Introduction

Le projet de compilation proposé consiste à développer :

- un compilateur de programmes, écrits dans un langage dit langage PROJET, qui produit un code objet de ces programmes ; le langage Projet permet d'écrire des applications composées d'unités compilables séparément,
- un éditeur de liens dont le rôle est de regrouper dans un unique programme exécutable les différentes unités objets résultant de la compilation séparée des composants d'une application.

Les programmes en langage PROJET ainsi compilés peuvent être exécutés sur une machine à pile virtuelle qui est ici fournie.

La base du compilateur à réaliser est générée par le logiciel ANTLR, présenté brièvement dans le chapitre 1 (page 5). ANTLR engendre, à partir d'une grammaire décrivant un langage, un analyseur lexical et un analyseur syntaxique pour ce langage. Vous devez utiliser le mécanisme de points de génération, à ajouter à la grammaire, pour réaliser des contrôles sémantiques supplémentaires et pour effectuer la production du code objet au cours de l'analyse syntaxique.

Le langage PROJET, langage des programmes que l'on souhaite compiler, est décrit dans le chapitre 2 (page 11).

Le code machine, obtenu après compilation, est destiné à être exécuté sur la machine virtuelle MAPILE dont le fonctionnement est décrit dans le chapitre 3 (page 16) ; un simulateur de cette machine est disponible sur le réseau.

Dans le chapitre 4 (page 23), on introduit les éléments utiles à la compilation des programmes écrits en langage PROJET. On propose d'abord des structures de données (programme objet, table des symboles, pile des reprises) puis on s'intéresse aux divers traitements effectués au cours de la compilation des constructions de base (déclaration/occurrence d'un identificateur, calcul/contrôle de type, production de code).

Le chapitre 5 (page 33) décrit les principes de la compilation des procédures : déclaration des procédures, appels de procédures.

Enfin, le chapitre 6 (page 39) est consacré aux implications induites par la possibilité d'écrire des unités compilables séparément : le compilateur doit préparer les informations utiles au regroupement des unités au sein d'une même application et il faut développer un éditeur de liens chargé de procéder à ce regroupement.

Le travail à effectuer avec les rendus demandés sont explcités à la fin du document.

Table des matières

1

Utilisation de ANTLR et compilateur

L'ossature du compilateur à produire est constituée par l'analyseur ; ce dernier est organisé en deux niveaux :

- l'analyseur lexical qui "découpe" le programme source en une séquence de terminaux (appelés items lexicaux ou tokens),
- l'analyseur syntaxique qui vérifie que cette séquence appartient bien au langage engendré par la grammaire du langage source (PROJET en ce qui nous concerne).

Le logiciel ANTLR (ANother Tool for Language Recognition) permet d'automatiser la phase d'écriture de ces analyseurs qui sont déduits directement de la grammaire.

ANTLR, développé par Terence Parr (http ://www.antlr.org), est un générateur d'analyseurs qui lit une grammaire en entrée et la convertit en un programme qui peut reconnaître un texte et le traiter selon les règles de cette grammaire.

Les analyseurs générés sont des analyseurs descendants gauche-droite (DGD) utilisant la notion LL(*) qui permet de choisir sans ambiguïté la règle à appliquer avec un nombre quelconque de caractères d'anticipation (sur le texte à reconnaître). ANTLR autorise le choix du nombre k de caractères d'anticipation et nous nous limitons ici à l'analyse LL(1) où un seul caractère d'anticipation est connu.

ANTLR peut produire du code Java, Python, C, C++, etc. à partir d'une grammaire donnée. Nous utilisons ici le langage cible Java.

1.1 Grammaire d'entrée

Pour une grammaire `exemple`, ANTLR produit deux classes, l'une pour l'analyse lexicale et l'autre pour l'analyse syntaxique, resp. `exempleLexer.java` et `exempleParser.java`, à partir d'un ou de deux fichier(s), suffixé(s) par `.g`, décrivant la grammaire. En effet, les règles syntaxiques et lexicales peuvent appartenir à un même fichier `exemple.g` ou bien être séparées en deux fichiers `exempleLexer.g` et `exempleParser.g`.

Le fichier descriptif d'une grammaire contient l'ensemble des règles de production, précédé éventuellement de rubriques ANTLR permettant de spécifier un certain nombre de caractéristiques de l'analyseur souhaité. Les caractéristiques de l'écriture des règles peuvent être résumées ainsi :

- les productions sont regroupées, pour une même partie gauche de règle, sous la forme
  `partie-gauche : partie-droite1 | ... | partie-droiteN ;`
- pour l'analyse syntaxique, les non terminaux sont les identificateurs figurant en partie gauche des productions (l'axiome est la partie gauche de la première règle) et doivent commencer par une minuscule,

- pour l'analyse lexicale, les identificateurs figurant en partie gauche des productions, et correspondant à des items lexicaux, doivent commencer par une majuscule,
- on peut employer des expressions régulières pour l'écriture des parties droites :
  - (...)* dénotant la répétition 0 ou plusieurs fois,
  - (...)+ dénotant la répétition au moins une fois,
  - (...) ? notation ANTLR, équivalente à la notation usuelle 0/1, dénotant 0 ou 1 fois,
  - (... | ...) pour noter le OU.
- des caractères (ou suites de caractères) appartenant au vocabulaire du texte à analyser (donc des terminaux) peuvent figurer entre quotes '...' dans les parties droites des règles.

Exemple :
Soit la grammaire `Exp1` d'expressions arithmétiques décrite dans `Exp1Lexer.g` pour la partie lexicale et dans `Exp1Parser.g` pour la partie syntaxique.

Le fichier `Exp1Lexer.g` contient :

```
lexer grammar Exp1;
options { language = Java; k = 1; }
NBENTIER :  ( '0'..'9' )+ ;
IDENT : ('a' ..'z' | 'A'..'Z')+ ;
```

Le fichier Exp1Parser.g contient :

```
parser grammar Exp1;
options { language = Java;
        k = 1;
        tokenVocab = Exp1Lexer; }
exp     :    terme ( '+' terme | ' -' terme )* ;
terme   :  primaire ( '*' primaire | ' div' primaire )* ;
primaire:   NBENTIER | IDENT | '(' exp ')' ;
```

N.B. Les items lexicaux (terminaux de l'analyse syntaxique) sont donc ici `NBENTIER`, `IDENT`, ainsi que `'+'`, `'-'`, `'*'`, `'div'`, `'('` et `')'`. Les non-terminaux de l'analyse syntaxique sont `exp` (qui est l'axiome de Exp1), `terme` et `primaire`.

À partir de ces deux fichiers, ANTLR produit `Exp1Lexer.java` et `Exp1Parser.java` qui permettent la reconnaissance d'expressions appartenant au langage généré par la grammaire `Exp1`.
Ainsi `(a+2)*cpt` et `55-(x div 2)*(Max-Min)` sont des entrées reconnues comme mots du langage décrit, contrairement à `99+div x` qui n'est pas un mot de ce même langage.

1.2 Traitements supplémentaires et points de génération

Pour compléter la reconnaissance de mots du langage considéré par des traitements et/ou des contrôles supplémentaires, il est possible d'ajouter du code dans la grammaire du fichier .g lu par ANTLR. Ce code doit être écrit dans le langage cible (Java pour nous) et peut correspondre à l'appel de points de génération.

Dans le cadre du compilateur à réaliser, on place des appels aux points de génération dans la grammaire du langage PROJET pour effectuer les contrôles syntaxiques supplémentaires et pour produire le code objet.

L'emplacement de ces points de génération dans les règles de la grammaire indique à quels moments on souhaite intervenir au cours de l'analyse syntaxique :

- l'appel à un point de génération est de la forme `PtGen.pt(n)`, où n est un entier ≤ 1 et ≥ 255, et correspond à un appel de fonction en Java qui doit figurer entre accolades,
- on peut faire figurer des séquences de points de génération n'importe où dans les parties droites des règles. Par exemple, pour la règle :
  `a : NBENTIER b a | IDENT;`
  on peut écrire :

```
a : NBENTIER { PtGen.pt(4); PtGen.pt(7); } b { PtGen.pt(3); } a
        | { PtGen.pt(4); } IDENT ;
```

- le code du traitement à effectuer lors d'un appel à un point de génération par `{ PtGen.pt(n); }` doit figurer dans le fichier `PtGen.java` dont le squelette est fourni.

1.3 Construction du compilateur

La version 3.5.2 de ANTLR est utilisée pour générer l'analyseur, base du compilateur à réaliser, à partir de la grammaire fournie dans `projet.g` (grammaire décrite au chapitre 2).

Dans le cadre de ce projet de compilateur, les règles syntaxiques et lexicales appartiennent à un seul fichier `projet.g` comme ANTLR le permet.

La section `TP Projet de compilation` sous `Moodle L3-CMPL` contient l'archive JAR de la version 3.5.2 de ANTLR, la grammaire initiale `projet.g` et différentes classes et utilitaires. En particulier, un script, `g2java.bat` pour Windows et `g2java.sh` pour Linux, permet d'exécuter ANTLR sur la grammaire `projet.g` et de générer les analyseurs lexical et syntaxique `projetLexer.java` et `projetParser.java`. Le fichier `projet.tokens`, contenant le codage des unités lexicales sous forme d'entiers, est également produit. **Notez** que les chemins d'accès dans ces scripts doivent être mis à jour avec la configuration de votre projet CMPL.

La **partie lexicale** est à la fin du fichier `projet.g` et ne doit pas du tout être modifiée. Cette partie définit en particulier les terminaux `nbentier` et `ident`. Leurs attributs lexicaux respectifs `UtilLex.valEnt` et `UtilLex.numIdCourant` (ainsi que la méthode d'affichage de la chaîne correspondant à un identificateur `UtilLex.chaineIdent`) sont gérés à l'aide de la classe UtilLex (`UtilLex.class` fourni).

La **partie syntaxique** de la grammaire `projet.g` est celle qui nous intéresse ici, et par la suite, les références à la grammaire `projet.g` concernent toujours la partie syntaxique.

La grammaire `projet.g` ne doit être modifiée que par ajout des appels aux points de génération afin de réaliser un compilateur complet.

Rappel : ces appels correspondant à du code Java, ils doivent être mis entre accolades et respecter la syntaxe Java. Par ex, l'appel à un point de génération, numéroté 255, qui créé, en fin de compilation, les fichiers contenant le code produit en version exécutable (fichier .obj) et en version lisible (fichier .gen) figure ainsi dans la règle :

```
unite           :   unitprog {PtGen.pt(255);} | unitmodule {PtGen.pt(255);} ;
```

Dans la classe `PtGen.java`, dans la méthode `pt`, on trouve alors le code Java :

```
case 255 : po.constObj(); po.constGen();
```

Le code associé aux points de génération doit permettre de produire le code objet d'un programme source, écrit en langage PROJET, et d'effectuer les contrôles supplémentaires liés à la compilation du programme source.

Pour construire progressivement votre compilateur, vous devez, à chaque étape :

1. placer des appels aux points de génération dans les règles de la grammaire `Projet.g`,
2. fournir le code Java associé à ces points de génération dans le squelette `PtGen.java` fourni,
3. exécuter ANTLR à chaque ajout/modification d'appels aux points de génération dans `Projet.g` pour générer à nouveau les analyseurs, et donc votre compilateur, mis à jour. Notez que, sous Eclipse, il faut à chaque fois "rafraîchir" votre projet Java pour que ProjetParser.java et ProjetLexer.java soient mis à jour.

Le compilateur obtenu est exécuté via `Projet.java` (à ne pas confondre avec `Projet.g`) qui lit le nom du programme source à compiler et lance la compilation à partir de l'axiome `unité` de la grammaire `Projet.g`.

`Projet.java`, qui n'a pas à être modifié, contient :

```
class Projet {
    public static String nomSourceComplet;
    private static void UneCompilation (String nomDuSource) {
        ANTLRFileStream input = new ANTLRFileStream(nomSourceComplet);
        ProjetLexer lexer = new ProjetLexer(input);
        CommonTokenStream token_stream = new CommonTokenStream(lexer);
        ProjetParser parser = new ProjetParser(token_stream);
        PtGen.pt(0);        // appel au point de génération d'initialisation(s)
        parser.unite(); (...)   // unite = axiome de Projet.g
}
```

Notez que le point de génération numéroté 0 est systématiquement appelé au lancement d'une compilation. Il doit donc contenir les (ré-)initialisations nécessaires à toute nouvelle compilation.

Remarque : Le compilateur réalisé permet de vérifier qu'un programme en langage PROJET est syntaxiquement correct, et de produire le code objet exécutable de ce programme. Le compilateur effectue donc une sorte de traduction des programmes, du langage PROJET (cf. chapitre 2, page 11) en un langage dit de plus bas niveau (plus proche du fonctionnement de la machine), ici le langage de la machine d'exécution virtuelle MAPILE (cf. chapitre 3, page 16).

Exercice :
Soit la grammaire `Exp2.g` contenant :

$exp4$: $exp5$ $('+'$ $exp5$ | $'-'$ $exp5)*$;
$exp5$: $primaire$ $('*'$ $primaire$ | $'div'$ $primaire)*$;
$primaire$: $nbentier$ | $ident$ | $'('exp4')'$;

où, ici, la partie lexicale définit `nbentier` comme une suite de chiffres et `ident` comme une suite de lettres.

N.B. : Le répertoire `exempleExpANTLR` de la section `TP Projet de compilation` sous Moodle contient les fichiers `Exp2.g`, `Exp2.java` et le squelette `PtGen2.java` **à compléter**. Les attributs lexicaux associés aux items lexicaux `nbentier` et `ident`, resp. `nbentier.valEntier` et `ident.idLu`, sont accessibles dans `PtGen2.java` où le code des points de génération doit être mis.

(a) Placer des points de génération dans la grammaire `Exp2.g` afin d'afficher l'expression analysée à raison d'un entier, ou symbole, par ligne.
Pour $(10 + a) - 4 * 3$ on doit afficher :

```
(
10
+
a
)
-
4
*
3
```

(b) Modifier l'emplacement des points de génération (il faut donc générer un nouvel analyseur syntaxique à l'aide de ANTLR) et les traitements associés afin d'afficher un pseudo-code MAPILE produisant :

```
empiler n           //pour un nbentier n,
contenug a          //pour un ident a
add, sous, mul, div //respectivement pour les opérateurs '+', '-', '*', 'div'
```

L'analyse de l'expression $(10 + a) - 4 * 3$ doit alors afficher :

```
empiler 10
contenug a
add
empiler 4
empiler 3
mul
sous
```

Grammaire du langage PROJET :

```
unite           :   unitprog | unitmodule ;
unitprog        :   'programme' ident ':' declarations corps ;
unitmodule      :   'module' ident ':' declarations ;
declarations    :   partiedef ? partieref ? consts ? vars ? decprocs ? ;
partiedef       :   'def' ident ( ',' ident )* ptvg ;
partieref       :   'ref' specif ( ',' specif )* ptvg ;
specif          :   ident ( 'fixe' '(' type ( ',' type )* ')' ) ?
                    ( 'mod' '(' type ( ',' type )* ')' ) ? ;
consts          :   'const' ( ident '=' valeur ptvg )+ ;
vars            :   'var' ( type ident ( ',' ident )* ptvg )+ ;
type            :   'ent' | 'bool' ;
decprocs        :   ( decproc ptvg )+ ;
decproc         :   'proc' ident parfixe ? parmod ? consts ? vars ? corps ;
ptvg            :   ';' | ;
corps           :   'debut' instructions 'fin' ;
parfixe         :   'fixe' '(' pf ( ';' pf )* ')' ;
pf              :   type ident ( ',' ident )* ;
parmod          :   'mod'  '(' pm ( ';' pm )* ')' ;
pm              :   type ident ( ',' ident )*;
instructions    :   instruction ( ';' instruction )*;
instruction     :   inssi | inscond | boucle | lecture | ecriture | affouappel | ;
inssi           :   'si' expression 'alors' instructions
                    ( 'sinon' instructions )?
                    'fsi' ;
inscond         :   'cond' expression ':' instructions
                    ( ',' expression ':' instructions )*
                    ( 'aut'  instructions |  ) 'fcond' ;
boucle          :   'ttq ' expression 'faire' instructions 'fait' ;
lecture         :   'lire' '(' ident ( ',' ident )* ')' ;
ecriture        :   'ecrire' '(' expression ( ',' expression )* ')' ;
affouappel      :   ident ( ':=' expression
                        | ( effixes ( effmods )? )? ) ;
effixes         :   '(' ( expression ( ',' expression )* )? ')' ;
effmods         :   '(' ( ident ( ',' ident )* )? ')' ;
expression      :   exp1 ( 'ou' exp1 )* ;
exp1            :   exp2 ( 'et' exp2 )* ;
exp2            :   'non' exp2 | exp3 ;
exp3            :   exp4 ( '=' exp4| '<>' exp4 | '>' exp4| '>=' exp4 | '<' exp4
                            | '<=' exp4 )? ;
exp4            :   exp5 ( '+' exp5 | '-' exp5 )* ;
exp5            :   primaire ( '*' primaire | 'div' primaire )* ;
primaire        :   valeur | ident | '(' expression ')' ;
valeur          :   nbentier | '+' nbentier | '-' nbentier | 'vrai' | 'faux' ;
```

2 Le langage PROJET à compiler

Le fichier `Projet.g` contient à la fois les règles concernant l'analyse lexicale et celles concernant l'analyse syntaxique du langage PROJET. La partie lexicale se situe à la fin du fichier et ne doit pas être modifiée. Elle définit les items lexicaux `nbentier` et `ident`. Les attributs lexicaux associés sont accessibles resp. par `UtilLex.valEnt` et par `UtilLex.numIdCourant` et correspondent à la valeur entière associée au dernier `nbentier` reconnu et au numéro d'identificateur du dernier `ident` reconnu. Les items lexicaux correspondant aux mots réservés et aux différents séparateurs du langage PROJET figurent directement dans les règles syntaxiques entre cotes '...', comme ANTLR le permet.

2.1 Syntaxe du langage PROJET

Les règles d'écriture (syntaxe) des unités sources exprimées en langage PROJET sont fournies par la grammaire de la page précédente (où `nbentier` et `ident` sont fournis par la partie lexicale).

Ces règles sont à compléter par des contraintes sémantiques non exprimables dans une grammaire hors-contexte et habituelles pour des langages impératifs, comme par exemple :

- tout identificateur manipulé dans le corps d'un programme ou d'une procédure doit avoir fait l'objet d'une déclaration,
- les deux membres d'une instruction d'affectation doivent être du même type.

Plutôt que d'établir ici une liste (difficilement exhaustive ...) de toutes ces contraintes, nous avons préféré les indiquer dans les paragraphes suivants, consacrés à la sémantique des différentes constructions.

2.2 Sémantique des constructions du langage PROJET

2.2.1 Constructions de base

Un programme PROJET peut **déclarer des constantes et des variables globales de type entier ou booléen**. Chaque identificateur manipulé dans le corps du programme doit faire l'objet d'une, et d'une seule, déclaration. Le corps du programme exécute un traitement spécifié par une séquence d'instructions.
Chaque **instruction** peut être :

- une affectation, notée `:=`, de la valeur d'une expression `exp` à une variable globale (par ex. `x`) du même type que l'expression (**`x` := `exp`**),
- une conditionnelle `si expbool alors instructions [sinon instructions] fsi` ayant la sémantique de l'instruction `if [else]` de Java,

- une conditionnelle

```
cond expbool1 : instructions,
     expbool2 : instructions,
     . . .
     [aut instructions | vide]
fcond
```

dont la sémantique est proche de celle de l'instruction `switch` de Java, i.e :

```
si expbool1 alors instructions sinon
si expbool2 alors instructions sinon
. . .
[sinon instructions | vide]
fsi . . . fsi,
```

- une itération `ttq expbool faire instructions fait` dont la sémantique est celle de l'instruction Java `while`,
- une instruction de lecture au clavier `lire (v1, v2, ...,vn)`, où `v1, v2, ...,vn` sont des variables, qui permet d'affecter à chacune de ces variables une valeur obtenue par saisie au clavier,
- une instruction d'affichage à l'écran `ecrire (exp1, exp2, ..., expn)` qui affiche à l'écran la valeur des expressions `exp1, exp2, ..., expn`.

Les **expressions**, de type entier ou booléen, combinent des variables, des constantes et des notations de valeurs (nombres entiers signés ou non, valeurs booléennes `vrai` et `faux`) à l'aide :

- d'opérateurs arithmétiques (`+, -, *, div`) à opérandes et résultat de type entier,
- d'opérateurs de comparaison (`=, <>, >, >=, <, <=`) à résultat de type booléen ; dans un but simplificateur nous imposons que les opérandes soient du type entier,
- d'opérateurs logiques (`ou, et, non`) à opérandes et résultat de type booléen.

Les règles de priorité appliquées pour l'évaluation des expressions sont les règles induites par la grammaire du langage PROJET.

Exemple :

```
programme simple :
    const moinscinq =-5 ;
    var ent i, n, x, s ;
        bool b ;
    debut
        lire (n) ; i:=n ; s:=0 ; b:=faux ;
        ttq i>0 faire
            lire (x) ; s:=s+x ;
            si x=moinscinq alors b:=vrai fsi ;
            i:=i-1 ;
        fait ;
        ecrire (s, b)
    fin
```

Si on fournit comme données 4 12 -5 1 10, l'exécution de ce programme provoque l'affichage de la valeur entière 18 puis du caractère 'V' (pour `vrai`).

2.2.2 Procédures

On peut **déclarer des procédures**, mais seulement au niveau global :

- l'entête de la procédure comporte le nom de celle-ci ainsi que le type (`ent / bool`), le mode de passage (`fixe / mod`) et le nom de chacun des paramètres formels,
- la procédure peut déclarer des constantes et des variables qui lui sont locales,
- le corps de la procédure est identique au corps d'un programme ; à l'intérieur du corps de la procédure, un paramètre `mod` se comporte comme une variable et un paramètre `fixe` est assimilable à une constante, on ne peut donc pas le faire figurer en partie gauche d'affectation, ni le transmettre en paramètre effectif `mod` dans un appel de procédure.

Le nom de la procédure est considéré comme un identificateur global alors que les paramètres formels et les identificateurs déclarés à l'intérieur de la procédure sont locaux à celle-ci ; il y a donc un masquage éventuel d'identificateurs globaux.

On peut **appeler une procédure** sous réserve qu'elle soit déclarée (les références en avant ne sont pas autorisées). Cet appel provoque l'exécution du corps de la procédure, en substituant, dans celui-ci, les paramètres effectifs aux paramètres formels. A la fin de la procédure, l'exécution continue à l'instruction qui suit l'appel. Il doit y avoir conformité en nombre, type et mode de passage des paramètres entre l'instruction d'appel et la déclaration de la procédure :

- le paramètre formel est déclaré `fixe` ⇒ le paramètre effectif (lors de l'appel) est une expression,
- le paramètre formel est déclaré `mod` ⇒ le paramètre effectif (lors de l'appel) est une variable.

Exemple :

```
programme deuxprocs :
    const n=10 ; var ent x ;

    proc p1 mod (ent n)
    {le paramètre n est modifiable dans p1}
        const m=3 ; var ent x, i ;
        {N.B. x, variable locale, "masque" x variable globale}
    debut
        i:=m ; n:=0 ;
        ttq i>0 faire lire (x) ; n:=n+x ; i:=i-1 fait
    fin ;

    proc p2 fixe (ent nbf, barre)
    {les paramètres nbf et barre ne peuvent pas être modifiés dans p2}
        var ent y, i, res ;
    debut
        i:=nbf ; res:=0 ;
        ttq i>0 faire p1( ) (y) ; si y>=barre alors res:=res+1 fsi ; i:=i-1 fait ;
        x:=res
    fin ;

    debut
        lire (x) ; p2(x+1, n) ( ) ; ecrire (x)
    fin
```

Pour les données 2 1 1 4 3 11 -2 15 -4 -3 ce programme affiche 1.

2.2.3 Compilation séparée

Une application développée en langage PROJET peut être composée non seulement d'un programme mais aussi de modules écrits (et compilés) séparément. Ces différentes unités (le programme et les modules) coopèrent en se partageant des procédures :

- toute procédure déclarée dans une unité (programme ou module) est visible à l'extérieur de l'unité si son nom figure dans la liste `def` des points d'entrée de l'unité,
- toute procédure dont la spécification (entête de procédure sans le nom des paramètres formels) est fournie dans la liste `ref` des références externes de l'unité est appelable depuis cette unité.

Tout point d'entrée ne doit figurer qu'une fois dans la liste `def` et doit faire l'objet d'une déclaration de procédure dans l'unité. Toute procédure citée dans la liste `ref` d'une unité ne peut y figurer qu'une fois et ne doit pas être redéclarée dans l'unité.

Il y a bien sûr des règles à respecter pour que "l'union" d'un programme et de modules constitue une application valide. Ces règles, et les contrôles qu'elles induisent, sont prises en compte par le logiciel qui effectue le regroupement des unités : il s'agit de l'éditeur de liens, celui-ci est présenté dans le chapitre 6.

Exemple :

Le programme `ess1` et les modules `m1ess1` et `m2ess1` suivants constituent une application PROJET dont l'exécution, pour les données 3 12 0 23 1 5 17 11 -1, provoque l'affichage de 4 0 5 3 11 17.
Le programme `ess1` utilise des références externes (`init1, int2, ..., ecr2`) qui sont définies en points d'entrée des modules `m1ess1` et `m2ess1`.

N.B. : ici aucun point d'entrée n'est défini par le programme mais il pourrait y en avoir. De même, un module peut utiliser des références externes.

```
programme ess1 :

    {les procédures suivantes ne sont pas déclarées dans cette unité programme}
    {mais définies en "ref", elles peuvent néanmoins être utilisées}
    ref init1, init2, ajout1 fixe (ent), ajout2 fixe (ent), ecr1, ecr2 ;

    const minx=0 ; maxx=20 ; moyx=10 ;
    var ent x ;
debut
    init1 ; init2 ; lire (x) ;
    ttq x<>-1 faire
        si x>=minx et x<=maxx alors
            si x<moyx alors ajout1 (x) sinon ajout2 (x) fsi ;
        fsi ;
        lire (x) ;
    fait ;
    ecr1 ; ecr2 ;
fin {ess1}
```

```
module m1ess1 :

    {les procédures suivantes sont déclarées dans cette unité module}
    {définies en "def", elles sont accessibles par d'autres unités}
    def init1, ajout1, ecr1 ;

    var ent n, min, max ;

    proc init1 debut n:=0 fin ; {init1}

    proc ajout1 fixe (ent y)
    debut
        si n=0 alors min:=y ; max:=y
        sinon si y<minalors min :=y
              sinon si y>max alors max:=y fsi
              fsi
        fsi ;
        n:=n+1 ;
    fin ; {ajout1}

    proc ecr1
    debut
        si n<>0 alors ecrire (n, min, max) sinon ecrire (0) fsi ;
    fin ; {ecr1}
```

```
module m2ess1 :
    def init2, ajout2, ecr2 ;
    var ent n, min, max ;

    proc init2 debut n:=0 fin ; {init2}

    proc ajout2 fixe (ent y)
    debut
        si n=0 alors min:=y ; max:=y
        sinon si y<min alors min :=y
              sinon si y>max alors max:=y fsi
              fsi
        fsi ;
        n:=n+1 ;
    fin ; {ajout2}

    proc ecr2
    debut
        si n<>0 alors ecrire (n, min, max) sinon ecrire (0) fsi ;
    fin ; {ecr2}
```

3 MAPILE : la machine à pile pour exécuter le code produit

3.1 Principe et utilisation

Le code produit par le compilateur, dit code MAPILE, est destiné à être exécuté par la machine à pile virtuelle fournie MAPILE. Le nom du fichier contenant le code MAPILE à exécuter doit être suffixé par `.map`, ou, à défaut, par `.obj`. Ce fichier est un fichier texte contenant un seul entier par ligne (qui peut être suivi de commentaires qui seront ignorés par MAPILE). Cet entier correspond soit au code d'une instruction MAPILE, soit à un argument d'une instruction MAPILE.

La machine à pile doit :

- charger en mémoire le fichier proposé par l'utilisateur dans un segment de code `po`,
- exécuter les instructions contenues dans le segment code en faisant progresser le compteur ordinal `ipo` (initialement à 1) jusqu'à la rencontre du code de l'instruction arrêt.

L'exécution des instructions **autres** que celles de branchement (`bincond` et `bsifaux`) et d'appel ou retour de procédure (`appel`, `retour`), provoque une augmentation du compteur ordinal de (1 + nombre d'arguments de l'instruction exécutée). Les instructions de branchement et d'appel ou retour de procédure provoquent une rupture dans cette progression séquentielle de `ipo`.

La réservation de mémoire pour les variables (globales ou locales) et pour les données de liaison (nécessaires pour traiter les appels de procédure), ainsi que les calculs liés à l'exécution des instructions, sont réalisés dans un segment de pile, nommé ici `pileExec`. Le principe de fonctionnement de ce segment de pile ressemble à celui d'une pile d'entiers, de sommet de pile (ou registre de pile) ip (initialement à -1), avec quelques particularités.

Les appels de procédures nécessitent la gestion d'un autre registre `bp`, dont la valeur initiale est 0. Les réservations de mémoire (pour les variables ou pour les appels de procédures) impliquent de pouvoir modifier le sommet de pile `ip` sans empiler de valeurs et de pouvoir accéder à/modifier certains emplacements du segment pile sans dépiler d'éléments.

Les accès au segment pile, nécessaires à l'exécution d'un code MAPILE, sont :

- `void pileExec.empiler(int val)` : empile l'entier val en sommet de pile
- `int pileExec.depiler()` : dépile le sommet de pile
- `void pileExec.reserver(int arg)` : augmente de l'entier arg le sommet de pile ip
- `int pileExec.contenu(int arg)` : récupère le (arg+1)eme élément du segment pile
- `void pileExec.modifier(int arg, int val)` : range l'entier val dans le `(arg+1)`eme élément du segment pile.

La machine à pile est activable en exécutant `Mapile.java` et nécessite les classes `ExecMapile.class` et `Mnemo.class`. Le nom du fichier (sans suffixe) contenant le code MAPILE à exécuter est alors demandé. Si, lors de la saisie du nom du fichier, on termine en tapant le caractère $*$, on obtient une trace complète de l'exécution, sous forme mnémonique.

3.2 Instructions élémentaires

Nous décrivons ici, en termes d'accès au segment pile, l'exécution des instructions MAPILE traduisant des programmes sources, écrits en langage PROJET, ne manipulant pas de procédures.

code	mnémonique	effet
	// réserver arg1 emplacements dans la pile pour des variables globales ou locales	
1	reserver arg1	`ip+=arg1 ;`
	// empiler une valeur entière ou booléenne (codée 0 ou 1 si booléenne)	
2	empiler arg1	`pileExec.empiler(arg1) ;`
	// empiler le contenu de la variable globale d'adresse arg1	
3	contenug arg1	`val=pileExec.contenu(arg1); pileExec.empiler(val) ;`
	// ranger le sommet de pile dans la variable globale d'adresse arg1	
4	affecterg arg1	`val=pileExec.depiler() ; pileExec.modifier(arg1, val).`
	// opérations booléennes **(vrai est codé 1, faux est codé 0)**	

5	ou	`val2=pileExec.depiler(); val1=pileExec.depiler() ;`
		`if (val1==1 || val2==1) pileExec.empiler(1) ;`
		`else pileExec.empiler(0) ;`

6	et	idem avec `if (val1==1 && val2==1)`
7	non	`val=pileExec.depiler() ; pileExec.empiler(1-val) ;`
	// opérations de comparaison **(vrai est codé 1, faux est codé 0)**	

8	inf	`val2=pileExec.depiler() ; val1=pileExec.depiler() ;`
		`if (val1 < val2) pileExec.empiler(1) ;`
		`else pileExec.empiler(0) ;`

9	infeg	idem avec `if (val1 <= val2)`
10	sup	idem avec `if (val1 >val2)`
11	supeg	idem avec `if (val1 >=val2)`
12	eg	idem avec `if (val1 == val2)`
13	diff	idem avec `if (val1 != val2)`
	// opérations arithmétiques	
14	add	`val2=pileExec.depiler().; val1=pileExec.depiler() ;` `val=val1+val2 ; pileExec.empiler(val) ;`
15	sous	`val2=pileExec.depiler().; val1=pileExec.depiler() ;` `val=val1-val2 ; pileExec.empiler(val) ;`
16	mul	`val2=pileExec.depiler().; val1=pileExec.depiler() ;` `val=val1*val2 ; pileExec.empiler(val) ;`
17	div	`val2=pileExec.depiler().; val1=pileExec.depiler() ;` `val=val1/val2 ; pileExec.empiler(val) ;`

code	mnémonique	effet
	// instructions de branchement	

18	bsifaux arg1	// branchement conditionné par la valeur du sommet de pile
		`val=pileExec.depiler();`
		`if (val==0) ipo=arg1`

19	bincond arg1	// branchement inconditionnel `ipo=arg1 ;`
	// instructions d'entrée/sortie	
20	lirent	// acquisition d'une donnée entière `pileExec.empiler(entier-lu) ;`

21	lirebool	// acquisition d'une donnée booléenne ('V' ou 'F')
		`if (donnee-lue=='V') pileExec.empiler(1) ;`
		`else if (donnee-lue=='F') pileExec.empiler(0)) ;`

22	ecrent	// affichage de la valeur d'une expression entière `val=pileExec.depiler() ; System.out.print(val) ;`

23	ecrbool	// affichage de la valeur d'une expression booléenne
		`val=pileExec.depiler() ;`
		`if (val ==1) System.out.print('V') ;`
		`else if (val ==0) System.out.print('F') ;`

	// arrêt de l'exécution du programme	
24	arret	

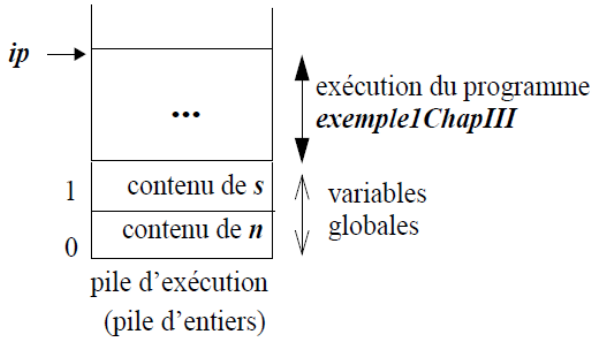
Exemple :

La colonne de gauche ci-dessous montre un petit programme en langage PROJET. La partie médiane donne le code MAPILE, en mnémonique (plus facilement lisible), que le compilateur doit produire pour ce programme. La colonne de droite donne le code MAPILE réellement exécutable, soit une suite d'entiers. Devant chaque code MAPILE, `ipo` donne la valeur du compteur ordinal pour ce code (`ipo` peut être vu comme l'index dans le segment de code).

L'exécution du code MAPILE suivant affiche 6 si les données fournies sont 3 et 20.

programme source PROJET		**code MAPILE en mnémonique**		**segment code**
programme exemple1ChapIII :				
	ipo		***ipo***	**po**
const m=10 ;				
var ent n, s ;	*1*	reserver 2	*1*	1
			2	2
debut				
lire (n, s) ;	*3*	lirent	*3*	20
	4	affecterg 0	*4*	4
			5	0
	6	lirent	*6*	20
	7	affecterg 1	*7*	4
			8	1
si n=0 **alors**	*9*	contenug 0	*9*	3
			10	0
	11	empiler 0	*11*	2
			12	0
	13	eg	*13*	12
	14	bsifaux 20	*14*	18
			15	20
n :=m	*16*	empiler 10	*16*	2
			17	10
	18	affecterg 0	*18*	4
			19	0
fsi ;				
ecrire (s **div** n)	*20*	contenug 1	*20*	3
			21	1
	22	contenug 0	*22*	3
			23	0
	24	div	*24*	17
	25	ecrent	*25*	22
fin exemple1ChapIII	*26*	arret	*26*	24

Principe d'exécution du code objet du programme *exemple1ChapIII* sur la machine à pile MAPILE

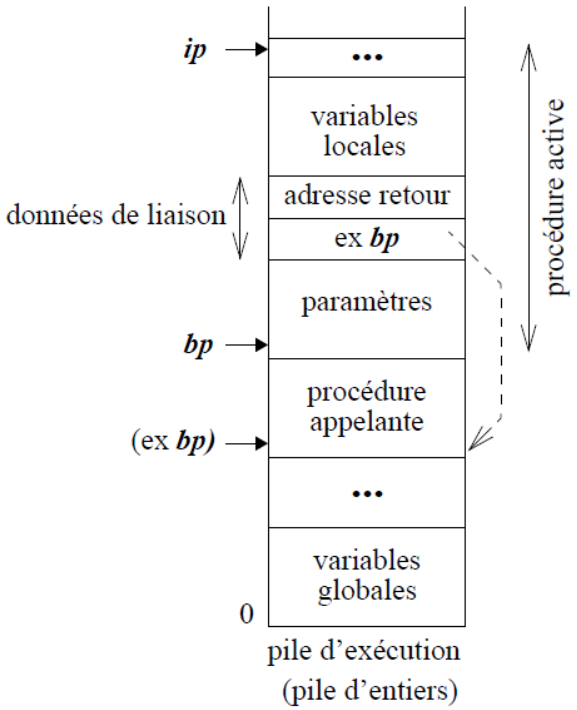


3.3 Traitement des procédures

Fonctionnement de la machine à pile MAPILE avec des procédures.

Le registre `bp` permet de repérer, dans la pile d'exécution, la procédure active (en cours d'exécution). Initialement, `bp` vaut 0 (c'est le programme principal qui est actif).
Deux **données de liaison** suffisent pour rétablir le contexte à la fin de l'exécution d'une procédure :

- l'une indique l'ancienne valeur de `bp`, c'est à dire l'emplacement, dans la pile, des paramètres et variables locales de la procédure appelante,
- l'autre fournit l'adresse à laquelle il faut continuer l'exécution du programme objet (*c.-à-d.* l'adresse de l'instruction qui suit l'instruction d'appel).



Pour gérer les procédures, on ajoute six instructions à la machine à pile, trois d'entre elles (`contenul`, `empileradl`, et `affecterl`) comportent un "bit" d'indirection `arg2`.

code	mnémonique	effet
	// transmission, en paramètre mod, d'une variable globale	
25	empileradg arg1	`pileExec.empiler(arg1) ;`
	// transmission, en paramètre mod, d'une variable locale (bit arg2=0) // ou d'un paramètre mod (bit arg2=1)	
26	empileradl arg1 arg2	`if (arg2==0) pileExec.empiler(bp+arg1) ;` `else pileExec.empiler(pileExec.contenu(bp+arg1)) ;`
	// obtention, en sommet de pile, de la valeur d'un paramètre fixe ou d'une variable // locale (bit arg2=0) ou de la valeur d'un paramètre mod (bit arg2=1)	

27	contenul arg1 arg2	`if (arg2==0) val=pileExec.contenu(bp+arg1) ;`
		`else val=pileExec.contenu(pileExec.contenu(bp+arg1)) ;`
		`pileExec.empiler(val) ;`

	// rangement de la valeur du sommet de pile dans une variable locale (bit arg2=0) // ou dans un paramètre mod (bit arg2=1)	

28	affecterl arg1 arg2	`val=pileExec.depiler() ;`
		`if (arg2==0) pileExec.modifier(bp+arg1, val) ;`
		`else pileExec.modifier(pileExec.contenu(bp+arg1), val) ;`

	// appel de procédure : arg1 indique l'adresse, dans po, de la procédure appelée.
	// Les paramètres effectifs, dont le nombre est arg2, ont été évalués et figurent
	// en sommet de pile ; l'instruction positionne les données de liaison et effectue
	// le changement de contexte pour basculer dans celui de la procédure appelée	

29	appel arg1 arg2	`pileExec.empiler(bp) ; bp=ip-arg2 ;` `pileExec.empiler(ipo+3) ; ipo=arg1 ;`

	// retour de procédure : arg1 est le nombre de paramètres et permet de retrouver,
	// à l'aide de bp, les données de liaison ; l'instruction met la pile à jour et restitue
	// le contexte de la procédure appelante	

30	retour arg1	`exbp=pileExec.contenu(bp+arg1) ;`
		`ipo=pileExec.contenu(bp+arg1+1) ;`
		`ip=bp-1 ; bp=exbp ;`

Exemple :

La colonne de gauche ci-dessous montre un programme en langage PROJET *exemple2ChapIII* avec deux procédures. La colonne de droite donne le code MAPILE, en mnémonique, que le compilateur doit produire pour ce programme. Devant chaque code MAPILE, `ipo` donne la valeur du compteur ordinal pour ce code (`ipo` peut être vu comme l'index dans le segment de code).

L'exécution du code MAPILE suivant affiche 2.

programme source PROJET	*ipo*	**code MAPILE en mnémonique**	
programme exemple2ChapIII :			
var ent n ;	*1*	reserver	1
	3	bincond	50
proc p **fixe** (**ent** x) **mod** (**ent** y)			
var ent j ;	*5*	reserver	1
debut			
j :=0 ;	*7*	empiler	0
	9	affecterl	4 0
y :=x ;	*12*	contenul	0 0
	15	affecterl	1 1
fin ; {p}	*18*	retour	2
proc q **mod** (**ent** z)			
var ent i ;	*20*	reserver	1
debut			
p(2)(i) ;	*22*	empiler	2
	24	empileradl	3 0
	27	appel	5 2
p(i)(z) ;	*30*	contenul	3 0
	33	empileradl	0 1
	36	appel	5 2
p(z)(i) ;	*39*	contenul	0 1
	42	empileradl	3 0
	45	appel	5 2
fin ; {q}	*48*	retour	1
debut			
q()(n) ;	*50*	empileradg	0
	52	appel	20 1
ecrire(n) ;	*55*	contenug	0
	57	ecrent	
fin ; {exemple2ChapIII}	*58*	arrêt	

4

Éléments de base du compilateur

Ce chapitre introduit les éléments de base du compilateur à réaliser pour la compilation de programmes en langage PROJET. Les éléments propres à la compilation des procédures et modules sont présentés dans les chapitres suivants.

4.1 Organisation du compilateur

Le contenu des points de génération, permettant de produire un code objet exécutable (à partir d'un programme en langage PROJET) et d'effectuer des contrôles sémantiques, est à fournir dans le fichier PtGen.java. Il s'agit donc de compléter la procédure `pt (int numGen)` dans `PtGen.java`.

Différentes structures de données sont nécessaires à la production de code et aux vérifications sémantiques.

La génération du code objet s'effectue via une instance de la classe `ProgObjet`. Les méthodes associées permettent de produire le code séquentiellement, de le modifier, de construire le fichier exécutable (`.obj`) et le fichier mnémonique (`.gen`) associés au code produit.

La génération de branchements dans le code nécessite l'utilisation d'une pile, dite pile des reprises, via une instance de la classe `TPileRep`.

Une table des symboles est nécessaire à la gestion des identificateurs pendant la compilation. Cette table des symboles et les utilitaires associés sont définis dans le fichier `PtGen.java`.

Enfin, le fichier `PtGen.java` contient la création d'une instance de la classe `ProgObjet` (nommée `po`), la création d'une instance de la classe `TPileRep` (nommée `pileRep`) et la définition de différentes constantes. Ces constantes sont en effet indispensables à la lisibilité du compilateur réalisé et doivent être utilisées. Par ex :

```
// pour la lisibilité des points de génération on dispose de constantes pour les codes MAPILE
    static final int
        RESERVER=1, EMPILER=2, CONTENUG=3, AFFECTERG=4, OU=5, ET=6,
        NON=7, INF=8, INFEG=9, SUP=10, SUPEG=11, EG=12, DIFF=13, ADD=14,
        SOUS=15, MUL=16, DIV=17, BSIFAUX=18, BINCOND=19, LIRENT=20,
        LIREBOOL=21, ECRENT=22, ECRBOOL=23, ARRET=24, EMPILERADG=25,
        EMPILERADL=26, CONTENUL=27, AFFECTERL=28, APPEL=29, RETOUR=30;
// valeurs du vecteur de translation (compilation séparée)
    static final int TRANSDON=1, TRANSCODE=2, REFEXT=3;
// codage des valeurs booléennes
    static final int VRAI=1, FAUX=0;
```

4.2 Structures de données du compilateur

4.2.1 Classe ProgObjet

Le programme objet MAPILE résultant de la compilation du programme source est produit en mémoire par une instance de la classe `ProgObjet`. Le code objet y est géré dans un tableau d'entiers `po`, d'indice de remplissage ipo. Les déclarations et méthodes sont les suivantes :

```
private static final int MAXOBJ = 1000 ;

private int ipo = 0 ; // indice de remplissage du tableau po
private int [] po = new int [MAXOBJ+1] ;
```

Dans le cadre des éléments de base du compilateur, les méthodes à utiliser sont les suivantes :

```
public int getIpo() // retourne l'index du dernier code produit

public int getElt( int i ) // retourne le code produit à l'index i
```

La méthode `produire` permet de produire des codes MAPILE séquentiellement.

```
public void produire (int codeouarg)
    // le paramètre est soit un code MAPILE, soit un argument d'une instruction MAPILE
```

La méthode `modifier` permet de modifier un code MAPILE déjà produit (mise à jour d'une adresse de branchement par exemple).

```
public void modifier (int i, int codeOuArg)
// modifie le code déjà produit à l'index i par codeOuArg
```

Les méthodes de construction de fichiers à partir du code produit sont les suivantes :

```
static void constGen ( )
    // recopie, usuellement en fin de compilation, le code produit, sous forme mnémonique,
    // dans le fichier nomSource''.gen'' (par l'utilisation de la classe Mnemo)

static void constObj ( )
    // recopie, usuellemnt en fin de compilation, le code produit dans le fichier
    // nomSource''.obj''
```

D'autres attributs et méthodes concernent la compilation séparée (cf. chapitre 6).

4.2.2 Table des symboles définie dans PtGen.java

La table des symboles `tabSymb` est nécessaire au traitement des identificateurs ; elle permet de mémoriser les informations fournies lors de la déclaration d'un identificateur (constante/variable, type, etc.). On consulte ces informations à chacune des occurrences d'utilisation de l'identificateur afin d'effectuer des vérifications (conformité de type, constante non autorisée en partie gauche d'affectation, etc.) et de réaliser la production du code objet adapté.
Le type d'un élément de la table des symboles est défini par :

```
class EltTabSymb {
    public int
            code,       // codage (“numIdCourant”) attribué à
                        // l'identificateur par l'analyseur lexical
            categorie,  // cf. catégories définies ci-dessous
            type,       // l'un des trois types permis
            info ;      // dépend de la catégorie (par ex adresse d'exécution
                        // d'une variable, valeur d'une constante, etc.)
    public EltTabSymb (int code, int categorie, int type, int info) {
        this.code=code; this.categorie=categorie; this.type=type; this.info=info;
    }
}
```

Les types permis sont (NEUTRE est utilisé pour remplir des cases non significatives) :

```
private static final int ENT=1, BOOL=2, NEUTRE=3 ;
```

Les catégories possibles sont :

```
private static final int CONSTANTE=1, VARGLOBALE=2, VARLOCALE=3,
            PARAMFIXE=4, PARAMMOD=5, PROC=6, DEF=7, REF=8, PRIVEE=9 ;
```

La table des symboles est définie par :

```
private static final int MAXSYMB = 300 ;
private static int it = 0 ; // indice de remplissage de la table des symboles
private static int bc = 1 ; // début du bloc courant dans la table des symboles
private static EltTabSymb [] tabSymb = new EltTabSymb [MAXSYMB+1] ;
```

Les utilitaires d'ajout et de recherche d'un identificateur dans la table des symboles sont les suivants :

```
private static void placeIdent (int c, int cat, int t, int v) {
    // placement d'un nouveau quadruplet dans tabSymb
    it++ ; tabSymb[it]=new EltTabSymb (c, cat, t, v) ;
}
```

L'utilitaire `presentIdent(int borneInf)` recherche si `tabSymb[borneInf..it]` contient une déclaration correspondant au dernier identificateur lu (codé `numIdCourant` par l'analyseur lexical). Le résultat est 0 si la recherche est infructueuse, sinon c'est l'indice i où figure cette déclaration. Pour prendre en compte la portée des identificateurs lorsqu'on manipule des procédures, la recherche privilégie la déclaration la plus récente.

```
private static int presentIdent (int borneInf) {
    int i=it ;
    while (i>=borneInf && tabSymb[i].code!=UtilLex.numIdCourant) i --
    if (i>=borneInf) return i ; else return 0 ;
}
```

4.2.3 TPileRep pour la pile des reprises

La traduction des instructions `si, cond` et `ttq` conduit à effectuer des branchements (`bsifaux`, `bincond`) "en avant". Le code étant produit en un seul (pseudo-)passage, il est nécessaire de mémoriser les indices de `po` correspondant aux arguments de ces branchements. La possible imbrication des instructions `si, cond` et `ttq` impose l'emploi d'une pile, définie dans la classe `TPileRep`.

Les méthodes associées à `TPileRep` sont les suivantes :

- `void empiler (int indexProgObjet)` : qui permet d'empiler (en sommet de pile) un entier correspondant à un indice du code produit (correspondant à une partie de code non encore définitive),
- `public int depiler ( )` : dépile le sommet de pile, et retourne ainsi l'indice d'une partie à modifier du code produit.

4.3 Traitements effectués par le compilateur

Pour présenter les différents concepts abordés nous nous appuyons sur l'exemple suivant, dans lequel nous avons matérialisé des "points d'observation" à l'aide de nombres entre parenthèses.

```
programme exempleChapIV :
    const val=10 ; b=faux ;
    var ent u, v ;
        bool x ;
        ent k, a;
debut                                   (1)
    lire (a,k,x) ;                      (2)
    ecrire (5+a*k-val>val et x  (3) ) ;     (4)
    u:=3 ;                              (5)
    si x alors                          (6)
        v:=u+1
    sinon                               (7)
        v:=u-1
    fsi ;                               (8)
    ttq v<100 faire                     (9)
        v:= v*v
    fait ;                              (10)
    ecrire(v) ;                         (11)
fin
```

4.3.1 Déclaration des identificateurs

Initialement la table des identificateurs `tabSymb` est vide (indice de remplissage `it=0`). À chaque déclaration d'identificateur, on contrôle (fonction `presentIdent`) l'unicité de déclaration dans le bloc courant et on ajoute la déclaration dans `tabSymb` (fonction `placeIdent`).

Lors de la **compilation de programme sans procédures**, le bloc courant correspond toujours au programme principal et l'indice de bloc courant `bc` dans `tabSymb` est toujours égal à 1.

Pour une déclaration de constante, le champ `info` reçoit la valeur de la constante (noter le codage des valeurs booléennes : **faux → 0, vrai → 1**). Dans le cas d'une variable, il reçoit l'adresse d'exécution (dans la pile d'exécution de MAPILE) ; cette adresse est absolue pour une variable globale.
Rappel : l'adresse d'exécution de la première variable déclarée est toujours 0 (cf. page 20).

Pour le programme `exempleChapIV` en (1), on a :

tabSymb

	code	catégorie	type	info
bc=1	`@val`	`CONSTANTE`	`ENT`	10
2	`@b`	`CONSTANTE`	`BOOL`	0
3	`@u`	`VARGLOBALE`	`ENT`	0
4	`@v`	`VARGLOBALE`	`ENT`	1
5	`@x`	`VARGLOBALE`	`BOOL`	2
6	`@k`	`VARGLOBALE`	`ENT`	3
it=7	`@a`	`VARGLOBALE`	`ENT`	4

ipo po (code produit)

1- réserver 5

La notation `@val` est utilisée pour désigner le code (entier positif) attribué à la chaîne "`val`" par l'analyseur lexical (*i.e.* c'est l'attribut lexical `numIdCourant` de cette chaîne).

Le seul code produit lors de la déclaration des identificateurs correspond à la réservation des emplacements (dans la pile d'exécution de MAPILE) pour les variables, ici : `réserver 5`

La compilation du corps du programme principal ne modifie pas la table des symboles mais l'utilise pour les vérifications de type et pour la production de code.

Par exemple, le code suivant est produit pour :

- récupérer la valeur de la constante `val` : `empiler 10`
- récupérer le contenu de la variable `k` : `contenug 3`
- affecter une nouvelle valeur à la variable `a` : `affecterg 4` (avec valeur en sommet de pile d'exécution)

4.3.2 Expressions - Calcul de type

Pour contrôler la validité du type des opérandes dans les expressions, on gère une variable entière `tCour` indiquant le type (`ENT` ou `BOOL`) de l'expression "courante" et on dispose de :

```
static void verifEnt ( )
    {if (tCour!=ENT) UtilLex.messErr(‘‘expression entière attendue’’);}
static void verifBool ( )
    {if (tCour!=BOOL) UtilLex.messErr(‘‘expression booléenne attendue’’);}
```

Lorsqu'une expression est réduite à une valeur, le positionnement de `tCour` est fixé par le type de cette valeur ; dans le cas d'un identificateur c'est `tabSymb` qui fournit l'information. Si l'expression correspond à une opération (booléenne ou arithmétique), la valeur attribuée à `tCour` est déterminée par l'opérateur (qui précise également le type attendu pour les opérandes, cf. chapitre 2).

Pour l'expression `5+a*k-val>val et x` qui figure dans le programme `exempleChapIV`, les contrôles effectués et le code produit jusqu'au point (3), sont les suivants :

expression courante	`tCour`	**contrôle**	*ipo*	`po`
`5`	`ENT`	`verifEnt`	*12* -	empiler 5
`a`	`ENT`	`verifEnt`	*14* -	contenug 4
`k`	`ENT`	`verifEnt`	*16* -	contenug 3
			18 -	mul
`a*k`	`ENT`	`verifEnt`	*19* -	add
`5+a*k`	`ENT`	`verifEnt`		
`val`	`ENT`	`verifEnt`	*20* -	empiler 10
			22 -	sous
`5+a*k-val`	`ENT`	`verifEnt`		
`val`	`ENT`	`verifEnt`	*23* -	empiler 10
			25 -	sup
`5+a*k-val>val`	`BOOL`	`verifBool`		
`x`	`BOOL`	`verifBool`	*26* -	contenug 2
			28 -	et
`5+a*k-val>val et x`	`BOOL`			

4.3.3 Instructions de base

Les instructions de base comprennent `lire`, `ecrire`, l'affectation `:=`, les instructions `si, ttq` et `cond`. Le code produit pour l'instruction `ecrire` dépend du type de l'expression à afficher, ce type est fourni par `tCour`. En ce qui concerne l'instruction `lire` et l'affectation, `tabSymb` contient les informations relatives à la variable destinataire de la valeur. Pour `exempleChapIV`, en (2), on obtient :

```
 3 -  lirent
 4 -  affecterg 4
 6 -  lirent
 7 -  affecterg 3
 9 -  lirebool
10 -  affecterg 2
```

en (4), on produit :

```
29 -  ecrbool
```

et en (11) :

```
72 -  ecrent
```

en (5), on obtient, pour l'instruction d'affectation :

```
30 -  empiler   3
32 -  affecterg 0
```

Pour **l'instruction** `si`, le code MAPILE contient une instruction `bsifaux` (et une instruction `bincond` lorsqu'il y a une clause `sinon`) dont l'argument n'est pas connu au moment de la production

du branchement. On mémorise, dans `pileRep`, l'indice de `po` correspondant à cet argument et la résolution d'adresse est réalisée à la rencontre de `fsi` . Ici en (6), `po` contient, pour la compilation de `si x alors` :

```
34 - contenug   2
36 - bsifaux    0   //et on empile l'indice ipo=37 dans pileRep
```

la compilation de la clause alors , en (7), produit :

```
38 - contenug   0
40 - empiler    1
42 - add
43 - affecterg   1   //on dépile 37, on effectue po[37] = 47 (début partie sinon)
45 - bincond     0   //et on empile l'indice ipo=46 dans pileRep
```

en (8) on dépile 46 et on effectue `po[46]` $\leftarrow$ 54 (`ipo` de ce qui suit `fsi`)

L'instruction `ttq` nécessite aussi l'emploi de `pileRep`, d'une part pour mémoriser l'adresse de "bouclage" (début de l'évaluation de l'expression booléenne) et, d'autre part, pour traiter la référence en avant induite par l'instruction `bsifaux` permettant de quitter l'instruction `ttq`. Ici en (9), la compilation de `ttq v>100 faire` produit :

```
54 -contenug    1   //et on empile l'indice 54 dans pileRep
                    // (début condition d'itération)
56 -empiler     100
58 -inf
59 -bsifaux     0   //et on empile l'indice ipo=60 dans pileRep
```

en (10), on obtient :

```
61 -contenug    1
63 -contenug    1
65 -mul
66 -affecterg   1   //et on dépile 60 pour effectuer po[60] = 70
                    // (ipo de ce qui suit fait)
68 -bincond     54  //on dépile 54 pour produire bincond 54
```

L'instruction `cond` équivaut à des instructions `si` en cascade dont toutes les clauses `alors` se terminent par un branchement `bincond` ; ces `bincond` ont un argument identique (l'adresse de l'instruction qui suit `fcond`). Pour éviter d'encombrer `pileRep` on chaîne entre elles les cases de `po` concernées et on mémorise dans `pileRep` le début de cette chaîne de reprise.

4.4 Exemples de programmes en langages Projet

Exercice :

Saisir, dans un fichier texte (de nom suffixé par `.obj` ou `.map`), le code MAPILE correspondant à un programme source PROJET proposé puis exécuter ce code sur la machine à pile. Pour l'exemple précédent, le fichier `exempleChapIV.obj` commence par :

```
1   réserver    5 // ceci est du commentaire, seul l'entier 1 est pris en compte
5
20  lirent
4   affecterg   4
4
. . .
```

Vous trouverez ci-après un ensemble de programmes (sans procédures) en langage PROJET (programmes sources disponibles dans le répertoire TestsProjet). Ces exemples de programmes ne comportent pas d'erreurs. **Il vous revient de tester également des programmes comportant des erreurs sémantiques (non syntaxiques) comme, par exemple, des erreurs de type dans les affectations, expressions, etc.**

```
programme exo1 : {somme de n valeurs}
    var ent n, som, i, x ;
debut
    lire (n) ; {nombre de valeurs}
    som:=0 ; i:=1 ;
    ttq i<=n faire
        lire (x) ; som:=som+x ; i:=i+1 ;
    fait ;
    ecrire (som)
fin {exo1}
```

```
programme exo2a :
 {on lit une suite de valeurs entières terminée par -1 et on dénombre les valeurs
 comprises dans l'intervalle 0..9 ainsi que les valeurs comprises dans
 l'intervalle 10..20}
    const minx=0 ; maxx=20 ; moyx=10 ;
    var ent n1, n2, x ;
debut
    n1:=0 ; {nombre de valeurs >=minx et <moyx}
    n2:=0 ; {nombre de valeurs >=moyx et <=maxx}
    lire (x) ;
    ttq x<>-1 faire
        si x>=minx et x<=maxx alors
            si x<moyx alors n1:=n1+1 sinon n2:=n2+1 fsi
        fsi ;
        lire (x) ;
    fait ;
    ecrire (n1, n2) ;
fin {exo2a}
```

```
programme exo2b :
 {fait la même chose que le programme exo2a}
    const minx=0 ; maxx=20 ; moyx=10 ;
    var ent n1, n2, x ;
debut
    n1:=0 ; n2:=0 ; lire (x) ;
    ttq x<>-1 faire
        cond
            x<minx ou x>maxx : ,
            x<moyx : n1:=n1+1 ,
            x>=moyx : n2:=n2+1
        fcond ;
        lire (x) ;
    fait ;
    ecrire (n1) ; ecrire (n2) ;
fin {exo2b}


programme exo3a :
{recherche d'une valeur dans une suite d'entiers terminée par -1}
    const vconnue=12 ; marqueur=-1 ;
    var ent vlue ; bool res ;
debut
    lire (vlue) ;
    ttq vlue<>marqueur et vlue<>vconnue faire lire (vlue) fait ;
    res:=vlue=vconnue ;
    ecrire (res) ;
fin {exo3a}


programme exo3b :
{comptage des occurrences d'une valeur dans une suite d'entiers terminée par -1}
    const vconnue=12 ; marqueur=-1 ;
    var ent vlue, nbv, nbautre ;
debut
    nbv:=0 ; nbautre:=0 ;
    lire (vlue) ;
    ttq vlue<>marqueur faire
        si vlue=vconnue alors nbv:=nbv+1
        sinon nbautre:=nbautre+1 fsi ;
        lire (vlue)
    fait ;
    ecrire (nbv, nbautre) ;
fin {exo3b}
```

```
programme exo4 : {min et max d'une suite d'entiers terminée par -1}
    const marqueur=-1 ;
    var ent nblu, min, max ;
debut
    lire (nblu) ;
    si nblu=marqueur alors min:=0 ; max:=0 sinon min:=nblu ; max:=nblu fsi ;
    ttq nblu<>marqueur faire
        cond
            nblu>max : max:=nblu,
            nblu<min : min:=nblu
        fcond ;
        lire (nblu) ;
    fait ;
    ecrire (min, max) ;
fin {exo4}


programme exo5 : {valeur d'un polynôme par la méthode de Horner}
    var ent n, i, x, px, a ;
debut
    lire (n, x) ; px:=0 ; i:=n ;
    ttq i>=0 faire
        lire (a) ; px:=a+px*x ; i:=i-1 ;
    fait ;
    ecrire (x, px) ;
fin {exo5}
```

5 Compilation des procédures

Rappel : la déclaration des procédures dans le langage Projet est décrite au chapitre 2, page 13.

Pour présenter les différents concepts abordés nous nous appuyons sur l'exemple suivant, dans lequel nous avons matérialisé des "points d'observation" à l'aide de cercles numérotés.

```
programme exempleChapV :
    const val=10 ; b=faux ;
    var ent u, v ;
        bool x ;
        ent k ; (1)
    proc q (2) (3)  fixe (ent a) mod (ent b ; bool c)
        var ent k ; (4)
    debut
        ecrire (5+a*k-val>b et x) ;
        lire (k) ;
        c:=k=-1 ; (7)
        si k>a (8) alors b:=b+1 fsi (9)
    fin ;(5)
debut (6)
    lire (u) ; v:=0 ;
    q (u*val) (v, x) ;
    ...
fin
```

De nouvelles déclarations et de nouveaux traitements sont à prévoir pour la **compilation de programmes avec des procédures**. Ceci implique aussi quelques modifications des traitements déjà prévus pour la compilation de programmes sans procédure.

5.1 Déclaration des identificateurs

Dans le cas de la déclaration d'une variable, le champ `info` de `tabSymb` reçoit l'adresse d'exécution, cette adresse est absolue pour une variable globale, mais elle est relative au registre MAPILE `bp` pour un paramètre formel ou une variable locale (l'adressage relatif commence à 0).

La déclaration de l'entête d'une procédure ajoute deux lignes dans `tabSymb`, ainsi qu'une ligne par paramètre formel. Les déclarations des constantes et variables locales à la procédure sont ajoutées à la

suite dans `tabSymb`.

Il convient de noter que la déclaration de l'entête d'une procédure reste dans `tabSymb` pendant toute la compilation du programme dans sa globalité (avec les caractéristiques de ses paramètres formels). Cela permet en effet de compiler les appels à cette procédure.

A l'inverse, les déclarations des constantes et variables locales à la procédure ne peuvent plus, par définition, être utilisées après la compilation de la procédure. Ces déclarations ne doivent alors plus apparaître dans `tabSymb` quand le corps de la procédure est totalement compilé. Pour le programme `exempleChapV`, en (1), on a :

tabSymb

	code	*categorie*	*type*	*info*
bc=1	@val	CONSTANTE	ENT	10
2	@b	CONSTANTE	BOOL	0
3	@u	VARGLOBALE	ENT	0
4	@v	VARGLOBALE	ENT	1
5	@x	VARGLOBALE	BOOL	2
it=6	@k	VARGLOBALE	ENT	3

ipo- *po (code produit)*

1- réserver 4
3- bincond 0

Ce branchement permet d'enjamber le code objet produit pour la procédure q, l'adresse de branchement ne sera connue qu'au début du programme principal

L'indice `ipo` de l'argument du `bincond` produit (branchement en avant) doit être empilé dans `pileRep`.

Lors de la compilation d'un programme avec procédure(s), le bloc courant dans `tabSymb` correspond soit au programme principal (`bc`=1), soit à une procédure en cours de compilation (`bc`>1). En (2), on est toujours au niveau global (`bc`=1), la déclaration de la procédure `q` occupe deux emplacements dans `tabSymb` :

tabSymb

	code	*categorie*	*type*	*info*	*ipo-*	*po (code produit)*
bc=1	@val	CONSTANTE	ENT	10	*1-*	réserver 4
2	@b	CONSTANTE	BOOL	0	*3-*	bincond 0
3	@u	VARGLOBALE	ENT	0	*5-*	réserver 1
4	@v	VARGLOBALE	ENT	1		
5	@x	VARGLOBALE	BOOL	2		
6	@k	VARGLOBALE	ENT	3		
7	@q	PROC	NEUTRE	5	←le code de q commence en po[5]	
it=8	-1	PRIVEE	NEUTRE	0	←nombre de paramètres de q (non connu là)	

En (3) on change de contexte pour compiler la procédure `q`, `bc` prend la valeur 9 (=`it`+1), la déclaration des paramètres formels et des variables locales est réalisée localement à ce contexte et, en (4), on a :

tabSymb

	code	*categorie*	*type*	*info*
1	@val	CONSTANTE	ENT	10
2	@b	CONSTANTE	BOOL	0
3	@u	VARGLOBALE	ENT	0
4	@v	VARGLOBALE	ENT	1
5	@x	VARGLOBALE	BOOL	2
6	@k	VARGLOBALE	ENT	3
7	@q	PROC	NEUTRE	5
8	-1	PRIVEE	NEUTRE	3
bc=9	@a	PARAMFIXE	ENT	0
10	@b	PARAMMOD	ENT	1
11	@c	PARAMMOD	BOOL	2
it=12	@k	VARLOCALE	ENT	5

ipo-	*po (code produit)*
1-	réserver 4
3-	bincond 0
	←le code de q commence en po[5]
	←nombre de paramètres de q mis à jour

NB : La rupture entre les adresses d'exécution du paramètre `c` (adresse 2) et de la variable locale `k` (adresse 5) est due aux deux données de liaison nécessaires à MAPILE pour effectuer le retour de procédure (voir 3.3).

Le corps de la procédure est compilé avec cette configuration de `tabSymb`. En (5), on quitte la procédure `q` et on revient au contexte global (`bc` ← 1). La variable locale `k` disparaît (mise à jour de `it`) et on "masque " (par -1) le code des paramètres formels car ceux-ci ne doivent pas être visibles à l'extérieur de `q`. Par contre leur catégorie et leur type sont nécessaires pour compiler les appels à `q`.
Enfin, on produit l'instruction retour et, en (6), on résout le branchement inconditionnel vers la première instruction du corps du programme. En (6), `tabSymb` devient :

tabSymb

	code	*categorie*	*type*	*info*
bc=1	@val	CONSTANTE	ENT	10
2	@b	CONSTANTE	BOOL	0
3	@u	VARGLOBALE	ENT	0
4	@v	VARGLOBALE	ENT	1
5	@x	VARGLOBALE	BOOL	2
6	@k	VARGLOBALE	ENT	3
7	@q	PROC	NEUTRE	5
8	-1	PRIVEE	NEUTRE	3
9	-1	PARAMFIXE	ENT	0
10	-1	PARAMMOD	ENT	1
it=11	-1	PARAMMOD	BOOL	2

ipo-	*po (code produit)*
1-	réserver 4
3-	bincond **61** ← mise à jour
5-	réserver 1
	...
	instructions de la procédure q
	...
59-	retour 3

5.2 Appel de procédure

Lors de l'appel `q (u*val) (v, x)` effectué dans le programme *exempleChapV*, `tabSymb` a la même configuration qu'en (6) et la fin de `po`, avant l'appel, est :

```
61 -  lirent
62 -  affecterg  0
64 -  empiler    0
66 -  affecterg  1
```

La recherche de `@q`, dans `tabSymb`, permet de récupérer l'adresse de début de `q` dans `po` (`q` commence à `ipo` = 5 dans `po`), le nombre de paramètres de `q` (3) et permet également de retrouver les caractéristiques (catégorie, type) du premier paramètre. On évalue les paramètres effectifs (ceux de l'appel) en contrôlant leur validité (en nombre, compatibilité de catégorie et identité de type) par rapport au modèle que constitue la déclaration présente dans `tabSymb`.

Il ne reste plus ensuite qu'à effectuer l'appel à la procédure :

```
68 -  contenug     0       // paramètre fixe : expression
70 -  empiler      10
72 -  mul
73 -  empileradg   1       // paramètres mod : on fournit l'adresse d'un ident
75 -  empileradg   2       // par l'instruction MAPILE empilerad
77 -  appel        5   3   // appel
```

5.3 Exemples de programmes en langage Projet

Vous trouverez ci-après un ensemble de programmes (**avec procédures**) en langage PROJET (programmes sources disponibles dans le répertoire TestsProjet). Ces exemples de programmes ne comportent pas d'erreur. **Il vous revient de tester également des programmes comportant des erreurs sémantiques (non syntaxiques)** comme, par exemple, des erreurs de nombre, de type ou de catégorie (`fixe` ou `mod`) de paramètres lors d'un appel de procédure.

Exercice

Compiler ces programmes en langage Projet, puis exécuter ce code sur la machine à pile MAPILE.

```
programme exo6 : {calcul de factorielle n}
    var ent n, fn ;
    proc fact fixe (ent i) mod (ent fi)
        var ent fi1 ;
    debut
        si i=1 alors fi:=1 sinon fact (i-1) (fi1) ; fi:=i*fi1 fsi ;
    fin ; {fact}
debut
    lire (n) ;
    fact (n) (fn) ;
    ecrire (fn) ;
fin {exo6}
```

```
programme exo7 :
{calcul de la somme (resp. du produit) d'une suite de nombres entiers terminée
par 0 (-1)}
    var ent res ; bool op ;

    proc recurs mod (ent r)
        var ent x, rx ;
    debut
        lire (x) ;
        cond
            x=0 : r:=0 ; op:=vrai , {opération d'addition}
            x=-1 : r:=1 ; op:=faux {opération de multiplication}
        aut
            recurs ( ) (rx) ;
            si op alors r:=x+rx sinon r:=x*rx fsi
    fcond
    fin ; {recurs}

debut
    recurs ( ) (res) ; ecrire (res) ;
fin {exo7}
```

```
programme exo8 : {idem exo7}
    var ent res ; bool op ;

    proc verifdrapeau fixe (ent v) mod (ent r ; bool b)
        var bool drap ;
    debut
        drap:=faux ;
        cond
            v=0 : drap:=vrai ; r:=0 ; op:=vrai , {addition}
            v=-1 : drap:=vrai ; r:=1 ; op:=faux {multiplication}
        fcond ;
        b:=drap ;
    fin ; {verifdrapeau}

    proc recurs mod (ent r)
        var ent x ; bool marq ;
    debut
        lire (x) ; verifdrapeau (x) (r, marq) ;
        si non marq alors
            recurs ( ) (r) ; si op alors r:=x+r sinon r:=x*r fsi
        fsi
    fin ; {recurs}

debut
    recurs ( ) (res) ; ecrire (res) ;
fin {exo8}
```

```
programme exo9 :
    var ent x1, y1, x2, y2, d ;

    proc carre fixe (ent v) mod (ent v2)
    debut
        v2:=v*v ;
    fin ; {carre}

    proc dist fixe (ent x1, y1, x2, y2) mod (ent d)
        var ent d1, d2 ;
    debut
        carre (x2-x1) (d1) ; carre (y2-y1) (d2) ; d:=d1+d2 ;
    fin ; {dist}

debut
    lire (x1,y1, x2, y2) ;
    dist (x1, y1, x2, y2) (d) ;
    ecrire (d) ;
fin {exo9}
```

6

Compilation séparée - Édition de liens

Le langage PROJET permet de développer des applications composées d'unités (`programme` ou `module`) écrites et compilées séparément. Chaque unité (`programme` ou `module`) peut définir des procédures visibles par les autres unités (liste des points d'entrée `def`) ou appeler des procédures définies dans d'autres unités (liste des références externes `ref`).

Une phase d'édition de liens est nécessaire pour regrouper les différents composants de l'application : une unité `programme` et d'éventuelles unités `module`. Le programme exécutable résultant de l'édition de liens est constitué par la concaténation du code objet de l'unité `programme` et du code objet des unités `module`. Un tel regroupement implique :

- des translations de données dans tout `module`, car l'adresse des variables globales calculée par le compilateur commence toujours à 0, or, dans le programme final, la première instruction (l'instruction réserver du `programme`) est chargée de réserver la totalité des variables globales (celle du `programme`, puis celles du premier `module`, ...) . Il faut aussi actualiser l'argument de l'instruction réserver de l'unité `programme`,
- des translations d'adresse dans le code objet de tout `module` pour tenir compte de la concaténation des codes objets, ceci concerne les instructions effectuant un branchement (`bsifaux`, `bincond`, `appel` de procédures locales au module),
- la résolution des adresses d'appel aux références externes, celles-ci étant bien sûr inconnues au moment de la compilation de l'unité (`programme` ou `module`).

6.1 Compilation séparée

Il faut que le compilateur prépare toutes les informations utiles à l'éditeur de liens pour intégrer l'unité compilée dans une application avec d'autres unités. Il doit, en particulier, construire le vecteur de translation pour repérer toutes les cases de `po` devant subir une modification au cours de l'édition de liens. La classe `ProgObjet` comprend la définition du vecteur de translation `vTrans` associé au programme objet construit. En fin de compilation, on dispose de la liste `transExt` des doublets (`adPo`, `X`) où `adPo` est l'indice `ipo` concerné et :

- `X=TRANSDON` → translation d'adresse de variable globale dans un `module`,
- `X=TRANSCODE` → translation d'adresse de branchement/appel de procédure dans un `module`,
- `X=REFEXT` → résolution d'un appel à une référence externe.

La méthode `vecteurTrans(X)` construit le vecteur de translation. Cette méthode est appelée dans `PtGen.java` par `modifierVecteurTrans(int x)` qui vérifie si il s'agit de la compilation d'un module ou d'un programme.

La liste `transExt` est recopiée, par la procédure `constObj()`, au début du fichier `.obj` devant les codes et arguments du programme MAPILE engendré par le compilateur.

Les autres informations que le compilateur doit préparer pour l'éditeur de liens sont placées dans un descripteur, structure composée de huit champs et déclarée à l'aide de la classe `Descripteur`. Cette classe comporte toutes les méthodes d'accès aux différents champs d'un descripteur et les méthodes `ecriredesc(String nomFichier)` et `lireDesc(String nomFichier)` :

```
class Descripteur {

    class EltDef {
        public String nomProc ;
        public int adPo, nbParam ;
        public EltDef (String nomProc, int adPo, int nbParam) {
            this.nomProc=nomProc ; this.adPo=adPo; this.nbParam=nbParam ;
        }
    }
    class EltRef {
        public String nomProc ;
        public int nbParam ;
        public EltRef (String nomProc, int nbParam) {
            this.nomProc=nomProc ; this.nbParam=nbParam ;
        }
    }
    private String unite ;  // ‘‘programme’’ ou ‘‘module’’
    private int tailleCode, // valeur de ipo en fin de compilation
            // nombre de variables globales déclarées dans l'unité
            tailleGlobaux,
            nbDef,  // nombre de points d'entrée dans la liste def
            nbRef,  // nombre de références externes dans la liste ref
            nbTransExt; // nombre de doublets dans la liste transext
    private static final int MAXDEF=10, MAXREF=10;
    // table des points d'entrée
    private EltDef [] tabDef = new EltDef [MAXDEF+1] ;
     // table des références externes
    private EltRef [] tabRef = new EltRef [MAXREF+1] ;
    public Descripteur ( ) {
        nbDef = 0 ; nbRef = 0 ; nbTransExt = 0 ;
        for (int i=0 ; i<=MAXDEF ; i++)
            tabDef[i] = new EltDef(‘‘inconnu’’, -2, -2) ;
        for (int i=0 ; i<=MAXREF ; i++)
            tabRef[i] = new EltRef(‘‘inconnu’’, -2) ;
    }
    // recopie le contenu du descripteur dans le fichier nomFichier.desc
    public void ecrireDesc (String nomFichier) {
        (...)
    }
    // affecte aux champs du descripteur le contenu du fichier nomFichier.desc
    public void lireDesc (String nomFichier) {
        (...)
    }
}
```

Pour simplifier, nous avons prévu comme **seul contrôle de cohérence** à effectuer lors de l'édition de liens : la vérification de l'égalité du nombre des paramètres pour chaque référence externe et le point d'entrée qui lui correspond, ce qui explique la présence d'un tel champ pour chaque élément de `tabDef` et de `tabRef`.

Chaque identificateur de procédure (la chaîne, pas le `numIdCourant` qui n'est connu que pendant la phase de compilation) figurant dans la liste `def` est placée dans `tabDef`. Il faut vérifier, en fin de compilation, qu'à tout point d'entrée correspond une déclaration de procédure (les champs `adPo` et `nbParam` sont alors remplis avec les valeurs adéquates).

Pour chaque procédure spécifiée dans la liste `ref`, on doit remplir un élément de `tabRef` à l'aide du nom (la chaîne) de la procédure et du nombre des paramètres extrait de la spécification. Il faut également recopier la spécification de la référence externe dans `tabSymb` afin de pouvoir compiler les appels à cette procédure (deux lignes pour la déclaration de la procédure + une ligne par déclaration de paramètre). Dans le champ `info` relatif à l'adresse de la procédure dans le code objet, on met le rang ($\geq$1) de la procédure dans la liste `ref`.

À la fin de la compilation de l'unité, `ess1.pro` par exemple, on doit :

- recopier le descripteur dans le fichier `ess1.desc` (méthode `ecrireDesc` de la classe `Descripteur`),
- créer le fichier objet `ess1.obj`, en y écrivant d'abord les couples de la liste `transExt` puis le contenu du tableau `po`. **Rq :** l'appel à `constObj()` le fait automatiquement.

Ainsi `ess1.obj` commencera par :

```
4   3   // liste transExt
7   3
42  3
49  3
57  3
60  3
1       // code reserver 1
1
29      // code appel init1
1
0
29      // code appel init2
2
0
```

puis la suite des codes MAPILE et des arguments associés.

Exemple

Dans les pages qui suivent nous fournissons les fichiers `.gen` et `.desc` qui résultent de la compilation de `ess1.pro`, `m1ess1.pro` et `m2ess1.pro` (*cf.* chapitre 2, pages 14).

programme source `ess1.pro`

```
programme ess1 :
    ref init1, init2, ajout1 fixe (ent), ajout2 fixe (ent), ecr1, ecr2 ;
    const minx=0 ; maxx=20 ; moyx=10 ;
    var ent x ;
debut
    init1 ; init2 ; lire (x) ;
    ttq x<>-1 faire
        si x>=minx et x<=maxx alors
            si x<moyx alors ajout1 (x) sinon ajout2 (x) fsi ;
        fsi ;
        lire (x) ;
    fait ;
    ecr1 ; ecr2 ;
fin {ess1}
```

code objet `ess1.gen`

1 -	réserver	1			*23* -	supeg			*41* -	appel	**3**	1	**ajout1(x)**
3 -	appel	**1**	0	**init1**	*24* -	contenug	0		*44* -	bincond	51		
6 -	appel	**2**	0	**init2**	*26* -	empiler	20		*46* -	contenug	0		
9 -	lirent				*28* -	infeg			*48* -	appel	**4**	1	**ajout2(x)**
10 -	affecterg	0			*29* -	et			*51* -	lirent			
12 -	contenug	0			*30* -	bsifaux	51		*52* -	affecterg	0		
14 -	empiler	-1			*32* -	contenug	0		*54* -	bincond	12		
16 -	diff				*34* -	empiler	10		*56* -	appel	**5**	0	**ecr1**
17 -	bsifaux	56			*36* -	inf			*59* -	appel	**6**	0	**ecr2**
19 -	contenug	0			*37* -	bsifaux	46		*62* -	arrêt			
21 -	empiler	0			*39* -	contenug	0						

descripteur `ess1.desc`

```
unite : ‘‘programme’’
tailleCode : 62
tailleGlobaux : 1
nbDef : 0
nbRef : 6
nbTransExt : 6
tabDef : aucun élément significatif
tabRef :
```

	nomProc	nbParam	transExt
1	"init1"	0	(4,3) (7,3) (42,3) (49,3) (57,3) (60,3)
2	"init2"	0	
3	"ajout1"	1	
4	"ajout2"	1	
5	"ecr1"	0	
6	"ecr2"	0	

module source `m1ess1.pro`

```
module m1ess1 :
    def init1, ajout1, ecr1 ;
    var ent n, min, max ;
    proc init1 debut n:=0 fin ; {init1}
    proc ajout1 fixe (ent y)
    debut
        si n=0 alors min:=y ; max:=y
        sinon si y<min alors min :=y sinon si y>max alors max:=y fsi fsi fsi ;
        n:=n+1 ;
    fin ; {ajout1}
    proc ecr1
    debut
        si n<>0 alors ecrire (n, min, max) sinon ecrire (0) fsi
    fin ; {ecr1}
```

code objet `m1ess1.gen`

1 -	empiler	0		***init1***	*32* -	bsifaux	**41**		*63* -	contenug	**0**	**ecr1**
3 -	affecterg	**0**			*34* -	contenul	0	0	*65* -	empiler	0	
5 -	retour	0			*37* -	affecterg	**1**		*67* -	diff		
7 -	contenug	**0**		**ajout1**	*39* -	bincond	**54**		*68* -	bsifaux	**81**	
9 -	empiler	0			*41* -	contenul	0	0	*70* -	contenug	**0**	
11 -	eg				*44* -	contenug	**2**		*72* -	ecrent		
12 -	bsifaux	**26**			*46* -	sup			*73* -	contenug	**1**	
14 -	contenul	0	0		47 -	bsifaux	**54**		*75* -	ecrent		
17 -	affecterg	**1**			*49* -	contenul	0	0	*76* -	contenug	**2**	
19 -	contenul	0	0		*52* -	affecterg	**2**		*78* -	ecrent		
22 -	affecterg	**2**			*54* -	contenug	**0**		*79* -	bincond	**84**	
24 -	bincond	**54**			*56* -	empiler	1		*81* -	empiler	0	
26 -	contenul	0	0		*58* -	add			*83* -	ecrent		
29 -	contenug	**1**			*59* -	affecterg	**0**		*84* -	retour	0	
31 -	inf				*61* -	retour	1					

descripteur `m1ess1.desc`

```
unite : ‘‘module’’
tailleCode : 85
tailleGlobaux : 3
nbDef : 3
nbRef : 0
nbTransExt : 21
tabDef :
```

	nomProc	adPo	nbParam	transExt
1	"init1"	1	0	(4,1) (8,1) (13,2) (18,1) (23,1) (25,2) (30,1)
2	"ajout1"	7	1	(33,2) (38,1) (40,2) (45,1) (48,2) (53,1) (55,1)
3	"ecr1"	63	0	(60,1) (64,1) (69,2) (71,1) (74,1) (77,1) (80,2)

```
tabRef : aucun élément significatif
```

module source `m2ess1.pro`

```
module m2ess1 :
    def init2, ajout2, ecr2 ;
    var ent n, min, max ;
    proc init2 debut n:=0 fin ; {init2}
    proc ajout2 fixe (ent y)
    debut
        si n=0 alors min:=y ; max:=y
        sinon si y<min alors min :=y sinon si y>max alors max:=y fsi fsi fsi ;
        n:=n+1 ;
    fin ; {ajout2}
    proc ecr2
    debut
        si n<>0 alors ecrire (n, min, max) sinon ecrire (0) fsi
    fin ; {ecr2}
```

code objet `m2ess1.gen`

1 -	empiler	0		**init2**	*32* -	bsifaux	**41**		*63* -	contenug	**0**	**ecr2**
3 -	affecterg	**0**			*34* -	contenul	0	0	*65* -	empiler	0	
5 -	retour	0			*37* -	affecterg	**1**		*67* -	diff		
7 -	contenug	**0**		**ajout2**	*39* -	bincond	**54**		*68* -	bsifaux	**81**	
9 -	empiler	0			*41* -	contenul	0	0	*70* -	contenug	**0**	
11 -	eg				*44* -	contenug	**2**		*72* -	ecrent		
12 -	bsifaux	**26**			*46* -	sup			*73* -	contenug	**1**	
14 -	contenul	0	0		*47* -	bsifaux	**54**		*75* -	ecrent		
17 -	affecterg	**1**			*49* -	contenul	0	0	*76* -	contenug	**2**	
19 -	contenul	0	0		*52* -	affecterg	**2**		*78* -	ecrent		
22 -	affecterg	**2**			*54* -	contenug	**0**		*79* -	bincond	**84**	
24 -	bincond	**54**			*56* -	empiler	1		*81* -	empiler	0	
26 -	contenul	0	0		*58* -	add			*83* -	ecrent		
29 -	contenug	**1**			*59* -	affecterg	**0**		*84* -	retour	0	
31 -	inf				*61* -	retour	1					

descripteur `m2ess1.desc`

`unite : ‘‘module’’`
`tailleCode : 85`
`tailleGlobaux : 3`
`nbDef : 3`
`nbRef : 0`
`nbTransExt : 21`
`tabDef :`

	nomProc	adPo	nbParam	transExt
1	"init2"	1	0	(4,1) (8,1) (13,2) (18,1) (23,1) (25,2) (30,1)
2	"ajout2"	7	1	(33,2) (38,1) (40,2) (45,1) (48,2) (53,1) (55,1)
3	"ecr2"	63	0	(60,1) (64,1) (69,2) (71,1) (74,1) (77,1) (80,2)

`tabRef : aucun élément significatif`

6.2 Édition de liens

L'éditeur de liens regroupe les codes objets d'un programme et d'au plus 5 modules dont les noms sont obtenus par dialogue. Le travail est réparti en deux phases bien distinctes.

6.2.1 Phase 1 - Préparation des translations

On commence par acquérir le nom du programme (`ess1`) et des modules (`m1ess1` et `m2ess1`), puis à l'aide des descripteurs `ess1.desc`, `m1ess1.desc` et `m2ess1.desc` (fichiers lus par la méthode `lireDesc` de la classe `Descripteur`) :

- on calcule les bases de décalage, soit, pour l'exemple traité :

	0	1	2
transDon	0	1	4
transCode	0	62	147

(0 pour ***ess1***, 1 pour ***m1ess1***, 2 pour ***m2ess1***)

- on construit un dictionnaire `dicoDef[1..60]` (au plus 6 unités et chacune définit au plus 10 points d'entrée), "union" des `def`, il ne faut bien sûr pas de double définition :

`dicoDef` :

	nomProc	***adPo***	***nbParam***
1	"init1"	63	0
2	"ajout1"	69	1
3	"ecr1"	125	0
4	"init2"	148	0
5	"ajout2"	154	1
6	"ecr2"	210	0

- enfin, on doit pouvoir associer chaque référence externe avec un point d'entrée (qui doit posséder le même nombre de paramètres) ; on en déduit `adFinale[0..5][1..10]` :

adFinale :		1	2	3	4	5	6
(ess1)	0	63	148	69	154	125	210
(m1ess1)	1	vide (pas de ***ref*** pour ***m1ess1***)					
(m2ess1)	2	vide (pas de ***ref*** pour ***m2ess1***)					

(rangs des ***ref*** dans l'unité)

6.2.2 Phase 2 - Concaténation de code

Les tables `transDon`, `transCode` et `adFinale` ayant été construites, il ne reste plus qu'à concaténer les codes objets en procédant aux translations/résolutions indiquées dans les listes `transExt`. On obtient alors un programme `ess1.map` exécutable sur MAPILE ; la procédure `Mnemo.creerFichier` de la classe `Mnemo` permet de créer le fichier mnémonique `ess1.ima`.

1	réserver	7	
3	appel	63	0
6	appel	148	0
9	lirent		
10	affecterg	0	
12	contenug	0	
14	empiler	-1	
16	diff		
17	bsifaux	56	
19	contenug	0	
21	empiler	0	
23	supeg		
24	contenug	0	
26	empiler	20	
28	infeg		
29	et		
30	bsifaux	51	
32	contenug	0	
34	empiler	10	
36	inf		
37	bsifaux	46	
39	contenug	0	
41	appel	69	1
44	bincond	51	
46	contenug	0	
48	appel	154	1
51	lirent		
52	affecterg	0	
54	bincond	12	
56	appel	125	0
59	appel	210	0
62	arrêt		

bien noter la nouvelle valeur de l'argument de la première instruction réserver du programme final.

63	empiler	0	
65	affecterg	1	
67	retour	0	
69	contenug	1	
1	empiler	0	
73	eg		
74	bsifaux	88	
76	contenul	0	0
79	affecterg	2	
81	contenul	0	0
84	affecterg	3	
86	bincond	116	
88	contenul	0	0
91	contenug	2	
93	inf		
94	bsifaux	103	
96	contenul	0	0
99	affecterg	2	
101	bincond	116	
103	contenul	0	0
106	contenug	3	
108	sup		
109	bsifaux	116	
111	contenul	0	0
114	affecterg	3	
116	contenug	1	
118	empiler	1	
120	add		
121	affecterg	1	
123	retour	1	
125	contenug	1	
127	empiler	0	
129	diff		
130	bsifaux	143	
132	contenug	1	
134	ecrent		
135	contenug	2	
137	ecrent		
138	contenug	3	
140	ecrent		
141	bincond	146	
143	empiler	0	
145	ecrent		
146	retour	0	

148	empiler	0	
150	affecterg	4	
152	retour	0	
154	contenug	4	
156	empiler	0	
158	eg		
159	bsifaux	173	
161	contenul	0	0
164	affecterg	5	
166	contenul	0	0
169	affecterg	6	
171	bincond	201	
173	contenul	0	0
176	contenug	5	
178	inf		
179	bsifaux	188	
181	contenul	0	0
184	affecterg	5	
186	bincond	201	
188	contenul	0	0
191	contenug	6	
193	sup		
194	bsifaux	201	
196	contenul	0	0
199	affecterg	6	
201	contenug	4	
203	empiler	1	
205	add		
206	affecterg	4	
208	retour	1	
210	contenug	4	
212	empiler	0	
214	diff		
215	bsifaux	228	
217	contenug	4	
219	ecrent		
220	contenug	5	
222	ecrent		
223	contenug	6	
225	ecrent		
226	bincond	231	
228	empiler	0	
230	ecrent		
231	retour	0	

Travail à effectuer

Préambule : la réalisation des TP nécessite un travail non négligeable en dehors des séances de TP. Vous avez de nombreux créneaux « libres » sur ADE qui peuvent être utilisés à préparer vos séances de TP.

Le projet de compilation proposé consiste à développer un compilateur d'unités écrites en langage Projet (programme et modules), puis un éditeur de liens permettant la réalisation d'une application complète à partir d'un programme et de modules compilés séparément. L'application ainsi obtenue peut alors être exécutée sur la machine virtuelle MAPILE mise à disposition.

Notez toutefois qu'une application peut se réduire à un programme seul (sans module). Ce sera le cas dans toute la première partie de ce projet.

L'organisation du travail et des séances de TP est la suivante :

- TP4 : Utilisation de la machine d'exécution Mapile.
  Compilation «à la main» de l'exercice 3b, poly p30, en code Mapile. Exécution du code obtenu sur la machine Mapile.
  (Vous pouvez aussi compiler «à la main», puis exécuter, les exercices 2a et 2b, poly p30)
- TP5 : Utilisation de ANTLR (générateur automatique d'analyseurs lexicaux et syntaxiques) Exercice sur les expressions arithmétiques, poly p8.
  Si possible, début du projet de compilation : compilation des déclarations et début expressions.
- TP6-7 : Compilation des déclarations, des expressions, des instructions écriture, lecture et affectation.
- TP8-9 : Compilation des instructions si, ttq, cond.
  Projet à rendre (projet.g et PtGen.java) sur Moodle au plus tard lundi 18/03 avant 20h.
- TP10-11 : Compilation des procédures (déclaration et appels). **Projet à rendre (projet.g et PtGen.java) sur Moodle au plus tard mardi 1/04 avant 20h.**
- TP12-13 : Compilation séparée afin de préparer l'édition de lien (programmes et modules).
- TP14-15 : Edition de liens.

Le projet complet est à rendre au plus tard JEUDI soir 18/04/2024 pour tous.

NB1 : Le contrôle de projet (**CC2**) aura lieu le **vendredi 19/04/2024 à 15h**.

NB2 : Le contrôle de projet porte uniquement sur l'ensemble de la compilation et édition de liens réalisées en TP (pas de nouvelle spécification, aucune nouvelle instruction, etc.). Aucun document n'est permis mais la grammaire, la liste des codes Mapile et les entêtes de procédures nécessaires sont rappelées.