COMPILATEUR

JULIEN GIRAUD – SON-MICHEL DINH
UNIVERSITE LYON 1

Table des matières

Historique 3

Structure et fonctionnement 4

Lien avec les interpréteurs 8

Le problème de l'amorçage (bootstrap) 8

Compilateur simple passe et multi passe 9

Compilateur de compilateur 10

Qualité 10

Chaîne de compilation 10

Compilation croisée 11

Autres compilations 11

Byte code ou code octet 11

Exemples 11

Etudions le langage Pascal 12

Présentation et histoire 13

Les fichiers sources 14

Le Turbo Pascal 16

Un exemple de code : Hello World 17

Delphi et la programmation fenêtrée 19

Un **compilateur** est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage informatique (le langage cible).

Pour qu'il puisse être exploité par la machine, le compilateur traduit le code source, écrit dans un langage de haut niveau d'abstraction, facilement compréhensible par l'humain, vers un langage de plus bas niveau, un langage d'assemblage ou langage machine. Dans le cas de langage semi-compilé (ou semi-interprété), le code source est traduit en un langage intermédiaire, sous forme binaire (code objet ou bytecode), avant d'être lui-même interprété ou compilé.

Inversement, un programme qui traduit un langage de bas niveau vers un langage de plus haut niveau est un décompilateur.

Un compilateur effectue les opérations suivantes : analyse lexicale, pré-traitement (préprocesseur), décomposition analytique (parsing), analyse sémantique, génération de code et optimisation de code.

Quand le programme compilé (code objet) peut être exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée.

La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable.

Historique

Les logiciels des premiers ordinateurs étaient écrit en langage assembleur. Les langages de programmation de plus haut niveau (dans les couches d'abstraction) n'ont été inventés que lorsque les avantages apportés par la possibilité de réutiliser le logiciel sur différents types de processeurs devenaient plus importants que le coût de l'écriture d'un compilateur. La capacité de mémoire très limitée des premiers ordinateurs a également posé plusieurs problèmes techniques dans le développement des compilateurs.

Vers la fin des années 1950, des langages de programmation indépendants des machines font pour la première fois leur apparition. Par la suite, plusieurs compilateurs expérimentaux sont développés. Le premier compilateur, A-0 System - pour le langage A-0 - est écrit par Grace Hopper, en 1952. L'équipe FORTRAN dirigé par John Backus d'IBM est considéré comme étant le développeur du premier compilateur complet, en 1957. COBOL est le premier langage à être compilé sur plusieurs architectures, en 1960.

Dans plusieurs domaines d'application, l'idée d'utiliser un langage de plus haut niveau d'abstraction s'est rapidement répandu. Avec l'augmentation des fonctionnalités supportées par les langages de programmation plus récents et la complexité croissante de l'architecture des ordinateurs, les compilateurs se sont de plus en plus complexifiés.

En 1962, le premier compilateur « auto-hébergé » - capable de compiler son propre code source en langage de haut niveau - est créé, pour le LISP, par Tim Hart et Mike Levin au Massachusetts Institute of Technology (MIT). À partir des années 1970, il est devenu très courant de développer un compilateur dans le langage qu'il doit compiler, faisant du Pascal et du C des langages de développement très populaires.

Structure et fonctionnement

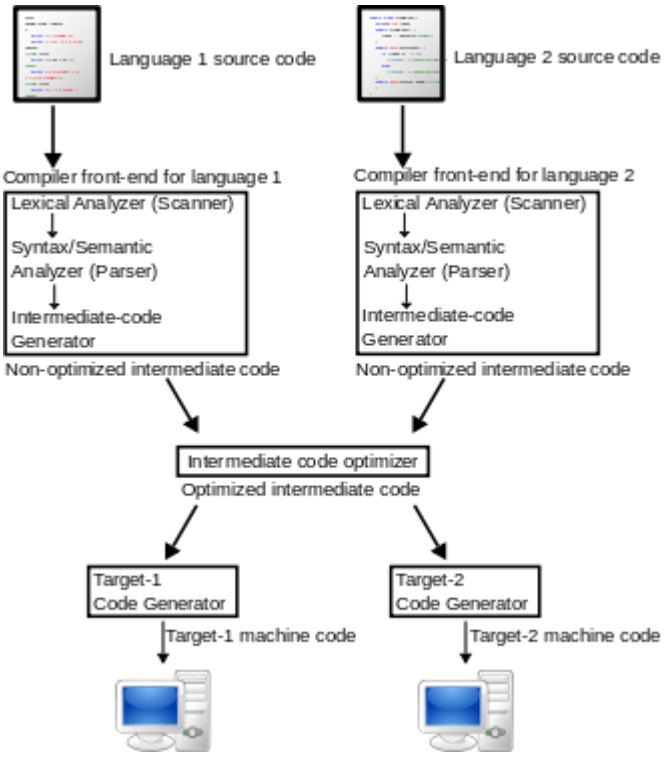


Schéma de compilation multi-source multi-cible

La tâche principale d'un compilateur est de produire un code objet correct qui s'exécutera sur un ordinateur. La plupart des compilateurs permettent d'optimiser le code, c'est-à-dire qu'ils vont chercher à améliorer la vitesse d'exécution, ou réduire l'occupation mémoire du programme.

En général, le langage source est « de plus haut niveau » que le langage cible, c'est-à-dire qu'il présente un niveau d'abstraction supérieur. De plus, le code source du programme est généralement réparti dans plusieurs fichiers.

Un compilateur fonctionne par analyse-synthèse : au lieu de remplacer chaque construction du langage source par une suite équivalente de constructions du langage cible, il commence par analyser le texte source pour en construire une représentation intermédiaire qu'il traduit à son tour en langage cible.

On sépare le compilateur en au moins deux parties : une partie avant (ou frontale), parfois appelée « souche », qui lit le texte source et produit la représentation intermédiaire ; et une partie arrière (ou finale), qui parcourt cette représentation pour produire le texte cible. Dans un compilateur idéal, la partie avant est indépendante du langage cible, tandis que la partie arrière est indépendante du langage source. Certains compilateurs effectuent des traitements substantiels sur la partie intermédiaire, devenant une partie centrale à part entière, indépendante à la fois du langage source et de la machine cible. On peut ainsi écrire des compilateurs pour toute une gamme de langages et d'architectures en partageant la partie

centrale, à laquelle on attache une partie avant par langage et une partie arrière par architecture.

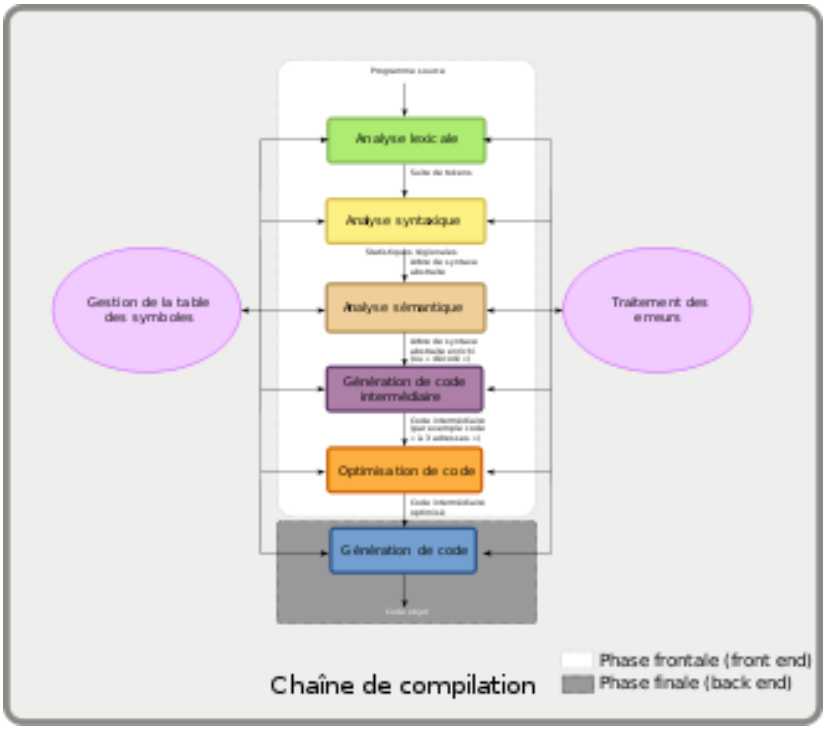


Schéma d'une chaîne de compilation classique

Les étapes de la compilation incluent :

- le prétraitement, nécessaire pour certaines langues comme C, qui prend en charge la substitution de macro et de la compilation conditionnelle.

  Généralement, la phase de prétraitement se produit avant l'analyse syntaxique ou sémantique ; par exemple dans le cas de C, le préprocesseur manipule les symboles lexicaux plutôt que des formes syntaxiques.

- l'analyse lexicale, qui découpe le code source en petits morceaux appelés jetons (tokens).

  Chaque jeton est une unité atomique unique de la langue (unités lexicales ou lexèmes), par exemple un mot-clé, un identifiant ou un symbole. La syntaxe de jeton est généralement un langage régulier, donc un automate à états finis construits sur une expression régulière peut être utilisé pour le reconnaître.

  Cette phase est aussi appelée *à balayage* ou *lexing* ; le logiciel qui effectue une analyse lexicale est appelé un analyseur lexical ou un scanner. Deux exemples classiques : lex et flex.

- l'analyse syntaxique implique l'analyse de la séquence jeton pour identifier la structure syntaxique du programme.

  Cette phase s'appuie généralement sur la construction d'un arbre d'analyse ; on remplace la séquence linéaire des jetons par une structure en arbre construite selon la grammaire formelle qui définit la syntaxe du langage. Par exemple, une condition est toujours suivie d'un test logique (égalité, comparaison...). L'arbre d'analyse est souvent modifié et amélioré au fur et à mesure de la compilation. Yacc et GNU Bison sont les analyseurs syntaxiques les plus utilisés.

- l'analyse sémantique est la phase durant laquelle le compilateur ajoute des informations sémantiques à l'arbre d'analyse et construit la table des symboles.

Cette phase vérifie le type (vérification des erreurs de type), ou l'objet de liaison (associant variables et références de fonction avec leurs définitions), ou une tâche définie (toutes les variables locales doivent être initialisées avant utilisation), peut émettre des avertissements, ou rejeter des programmes incorrects.

L'analyse sémantique nécessite habituellement un arbre d'analyse complet, ce qui signifie que cette phase fait suite à la phase d'analyse syntaxique, et précède logiquement la phase de génération de code ; mais il est possible de replier ces phases en une seule passe.

- la transformation du code source en code intermédiaire ;
- l'application de techniques d'optimisation sur le code intermédiaire : c'est-à-dire rendre le programme « meilleur » selon son usage (voir *infra*).
- la génération de code avec l'allocation de registres et la traduction du code intermédiaire en code objet, avec éventuellement l'insertion de données de débogage et d'analyse de l'exécution ;
- et finalement l'édition des liens.

L'analyse lexicale, syntaxique et sémantique, le passage par un langage intermédiaire et l'optimisation forment la partie frontale. La génération de code et l'édition de liens constituent la partie finale.

Ces différentes étapes font que les compilateurs sont toujours l'objet de recherches.

Lien avec les interpréteurs

L'implémentation (réalisation concrète) d'un langage de programmation peut être interprétée ou compilée. Cette réalisation est un compilateur ou un interpréteur, et un langage de programmation peut avoir une implémentation compilée, et une autre interprétée.

Le problème de l'amorçage (bootstrap)

Article détaillé : Bootstrap (compilateur).

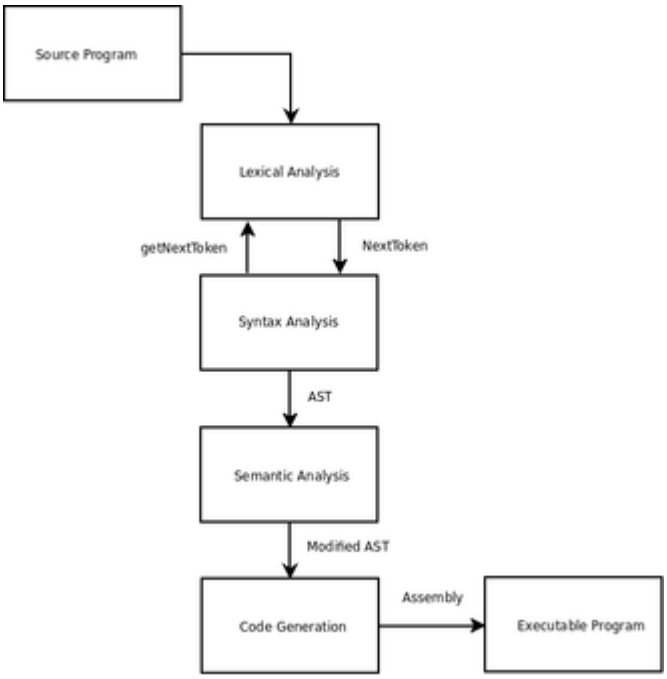
Les premiers compilateurs ont été écrits directement en langage assembleur, un langage symbolique élémentaire correspondant aux instructions du processeur cible et quelques structures de contrôle légèrement plus évoluées. Ce langage symbolique doit être assemblé (et non compilé) et lié pour obtenir une version exécutable. En raison de sa simplicité, un programme simple suffit à le convertir en instructions machines.

Les compilateurs actuels sont généralement écrits dans le langage qu'ils doivent compiler ; par exemple un compilateur C est écrit en C, SmallTalk en SmallTalk, Lisp en Lisp, etc. Dans la réalisation d'un compilateur, une étape décisive est franchie lorsque le compilateur pour le langage X est suffisamment complet pour se compiler lui-même : il ne dépend alors plus d'un autre langage (même de l'assembleur) pour être produit.

Il est complexe de détecter qu'un compilateur bug. Par exemple, si un compilateur C comporte un bug, les programmeurs en langage C auront naturellement tendance à mettre en cause leur propre code source, non pas le compilateur. Pire, si ce compilateur buggé (version V1) compile un compilateur (version V2) non buggé, l'exécutable compilé (par V1) du compilateur V2 pourrait être buggé. Pourtant son code source est bon. Le bootstrap oblige donc les programmeurs de compilateurs à contourner les bugs des compilateurs existants.

Compilateur simple passe et multi passe

La classification des compilateurs par nombre de passes a pour origine le manque de ressources matérielles des ordinateurs. La compilation est un processus couteux et les premiers ordinateurs n'avaient pas assez de mémoire pour contenir un programme devant faire ce travail. Les compilateurs ont donc été divisés en sous programmes qui font chacun une lecture de la source pour accomplir les différentes phases d'analyse lexicale, d'analyse syntaxique et d'analyse sémantique.



Compilateur multi-passes typique

La capacité de combiner le tout en un seul passage a été considérée comme un avantage car elle simplifie la tâche d'écriture d'un compilateur et il compile généralement plus rapidement qu'un compilateur multi passe. Ainsi, suivant les ressources limitées des premiers systèmes, de nombreux langages ont été spécifiquement conçus afin qu'ils puissent être compilés en un seul passage (par exemple, le langage Pascal).

Dans certains cas, la conception d'une fonctionnalité de langage a besoin d'un compilateur pour effectuer plus d'une passe sur la source. Par exemple, considérons une déclaration figurant à la ligne 20 de la source qui affecte la traduction d'une déclaration figurant à la ligne 10. Dans ce cas, la première passe doit recueillir des renseignements sur les déclarations figurant après les déclarations qu'ils affectent, avec la traduction proprement dite qui s'effectue lors d'un passage ultérieur.

L'inconvénient de la compilation en un seul passage est qu'il n'est pas possible d'exécuter la plupart des optimisations sophistiquées nécessaires pour générer du code de haute qualité. Il peut être difficile de dénombrer exactement le nombre de passes qu'un compilateur optimisant effectue.

Le fractionnement d'un compilateur en petits programmes est une technique utilisée par les chercheurs intéressés à produire des compilateurs performants. Prouver la justesse d'une série de petits programmes nécessite souvent moins d'effort que de prouver la justesse d'un plus grand programme unique équivalent.

Compilateur de compilateur

Article détaillé : Compilateur de compilateur.

Un compilateur de compilateur est un programme qui peut générer une, voire toutes les parties d'un compilateur.

Qualité

Article connexe : Optimisation de code.

Selon l'usage et la machine qui va exécuter un programme, on peut vouloir optimiser la vitesse d'exécution, l'occupation mémoire, la consommation d'énergie, la portabilité sur d'autres architectures, ou le temps de compilation.

Chaîne de compilation

Article détaillé : Chaîne de compilation.

Compilation croisée

La compilation croisée fait référence aux chaînes de compilation capables de traduire un code source en code objet dont l'architecture processeur diffère de celle où la compilation est effectuée. Ces chaînes sont principalement utilisés en informatique industrielle et dans les Systèmes embarqués.

Autres compilations

Byte code ou code octet

Certains compilateurs traduisent un langage source en langage machine virtuel, c'est-à-dire en un code (généralement binaire) exécuté par une machine virtuelle : un programme émulant les principales fonctionnalités d'un ordinateur. Le portage d'un programme ne requiert ainsi que le portage de la machine virtuelle. C'est le cas du compilateur Java, qui traduit du code Java en bytecode Java (code objet).

Exemples

Cette section est vide, insuffisamment détaillée ou incomplète. Votre aide est la bienvenue !

Si la plupart des compilateurs traduisent un code d'un langage de programmation vers un autre, ce n'est pas le cas de tous les compilateurs. Par exemple, le logiciel LaTeX compile un code écrit dans le langage de formatage de texte LaTeX, pour le convertir en un autre langage de présentation, par exemple DVI, HTML, PostScript...

Certains compilateurs traduisent, de façon incrémentale ou interactive, le programme source (tapé par l'utilisateur) en code machine. Par exemple, certaines implémentations de Common Lisp (comme SBCL) traduisent un bout de programme en code machine (en mémoire).

Les compilateurs à la volée (Just in time) traduisent une représentation intermédiaire en code machine, de manière progressive.

Etudions le langage Pascal

Pascal est un langage de programmation impératif qui, conçu pour l'enseignement, se caractérise par une syntaxe claire, rigoureuse et facilitant la structuration des programmes1.

Programmation Pascal

En dehors de la syntaxe et de sa rigueur, le langage Pascal possède de nombreux points communs avec le C (voir les pointeurs). Le langage Pascal de base était conçu à usage purement éducatif et était assez limité. Par exemple, les chaînes de caractères, absentes du langage d'origine, ont rapidement été intégrées2,3. Les développements qu'il a connus par la suite en ont fait un langage complet et efficace. Plus récemment, la généricité a été ajoutée dans Delphi 20094 et dans Free Pascal depuis la version 2.25,6.

Les implémentations actuelles de Pascal, utilisées hors du monde éducatif, sont des extensions telles que Turbo Pascal (mode texte), Pascal Objet (programmation objet), et Delphi (fenêtré). Il existe des versions libres comme Free Pascal et Lazarus (fenêtré). On peut programmer en Pascal sous DOS, Windows, Mac OS ou encore sous Linux/Unix ou Palm OS.

Le système d'exploitation des ordinateurs Apollo7, ainsi qu'une partie du système du Macintosh ont été écrits en Pascal. La première version d'Adobe Photoshop également8. Le

compilateur GCC a été développé par Richard Stallman à partir d'un compilateur du LLNL, qui était écrit en langage Pastel, une extension du langage Pascal9.

La syntaxe du langage a été adaptée à d'autres langages comme Ada, Modula-2 (puis Modula-3) ou Oberon.

Présentation et histoire

Le langage de programmation Pascal (dont le nom vient du mathématicien français Blaise Pascal) a été inventé par Niklaus Wirth dans les années 1970. Il a été conçu pour servir à l'enseignement de la programmation de manière rigoureuse mais simple, en réaction à la complexité de l'Algol 68. Le premier compilateur a été conçu sur un CDC 64001,10.

Ce langage est l'un de ceux qui ont servi à enseigner la programmation structurée. Le goto ou saut n'importe où dans le programme (dit « branchement ») est fortement déconseillé dans ce langage, le programme est un assemblage de procédures et de fonctions, dans lesquelles on peut utiliser des blocs conditionnels (if, case) et répétitifs (while, for, repeat) ayant chacun une entrée et une sortie afin de faciliter les contrôles, ce qui aboutit à des mises au point rapides et sûres.

Le langage est de plus fortement et statiquement typé, c'est-à-dire que toutes les variables doivent avoir un type défini au moment de la compilation. En revanche son manque de souplesse pour gérer les passages du type caractère au type chaîne de caractères est l'un de ses points faibles.

Il a largement pénétré le monde de l'éducation et de la recherche (universités), puis dans une moindre mesure celui de l'industrie.

Le compilateur P4 a été diffusé en source dans les universités, à un prix très réduit. Il générait du P-Code, un code pour une machine virtuelle. Les programmes Pascal étaient donc facilement portables sur une machine. Il suffisait d'écrire pour elle un interpréteur de P-Code.

Il y eut donc rapidement des portages sur 6502, 8080, Z80 et DEC PDP-11, les principaux microprocesseurs de l'époque.

Le compilateur UCSD Pascal, de l'université de Californie à San Diego, eut beaucoup de succès, notamment sur des machines comme l'Apple II qui furent très diffusées11.

Mais le coup de « turbo » sera donné par la société Borland, créée en 1983 qui commercialisa le compilateur Turbo Pascal pour un prix très modique (49 $ de l'époque alors que le compilateur Pascal Microsoft était à plus de 500 $ et ne possédait pas d'éditeur intégré ni de compilation in core). En fait, il y a aura bien un concurrent direct du Turbo Pascal chez Microsoft (Quick Pascal) mais il sera commercialisé bien trop tard pour inverser la tendance.

Des compilateurs ont été produits pour divers ordinateurs, notamment des fabricants Sun12, HP13, SGI14, CDC15, IBM16, Unisys17, Texas Instruments18.

Le Pascal a fait l'objet des normes ISO 7185 (1983)19 et ISO 10206 (1990)20.

Les fichiers sources

En général, on reconnaît un programme en Pascal par l'extension du fichier qui le contient : .pas, .p ou .pp, les deux dernières étant plus fréquentes sur les systèmes de type UNIX. L'utilisation d'environnements fenêtrés (Delphi et Lazarus) a entrainé l'apparition de nouvelles extensions.

Le code source est organisé suivant différentes possibilités :

- Un programme principal commence par le mot-clé program, facultatif dans certaines implémentations. Il est compilé et lié avec les unités qu'il utilise en un fichier exécutable. Cette phase de compilation est le cas le plus fréquent dans les compilateurs récents. Au contraire, le Pascal UCSD, par exemple, produisait du bytecode.

- Une bibliothèque dynamique commence par le mot-clé library. Elle est compilée en un fichier portant l'extension .dll sous Windows, .so sous UNIX. Les bibliothèques dynamiques sont apparues dans le langage Pascal en même temps que Delphi.

- Un fichier unité commence par le mot-clé unit. Il est inclus dans des programmes ou des bibliothèques, et peut être compilé en un code objet standard (extension .o ou .obj) ou dans un format particulier, tel que .dcu pour Delphi.

Avant l'apparition des bibliothèques, Turbo Pascal permettait d'utiliser des Overlay (en), technique habituelle sous DOS pour les programmes de grande taille21. Il s'agissait de fichiers séparés du fichier exécutable principal et qui pouvaient être chargés ponctuellement.

Il est possible d'inclure du code dans un programme autrement qu'en écrivant une unité et ce en faisant simplement un include, c'est-à-dire en indiquant au compilateur d'inclure le texte d'un fichier dans un programme avec la directive $I :

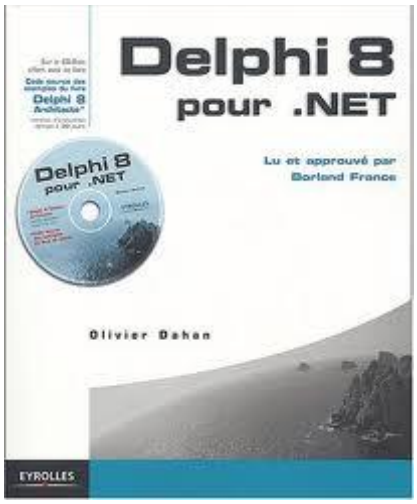
```
program test;
uses Crt, Dos; { utilisation habituelle des unités }
{$I outil.inc} { utilisation directe de code externe }
begin
     { utiliser les fonctions du fichier outil.inc }
end.
```

Toutefois, cette méthode n'est pas recommandée au regard des inconvénients qu'elle présente - notamment si un identificateur est déclaré plusieurs fois dans des fichiers différents - et du manque de contrôle du compilateur sur ce type de fichier.

Le Turbo Pascal

Le logiciel Turbo Pascal a été écrit par Anders Hejlsberg22,23 : il s'est appelé auparavant Compass Pascal, puis Poly Pascal. Très compact (12 kilo-octets) et très rapide car travaillant essentiellement en mémoire vive, il compilait en une passe et produisait du code machine x86 sous DOS et non pas du bytecode. Il était livré avec un environnement complet (un éditeur de texte et une aide en ligne, innovation à l'époque, particulièrement compacte grâce à un système de *substitution*).

Chaque nouvelle version de Turbo Pascal a apporté son lot d'innovations, comblant en partie des lacunes du langage original. C'est ainsi qu'en 1987 la version 4 apparaît avec un véritable environnement de développement intégré, en 1989 la version 5.5 introduit les objets24, en 1990 la version 6 permet la programmation de fenêtres dans la console DOS, par le biais de la Turbo Vision, apportant les prémisses de la programmation événementielle. Enfin, en 1992 sort la version 7 pour DOS, qui sera la dernière, ainsi qu'une version pour Windows, rapidement supplantée par Delphi.



Delphi 8 pour .NET - Eyrolles

Un exemple de code : Hello World

```
program HelloWorld(output);
begin
writeln ('Hello World');
readln;
end.
```

Le paramètre Output qui suit le nom du programme est aujourd'hui facultatif (il n'était obligatoire qu'avec les premières versions des implémentations Pascal). De nos jours, il est absent la plupart du temps.

Contrairement au C, le Pascal n'est pas sensible à la casse, c'est-à-dire que les mots réservés (comme begin) ou les identificateurs (comme write ou la variable a) peuvent être indifféremment écrits en majuscules ou en minuscules.

Toujours contrairement au C, les déclarations (var dans l'exemple ci-dessus) se font dans une partie clairement séparée du code. Les déclarations locales sont faites en début de procédure ou de fonction, les déclarations globales, elles, étant faites n'importe où avant le programme principal. Ceci ajoute de la clarté au langage au prix d'un certain manque de souplesse. On ne peut pas déclarer de variable au beau milieu d'une fonction. Notons qu'en Pascal, les

déclarations doivent précéder toute utilisation ; il est notamment interdit d'utiliser une procédure ou une fonction qui n'a pas encore été déclarée. Il est possible de contourner ce problème à l'aide du mot-clé *forward*.

Enfin, la distinction entre procédures et fonctions permet d'éviter certaines erreurs de programmation - par défaut, car une directive de compilation de Delphi permet d'activer une syntaxe « étendue » qui offre le même laxisme que le C.

Delphi et la programmation fenêtrée

En 1995, pour contrecarrer Microsoft et la programmation visuelle du Visual Basic, Borland sort Delphi. Contrairement à VB qui produit du p-code, Delphi produit du code machine, plus rapide. On voit également apparaître la bibliothèque VCL servant d'interface aux bibliothèques système (en) de Windows, facilitant grandement le développement.

Au début des années 2000, Borland produit Kylix, l'équivalent de Delphi pour le monde Linux, qui n'aura pas un grand succès. Au début des années 2010, Embarcadero, qui a repris les activités d'outils de développement de Borland, produit Delphi XE, dont les versions récentes sont compatibles avec Windows, OS X et iOS.

Lazarus, une version open-source légèrement différente, permet de compiler sur différentes plateformes.

Exemple de source Lazarus

Voici un exemple de fichier Pascal associé à une fenêtre (Form1: TForm1) contenant un label (un texte). Ce fichier est généré automatiquement et sert de structure de base pour programmer. C'est-à-dire qu'on peut le modifier, le compléter etc. Les petits points sont des repères lors de l'édition. Ils sont invisibles à l'exécution. Le source et les fenêtres Delphi sont très semblables.

```
unit Unit1;
     {$mode objfpc}{$H+}
interface
uses
     Classes, SysUtils, LResources, Forms, Controls,
Graphics, Dialogs, StdCtrls;
type
     { TForm1 }
     TForm1 = class (TForm)
          Label1: TLabel; { le label "Hello world!" posé
     sur la fenêtre }
     private
          { private declarations }
     public
          { public declarations }
     end;
var
     Form1: TForm1;
implementation
initialization
     {$I unit1.lrs}
end.
```

Petite explication : la directive `{$I unit1.lrs}` permet de lier la classe TForm1, décrivant une fenêtre, au fichier de ressource unit1.lrs qui contient le design de la fenêtre. Avec Lazarus, le fichier lrs est un fichier intermédiaire créé automatiquement par le compilateur à partir des informations du fichier lfm et des directives de compilation (notamment la possibilité de choisir la bibliothèque de widget). Avec Delphi, la directive équivalente aurait été {$R unit1.dfm} et il n'y a pas de fichier intermédiaire. Par ailleurs, elle aurait été placée dans la partie interface.