MASTER 1re ANNÉE, COMPILATION

Projet : Compilateur pseudo-pascal

UNIVERSITÉ CLAUDE BERNARD LYON 1, DÉPARTEMENT INFORMATIQUE

Révision : 1.3.147
En date du : 16 octobre 2014
Date de génération : 16 octobre 2014

Auteur:
Erwan Guillou

Année:
2014-2015

Table des matières

Chapitre 1

Le langage

Sommaire

1.1 Les identificateurs

Ils servent à donner un nom à un objet.

Syntaxe
On appelle lettre un caractère de 'a'..'z' ou 'A'..'Z' ou '_'.
On appelle digit un caractère de '0'..'9'.
Un identificateur Pascal est une suite de lettres ou de digit accolées, commençant par une lettre.

Exemples
x, y1, jour, mois, annee, NbCouleurs, longueur_ligne.

Remarques
- Il n'y a pas de différence entre minuscules et majuscules.
- On n'a pas le droit de mettre d'accents, ni de caractères de ponctuation.
- Un identificateur doit être différent des mots clés (begin, function, real, . . .).

On se sert des identificateurs pour : le nom du programme, les noms de variables, les noms de constantes, les noms de types.

1.2 Les types primitifs

Un type décrit un ensemble de valeurs et un ensemble d'opérateurs sur ces valeurs.

1.2.1 Type entier : integer

Entier signé en complément à deux sur 16 bits.
Sur 16 bits, un entier à sa valeur dans $-32768...+32767$.

Opérateurs sur les entiers
- abs(x) valeur absolue de x.
- pred(x) x - 1.
- succ(x) x + 1.
- odd(x) true si x est impair, false sinon.
- sqr(x) le carré de x.
- + x identité.
- - x signe opposé.
- x + y addition.
- x - y soustraction.
- x * y multiplication.
- x / y division, fournissant un résultat de type réel.
- x div y dividende de la division entière de x par y.

— x mod y reste de la division entière, avec y non nul.

Remarques
— Attention, les opérateurs /, div et mod, produisent une erreur à l'exécution si y est nul.
— Lorsqu'une valeur (ou un résultat intermédiaire) dépasse les bornes au cours de l'exécution, on a une erreur appelée débordement arithmétique.

1.2.2 Type réel : real

Leur domaine de définition dépend de la machine et du compilateur utilisés. On code un réel avec une certaine précision, et les opérations fournissent une valeur approchée du résultat dit < juste >.

Exemples de real 0.0 ; -21.4E3 (= $-21.4 * 10^3$ = -21400) ; 1.234E-2 (= $1.234 * 10^{-2}$)

Opérateurs sur les réels :
— + x identité.
— - x signe opposé.
— x + y addition.
— x - y soustraction.
— x * y multiplication.
— x / y division, fournissant un résultat de type réel.
— abs(x) valeur absolue de x,
— sqr(x) carré de x
— sqrt(x) racine carrée de x
— trunc(x) partie entière de x
— round(x) entier le plus proche de x
— sin(x), cos(x), exp(x), ln(x)

Remarques
— Si l'un au moins des 2 arguments est réel, le résultat est réel pour : x - y, x + y, x * y.
— Résultat réel que l'argument soit entier ou réel : x / y (y doit être non nul)

1.2.3 Type caractère : char

Le jeux des caractères comportant les lettres, les digits, l'espace, les ponctuations, etc, est codé sur un octet non signé.

Le choix et l'ordre des 256 caractères possible dépend de la machine et de la langue. Sur PC, on utilise le code ASCII, où 'A' est codé par 65, 'B' par 66, 'a' par 97, ' ' par 32, '' par 123, etc.

Le code ascii est organisé comme suit : de 0 à 31, sont codés les caractères de contrôle (7 pour le signal sonore, 13 pour le saut de ligne, etc). De 32 à 127, sont codés les caractères

et ponctuations standards et internationaux. Enfin de 128 à 255, sont codés les caractères accentués propres à la langue, et des caractères semi-graphiques.

Opérateurs sur les caractères :

- ord(c) numéro d'ordre dans le codage ; ici < code ascii >.
- chr(a) le résultat est le caractère dont le code ascii est a.
- succ(c) caractère suivant c dans l'ordre ascii , chr(ord(c)+1)
- prec(c) caractère précédent c dans l'ordre ascii.

Remarques

- Le caractère apostrophe se note "'".
- Une suite de caractères telle que 'Il y a' est une chaîne de caractères ; il s'agit d'un objet de type string, que l'on verra plus loin.

1.2.4 Type booléen : boolean

Utilisé pour les expressions logiques.
Deux valeurs : false (faux) et true (vrai).

Opérateurs sur les booléens :

- not (négation), and (et), or (ou).
- Opérateurs de comparaison (entre 2 entiers, 2 réels, 1 entier et 1 réel, 2 chars, 2 booléens) : <, >, <=, >=, = (égalité, à ne pas confondre avec l'attribution :=), <> (différent).
  Le resultat d'une comparaison est un booléen.
  On peut comparer 2 booléens entre eux, avec la relation d'ordre false < true.

Remarques

- En mémoire, les booléens sont codés sur 1 bit, avec 0 pour false et 1 pour true. De là les relations d'ordre.
- Les opérateurs booléens not, and, or s'apparentent approximativement à (1 - x), *, +.

1.2.5 Type chaîne : string

On code une chaîne de caractère telle que 'bonjour' dans un objet de type string.

```
string [m]
```

où m est une constante entière donnant le nombre maximum de caractères pouvant être mémorisés. Exemple :

```
VAR s : string[80];
BEGIN
    s := 'Le ciel est bleu.';
```

```
    writeln (s);
END.
```

Codage :
Ayant déclaré s : string[80], comment sont codés les caractères ? En interne, Pascal réserve un array [0..80] of char. Le premier caractère est s[1], le deuxième est s[2], etc. La longueur courante de la chaîne est codé dans la case 0 (ie : ord(s[0])).

Remarques :

— Affecter une chaîne plus longue que l'espace réservé à la déclaration est une erreur.
— Comme la longueur courante est codée sur un char, elle est limitée à 255.

Opérateurs sur les chaînes :

— a := '' Chaîne vide (longueur 0).
— a := b Recopie de b dans a.
— a := c + d Concaténation en une seule chaîne. c et d de types string ou char ; le résultat est un string.
— length(a) Longueur courante de a, résultat entier.

```
CONST Slogan = 'lire la doc';
VAR s1, s2 : string[100];
    i : integer;
BEGIN
    s1 := 'veuillez ';
    s2 := s1 + Slogan;
    writeln ('s2 = ''', s2, '''');
    writeln ('Longueur courante de s2 : ', length(s2) );
    write ('Indices des ''l'' dans s2 : ');
    for i := 1 to length(s2) do
        if s2[i] = 'l' then write(i, ' ');
    writeln;
END.
```

Comparaison entre 2 chaînes : les opérateurs =, <>, <, >, <=, >=, sont utilisables, et le résultat est un booléen. La comparaison se fait selon l'ordre lexicographique du code ASCII.

1.3 Les types construits

1.3.1 Type intervalle

C'est un sous-ensemble de valeurs consécutives d'un type hôte.

```
N .. M
```

où N et M sont des constantes du même type, et sont les bornes inférieures et supé- rieures de l'intervalle, N et M inclus.

```
VAR
    pourcentage : 0 .. 100; { le type hote est integer }
    digit : '0' .. '9'; { le type hote est char }
    reponse : false .. true; { le type hote est boolean }
```

Remarques

— Il faut impérativement que le type hôte soit codé sur un entier (signé ou non, sur un nombre de bits quelconque). On dit alors que ce type hôte est un type ordinal.
— Ainsi les types integer, char et boolean sont des types ordinaux.
— Seul un type ordinal admet les opérateurs pred, succ et ord (le précédent, le successeur et le numéro d'ordre dans le codage).
— Par contre le type real n'est pas ordinal, et donc on ne pas créer un type intervalle avec des réels, il n'y a pas de notion de < réels consécutifs >.
— Un autre cas de type non ordinal est le type string pour les chaînes de carac- tères, qui n'est pas codé sur un mais sur plusieurs entiers. On ne peut donc pas déclarer 'aaa'..'zzz'.

Bonne habitude : Utiliser des constantes nommées pour borner les intervalles. De la sorte on pourra consulter ces valeurs pendant le programme, et ces bornes ne seront écrites qu'une seule fois.

```
CONST
    PMin = 0;
    PMax = 100;
VAR
    pourcentage : PMin .. PMax;
BEGIN
    writeln ('L''intervalle est ', PMin, ' .. ', PMax);
END.
```

1.3.2 Type énuméré

Il est fréquent en programmation que l'on aie à distinguer plusieurs cas, et que l'on cherche à coder le cas à l'aide d'une variable.

Exemple

```
VAR
    feux : 0..3; { rouge, orange, vert, clignotant }
BEGIN
{ ... }
    if feux = 0
    then Arreter
    else if feux = 1
    then Ralentir
    else if feux = 2
{ ... }
END.
```

Ceci est très pratique mais dans un programme un peu long cela devient rapide- ment difficile à comprendre, car il faut se souvenir de la signification du code. D'où l'intérêt d'utiliser un type énuméré, qui permet de donner un nom aux valeurs de code :

```
VAR
    feux : (Rouge, Orange, Vert, Clignotant);
BEGIN
{ ... }
    if feux = Rouge
    then Arreter
    else if feux = Orange
    then Ralentir
    else if feux = Vert
{ ... }
END.
```

— En écrivant cette ligne, on déclare en même temps :
  — la variable feux, de type énuméré (toujours codée sur un entier),
  — et les constantes nommées Rouge, Orange, Vert et Clignotant.
— à ces constantes sont attribuées les valeurs 0, 1, 2, 3 (la première constante prend toujours la valeur 0).
  — On ne peut pas choisir ces valeurs soi-même, et ces identificateurs ne doivent pas déjà exister.
  — L'intérêt n'est pas de connaître ces valeurs, mais d'avoir des noms explicites.
— Le type énuméré étant codé sur un entier, il s'agit d'un type ordinal et on peut :
  — utiliser les opérateurs pred, succ et ord (exemple : pred(Orange) est Rouge, succ(Orange) est Vert, ord(Orange) est 1).
  — déclarer un type intervalle à partir d'un type énuméré (exemple : Rouge..Vert).

1.3.3 Type enregistrement

Il s'agit simplement de regrouper des variables V1, V2, ... de différents types T1, T2, ... dans une variable "à tiroirs".

```
Record
    V1 : T1;
    V2 : T2;
    { ... }
End;
```

Soit r une variable de ce type ; on accède aux différents champs de r par r.V1, r.V2, ... Reprenons l'exemple du programme portrait.

```
{ ... }
TYPE
{ ... }
    personne_t = Record
        taille : taille_t;
        cheveux : cheveux_t;
        yeux : yeux_t;
```

```
    End;
VAR
    bob, luc : personne_t;
BEGIN
    bob.taille := 180;
    bob.cheveux := Brun;
    bob.yeux := Noir;
    luc := bob;
END.
```

Remarque La seule opération globale sur un enregistrement est : recopier le contenu de r2 dans r1 en écrivant : r2 := r1 ;

Ceci est équivalent (et plus efficace) que de copier champ à champ ; en plus on ne risque pas d'oublier un champ.

Il y a une condition : les 2 variables doivent être exactement du même type.

Intérêt de ce type Il permet de structurer très proprement des informations qui vont ensemble, de les recopier facilement et de les passer en paramètres à des procédures (on y reviendra).

Remarque générale Lorsqu'on crée un type T2 à partir d'un type T1, ce type T1 doit déjà exister ; donc T1 doit être déclaré avant T2.

1.3.4 Type tableau

```
array [ I ] of T
```

I étant un type intervalle, et T un type quelconque. Ce type définit un tableau comportant un certain nombre de cases de type T, chaque case est repérée par un indice de type I.

Exemple

```
TYPE vec_t = array [1..10] of integer;
VAR v : vec_t;
```

v est un tableau de 10 entiers, indicés de 1 à 10.

indice : 1 2 3 4 5 6 7 8 9 10

case mémoire :

— à la déclaration, le contenu du tableau est indéterminé, comme toute variable.

— On accède à la case indice i par v[i] (et non v(i)).

— Pour mettre toutes les cases à 0 on fait

```
for i := 1 to 10 do v[i] := 0;
```

Remarque : L'intervalle du array peut être de tout type intervalle, par exemple 1..10, 'a'..'z', false..true, ou encore un intervalle d'énumérés Lundi..Vendredi. On aurait pu déclarer vecteur comme ceci (peu d'intérêt) :

```
TYPE interv = 1..10 ; vec_t = array [ interv ] of integer;
```

Contrôle des bornes Il est en général conseillé de repérer les bornes de l'intervalle avec des constantes nommées : si on décide de changer une borne, cela est fait à un seul endroit dans le programme.

L'écriture préconisée est donc

```
CONST vec_min = 1; vec_max = 10;
TYPE vec_t = array [vec_min..vec_max] of integer;
```

Règle 1 Il est totalement interdit d'utiliser un indice en dehors de l'intervalle de déclaration, sinon on a une erreur à l'exécution. Il faut donc être très rigoureux dans le programme, et ne pas hésiter à tester si un indice i est correct avant de se servir de v[i].

Exemple Programme demandant à rentrer une valeur dans le vecteur.

```
CONST vec_min = 1; vec_max = 10;
TYPE vec_t = array [vec_min..vec_max] of integer;
VAR v : vect_t; i : integer;
BEGIN
    write ('i ? '); readln(i);
    if (i >= vec_min) and (i <= vec_max)
    then begin
        write ('v[', i, '] ? '); readln(v[i]);
    end
    else writeln ('Erreur, i hors intervalle ', vec_min, '..',
       vec_max);
END.
```

Règle 2 Le test d'un indice i et de la valeur en cet indice v[i] dans la même expression sont interdits.

Exemple

```
if (i >= vec_min) and (i <= vec_max) and (v[i] <> -1) then ...
else ...;
```

Une expression est toujours évaluée en intégralité ; donc si ($i <= vec\_max$), le test ($v[i] <> -1$) sera quand même effectué, alors même que l'on sort du vecteur !

Solution : séparer l'expression en 2.

```
if (i >= vec_min) and (i <= vec_max)
then if (v[i] <> -1) then ...
else ...
else ... { erreur hors bornes } ;
```

Recopie En Pascal, la seule opération globale sur un tableau est : recopier le contenu d'un tableau v1 dans un tableau v2 en écrivant : v2 := v1 ;

Ceci est équivalent (et plus efficace) que

```
for i := vec_min to vec_max do v2[i] := v1[i];
```

Il y a une condition : les 2 tableaux doivent être exactement de mêmes types, i.e issus de la même déclaration.

```
TYPE
    vecA = array [1..10] of char;
    vecB = array [1..10] of char;
VAR
    v1 : vecA; v2 : vecA; v3 : vecB;
BEGIN
    v2 := v1; { legal car meme type vecA }
    v3 := v1; { illegal, objets de types <> vecA et vecB }
    ...
END
```

1.4 Déclarations

1.4.1 Déclarations de constantes

Une constante est désignée par un identificateur et une valeur, qui sont fixés en début de programme, entre les mots clés CONST et VAR. La valeur ne peut pas être modifiée, et ne peut pas être une expression.

```
identificateur = valeur_constante;
{ ou }
identificateur : type = valeur_constante;
```

Dans la première forme, le type est sous-entendu (si il y a un point, c'est un réel, sinon un entier ; si il y a des quotes, c'est un caractère (un seul) ou une chaîne de caractères (plusieurs).

```
PROGRAM constantes;
CONST
        faux = false;
        entier = 14; { constantes NOMMEES }
        reel = 0.0;
        carac = 'z';
        chaine = 'hop';
        pourcent : real = 33.3; { seconde forme avec type }
VAR
{ variables }
BEGIN
{ instructions }
END.
```

1.4.2 Déclarations de variables

Une variable représente un objet d'un certain type ; cet objet est désigné par un identificateur. Toutes les variables doivent être déclarées après le VAR.

```
identificateur : type ;
```

On peut déclarer plusieurs variables de même type en même temps, en les séparant par des virgules (voir exemple ci-dessous). à la déclaration, les variables ont une valeur indéterminée. On initialise les variables juste après le BEGIN (on ne peut pas le faire dans la déclaration). Utiliser la valeur d'une variable non initialisée est une erreur grave !

```
VAR
        a, b, c : integer;
BEGIN
{ Partie initialisation }
END
```

1.4.3 Déclarations de type

Créer un type, c'est bien, mais le nommer, c'est mieux. On déclare les noms de types entre les mots clés TYPE et VAR.

```
nom_du_type = type;
```

Exemple

```
TYPE
        couleurs_feux_t = (Rouge, Orange, Vert, Clignotant);
VAR
        feux : couleurs_feux_t;
```

De la sorte couleurs_feux_t est un nom de type au même titre que integer ou char.

Exemple complet

```
PROGRAM portrait;
CONST
        TailleMin = 50; { en cm }
        TailleMax = 250;
TYPE
        taille_t = TailleMin .. TailleMax;
        couleurs_t = (Blond, Brun, Roux, Bleu, Marron, Noir, Vert)
           ;
        cheveux_t = Blond .. Roux;
        yeux_t = Bleu .. Vert;
VAR
        taille_bob, taille_luc : taille_t;
        cheveux_bob, cheveux_luc : cheveux_t;
        yeux_bob, yeux_luc : yeux_t;
BEGIN
        taille_bob := 180;
        cheveux_bob := Brun;
        yeux_bob := Noir;
END.
```

1.5 Procédures et fonctions

1.5.1 Les procédures

Une procédure est un sous-programme. écrire des procédures permet de découper un programme en plusieurs morceaux. Chaque procédure définit une nouvelle instruction, que l'on peut appeler en tout endroit du programme. On peut ainsi réutiliser le code d'un sous-programme. Lorsqu'on découpe un problème en terme de procédures, puis qu'on implémente ces procédures, on fait ce qu'on appelle une analyse descendante : on va du plus général au détail.

Pseudo-passage de paramètres

Ecrivons une procédure Produit qui calcule z = xy.

```
PROGRAM exemple5;
VAR x, y, z, a, b, c, d : real;
PROCEDURE Produit;
BEGIN
        z := x * y;
END;
```

On veut se servir de Produit pour calculer c = ab et d = (a - 1)(b + 1).

```
BEGIN
        write ('a b ? '); readln (a, b);
        x := a; y := b; { donnees }
        Produit;
        c := z; { resultat }
        x := a-1; y := b+1; { donnees }
        Produit;
        d := z; { resultat }
        writeln ('c = ', c, ' d = ', d);
END.
```

Remarques

— L'écriture est un peu lourde.
— Il faut savoir que la procédure < communique > avec les variables x, y, z.
— Cela interdit de se servir de x, y, z pour autre chose que de communiquer avec la procédure ; sinon gare aux effets de bord !
— Deux sortes de paramètres : données et résultats.

Paramétrage

La solution élégante consiste à déclarer des paramètres à la procédure :

```
PROGRAM exemple5bis;
VAR a, b, c, d : real;
PROCEDURE Produit (x, y : real; var z : real); { parametres }
```

```
BEGIN
        z := x * y;
END;
BEGIN
        write ('a b ? '); readln (a, b);
        Produit (a, b, c); { passage de }
        Produit (a-1, b+1, d); { parametres }
        writeln ('c = ', c, ' d = ', d);
END.
```

Il y a deux sorte de passage de paramètres : le passage par valeur et le passage par référence.

— Passage par valeur : à l'appel, le paramètre est une variable ou une expression. C'est la valeur qui est transmise, elle sert à initialiser la variable correspondante dans la procédure (ici x est initialisé à la valeur de a et y à la valeur de b).

— Passage par référence : à l'appel, le paramètre est une variable uniquement (jamais une expression). C'est l'adresse mémoire (la référence) de la variable qui est transmise, non sa valeur. La variable utilisée dans la procédure est en fait la variable de l'appel, mais sous un autre nom (ici z désigne la même variable (zone mémoire) que a).

C'est le mot-clé var qui dit si le passage se fait par valeur (pas de var) ou par référence (présence du var). Pas de var = donnée ; présence du var = donnée/résultat.

1.5.2 Les fonctions

Une fonction est une procédure qui renvoie un résultat, de manière à ce qu'on puisse l'appeler dans une expression. Exemples y := cos(x) + 1 ; c := chr(x + ord('0')) ;

Syntaxe

```
FUNCTION nom_fonction ( parametres : types_params ) :
   type_resultat;
VAR locales : types_locales;
        res : type_resultat;
BEGIN
        { ... }
        nom_fonction := res;
END;
```

Tout ce que l'on a dit sur le paramétrage des procédures reste valable pour les fonctions.

Procédure vs fonction

Exemple du produit.

```
PROGRAM exemple5ter;
VAR a, b, c, d : real;
FUNCTION Produit (x, y : real) : real;
AR res : real;
BEGIN
        res := x * y;
```

```
        Produit := res;
END;
BEGIN
        write ('a b ? '); readln (a, b);
        c := Produit (a, b);
        d := Produit (a-1, b+1);
        writeln ('c = ', c, ' d = ', d);
END.
```

Passage de types enregistrement

Exemple On veut savoir si un couple d'amis est assorti. On fixe les règles suivantes : le couple est assorti si ils ont moins de 10 ans d'écart, ou si le mari est agé et riche.

```
PROGRAM assorti;
TYPE
        humain_t = Record
                age, taille : integer;
                riche : boolean;
        End;
        couple_t = Record
                homme, femme : humain_t;
                nb_enfant : integer;
        End;
VAR amis : couple_t;
FUNCTION difference_age (h, f : humain_t) : integer;
VAR res : integer;
BEGIN
        res := abs (h.age - f.age);
        difference_age := res;
END;
FUNCTION couple_assorti (c : couple_t) : boolean;
VAR res : boolean;
BEGIN
        res := false;
        if difference_age (c.homme, c.femme) < 10 then res := true
            ;
        if (c.homme.age > 75) and c.homme.riche then res := true;
        couple_assorti := res;
END;
BEGIN
        { ... }
        write ('Ce couple avec ', amis.nb_enfant, ' enfant(s) est
            ');
        if couple_assorti (amis) then writeln ('assorti.')
        else writeln ('non assorti.');
END.
```

Chapitre 2

La machine cible

Sommaire

2.1 Description

La machine cible est de type 80x86 sur une architecture 32 bits. Le jeu d'instructions disponibles est similaire à celui d'Intel mais moins complexe.

L'objectif est de générer un exécutable pour cette machine. L'outil utilisé pour transformer le code assembleur généré en code machine sera Nasm [1].

2.2 Représentation des données en mémoire

Type	Taille	Description
booléen	8 bits	0 équivaut à faux, le reste à vrai
char	8 bits	code ascii du caractère
entier	16 bits	entier signé
réel	32 bits	nombre réel en virgule flottante
chaine	8 + n*8bits	le premier octet contient la longueuer de la chaine les octets suivants contiennent les codes ascii des caractères
intervalle	n bits	dépend du type hôte
énumération	8 bits	on limite le nombre de "valeurs" à 256
structure	n bits	dépend de la description de la structure

2.3 Gestion des données en mémoire

Les variables globales, ainsi que les chaines de caractères statiques, sont définies dans un espace ad-hoc réservé de la mémoire. L'accès à ces données se fera par le biais de leur adresse effective.

Les variables locales sont allouées dynamiquement dans la pile. Les arguments d'un sous-programmes sont passés dans la pile. L'accès à ces données se fera par le biais d'un adressage relatif par rapport au registre *BP*.

Les arguments sont empilés dans l'ordre inverse de leur passage : le premier argument doit être en sommet de pile au moment de l'appel effectif du sous-programme. Les arguments seront dépilés : en fin de sous-programme en code 3 adresses (cf instruction *retourner*) et par l'appelant en code assembleur (cf exemples fournis).

On considerera que tous les calculs intermédiaires seront réalisés, et stockés, en mémoire (dans la pile) : le compilateur ne réalisera pas la phase d'allocation de registres. L'espace alloué à ces *temporaires* se trouvera à la suite de celui alloué aux variables locales dans la pile. L'accès aux temporaires se fera de la même manière que pour les variables locales.

2.4 Les registres

La machine cible dispose, notamment, de

1. http ://nasm.us/

- 4 registres de donnée 32 bits : EAX, EBX, ECX et EDX. Ces registres sont utilisables
  - en 32 bits
  - en 16 bits : AX, BX, CX et DX. qui correspondent aux 2 octets de poids faible du registre 32 bits.
  - en 8 bits : AH et AL pour AX par exemple, qui correspondent respectivement à l'octet de poids fort et de poids faible du registre 16 bits (idem pour BX, CX et DX).
- 2 registres 32 bits d'indexation : ESI et EDI (décomposable en 2 registres 16 bits SI et DI, cf ci-dessus)
- un registre 32 bits d'accès à la pile : EBP

2.5 Modes d'adressage

Dans les instructions assembleurs, les opérandes peuvent être spécifiée de plusieurs façons :

- immédiate : on donne directement la valeur comme dans *immXX*, imm8 définit une valeur 8 bits, imm16 une valeur 16 bits et imm32 une valeur 32 bits.
- registre : on spécifie le registre contenant la valeur *regXX*, reg8 définit un registre 8 bits, ...
- adressage : on spécifie l'adresse de la case mémoire contenant la donnée. Plusieurs modes d'adressage sont possibles :
  - adressage direct : on spécifie le déplacement en mémoire pour obtenir l'adresse effective de la donnée en mémoire : $[depl]$. Dans la pratique, on utilisera des étiquettes pour spécifier cette adresse.
  - adressage indirect par registre : on spécifie le registre contenant l'adresse effective de la donnée $[base]$
  - adressage basé : on spécifie le registre contenant l'adresse de base ainsi qu'un décalage (en octets) $[base + depl]$
  - adressage indexé : on spécifie l'adresse de base par une valeur immédiate et un décalage par un registre d'index. Il est possible d'appliquer un facteur d'échelle (1, 2, 4 ou 8) à l'index. Cet adressage est notamment utilisé pour les accès aux éléments d'un tableau (de type primitif) : $[index * scale + depl]$
  - adressage basé indexé : on spécifie le registre contenant l'adresse de base, le registre d'index (avec potentiellement un facteur d'échelle) et le décalage (par une valeur immédiate) $[base + (index * scale) + depl]$

depl	**base**	**index**	**scale**
aucun	*EAX*	*EAX*	1
*imm*16	*EBX*	*EBX*	2
*imm*32	*ECX*	*ECX*	4
	EDX	*EDX*	4
	ESI	*ESI*	

suite sur la prochaine page

depl	base	index	scale
	EDI *EBP* *ESP*	*EDI* *EBP*	

Lors de l'utilisation d'un adressage direct, indirect ou autre, il est parfois nécessaire de spécifier (forcer) la taille des arguments en utilisant les mots clefs : *byte* (8 bits), *word* (16 bits) ou *dword* (32 bits). Par exemple : $word[abx + esi + decalage]$ définit les 2 octets contenus à l'adresse $abx + esi + decalage$ et $abx + esi + decalage + 1$

2.6 Gestion des sous-programmes

Chapitre 3

Fonctionnement du compilateur

Sommaire

3.1 Arguments de la ligne de commande

Le compilateur que vous implémentez (et qui sera nommé **ppc**) devra accepter comme arguments :

— *-c fichier.pas* le nom du fichier source à compiler. Ce fichier sera situé dans le répertoire **pascal**.
— *-O* active les optimisations
— *-a* active la génération du code assembleur

3.2 Affichage des messages

Le compilateur devra s'arrêter à la première erreur rencontrée en fournissant un message d'erreur informatif de la forme :

$$Erreur\,fichier : ligne.colonne : fonction, message$$

Lors de la détection de certains problèmes ne générant pas d'erreur (comme la perte de précision lors de la conversion d'un réel vers un entier, une division par 0, ou encore une boucle infinie), un warning devra être généré sous la forme :

$$Warning\,fichier : ligne.colonne : fonction, message$$

3.3 Sorties obligatoires

Dans ce qui suis, le terme *p*refixe correspond au nom du fichier à compiler sans son extension. Le terme *X*XX décrit le nom d'un sous-programme.

Votre compilateur devra au minimum générer les fichiers suivants :

— sauvegarder la table des identificateurs, avant optimisation, dans un fichier nommé *prefixe.ti*
— sauvegarder les tables des symboles, avant optimisation, dans des fichiers nommés *prefixe.XXX.ts*
— sauvegarder le code 3 adresses non optimisé dans des fichiers nommés *prefixe.XXX.3ad*

Les fichiers intermédiaires (ti, ts, code 3 adresses) seront générés dans le répertoire **intermediaire**.

Vous devrez implémenter les optimisations vues en cours, qui ne seront utilisées que si l'option *-O* est utilisée. Si les optimisations sont activées, votre compilateur devra :

— sauvegarder le graphe de flot de chaque sous programme dans des fichiers nommés *prefixe.XXX.flot* dans le répertoire **intermediaire**.
— sauvegarder les données intermédiaires post optimisation (ti, ts, code 3 adresses) seront sauvegardées dans des fichiers nommées *prefixe.optim.ti*, *prefixe.XXX.optim.ts* et *prefixe.XXX.optim.3ad*.

3.4 Sorties optionnelles

Une partie optionnelle du projet concerne les sorties effectives pour la machine cible :
— sauvegarder le code assembleur pré-optimisation dans un fichier nommé *prefixe.asm* dans le répertoire **asm**.
— sauvegarder le code assembleur post-optimisation dans un fichier nommé *prefixe.optim.asm* dans le répertoire **asm**.

3.5 Exemple

En utilisant la ligne de commande suivante :

$$ppc\ -c\ pgcd.pas\ -O\ -a$$

Le code source sera lu depuis le fichier **pascal/pgcd.pas** et le compilateur génerera les fichiers (voir code source en fin de document) :
— tables des identificateurs : **intermediaire/pgcd.ti** et **intermediaire/pgcd.optim.ti**
— tables des symboles du programme principal : **intermediaire/pgcd.pgcd.ts** et **intermediaire/pgcd.pgcd.optim.ts**
— tables des symboles de la fonction de calcul : **intermediaire/pgcd.calculpgcd.ti** et **intermediaire/pgcd.calculpgcd.optim.ti**
— code 3 adresses du programme principal : **intermediaire/pgcd.pgcd.3ad** et **intermediaire/pgcd.pgcd.optim.3ad**
— code 3 adresses de la fonction de calcul : **intermediaire/pgcd.calculpgcd.3ad** et **intermediaire/pgcd.calculpgcd.optim.3ad**
— graphe de flot du programme principal : **intermediaire/pgcd.pgcd.flot**
— graphe de flot de la fonction de calcul : **intermediaire/pgcd.calculpgcd.flot**
— code assembleur : **asm/pgcd.asm** et **asm/pgcd.optim.asm**

Chapitre 4

La grammaire effective du langage

Sommaire

4.1 Spécificités du langage Pseudo-Pascal

La grammaire utilisée est une version simplifiée de la grammaire réelle du langage pascal :

— Les expressions utilisées dans les déclarations de constantes doivent être calculables (et donc calculées) lors de la phase de compilation du code source.
— D'un point de vue sémantiques, les fonctions et procédures doivent être déclarées dans l'ordre (ie de manière à ce qu'un appel soit résolvable lors de son analyse sémantique).
— Dans le cas d'une procédure ou fonction n'acceptant aucun argument, le parenthésage est obligatoirement omis.

4.2 Restrictions du langage

— Le formalisme utilisé interdit les déclarations imbriquées de procédure ou fonction.
— La déclaration de variables de types complexes doit passer par une première phase de déclaration du type complexe.
— Comme pour les variables, le type de retour d'un fonction (s'il est complexe) doit passer par une phase de déclaration de type.
— Aucune des fonctions prédéfinies sur les types primitifs n'est utilisable (ord, chr, ...).

4.3 La grammaire

Symbole	**Dérivation**
Program	program id ; Block .
Block	BlockDeclConst BlockDeclType BlockDeclVar BlockDeclFunc BlockCode
BlockSimple	BlockDeclConst BlockDeclVar BlockCode
BlockDeclConst	const ListDeclConst *epsilon*
ListDeclConst	ListDeclConst DeclConst DeclConst
DeclConst	id = Expression ; id : BaseType = Expression ;
BlockDeclType	type ListDeclType *epsilon*
ListDeclType	ListDeclType DeclType DeclType
DeclType	id = Type ;
BlockDeclVar	var ListDeclVar *epsilon*

suite sur la prochaine page

Symbole	Dérivation
ListDeclVar	ListDeclVar DeclVar DeclVar
DeclVar	ListIdent : SimpleType ;
ListIdent	ListIdent , id id
BlockDeclFunc	ListDeclFunc ; *epsilon*
ListDeclFunc	ListDeclFunc ; DeclFunc DeclFunc
DeclFunc	ProcDecl FuncDecl
ProcDecl	ProcHeader ; BlockSimple
FuncDecl	FuncHeader ; BlockSimple
ProcHeader	procedure id FormalArgs
FuncHeader	function id FormalArgs : SimpleType
FormalArgs	(ListFormalArgs) *epsilon*
ListFormalArgs	ListFormalArgs ; FormalArg FormalArg
FormalArg	ListIdent : SimpleType var ListIdent : SimpleType
Type	UserType SimpleType
UserType	EnumType InterType ArrayType RecordType
SimpleType	id BaseType
BaseType	integer real boolean char string
EnumType	(ListEnumValue)
ListEnumValue	ListEnumValue , id id
InterType	NSInterBase .. NSInterBase
NSInterBase	id TOKINTEGER TOKCHAR

suite sur la prochaine page

La grammaire effective du langage

Symbole	Dérivation
ArrayType	array [ArrayIndex] of SimpleType
ArrayIndex	id InterType
RecordType	record RecordFields end
RecordFields	RecordFields ; RecordField RecordField
RecordField	ListIdent : SimpleType
BlockCode	begin ListInstr end begin ListInstr ; end begin end
ListInstr	ListInstr SEPSCOL Instr Instr
Instr	while Expression do Instr repeat ListInstr until Expression for id := Expression to Expression do Instr for id := Expression downto Expression do Instr if Expression then Instr if Expression then Instr else Instr VarExpr := Expression write (ListeExpr) writeln (ListeExpr) read (VarExpr) id (ListeExpr) id BlockCode
Expression	MathExpr CompExpr BoolExpr AtomExpr VarExpr id (ListeExpr)
MathExpr	Expression + Expression Expression - Expression Expression * Expression Expression / Expression Expression div Expression Expression mod Expression Expression ** Expression - Expression + Expression
CompExpr	Expression = Expression

suite sur la prochaine page

Symbole	Dérivation
	Expression <> Expression Expression < Expression Expression <= Expression Expression > Expression Expression >= Expression
BoolExpr	Expression and Expression Expression or Expression Expression xor Expression not Expression
AtomExpr	(Expression) TOKINTEGER TOKREAL TOKBOOLEAN TOKCHAR TOKSTRING
VarExpr	id VarExpr [ListIndices] VarExpr . id
ListeExpr	ListeExpr , Expression Expression
ListIdent	ListIdent , id id

Chapitre 5

Le code intermédiaire

Sommaire

Le code 3 adresses est le langage intermédiaire permettant de faire certaines optimisations indépendantes de la machine cible. Les opérandes d'une instruction sont définies soient par des valeurs, soit par des références aux identificateurs. Les instructions définies sont non typées (le typage est donné par la table des symboles).

5.1 Le jeu d'instructions

Le jeux d'instruction disponible est le suivant :

instruction	description	arguments
nop	ne rien faire	
fin	termine l'exécution du programme	
appeler x	appeler un sous-programme	x est l'étiquette du sous-programme
reserver n	réserve la mémoire pour les données locales	n est le nombre d'octets à réserver pour les variables locales et les temporaires du sous-programme
retourner n	fin d'un sous-programme et retour à l'appelant	n correspond à la taille de l'espace mémoire alloué pour les arguments du sous-programme (y compris l'adresse du résultat dans le cas d'une fonction)
renvoyer x	initialise le résultat d'une fonction avec la valeur x attention ceci ne termine pas l'exécution du sous-programme	x est soit un identificateur soit une constante
x := y + z x := y - z x := y * z x := y / z x := - y	réalise l'opération mathématique entre y et z stocke le résultat dans x	x fait référence à un identificateur y et z sont soit des identificateurs soit des constantes
x := y & z x := y | z x := ! y	réalise l'opération booléenne entre y et z stocke le résultat dans x	x fait référence à un identificateur y et z sont soit des identificateurs soit des constantes
x := y < z	compare y et z stocke le résultat dans x	x fait référence à un identificateur y et z sont soit des identificateurs soit des constantes

suite sur la prochaine page

instruction	description	arguments
x := y > z x := y <= z x := y >= z x := y == z x := y != z		
x := y	recopie la valeur de y dans x	x fait référence à un identificateur y est soit un identificateur soit une constante
x := & y	stocke l'adresse de y dans x	x et y font référence à des identificateurs
x := * y	stocke dans x le contenu de la case mémoire pointée par y	x et y font référence à des identificateurs
* x := y	recopie la valeur de y dans la case mémoire pointée par x	x fait référence à un identificateur y est soit un identificateur soit une constante
aller a x	la prochaine instruction à exécuter est celle définie par l'étiquette x	x est un identificateur d'étiquette
si y aller a x	la prochaine instruction à exécuter est celle définie par l'étiquette x si la valeur de y est vrai	x est un identificateur d'étiquette y est un identificateur
empiler x	stocke la valeur de x en sommet de pile	x fait référence à un identificateur ou une constante
empiler & x	stocke l'adresse de x en sommet de pile	x fait référence à un identificateur
empiler * x	stocke le contenu de la cas mémoire pointée par x en sommet de pile	x fait référence à un identificateur
depiler x	dépile le sommet de pile et stocke la valeur dans x	x fait référence à un identificateur
depiler * x	dépile le sommet de pile et stocke la valeur dans la case mémoire pointée par x	x fait référence à un identificateur
ecrire x	écrit la valeur de x sur la sortie standard	x de type primitif
lire x	lit la valeur de x sur l'entrée standard	x de type primitif
sautdeligne	passe à la ligne sur la sortie standard	

5.2 Nommage des temporaires et étiquettes

— les temporaires (stockage des calculs intermédiaires) seront nommés selon le schéma suivant :

.__tempddd

où ddd est une numérotation globale au programme (démarrant à 0).

— les étiquettes de branchement seront nommées selon la structure de contrôle utilisée, avec une numérotation liée à cette structure de contrôle (numérotation globale au programme, voir la partie schéma de traduction du cours).

$$.siXXX, .alorsXXX, .sinonXXX, .finsiXXX$$

$$.whileXXX, .bclwhileXXX, .finwhileXXX$$

$$.repeatXXX, .finrepeatXXX$$

$$.forXXX, .testforXXX, .finforXXX$$

— les étiquettes des sous-programmes seront nommées selon le schéma suivant :

$$xxxxx$$

où xxxxx est le nom du sous-programme.

Pour plus de clarté, les étiquettes seront placées sur des instructions vides (ie *nop*, voir ci-dessous).

Chapitre 6

Schéma de traduction

Sommaire

6.1 Les expressions

6.1.1 Les opérateurs binaires

$$E \rightarrow E_1 \text{ op } E_3$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

Code 3 adresses
E_1.code E_3.code E.place = E_1.place op E_3.place

6.1.2 Les opérateurs unaires

$$E \rightarrow \text{ op } E_2$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

Code 3 adresses
E_2.code E.place = op E_2.place

6.1.3 Les valeurs constantes

$$E \rightarrow \textit{valeur}|\text{id}$$

Seuls les attributs type et valeur de l'expression seront initialisés. Aucun code n'est généré.

6.1.4 Les identificateurs

$$E \rightarrow \text{ id}$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

Code 3 adresses
si id fait référence à un argument passé par @ E.place = * id

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

Code 3 adresses

si id fait référence à une variable
  E.place@ = & id
si id fait référence à un argument passé par @
  E.place@ = id

6.1.5 Les structures

$$E \rightarrow E_1 \text{ . id}$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

Code 3 adresses
E_1.code@ temp = E_1.place@ + decalage (id, E_1.type) E.place = * temp

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

Code 3 adresses
E_1.code@ E.place@ = E_1.place@ + decalage (id, E_1.type)

6.1.6 Les tableaux

$$E \rightarrow E_1[E_3]$$

Après vérification des cas d'erreur, le code 3 adresses généré aura la forme suivante

Code 3 adresses
E_1.code@ E_3.code temp1 = E_3.place * taille (E_1.type.typeelt) temp2 = E_1.place@ + temp1 E.place = * temp1

le code 3 adresses généré pour calculer l'adresse de E aura la forme suivante

Code 3 adresses
E_1.code@ E_3.code temp1 = E_3.place * taille (E_1.type.typeelt) E.place@ = E_1.place@ + temp1

6.1.7 Les appels de fonction

$$E \rightarrow \text{id} \mid \text{id}\ (E_1, E_2, \cdots, E_n)$$

Après vérification des cas d'erreur, le code 3 adresses généré pour calculer la valeur de E aura la forme suivante

Code 3 adresses

empiler & E.place
pour chaque arg E_i (i : n -> 1)
  passage par valeur
  E_i.code
  empiler E_i.place
  passage par adresse
  E_i.code@
  empiler E_i.place@
appeler id

6.2 Les instructions

6.2.1 Les affectations

$$I \rightarrow E_1 := E_3$$

Dans le cas général :

Code 3 adresses
E_1.code@ E_3.code * E_1.place@ = E_3.place

Cas particulier E_1 = id (id de fonction)

Code 3 adresses
E_3.code renvoyer E_3.place

Cas particulier E_1 = id

Code 3 adresses
E_3.code id = E_3.place

6.2.2 Les conditionnelles

$$I \rightarrow \text{if } E \text{ then } I_4$$

Code 3 adresses
.siXXX : nop
E.code
si E.place aller a .alorsXXX
aller a .finsiXXX
.alorsXXX : nop
I_4.code
.finsiXXX : nop

$$I \rightarrow \text{if } E_2 \text{ then } I_4 \text{ else } I_6$$

Code 3 adresses
.siXXX : nop
E.code
si E.place aller a .alorsXXX
aller a .sinonXXX
.alorsXXX : nop
I_4.code
aller a .finsiXXX
.sinonXXX : nop
I_6.code
.finsiXXX : nop

6.2.3 Les boucles

$$I \rightarrow \text{while } E \text{ do } I_4$$

Code 3 adresses
.whileXXX : nop
E.code
si E.place aller a .bclwhileXXX
aller a .finwhileXXX
.bclwhileXXX : nop
I_4.code
aller a .whileXXX
.finwhileXXX : nop

$$I \rightarrow \text{repeat } I_2 \text{ until } E$$

Code 3 adresses
.repeatXXX : nop
I_2.code
E.code
si E.place aller a .finrepeatXXX

suite sur la prochaine page

Code 3 adresses
aller a .repeatXXX .finrepeatXXX : nop

$$I \rightarrow \text{for id} := E_4 \text{ to } E_6 \text{ do } I_8$$

Code 3 adresses
.forXXX : nop E_4.code id = E_4.place .testforXXX : nop E_6.code temp = id > E_6.place si temp aller a .finforXXX I_8.code id = id + 1 aller a .testforXXX .finforXXX : nop

$$I \rightarrow \text{for id} := E_4 \text{ downto } E_6 \text{ do } I_8$$

Code 3 adresses
.forXXX : nop E_4.code id = E_4.place .testforXXX : nop E_6.code temp = id < E_6.place si temp aller a .finforXXX I_8.code id = id - 1 aller a .testforXXX .finforXXX : nop

6.2.4 Les appels de procédure

$$I \rightarrow \text{id} \mid \text{id } (E_1, E_2, \cdots, E_n)$$

Code 3 adresses
pour chaque arg E_i (i : n -> 1) passage par valeur E_i.code empiler E_i.place passage par adresse

suite sur la prochaine page

Code 3 adresses
E_i.code@
empiler E_i.place@
appeler id

Chapitre 7

Implémentation

Sommaire

7.1 Conseils d'organisation

Afin de vous simplifier le travail, pensez bien à l'ordre dans lequel vous implémenterez les différents aspects du compilateur. Par exemple :

— étape 1 : remplissage de la table des identificateurs, remplissage de la table des symboles avec les variables sur les types primitifs
— étape 2 : ajouter le calcul du type et de la valeur des expressions calculables à la compilation et remplissage de la TS avec les déclarations de constantes
— étape 3 : génération du code 3 adresses pour les expressions simples (hors types construits) et les affectations
— étape 4 : génération du code 3 adresses pour les structures de contrôle (boucles et conditionnelles)
— étape 5 : construction du graphe de flot de controle et optimisation (réduction des sous-expressions communes)
— étape 6 : ajout des types construits intervalles et tableaux
— étape 7 : ajout des types construits énumérations
— étape 8 : ajout des types construits structures
— étape 9 : ajout des types construits intervalles et tableaux
— étape 10 : ajout de la génération du code assembleur

L'ordre des étapes 1 à 5 est important pour partir sur de bonnes bases. Les étapes 6 à 9 quant à elles peuvent être faites dans un ordre différent.

7.2 La table des identificateurs

C'est la table qui associe un numéro unique à chaque identificateur (que ce soit un identificateur du programme source ou un identificateur généré par le compilateur).

Le langage pascal ne différenciant pas les caractères miniscules et majuscules, les chaines définissant les noms des identificateurs seront converties en minuscules.

Le numéro unique sera représenté par un **entier non signé**. La numérotation commencera à **1** (la valeur 0 étant réservée).

Vous devez disposer, au minimum, des fonctions (ou méthodes selon que vous fassiez du C ou du C++) :

```
/*
  ajoute un identificateur a la table
  renvoie le numero unique associe
  si il est deja present, ne fait que renvoyer son numero
*/
unsigned int ajoutIdentificateur ( const char* );
/*
  recupere le nom associe a un numero unique
*/
const char* getNom ( const unsigned int );
/*
  affiche la table sur la sortie standard
```

```
*/
void afficherTableIdent ();
/*
  sauvegarde la table dans un fichier
  le nom du fichier est passe en argument
*/
void sauvegarderTableIdent ( const char* );
```

7.3 Les tables des symboles

C'est la table qui contient la description de la signification d'un identificateur dans un contexte donné. Les identificateurs sont décrits par leurs numéros uniques.

Les signification possibles sont : programme, fonction, procedure, type, constante, variable, argument, champs, valenum, temporaire, etiquette, chaine.

La signification

— champs fait référence aux champs d'une structure.

— valenum fait référence aux différentes valeurs d'une énumération.

— chaine fait référence aux chaines de caractères statiques définies dans le programme source.

Ainsi la table des symboles est une table associative faisant le lien entre un numéro unique et des données hétérogènes. Pour la mettre en oeuvre, vous avez différentes possibilités dont :

— utiliser l'héritage en C++, avec une classe mère **Symbole**. La table conserve des pointeurs sur cette classe mère mais qui sont initialisés en créant des instances des classes filles.

— utiliser des structures en C avec une structure **Symbole** définie par une union C.

Chaque table des symboles est associée à un contexte (défini par son nom) et connait le contexte parent (pointeur sur la table des symboles du contexte parent).

7.4 Le code 3 adresses

Les instructions 3 adresses sera décrite par les structures C suivantes (adaptable en classes si vous préférez faire du C++) :

```
typedef enum {
    OP_NONE,
    OP_CHAR,
    OP_INT,
    OP_BOOL,
    OP_FLOAT,
    OP_IDENT
} TypeOperande;
typedef struct {
    TypeOperande type;
    union {
```

```
        char valchar;
        byte valbool;
        short valint;
        float valfloat;
        unsigned int validentˆ;
    } valeur;
} Operande;
typedef struct {
    unsigned int label;
    unsigned int code;
    Operande x, y, z;
} Instruction;
typedef std::vector < Instruction > BlocInstruction;
```

7.5 L'analyse sémantique

Chapitre 8

Exemples

Sommaire

8.1 Hello World !

Le programme de base dans tout langage...

8.1.1 Code pascal

```
program helloworld;
begin
        writeln ('Hello world!');
end.
```

Listing 8.1 helloworld

8.1.2 Code assembleur

```
%include "asm/include/macros.mac"
%include "asm/include/write.mac"

global _main

; ====================================
section     .text
helloworld:
    subprogram.prologue
; writeln ('Hello world!');
    push dword __ch1
    subprogram.call writeString
    subprogram.call newLine
    subprogram.epilogue
; ------------------------------------
section     .bss
; decalage pour acceder aux arguments
    .args_size equ 0
; variables locales et temporaires
    .vars_size equ 0
    .temps_size equ 0
; ------------------------------------
section     .data
; variables globales
; chaines de caracteres statiques
    __ch1 db 12, "Hello world!"

; ====================================
section    .text
_main:
    program.prologue
    subprogram.call helloworld
    program.epilogue
```

Listing 8.2 helloworld.asm

8.2 Hello

Demande son nom à l'utilisateur pour lui dire boujour.

8.2.1 Code pascal

```
program hello;
var              name : String;
begin
         write ('Please tell me your name: ');
         readln (name);
         writeln ('Hello, ',name,'!');
end.
```

Listing 8.3 hello

8.2.2 Code assembleur

```
%include "asm/include/macros.mac"
%include "asm/include/write.mac"
%include "asm/include/read.mac"

global _main

; ====================================
section .text
hello:
    subprogram.prologue
; write ('Please tell me your name: ');
    push dword __ch1
    subprogram.call writeString
; readln (name);
    push dword name
    subprogram.call readString
; writeln ('Hello, ',name,'!');
    push dword __ch2
    subprogram.call writeString
    push dword name
    subprogram.call writeString
    byte.push '!'
    subprogram.call writeChar
    subprogram.call newLine
    subprogram.epilogue
; ------------------------------------
section .bss
; decalage pour acceder aux arguments
    .args_size equ  0
; variables locales et temporaires
    .vars_size equ 0
```

```
    .temps_size equ 0
; ------------------------------------
section .data
; variables globales
    name db 0
         times 255 db 0
; chaines de caracteres statiques
    __ch1 db 26, "Please tell me your name: "
    __ch2 db 7, "Hello, "

; ====================================
section .text
_main:
    program.prologue
    subprogram.call hello
    program.epilogue
```

Listing 8.4 hello.asm

8.3 Calcul de pgcd

Calcul du pgcd de deux nombres saisis au clavier. Il s'agit de la version récursive.

8.3.1 Code pascal

```
program calculPGCD;

var x, y, resultat : integer;

function pgdc ( a : integer, b : integer ) : integer;
begin
        if a = b then
                pgcd := a;
        else if a > b then
                pgcd := pgcd ( a - b, b );
        else
                pgcd := pgcd ( a, b - a );
end

begin
        write ( 'Saisir la premiere valeur: ' );
        read ( x );
        write ( 'Saisir la seconde valeur: ' );
        read ( y );
        resultat := pgcd ( x, y );
        writeln ( 'le pgcd de ', x, ' et de ', y, ' est ',
           resultat );
end.
```

Listing 8.5 pgcd

8.3.2 Code assembleur

```
%include "asm/include/macros.mac"
%include "asm/include/write.mac"
%include "asm/include/read.mac"

%include "asm/include/debugger.mac"

global _main

section .text
pgcd:
    subprogram.prologue
;   __temp000 = a = b
    mov byte [ebp - 1 - .__temp000], 0
    mov ax, [ebp+8+.a]
    cmp ax, [ebp+8+.b]
    ;mov bx, [ebp+8+.b]
    ;cmp ax, bx
    jnz .not001
    mov byte [ebp - 1 - .__temp000], 1
.not001:
;   si __temp000 aller a alors002
    cmp byte [ebp - 1 - .__temp000], 1
    jz  .alors002
;   aller a sinon002
    jmp .sinon002
.alors002:
;   renvoyer a
    mov ebx, dword [ ebp + 8 + .result ]
    mov ax, word [ ebp + 8 + .a ]
    mov word [ebx], ax
;   aller a finsi002
    jmp .finsi002
.sinon002:
;   __temp001 = a > b
    mov byte [ebp - 1 - .__temp001], 0
    mov ax, [ebp+8+.a]
    mov bx, [ebp+8+.b]
    cmp ax, bx
    jle .not002
    mov byte [ebp - 1 - .__temp001], 1
.not002:
;   si __temp001 aller a alors001
    cmp byte [ebp - 1 - .__temp001], 1
    jz  .alors001
;   aller a sinon001
    jmp .sinon001
.alors001:
;   empiler & __temp3
    mov eax, ebp
    add eax, - 1 - .__temp003
```

```
    push                eax
;   empiler b
    word.push [ ebp + 8 + .b ]
;   __temp2 = a - b
    mov ax, word [ ebp + 8 + .a ]
    sub ax, word [ ebp + 8 + .b ]
    mov word [ ebp - 1 - .__temp002 ], ax
;   empiler __temp2
    word.push [ ebp - 1 - .__temp002 ]
;   appeler pgcd
    subprogram.call     pgcd
;   renvoyer __temp3
    mov ebx, dword [ ebp + 8 + .result ]
    mov ax, word [ ebp - 1 - .__temp003 ]
    mov word [ebx], ax
;   aller a finsin001
    jmp .finsi001
.sinon001:
;   empiler & __temp5
    mov eax, ebp
    add eax, - 1 - .__temp005
    push                eax
;   __temp4 = b - a
    mov ax, word [ ebp + 8 + .b ]
    sub ax, word [ ebp + 8 + .a ]
    mov word [ ebp - 1 - .__temp004 ], ax
;   empiler __temp4
    word.push [ ebp - 1 - .__temp004 ]
;   empiler a
    word.push [ ebp + 8 + .a ]
;   appeler pgcd
    subprogram.call     pgcd
;   renvoyer __temp5
    mov ebx, dword [ ebp + 8 + .result ]
    mov ax, word [ ebp - 1 - .__temp005 ]
    mov word [ebx], ax
.finsi001:
.finsi002:
    subprogram.epilogue
; -----------------------------------
section .bss
; decalage pour acceder aux arguments
.args_size      equ 4 + 4
.a              equ 0
.b              equ 2
.result         equ 4
; variables locales et temporaires
.vars_size      equ 0
.temps_size     equ 10
.__temp000      equ 0
.__temp001      equ 1
.__temp002      equ 2
```

```
.__temp003      equ 4
.__temp004      equ 6
.__temp005      equ 8
; -----------------------------------

; ===================================
section .text
calculpgcd:
    subprogram.prologue
;   ecrire ( ... )
    push dword __ch1
    subprogram.call writeString
;   lire ( x )
    push dword x
    subprogram.call readInt
;   ecrire ( ... )
    push dword __ch2
    subprogram.call writeString
;   lire y
    push dword y
    subprogram.call readInt
;   empiler & __temp6
    mov eax, ebp
    add eax, - 1 - .__temp006
    push eax
;   empiler y
    word.push [ y ]
;   empiler x
    word.push [ x ]
;   appeler pgcd
    subprogram.call pgcd
;   resultat = __temp6
    mov ax, word [ ebp - 1 - .__temp006 ]
    mov word [ resultat ], ax
;   ecrire ( ... )
    push dword __ch3
    subprogram.call writeString
;   ecrire ( ... )
    word.push [x]
    subprogram.call writeInt
;   ecrire ( ... )
    push dword __ch4
    subprogram.call writeString
;   ecrire ( ... )
    word.push [y]
    subprogram.call writeInt
;   ecrire ( ... )
    push dword __ch5
    subprogram.call writeString
;   ecrire ( ... )
    word.push [resultat]
    subprogram.call writeInt
```

```
;   sautdeligne
    subprogram.call newLine
;   fin
    subprogram.epilogue
; -------------------------------------
section .bss
; decalage pour acceder aux arguments
.args_size      equ 0
; variables locales et temporaires
.vars_size      equ 0
.temps_size     equ 2
.__temp006      equ 0
; -------------------------------------
section .data
; variables globales
x               dw 0
y               dw 0
resultat        dw 0
; chaines de caracteres statiques
__ch1           db 27, "Saisir la premiere valeur: "
__ch2           db 26, "Saisir la seconde valeur: "
__ch3           db 11, "le pgcd de "
__ch4           db 7, " et de "
__ch5           db 5, " est "

; =====================================
section .text
_main:
    program.prologue
    subprogram.call calculpgcd
    program.epilogue
```

Listing 8.6 pgcd.asm

8.4 Macros utiles

```
;-------------------------------------
; Macros de gestion de la pile
;   - (des)empilement d'un octet
;     byte.push arg
;     byte.pop arg
;   - (des)empilement d'un mot
;     word.push arg
;     word.pop arg
;-------------------------------------

%macro byte.push 1
    sub esp, 1
    push eax
```

```
    mov al, %1
    mov byte [esp+4], al
    pop eax
%endmacro

%macro byte.pop 1
%if %1 = al
    mov al, byte [esp]
%else
    push eax
    mov al, byte [esp+4]
    mov %1, al
    pop eax
%endif
    add esp, 1
%endmacro

%macro word.push 1
    sub esp, 2
    push eax
    mov ax, %1
    mov word [esp+4], ax
    pop eax
%endmacro

%macro word.pop 1
%if %1 = ax
    mov ax, word [esp]
%else
    push eax
    mov ax, word [esp+4]
    mov %1, ax
    pop eax
%endif
    add esp, 2
%endmacro

;--------------------------------------
; Macros de gestion du programme
;    - prologue d'entrée
;      program.prologue
;    - épilogue de sortie
;      program.epilogue
;--------------------------------------

%macro program.prologue 0
    pop eax
    pop ebx
    mov ebp, esp
    and esp, 0xFFFFFFF0
.begin:
%endmacro
```

```
%macro program.epilogue 0
.end:
    mov eax, 1
    push dword 0
    sub esp, 4
    int 0x80
%endmacro

;--------------------------------------
; Macros de gestion des sous-programme
;    - prologue d'entrée
;      subprogram.prologue arg
;    - épilogue de sortie
;      subprogram.epilogue arg
;    - appel
;      subprogram.call arg
;--------------------------------------

%macro subprogram.prologue 0-1 0
    enter .vars_size + .temps_size, 0
    pushad
%if %1 = 1
    fsave [.fpu.state]
%endif
.begin:
%endmacro

%macro subprogram.epilogue 0-1 0
.end:
%if %1 = 1
    frstor [.fpu.state]
%endif
    popad
    leave
    ret
%endmacro

%macro subprogram.call 1
    call %1
    add esp, %1.args_size
%endmacro

;--------------------------------------
; Macros d'encapsulation de la libc
;      clib_prolog
;      clib_epilog
;--------------------------------------

%macro clib_prolog 1
    mov ebx, esp
    and esp, 0xFFFFFFF0
```

```
    sub esp, 12
    push ebx
    sub esp, %1
%endmacro

%macro clib_epilog 1
    add esp, %1
    pop ebx
    mov esp, ebx
%endmacro
```

Listing 8.7 include/macros.mac