(DIV,$2,$3,$
      | t_cop t_entier t_entier      .      {place(COP,$2,$3,0
      | t_afc t_entier t_entier      .      {place(AFC,$2,$3,0
      | t_pri t_entier       .       .      {place(PRI,$2,0,0)
      | t_jmf t_entier t_entier      .      {place(JMF,$2,$3,0
      | t_jmp t_entier       .       .      {place(JMP,$2,0,0)
      | t_inf t_entier t_entier t_entier    {place(INF,$2,$3,$
      | t_sup t_entier t_entier t_entier    {place(SUP,$2,$3,$
      | t_equ t_entier t_entier t_entier    {place(EQU,$2,$3,$
;
```

FIG 19: Règles de grammaires pour l'interpréteur du code assembleur (yacc)

Nous avons implémenté une table des symboles à taille dynamique (comme pour la réalisation du compilateur) ainsi qu'une fonction permettant de traiter et d'afficher les bonnes valeurs en fonction de l'instruction voulue.

```
switch (code[numInst].inst) {

case AFC :
.       printf("AFC - ligne %d\n",numInst
break;

case COP :
.       printf("COP - ligne %d\n",numInst
break;

case ADD :
.       printf("ADD - ligne %d\n",numInst
break;

case SOU :
.       printf("SOU - ligne %d\n",numInst
break;
```

FIG 20: Exemple succinte de traitement du code assembleur

ARCHITECTURES MATERIELLES

« Conception d'un microprocesseur RISC avec pipe-line »

MIETLICKI Pascal - OTHMANI Jihed

4 GI – Groupe A

2007/2008

REALISATION DU MICROPROCESSEUR

Jeu d'instructions

Voici un tableau récapitulatif du jeu d'instructions que notre chemin de données doit être capable d'exécuter.

Opération	Code	OP	A	B	C	Description
Affectation	X"00"	AFC	Ri	j	_	[Ri]=j
Copie	X"01"	COP	Ri	Rj	_	[Ri]= [Rj]
Addition	X"02"	ADD	Ri	Rj	Rk	[Ri]= [Rj]+ [Rk]
Multiplication	X"03"	MUL	Ri	Rj	Rk	[Ri]= [Rj]* [Rk]
Division	X"04"	DIV	Ri	Rj	Rk	[Ri]= [Rj]/[Rk]
Soustraction	X"05"	SUB	Ri	Rj	Rk	[Ri]= [Rj]- [Rk]
Chargement	X"06"	LOAD	Ri	@j	_	[Ri]= [@j]
Sauvegarde	X"07"	STORE	@i	Rj	_	[@i]= [Rj]
NOP	X"08"	NOP	_	_	_	
Saut	X"09"	JMP	j	_	_	

Les codes opération ont été déclaré comme des constantes dans un package à part appelé code_op pour améliorer la lisibilité et la compréhension des instructions.

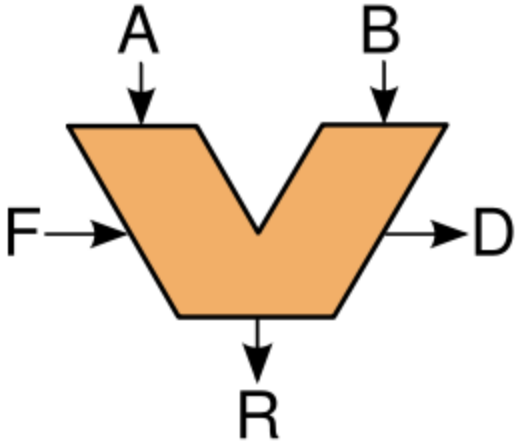
Voici un extrait :

```
package code_op is
  constant OP_AFC          : std_logic_vector(7 downto 0) := X"00";
  constant OP_COP          : std_logic_vector(7 downto 0) := X"01";
  constant OP_ADD          : std_logic_vector(7 downto 0) := X"02";
  constant OP_MUL          : std_logic_vector(7 downto 0) := X"03";
  constant OP_DIV          : std_logic_vector(7 downto 0) := X"04";
…
end code_op;
```

Les composants du microprocesseur

Unité Arithmétique et logique

Une UAL possède deux entrées A et B sur lesquelles on présentera les données à traiter. L'entrée F désignera l'opération à effectuer. Enfin, celle-ci possède deux sorties, R qui sera le résultat de l'opération, et D les drapeaux qui indiqueront soit qu'il y a eu erreur : division par zéro, dépassement de capacité (flag O), soit des codes conditions, résultat nul (flag Z), résultat négatif (flag N), retenu (C).



L'UAL réalisée est capable de réaliser les quatre opérations arithmétiques de base. Ces dernières sont réalisées bit à bit et non pas à partir des fonctions correspondantes fournies dans la librairie de Xilinx.

F(2 :0)	Opération

000	Addition
	`r(i):= (A(i) xor B(i)) xor c;`
	`c := (A(i) and B(i)) or (c and (A(i) or B(i)));`

001	Soustraction
	`r(i):= (A(i) xor not B(i)) xor c;`
	`c := (A(i) and not B(i)) or (c and (A(i) or not B(i)));`

010	Multiplication `if B(i) = '1' then r((i+8) downto i) := r((i+8) downto i) + ('0'&A);`

011	Division
	`if A_16(i downto i-7)>= B then r(i-7):='1';`
	`A_16(i downto i-7) := A_16(i downto i-7) - B;`
	`else r(i-7):='0';`

A la fin du calcul, on teste le résultat et on met à 1 les flags correspondants.

`if(r=X"0000") then NOZC(1) <= '1';` On met le flag Z à 1 si le résultat est nul.

`if(r(15 downto 8)>0) then NOZC(2)<='1';` On met le flag O à 1 si on détecte un débordement.

`NOZC(0) <= c;` Le flag C prend la valeur du retenu.

`NOZC(3)<=r(7);` On met le flag N à 1 si le bit de signe (8ème bit de R) est égal à 1.

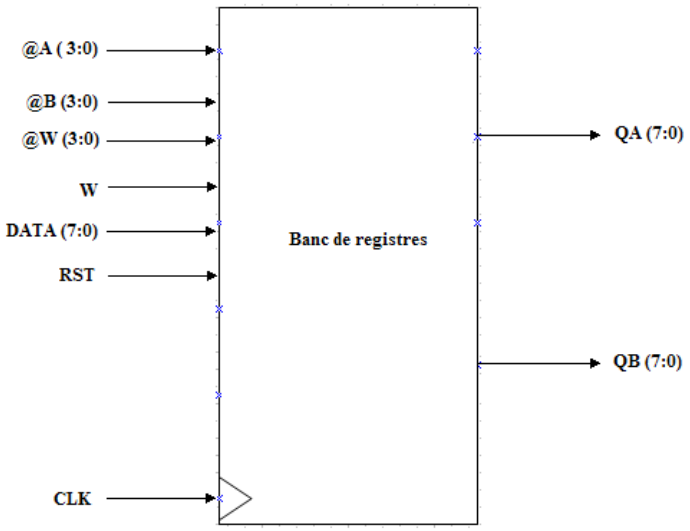
Banc de registres :

On a réalisé un banc de 16 registres de 8 bits avec un double accès en lecture et écriture.

Le signal RST est actif à 0, il permet de mettre à zéro le contenu de tous les registres.

Les entrées @A et @B permettent de lire deux registres simultanément, les valeurs correspondantes sont propagés vers les sorties QA et QB.

W spécifie si une écriture doit être réalisée, elle est active à 1. Quand cette entrée est active, les données présentent sur DATA sont copiés dans le registre d'adresse @W.

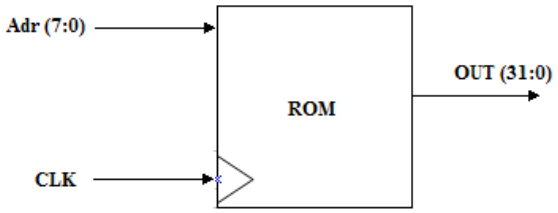


Mémoire des instructions :

Cette mémoire est assimilable à une mémoire ROM.

A chaque top d'horloge, on obtient en sortie le contenu de la case mémoire d'adresse Adr.

Cette mémoire contiendra les instructions de notre programme.



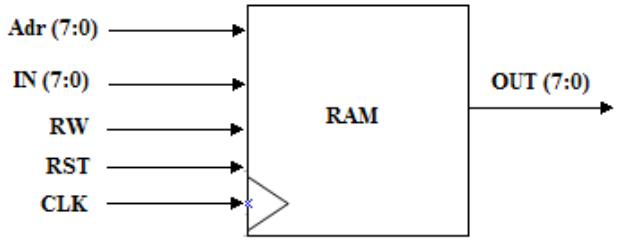
Mémoire des données :

La mémoire des données est assimilable à une RAM, elle permet un accès en lecture et écriture.

Lors d'une lecture (RW='1'), la sortie out reçoit le contenu de la zone mémoire d'adresse « Adr ».

Lors d'une écriture (RW='0'), on écrit les données se trouvant à l'entrée IN à la zone mémoire d'adresse « Adr ».

Le reset est actif à 0 et est synchrone avec l'horloge CLK tout comme la lecture et l'écriture.



Pipeline :

Le **pipeline** (ou *pipelining*) est une technologie visant à permettre une plus grande vitesse d'exécution des instructions en parallélisant des étapes.

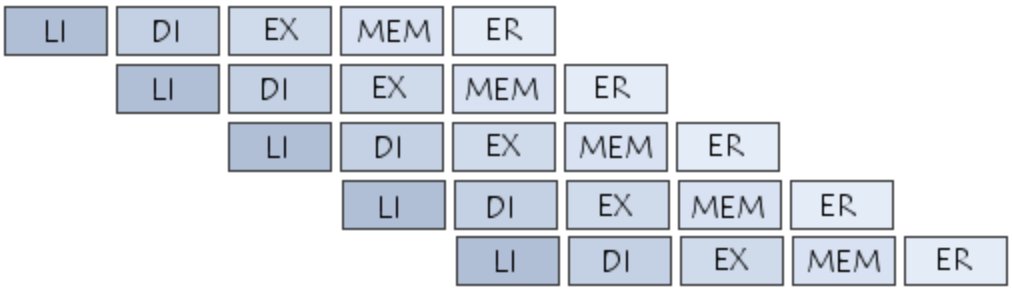
Pour comprendre le mécanisme du pipeline, il est nécessaire au préalable de comprendre les phases d'exécution d'une instruction. Les phases d'exécution d'une instruction pour un processeur contenant un pipeline « classique » à 5 étages sont les suivantes :

- **LI** : (*Lecture de l'Instruction* (en anglais *FETCH instruction*) depuis le cache ;
- **DI** : *Décodage de l'Instruction* (*DECODe instruction*) et recherche des opérandes (Registre ou valeurs immédiate);
- **EX** : *Exécution de l'Instruction* (*EXECute instruction*) (si ADD, on fait la somme, si SUB, on fait la soustraction, etc.);
- **MEM** : *Accès mémoire* (*MEMory access*), écriture dans la mémoire si nécéssaire ou chargement depuis la mémoire ;
- **ER** : *Ecriture* (*Write instruction*) de la valeur calculée dans les registres.

Les instructions sont organisées en file d'attente dans la mémoire, et sont chargées les unes après les autres.

Grâce au pipeline, le traitement des instructions nécessite au maximum les cinq étapes précédentes. Dans la mesure où l'ordre de ces étapes est invariable (LI, DI, EX, MEM et ER), il est possible de créer dans le processeur un certain nombre de circuits spécialisés pour chacune de ces phases.

L'objectif du pipeline est d'être capable de réaliser chaque étape en parallèle avec les étapes amont et aval, c'est-à-dire de pouvoir lire une instruction (LI) lorsque la précédente est en cours de décodage (DI), que celle d'avant est en cours d'exécution (EX), que celle située encore précédemment accède à la mémoire (MEM) et enfin que la première de la série est déjà en cours d'écriture dans les registres (ER).



Il faut compter en général 1 à 2 cycles d'horloge (rarement plus) pour chaque phase du pipeline, soit 10 cycles d'horloge maximum par instruction. Pour deux instructions, 12 cycles d'horloge maximum seront nécessaires (10+2=12 au lieu de 10*2=20), car la précédente instruction était déjà dans le pipeline. Les deux instructions sont donc en traitement dans le processeur, avec un décalage d'un ou deux cycles d'horloge). Pour 3 instructions, 14 cycles d'horloge seront ainsi nécessaires, etc.

Le principe du pipeline est ainsi comparable avec une chaîne de production de voitures. La voiture passe d'un poste de travail à un autre en suivant la chaîne de montage et sort complètement assemblée à la sortie du bâtiment. Pour bien comprendre le principe, il est nécessaire de regarder la chaîne dans son ensemble, et non pas véhicule par véhicule. Il faut ainsi 3 heures pour faire une voiture, mais pourtant une voiture est produite toute les minutes !

Il faut noter toutefois qu'il existe différents types de pipelines, de 2 à 40 étages, mais le principe reste le même.

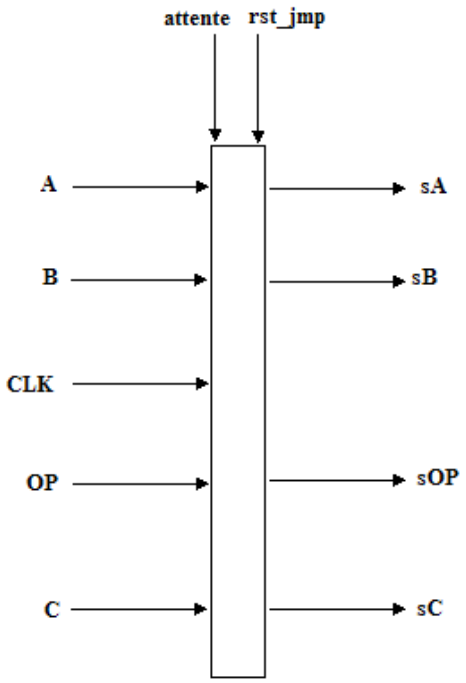
On a conçu un composant pipeline qui possède sept entrées et quatre sorties.

Les entrées A,B,C sont codés sur 8 bits et peuvent représenter soit un numéro de registre, une adresse ou bien une valeur dans le cas d'une affectation.

L'entrée « rst_jmp » permet d'injecter une instruction « nop » sur les sorties du pipeline dans le cas où on détecte un saut conditionnel ou unconditionnel afin d'éviter l'exécution d'une mauvaise instruction.

L'entrée « attente » permet de bloquer le pipeline afin de traiter les aléas, ce qui revient à attendre la fin d'une instruction avant l'éxécution d'une autre. Typiquement, attendre la fin d'une écriture sur un registre i avant sa lecture.

Les sorties sA, sB, sC et sOP ne sont qu'une recopie des entrées A, B, C et OP dans le cas où aucune aléa ni saut n'ont été détecté.



Chemin de données :

Le chemin de données a été conçu en plusieurs étapes. Chaque étape correspond à la prise en charge d'une nouvelle instruction par notre micro-processeur.

D'après les consignes données dans le fascicule, on a remarqué que le chemin de données peut être décomposé en plusieurs étages. Ces derniers peuvent être modifiés séparément au fur et à mesure de l'avancement du projet.

Partant de là, on a conçu quatre étages :

1/ Le premier est composé du compteur d'instructions, de la rom (mémoire des instructions), d'un pipeline et du banc de registres.

2/ Le deuxième est composé d'un pipeline et l'unité arithmétique et logique.

3/ Le troisième est composé d'un pipeline et la mémoire des données.

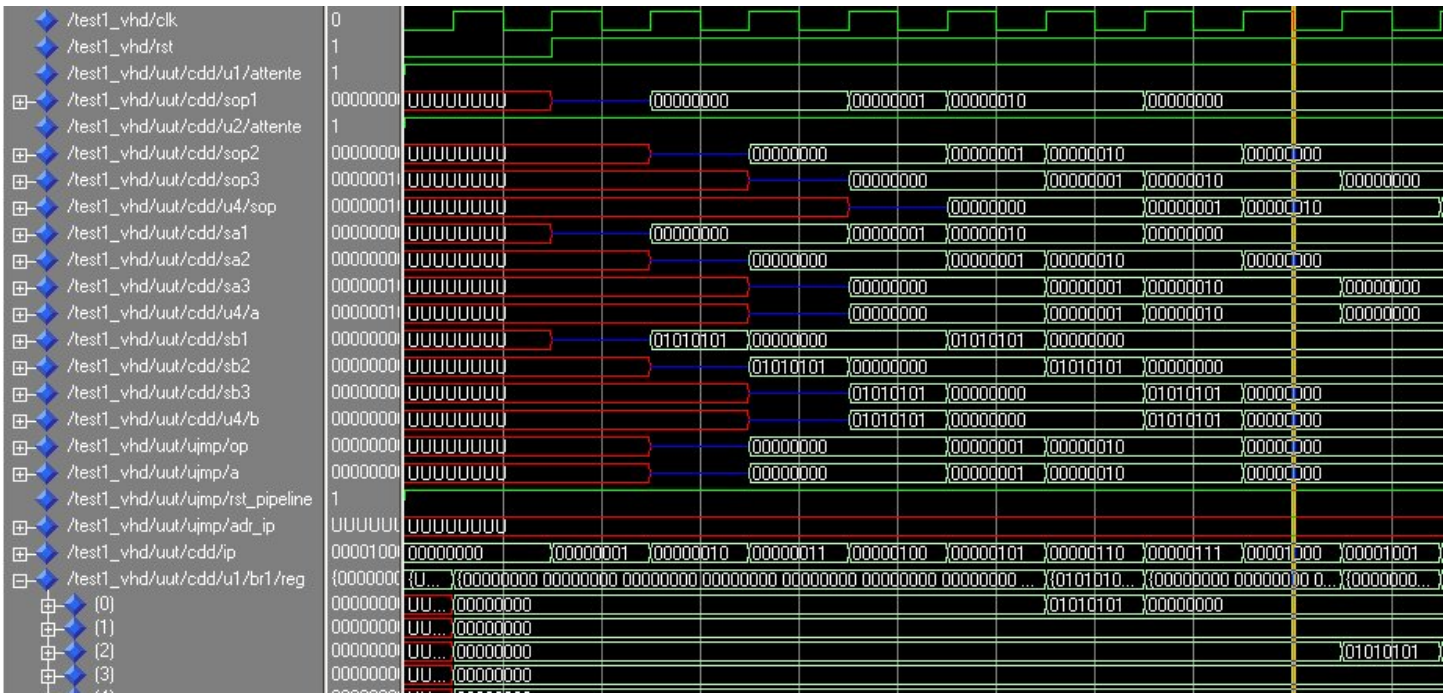
4/ Le quatrième est composé simplement d'un pipeline et un composant LC.

Remarque : Les différents LCs et multiplexeurs ont été directement codés dans les différents étages, aucun composant n'a été créé à cet effet.

– Exemple d'exécution de quelques instructions :

Code : Voici le contenu de la mémoire des instructions correspondant à un test des instructions AFC, COP et ADD.

variable rom : memInstructions:=(OP_AFC&X"005500", OP_AFC&X"005500", OP_COP&X"010000", OP_ADD&X"020001", OP_ADD&X"020001", others => X"00000000");



On affecte la valeur 55 au registre 0, on copie dans le registre 1 le contenu du registre 0 et finalement, on additionne les valeurs continues dans le registres 0 et 1 dans le registre 1.

Gestion des aléas

Dans un pipeline, la prochaine instruction commence avant que la précédente se termine. Ceci pose un problème si les opérandes d'une instruction dépendent du résultat de l'autre instruction. Le fait d'exécuter dan pipeline change l'ordre d'exécution réel des choses. (Normalement l'instruction 1 se fait avant l'instruction 2, mais dans un pipeline l'étage 5 de l'instruction 1 se fait après l'étage 2 de l'instruction 2, par exemple).

Ce changement d'ordre entraine deux types d'aléas :

2. Aléa de données : utilisation de la mauvaise donnée.
3. Aléa de branchement : une instruction incorrecte est recherchée (car le PC n'est pas modifié au même moment lors d'un branchement) .

Les types de dépendances pouvant produire des aléas de données :

- Lecture après écriture : (Read After Write - RAW) une instruction utilise les données produites par une instruction précédente. S'appel aussi dépendance de flux de données.

- Écriture après lecture : (Write After Read - WAR) une instruction écrit dans un emplacement qui est utilisé comme opérande dans une instruction précédente. S'appel aussi anti-dépendance.

- Écriture après écriture : (Write After Write - WAW) une intruction écrit dans le même emplacement qu'un instruction précédente. S'appel aussi dépendance de sortie.
  Peut-il y avoir un aléa lorsqu'il y a lecture après lecture? Non.

Solution aux aléas de données :

- Programme écrit pour qu'aucun aléa se produise.

- Détection automatique à l'exécution :
  - calage : attente (perte de performance)
  - transit de données ("data forwarding") : passer les résultats sans passer par les registres

Solution apportée :

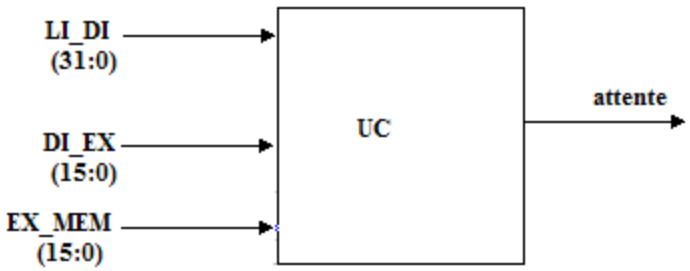
Pour notre projet, on a choisit de concevoir une unité de contrôle qui permet de détecter les aléas et de bloquer les pipelines afin d'éviter le chevauchement d'instructions critiques.

Cette unité possède trois entrées :

- LI_DI : c'est l'entrée du premier pipeline.
- DI_EX : c'est l'entrée du deuxième pipeline.
- EX_MEM : c'est l'entrée du troisième pipeline.

Et une sortie :

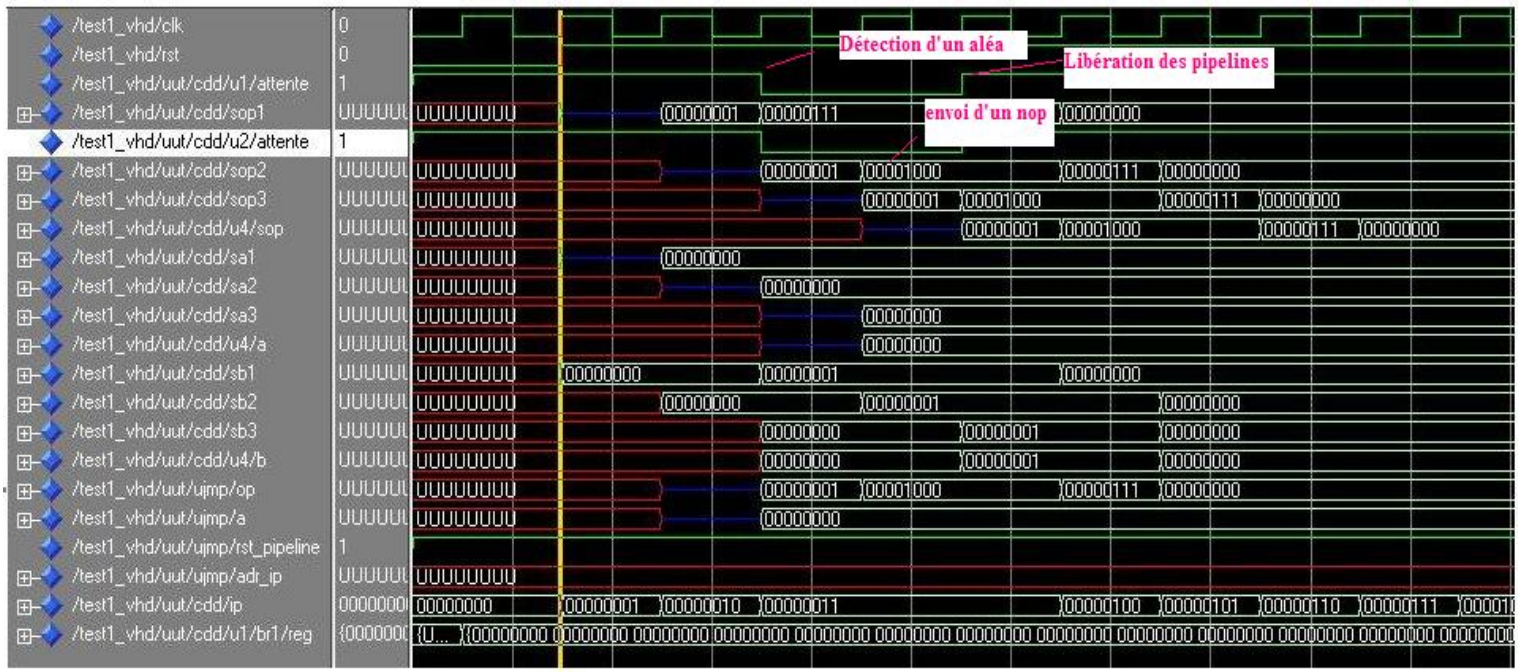
- Attente : codé sur un seul bit et active à '0'. Cette sortie est égale à '0' lorsqu'on détecte une opération de lecture au niveau du premier pipeline et une opération d'écriture au niveau du deuxième ou troisième pipeline sur le même registre.
- Quand le signal attente est actif, on bloque l'instruction au niveau de du premier pipeline, on envoi un NOP et bloque le compteur d'instructions bien sur.



Exemple :

- Voici le code à exécuter contenu dans la mémoire des instructions:

```
variable rom : memInstructions:=(OP_COP&X"000000", OP_STORE&X"000100",
others => X"00000000");
```



Gestion des sauts

On n'a géré que les sauts inconditionnels vu qu'ils sont les plus simples à réaliser et qu'il ne nous restait pas beaucoup de temps lors de la dernière séance.

Pour ce, on a réalisé un composant appelé « ujmp » qui se charge de détecter les instructions de sauts, de réinitialiser les pipelines et d'injecter au niveau du compteur d'instructions le bon numéro d'instruction à exécuter.

Exemple :

- Voici le code à exécuter contenu dans la mémoire des instrctions:

variable rom : memInstructions:=(OP_ADD&X"000100", OP_JMP&X"030000", OP_MUL&X"000100" , OP_MUL&X"000100", OP_MUL&X"000100", others => X"00000000");

On commence par une instruction d'addition puis on réalise un saut inconditionnel à l'instruction

numéro 3.

Conclusion générale

Au travers de ce projet, nous avons intégré de manière concrète non seulement le fonctionnement d'un compilateur avec les différents analyseurs (lexical, syntaxique et sémantique) ainsi que la structure et le fonctionnement d'un micro processeur. Ce travail nous a également conduit à nous poser des problèmes purement logiques mais aussi techniques (synchronisation des modules entre eux).

Il faut savoir que, bien que le projet soit intéressant, il est aussi très conséquent. Il n'a pas été de tout repos de devoir apprendre à gérer les outils proposés tel que lex et yacc que l'on utilisait pour la première fois. Ce projet a nécessité une importante charge de travail.

Néanmoins, nous avons su répondre aux attentes du cachier des charges et même un peu plus grâce à la gestion des sauts (if et while) ainsi que la taille du dictionnaire de données gérée dynamiquement (automatiquement augmentée en cas de dépassement) pour la partie lex et yacc et la gestion des aléas et des jumps inconditionnels pour la réalisation du microprocesseur.

Sur le plan pédagogique, ce projet nous a permis d'approfondir les concepts vus en cours et de les appliquer au sein d'un travail concret. Nous avons ainsi pu découvrir comment mettre en œuvre les apprentissages étudiés au préalable avec les outils et points importants associés à ces concepts.