M1IF
Projet Compilation

Paul Feautrier

21 janvier 2009

1 Introduction

L'objectif du projet est la réalisation d'un compilateur pour le langage Pascal- (voir `www.cs.uleth.ca/~hossain/cs4600a/cs4600W2Outline.pdf`). Ce langage est un sous-ensemble de Pascal. Les différences essentielles sont :

- Les seuls types de base sont les entiers et les booléens. Il n'y a ni réels ni caractères.
- Les types construits doivent être nommés à l'aide d'une construction `type`.
- Dans une déclaration, on doit toujours utiliser un nom de type et non pas le type lui-même.
- les seules structures de données sont les tableaux (`array`) et les structures (`record`).
- Il n'y a pas de fonction, mais seulement des procédures. Ceci n'est pas en fait une véritable restriction.
- L'opérateur `new` n'existe pas. Toutes les données sont logées dans la pile d'exécution.

Les aspects essentiels de Pascal- sur lesquels votre compilateur devra se concentrer sont :

- Le contrôle des types.
- La gestion de la pile d'exécution, en particulier en présence de procédures imbriquées.

La cible de la compilation est la machine RISC en FPGA objet du TD9 du cours d'architecture (voir `perso.ens-lyon.fr/jeremie.detrey/05-archi`). Comme cette machine est très simple, il est possible de générer du code binaire sans utiliser d'assembleur.

Le langage d'implémentation du projet est laissé libre. On notera cependant que l'utilisation de générateurs d'analyseurs syntaxiques du type de Lex

et Yacc est presque obligatoire. Ceci milite soit en faveur de C/C++, soit en faveur d'OCaml [LDG+00]. Il existe probablement des versions de Lex et Yacc en Java, mais elles n'ont pas l'autorité des versions C et OCaml. Enfin, un compilateur manipule nécessairement des structures de données dynamiques ; l'existence d'un ramasse-miette (*garbage collector*) est une aide appréciable, ce qui favorise Java et OCaml. Au total, le langage conseillé est OCaml, mais vous êtes libres de ne pas suivre cette recommandation (à votre grand dam).

Le projet sera matérialisé sous la forme d'un compilateur (source et binaire) et de programmes tests dont on vérifiera la bonne compilation et la bonne exécution[1]. Le tout sera accompagné d'un rapport d'implémentation et d'un guide d'utilisation. Chaque projet donnera lieu à une soutenance.

Le projet doit être réalisé par groupes de deux à trois personnes au maximum. Il n'est pas recommandé de travailler seul.

Certaines parties de ce travail se réfèrent aux TD de Compilation de Florent de Dinechin [dD02] ; d'autres se réfèrent aux TD d'architecture de Jérémie Detrey.

Ce document est accessible à travers la page Web `perso.ens-lyon.fr/Paul.Feautrier` et sera régulièrement mis à jour. Vous y trouverez également la grammaire de Pascal-. D'autres informations (par exemple des compléments de cours), seront ajoutées en temps réél.

2 Le langage

Le langage Pascal- pour lequel il faut réaliser un compilateur est un sous-ensemble de Pascal. En particulier, en cas de doute sur la correction d'un programme, vous pouvez toujours vous référer à une définition canonique de Pascal (par exemple [Gro92]) ou même avoir recours au compilateur standard (par exemple `gpc`).

La grammaire de Pascal- a été extraite du site `www.cs.uleth.ca/~hossain/cs4600a/cs4600W2Outline.pdf`. Vous la trouverez telle quelle dans le fichier `implementation.tar`. Cette grammaire est écrite en BNF avec les conventions suivantes :

- Les non-terminaux sont des identificateurs commençant par une lettre majuscule (attention si vous utilisez Ocamlyacc).
- Les terminaux sont écrits entre apostrophes (par exemple `'program'`).
- Les non-terminaux dont le nom est de la forme `xxxNameDef` et `xxxNameUse` représentent les identificateurs, c'est-à-dire les chaînes de caractères

[1] Il est recommandé que la promotion mette en commun ses programmes tests.

commençant par une lettre suivie de lettres ou de chiffres. La première forme indique qu'il s'agit de la première apparition de l'identificateur dans une portée lexicale (l'identificateur doit être *rangé* dans une table de symboles), et la deuxième forme qu'il s'agit d'une utilisation (l'identificateur doit être *trouvé* dans une table des symboles).
- Le non-terminal `Numeral` représente un nombre entier (suite de chiffres).
- Le non-terminal `Empty` représente la chaîne vide.
- La construction `[ syntagme ]` indique que la présence du syntagme est facultative.
- La construction `{ syntagme }` indique que le syntagme peut être répété un nombre arbitraire (y compris nul) de fois.

Ces règles ont peu de chances de convenir au générateur d'analyseur lexical et sémantique que vous allez choisir. C'est à vous de faire l'adaptation.

2.1 Sémantique

La sémantique est en général celle de Pascal.

2.1.1 Tableaux

Les tableaux peuvent avoir autant de dimensions que l'on veut. Chaque dimension est spécifiée à l'aide d'une borne inférieure et d'une borne supérieure qui peuvent avoir des valeurs quelconques mais constantes. En Pascal, avant tout accès à un tableau, on doit vérifier que les indices sont dans les bornes pour chaque dimension.

Les tableaux sont alloués dynamiquement sur la pile d'exécution par le prélude de la fonction qui les contient. Ils doivent être détruits par le postlude correspondant.

2.1.2 Variables entières

Pascal- traite les variables scalaires entières, qui peuvent figurer dans des indices, des bornes de boucles et des expressions arithmétiques entières. Les constantes entières ont la forme usuelle. Les entiers sont codés sur 16 bits, pour se conformer à la structure de la machine cible.

La création et la destruction des variables se font de la même façon que pour les tableaux.

2.1.3 Variables et expressions booléennes

Il existe des expressions booléennes, des variables et des constantes booléennes. Les expressions booléennes sont construites à partir d'expressions

entières, de variables et de constantes à l'aide des opérateurs de comparaison `<, >, <=, >=, =, <>` et des opérateurs booléens (`and, or, not`).

On remarquera que la grammaire n'interdit pas des expressions invalides, comme `false + 1`. C'est le contrôle de type qui doit détecter ces erreurs.

Un booléen peut être stocké dans un mot (16 bits, plus rapide) ou dans un bit (économie de mémoire). Dans la deuxième solution, l'accès est une opération complexe (masquages et décalages). A vous de choisir.

2.1.4 Instructions

Les instructions sont séparées les unes de autres par un point-virgule.

Instructions d'affectation La syntaxe est usuelle : `lhs := rhs`. Le membre gauche (lhs) doit être une référence à une variable. Le membre droit (rhs) doit être une expression du même type que le membre gauche.

Entrées et sorties Pour pouvoir vous assurer du bon fonctionnement d'un programme compilé, il est indispensable de pouvoir faire des lectures et des écritures. Plutôt qu'utiliser les instructions d'entrée / sortie de Pascal, assez complexes, vous définirez les procédures `readInt, readBool, writeInt` et `writeBool`. Vous devrez porter ces procédures à la connaissance du vérificateur de type, et prévoir leur implémentation directement en langage machine dans la bibliothèque d'exécution (voir plus loin Sect. 4.6).

Instructions conditionnelles Une instruction conditionnelle peut avoir l'une des deux formes :

```
if expression then statement
```

ou :

```
if expression then statement
else statement
```

On doit vérifier que le type de l'expression est bien booléen. On notera que si cette grammaire est soumise à Yacc ou à un générateur de la même famille, on obtient un *shift/reduce conflict* qui est correctement résolu.

Boucle `while`

```
while expression do statement
```

Le nombre d'itération d'une boucle `while` n'est pas fixé d'avance. La boucle se termine quand l'expression testée devient fausse.

Procédures Les procédures de Pascal peuvent utiliser deux modes de transmission des paramètres ([Ris] Sect. 6.3) :

- La transmission par valeur : le paramètre effectif est recopié sur la pile.
- La transmission par référence : c'est un pointeur qui est copié sur la pile.

Il est conseillé de toujours utiliser un mot (16 bits) pour ces recopies, quelque soit le type du paramètre.

Le programme En Pascal-, le programme principal peut contenir du code et des déclarations. Il doit typiquement engendrer le point d'entrée du programme objet.

2.1.5 Déclarations

Par rapport à Pascal, le mécanisme des déclarations a été simplifié (pour le compilateur, mais pas pour l'utilisateur). La déclaration d'une variable ne peut utiliser qu'un type prédéfini. Celui-ci est l'un des types de base `integer` ou `real`, ou a été défini antérieurement dans une déclaration de type. Noter en particulier comment les tableaux à plusieurs dimensions sont déclarés.

```
type row = array [1..10] of integer;
type matrix = array [1..10] of row;

var i : integer;
    M : matrix;
```

Une procédure a également un type et celui-ci doit être vérifié. Ceci revient à vérifier que les paramètres effectifs ont un type conforme.

C'est également à ce niveau que doivent être traitées les constantes : typage et préparation de la génération de code. En général, un compilateur range les constantes dans une zone de mémoire statique, par exemple juste après le code du programme. Il peut être intéressant d'éliminer les redondances.

2.2 Exemples

Vous trouverez de très nombreux exemples de programmes Pascal (à convertir, si possible, en Pascal-) dans le livre de Grogono [Gro92].

3 Organisation de l'exécution

Avant toute tentative de compilation, vous devez décider de la forme que prendra le programme à l'exécution. Les aspects principaux sont, dans l'ordre d'importance :

- La gestion de la mémoire. Les particularités de la machine FPGA font que le programme doit débuter à l'adresse `0xA00`, et ne pas s'étendre au dela de l'adresse `0xFF0`. On propose d'utiliser cet espace de la façon suivante :
  - la bibliothèque d'exécution (et en particulier le code de démarrage) à partir de l'adresse `A000`.
  - Le programme compilé.
  - La pile d'exécution commence à l'adresse `0xFEFF` et s'étend vers le bas.
- La faible taille de la zone d'adressage immédiat des instructions (6 bits) complique la manipulation des adresses. Heureusement, cet inconvénient est compensé par le grand nombre de registres. On pourra donc réserver des registres pour accéder à la pile d'exécution, à une zone de constantes, aux points d'entrée des fonctions, etc.
- La mécanique d'utilisation des fonctions et la gestion de la pile d'exécution. On distingue en général la séquence d'appel (exécutée par l'appelant), le prélude (exécuté au point d'entrée de la fonction), le postlude (exécuté quand la fonction a terminé son travail) et la séquence de retour (exécutée par l'appelant). Pour la conception de ces séquences, on s'impose en général les règles suivantes :
  - après la séquence de retour, la pile d'exécution doit avoir la même hauteur qu'avant la séquence d'appel.
  - On s'attachera à ce que la séquence de retour défasse ce qu'à fait la séquence d'appel, et de même pour le prélude et le postlude.
- Vous devez le plus tôt possible spécifier la structure de la pile d'exécution (voir [Ris] Sect. 6). Voici quelques conseils dans ce but :
  - En général, on réserve deux registres pour l'adresse de l'enregistrement d'activation courant et pour le pointeur de sommet de pile.
  - Au moment de la compilation d'une procédure, le compilateur doit établir la carte de l'enregistrement d'activation et préparer l'accès aux variables locales et aux paramètres effectifs.
  - De plus, Pascal- autorise la définition de procédures locales. La règle de visibilité spécifie qu'un identificateur réfère la variable locale la plus proche en allant vers le fond de la pile[2]. Il est possible d'adresser

[2]Noter qu'en conséquence les variables locales d'une activation antérieure d'une procé-

une variable non locale en remontant la chaîne des enregistrements d'activation, mais c'est une perte de temps. La solution la plus courante est de créer une structure de donnée de taille fixe (pourquoi ?), le *display*, dans laquelle on enregistre les adresses des enregistrement d'activation accessibles. La gestion du display devra être prévue dans le prélude de chaque procédure. Il faudra réserver un registre pour mémoriser l'adresse du display.

- Vous devez choisir une politique de gestion des registres au moment des appels de procédures. Les compilateurs anciens sauvegardent tous les registres sur la pile au moment de l'appel. Sur les machines modernes, ou le nombre de registres est très élevé, cette opération prendrait trop de temps. On convient donc qu'au moment d'un appel, les registres ne contiennent rien d'indispensable. Cette décision a des incidences sur la génération du code et sur d'éventuelles optimisations.
- Enfin, n'oubliez que vous devez surveiller les dépassements de capacité de la pile.

4 Structure du compilateur

Pour l'écriture du compilateur, on propose d'utiliser le paradigme des transformations de programme. A l'issue de la phase d'analyse syntaxique, le programme est mis sous la forme d'une structure de données appelée représentation intermédiaire (ou représentation interne, RI). Une première phase permet de vérifier que le programme est conforme aux règles qui ne peuvent pas être exprimées dans le cadre d'une grammaire hors contexte (par exemple, les contraintes de type). Les phases suivantes modifient la représentation intermédiaire jusqu'à ce que le programme soit équivalent à un programme en assembleur. La dernière phase du compilateur consiste en la fabrication d'un fichier binaire *image* de la mémoire au début de l'exécution.

On s'attachera à bien séparer ce qui ne dépend que du langage source (Pascal-) et ce qui est lié à la machine cible. On tachera de confiner l'influence de la machine cible à la dernière étape du compilateur (sect. 4.5). Par exemple, dans l'étape (4.4.1), on pourra exprimer le résultat de l'expansion en "code trois-adresse" : `x = y + z` où `x, y` et `z` sont des registres. Ce n'est qu'à la génération de code que l'on traduira :

```
MTA [y]
ADD [z]
MTR [x]
```

dure récursive sont invisibles.

Les problèmes de vitesse de compilation sont secondaires dans ce projet. On prendra cependant garde que les temps de compilation ne soient pas prohibitifs.

Certains langages (dont OCaml et Java) possèdent des fonctions de sérialisation (*marshalling*) qui permettent de lire et d'écrire très simplement des structures de données complexes (par exemple la RI). Ceci permet de réaliser les phases du compilateur sous forme de programmes indépendants sur lesquels plusieurs personnes peuvent travailler en parallèle. Attention, aux dernières nouvelles, bien que Ocaml soit maintenant un langage à objets, il n'est pas possible de sérialiser des objets en OCaml.

4.1 La représentation intermédiaire

La RI est le cœur du compilateur. Les compilateurs anciens utilisent une RI de très bas niveau, où une grande partie des informations utiles ont été perdues (par exemple, la structure des boucles). Les compilateurs modernes, au contraire, utilisent comme RI l'arbre d'analyse grammaticale (*Abstract Syntax Tree* ou AST) du programme, tel qu'il est construit par la phase d'analyse syntaxique, auquel on adjoint une table des symboles.

La table des symboles associe à chaque identificateur du programme des informations de type et de taille. Ces informations peuvent être enrichies au fur et à mesure du déroulement de la compilation. Une table de symbole s'implémente en général à l'aide d'une table de hachage. Certains langages fournissent des outils de gestion de ce type de structure de données.

Le langage Pascal- a une structure hiérarchique, qui devra être reflétée dans la structure de la table des symboles [Ris].

Utiliser l'AST comme représentation intermédiaire n'est pas entièrement satisfaisant, pour les raisons suivantes :

- La grammaire hors contexte du langage n'en représente pas tous les aspects (par exemple les contraintes de type).
- Il est souhaitable que la même RI serve d'un bout à l'autre du compilateur. Or, le langage objet peut contenir des traits qui ne figurent pas dans le langage source (par exemple, les `goto`s).
- Enfin, dans l'approche transformationnelle, il faut pouvoir modifier la RI, et les modifications sont souvent locales. Suivant le langage utilisé (fonctionnel ou impératif) ceci peut être plus ou moins facile et efficace.

La définition détaillée de la RI est la première étape du projet et sans doute le premier chapitre du rapport d'implémentation. On partira de l'AST du programme, mais on essaiera d'anticiper les besoins de l'ensemble du compilateur. L'expérience montre cependant que au cours de l'élaboration du compilateur, la RI devra être retouchée. On étudiera les moyens de rendre ces

retouches aussi indolores que possible (approche objet, constructeurs, etc.). Une suggestion : il est possible en OCaml de construire une RI modifiable en utilisant l'attribut `mutable` dans un type *record* (voir [LDG+00] chap. 1).

L'état de la RI à un instant donné est une information fondamentale pour la mise au point du compilateur. On développera donc en même temps que la RI un imprimeur qui en présente le contenu de façon lisible.

4.2 Analyse syntaxique

Il s'agit d'une application standard de Lex et Yacc, ou d'autres outils analogues.

4.3 Analyse sémantique

Cette phase a plusieurs responsabilités :

- On peut retarder la constructions des tables de symboles à ce moment, ou l'effectuer au vol pendant l'analyse syntaxique (déconseillé).

  > En général, les débutants ont tendance à vouloir faire le plus de traitements possibles pendant l'analyse syntaxique (i.e. dans les actions de `yacc`). Cette tactique est déconseillée, car elle diminue la lisibilité du programme, et parce que l'ordre d'exécution des actions par Yacc est difficile à maîtriser.

- Vérification des types : tout identificateur doit être déclaré. On en déduit le type de chaque expression. On vérifie que dans chaque instruction d'affectation le membre gauche et le membre droit on le même type. On vérifie que les expressions utilisées pour les indices et les bornes de boucles sont entières. Le résultat de cette phase devra enrichir la RI. On notera que la vérification des types assure en particulier que les paramètres formels et effectifs sont en nombres égaux. On essaiera de rendre cette phase la plus générique possible (voir la Sect. 5.2).

4.4 Transformations du programme

4.4.1 Expansion des expressions

En langage machine, surtout s'il s'agit d'une machine RISC, chaque ligne de code ne peut éxécuter qu'une seule opération de calcul ou d'accès à la mémoire. En Pascal-, au contraire, une instruction peut être aussi complexe que l'on veut. Il est donc nécessaire d'écrire une phase d'expansion qui remplace une instruction Pascal- par autant (ou plus) d'instructions assembleur qu'il y a d'opérateurs dans le membre droit.

Cette opération nécessite l'introduction de variables temporaires. Une bonne idée est de donner à ces variables des noms de registres. L'algorithme d'expansion peut être une simple visite de l'arbre de syntaxe abstraite. Vous pouvez également utiliser des algorithmes plus sophistiqués : priorité à la sous-expression la plus complexe, utilisation de l'associativité, de la commutativité, voire de la distributivité. Si votre algorithme de génération des noms de registre n'est pas (trop) stupide, vous pourrez, dans une première étape, ne pas vous soucier du nombre de registres disponibles (il y en a 64 moins ceux que vous aurez réservé pour des besoins spéciaux).

Vous n'oublierez pas que parmi les opérations complexes qui figurent dans le code Pascal-, il y a les accès aux tableaux et les accès aux structures.

4.4.2 Construction des boucles et des instructions conditionnelles

Cette partie de la compilation n'est pas facile à écrire à partir de l'arbre de syntaxe abstrait. Une des possibilités est de construire une deuxième RI, le graphe de contrôle, dont les sommets sont des blocs d'instructions s'exécutant en séquence, et dont les arcs représentent les successions possibles. Par exemple, pour un test `IF p THEN s ELSE t`, `p`, `s` et `t` seront des sommets, et il y aura un arc de `p` vers `s` et un arc de `p` vers `t`. Il faudra d'autre part identifier l'instruction qui suit le test, et créer les arcs correspondants. L'écriture du code de contrôle est alors une simple visite de ce graphe.

Une autre solution est d'écrire des "méthodes" réalisant une fois pour toute l'extraction des informations indispensables à partir de l'AST : détermination des instructions qui suivent une instruction donnée, des instructions précédentes, marquage, etc.

4.5 Génération du code

La machine FPGA pose des problèmes particuliers de de génération de code, à cause d'une part du très petit nombre d'instructions, et d'autre part de la faiblesse de son mécanisme d'adressage. Pour chaque opération du langage, il faudra développer une séquence d'instructions qui la réalise. Par exemple, pour effectuer un changement de signe, il faut amener l'argument dans un registre, remettre l'accumulateur à zéro et faire une soustraction. On peut remettre l'accumulateur à zéro soit en chargeant la constante 0, soit à l'aide d'un XOR. On suggère de trouver un moyen de paramétrer ce processus de traduction, par exemple à l'aide d'une table ou d'un fichier séparé.

Le point le plus difficile est sans doute celui de l'introduction des constantes dans le code. Il existe pour cela 5 méthodes :

- On peut mettre l'accumulateur à zéro par un MTA et utiliser ce zéro ailleurs.
- On peut lire la valeur du compteur ordinal, mais il est difficile de savoir précisément ce que l'on y trouve (sauf pour les instructions JRP et JRN).
- On peut utiliser les immédiats pour les nombres n'occupant pas plus de 6 bits.
- On peut construire une constante quelconque par une suite d'immédiats et de décalage. Par exemple, pour mettre `0xF1F1` dans l'accumulateur :

  ```
  MTA 0xF
  LSL 4
  OR  0x1
  LSL 4
  OR  0xF
  LSL 4
  OR  0x1
  ```

  Cette séquence de 7 instructions peut être ramenée à 5 si on sait compter en octal.
- Enfin, en supposant que l'on a préalablement initialisé un registre par la méthode précédente, on dispose d'une zone de 64 mots dans laquelle on peut enregistrer des constantes, que l'on récupère de la façon suivante :

  ```
  MTA [r]
  ADD disp
  LDA [0]
  ```

  On a supposé que l'accumulateur est le registre 0, que `r` est le registre adressant la zone des constantes, et que `disp` est le déplacement de la constante cherchée.

De même, l'exécution des branchements est complexe. On peut utiliser les branchements relatifs (JRP et JRN) si la distance est inférieure à 64, mais dans le cas contraire, il faut stocker une adresse dans un vecteur de transfert, la récupérer comme ci-dessus et utiliser JPR. Comme de plus la distance entre deux instructions dépend du code intermédiaire, l'écriture des branchements est nécessairement un calcul de point fixe. On suggère la méthode suivante :

- Ecrire une première version du code n'utilisant que JPR.
- Tant que des améliorations sont possibles :
  - Choisir un JPR.
  - Si sa cible est située à moins de 64 mots en avant ou en arrière, le remplacer par un JRP ou JRN et compacter le code.

Le résultat de la génération de code doit être un fichier image de la mémoire, dans un format prêt à l'emploi.

4.6 Bibliothèque d'exécution

La bibliothèque d'exécution devra être écrite en langage machine. Heureusement, elle est très rudimentaire. On y trouvera essentiellement les fonctions d'entrée/sortie et un prélude permettant l'initialisation du processeur au démarrage d'un programme (essentiellement la création de la pile et l'initialisation des registres que vous aurez réservé).

Vous devrez également vous poser la question des multiplications et divisions, que la machine FPGA ne sait pas exécuter. On suggère d'écrire un sous-programme de multiplication (d'ailleurs indispensable pour la gestion des tableaux à plusieurs dimension) mais d'interdire la division, du moins dans la première version du compilateur.

5 *To Do List*

5.1 Obligatoire

- La RI et son pretty-printer.
- Analyse syntaxique : Lex/Yacc et impression de la RI.
- Analyse sémantique : contrôle des types.
- Spécification de la structure de la mémoire et de la pile, du display, des séquences d'appel et de retour, du prélude et du postlude.
- Expansion des instructions.
- Expansion des tests et boucles.
- Ecriture de la bibliothèque d'exécution.
- Génération du code et tests.

5.2 Facultatif

Le langage Pascal- peut être étendu de diverses façons pour se rapprocher du Pascal complet. Vous pouvez donner ici libre cours à votre imagination. Voici cependant quelques suggestions :

- L'introduction du type caractère, dont la manipulation n'est pas facile pour la machine FPGA (il n'y a pas d'adressage à l'octet).
- L'introduction de tableaux à bornes variables. La production concernée est :

  ```
  IndexRange --> Expression '..' Expression
  ```

  Quelles restrictions faut-il imposer aux expressions utilisées pour que le programme reste raisonablement compilable ?
- Optimisations diverses (propagation des constantes, élimination du code mort, *peephole optimizations*, minimisation du nombre de registres). La

détection des sous-expressions communes et la réduction de force sont des optimisations importantes pour la bonne gestion des tableaux.
- Implémentation de la division.

Dans tous les cas vous devrez documenter ces extensions :
- Description syntaxique et sémantique de l'extension.
- Spécification précise de l'optimisation.
- Notes d'implémentation.
- Exemples, avec preuves d'efficacité ou d'utilité.

Bon travail.

Références

[dD02] Florent de Dinechin. Mim2, td de compilation, 2002. `www.ens-lyon.fr/~fdedinec/enseignement/compil`.

[Gro92] Peter Grogono. *La Programmation en Pascal*. InterEditions, 1992.

[LDG+00] Xavier Leroy, Damien Doliguez, Jacques Garrigue, Didier Rémy, and Jerôme Vouillon. The objective caml system, release 3.00, 2000. `pauillac.inria.fr/caml/FAQ/general-fra.html`.

[Ris] Tanguy Risset. Cours de compilation. polycopié MIM2.