Introduction à la compilation

Christine Paulin-Mohring

Université Paris Sud

Master Informatique 2011-2012

Objectifs du cours

Comprendre les principes de construction d'un compilateur

Programme: description d'une suite d'opérations qui étant donné une entrée va produire un résultat.

Les opérations sont décrites dans plusieurs modèles de calcul (impératif, fonctionnel, objet, assembleur . . .)

Example (1)

Programme source en ocaml:

```
open Printf
let _ = let sum = ref 0 in
  for i = 1 to 100 do sum := !sum + (i*i) done;
 printf "Result␣:␣%d\n" !sum
```

Example (2)

Byte-code engendré par `ocamlc -c`:

```
 0  CONST0                  18  PUSH
 1  PUSHCONST1              19  OFFSETINT 1
 2  PUSHCONSTINT 100        21  ASSIGN 2
 4  PUSH                    23  ACC1
 5  PUSHACC2                24  NEQ
 6  GTINT                   25  BRANCHIF 9
 7  BRANCHIF 27             27  CONST0
 9  CHECK_SIGNALS           28  POP 2
10  ACC1                    30  ACC0
11  PUSHACC2                31  PUSHGETGLOBAL "Result : %d\n"
12  MULINT                  33  PUSHGETGLOBALFIELD Printf, 1
13  PUSHACC3                36  APPLY2
14  ADDINT                  37  POP 1
15  ASSIGN 2                39  ATOM0
17  ACC1                    40  SETGLOBAL Sum
```

Example (3)

Code machine (partiel) engendré par `ocamlopt`:

```
00000000 <camlSum__entry>:
   0: 83 ec 04                sub    $0x4,%esp
   3: b9 01 00 00 00          mov    $0x1,%ecx
   8: b8 03 00 00 00          mov    $0x3,%eax
   d: 3d c9 00 00 00          cmp    $0xc9,%eax
  12: 7e 05                   jle    19 <camlSum__entry+0x19>
  14: 89 0c 24                mov    %ecx,(%esp)
  17: eb 1c                   jmp    35 <camlSum__entry+0x35>
  19: 89 c2                   mov    %eax,%edx
  1b: d1 fa                   sar    %edx
  1d: 89 c3                   mov    %eax,%ebx
  1f: 4b                      dec    %ebx
  20: 0f af da                imul   %edx,%ebx
  23: 01 d9                   add    %ebx,%ecx
  25: 89 0c 24                mov    %ecx,(%esp)
  .....
```

Autres apports

- maitriser les techniques de base pour la transformation textuelle: (lex, yacc)
- comprendre les caractéristiques des langages de programmation (sémantique, efficacité)
- programmation: structures de données et algorithmes avancés

Organisation

- TDs à partir de recueils d'exercices
  début mercredi 14 septembre
- Projet associé (responsable Kim Nguyen)
- Utilisation du langage ocaml
- Partiel + Controle Continu + Examen

Calendrier prévisionnel

- 12/09 Cours 1-2 : Introduction aux compilateurs, Analyse lexicale
  Cours 3 : Analyse syntaxique
  TD 1 - Projet 1
- 19/09 Cours 4 : Analyse sémantique élémentaire
  TD 2 - Projet 2
- 26/09 Cours 5 : Génération de code 1
  TD 3 - Projet 3
- 03/10 Cours 6 : Génération de code 2
  TD 4 - Projet 4
- 10/10 Cours 7 : Génération de code 3
  TD 5 - Projet 5
- 17/10 Pas de cours (?)
  TD 6 - Projet 6
- 24-28/10 Partiel + rattrapage TD 31 octobre (?)

Calendrier-suite

31/10	Cours 8 : Génération de code 4 TD 7 - Projet 7
07/11	Cours 9 : Organisation de la mémoire TD 8 - Projet 8
14/11	Cours 10 : Analyse sémantique avancée 1 TD 9 - Projet 9
21/11	Cours 11 : Analyse sémantique avancée 2 TD 10 - Projet 10
28/11	TD 11 - Projet 11
??	Projet 12 - soutenances

Pratique

Page WEB du cours :
`http://www.lri.fr/~paulin/COMPIL`

Page WEB du projet :
`http://www.lri.fr/~kn/compil`

Coordonnées :

Adresse électronique	`Christine.Paulin@lri.fr`
Téléphone	01 72 92 59 05
Bureau	PCRI - bat 650 - bureau 74 (RdC Est)

Merci d'utiliser prioritairement le courrier électronique.

Prérequis

- Théorie des langages formels: alphabet, mot, expressions régulières, automates, grammaires, ambiguité
- Langages de programmation: typage, organisation de la mémoire
- Algorithmique : tables de hachage, arbres, graphes. . .

Choix de ocaml

- Un langage pour un développement efficace:
  - Concision du code
  - Bibliothèques
  - Typage fort (documentation, détection des erreurs)
  - Efficacité du code généré
- Un langage particulièrement adapté aux manipulations symboliques:
  - Structures d'arbre
- Concepts avancés de théorie des langages de programmation: ordre supérieur, inférence de type, modules ...

Bibliographie

Compilation

- A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998
- J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Unix Programming Tools. O' Reilly, 1995

ocaml

- E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000
- Claude Marché and Ralf Treinen. Formation au langage CAML. Université Paris Sud. Notes du Cours LGL
- Jean-Christophe Filliâtre. Intiation à la programmation fonctionnelle. Université Paris Sud. Notes de Cours - Master Informatique M1

Assembleur

- Jim Larus. *Computer Organization and Design: The Hardware/Software Interface*, chapter Assemblers, Linkers, and the SPIM Simulator. Morgan Kaufmann, 2004.

  `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`

Principes de fonctionnement du cours

- Présence attentive obligatoire en cours/TD
- Pas de téléphone, ordinateur allumé . . .
- Questions bienvenues
- Des tests de connaissance courts : bonus sur note partiel
- Documents, notes de cours interdits en partiel examen,
  une page A4 manuscripte autorisée.

Plan-1

Introduction à la compilation

- Structure d'un compilateur, exemple
- Quelques notions de sémantique
- Evaluateur versus compilateur, techniques de construction de compilateurs

Analyse lexicale et syntaxique

- Mise en oeuvre d'un analyseur lexical
- Analyse descendante, analyse ascendante
- Ambiguités et précédences
- Actions sémantiques: arbres de syntaxe abstraite

Analyse sémantique élémentaire

- Analyse de portée

Plan-2

Génération de code

- Machines abstraites
- Assembleur
- Appels de fonctions, récursion terminale
- Allocation de registres
- Gestion de la mémoire

Analyse sémantique avancée

- Typage : surcharge, inférence de types
- Analyses statiques

Rappels de vocabulaire

- Alphabet : ensemble fini d'objets appelé caractères
- Mot (sur un alphabet) : suite finie de caractères.
  - Le mot vide sera noté ϵ;
  - l'opération de concaténation de deux mots m_1 et m_2 est notée par simple juxtaposition $m_1 m_2$.
- Langage ensemble de mots.
- Reconnaissance : étant donnés un mot m et un langage L, est-ce que $m \in L$?
  - peut-on construire un programme qui étant donné m décide si $m \in L$?
  - peut-on pour une classe de langages L engendrer un programme qui permet de décider si un mot m appartient à L.
- En compilation, on ne se contente pas de reconnaître l'appartenance d'un mot à un langage, on doit également transformer l'entrée afin d'avancer dans la production de code.

Exemple

- Alphabet
- Mot
- Langage fini
- Langage infini
- Reconnaissance
- Tranformation

Sommaire

1. Préambule
2. Introduction à la compilation
   - Description d'un compilateur
   - Exemple - analyse syntaxique
     - Principes
     - Mise en œuvre
   - Sémantique des langages
   - Exemple: Génération de code
   - Compilateur versus interpréteur
   - Techniques de construction de compilateurs
   - Eléments d'assembleur

Qu'est-ce qu'un compilateur ?

- Un compilateur est un traducteur qui permet de transformer un programme écrit dans un langage L_1 en un autre programme écrit dans un langage machine L_2.
- En pratique on s'arrête souvent à un langage intermédiaire (assembleur ou encore langage d'une machine abstraite).

Qu'est-ce qu'un programme ?

- Description de comment à partir d'une entrée, produire un résultat.
- Le compilateur peut rejeter des programmes qu'il considère incorrects, dans le cas contraire, il construit un nouveau programme (phase statique) que la machine pourra exécuter sur différentes entrées.
- L'exécution du programme sur une entrée particulière peut ne pas terminer ou échouer à produire un résultat (phase dynamique).

Qu'attend-on d'un compilateur ?

Détection des erreurs :

- Identificateurs mal formés, commentaires non fermés ...
- Constructions syntaxiques incorrectes
- Identificateurs non déclarés
- Expressions mal typées `if 3 then "toto" else 4.5`
- Références non instanciées
- ...

Les erreurs détectées à la compilation s'appellent les erreurs statiques.

Les erreurs détectées à l'exécution s'appellent les erreurs dynamiques: division par zéro, dépassement des bornes dans un tableau...

Qu'attend-on d'un compilateur ? (2)

Efficacité

- Le compilateur doit être si possible rapide (en particulier ne pas boucler)
- Le compilateur doit produire un code qui s'exécutera aussi rapidement que possible

Correction

- Le programme compilé doit représenter le même calcul que le programme original.
- Nécessite d'avoir une description indépendante des calculs représentés par les programmes du langage source.

Structure d'un compilateur

Un compilateur se construit en plusieurs phases.

- Analyse syntaxique :
  - Transforme une suite de caractères en un arbre de syntaxe abstraite (ast) représentant la description des opérations à effectuer.
  - Combinaison de l'analyse lexicale qui reconnait des “mots” aussi appelés token ou entités et de l'analyse syntaxique qui reconnait des phrases bien formées.
- Analyse sémantique : analyse l'arbre de syntaxe abstraite pour calculer de nouvelles informations permettant de:
  - rejeter des programmes incorrects (portée, typage. . .)
  - préparer la phase de génération de code (organisation de l'environnement, construction de tables pour les symboles, résolution de la surcharge . . .)

Structure d'un compilateur (2)

- Génération de code : passage par plusieurs langages intermédiaires pour produire un code efficace
  - Organisation des appels de fonctions dans des tableaux d'activation
  - Instructions simplifiées (code à trois opérandes)
  - Analyses de flots
  - Allocation de registres
  - ...

Après la compilation

- La compilation du programme se fait souvent de manière modulaire. Les dépendances par rapport aux autres parties du programme ou aux bibliothèques sont résolues par l'édition de liens.
- Le programme peut être intégré à un runtime, qui va offrir des services indépendants du programme (accès systèmes, gestionnaire de mémoire . . .).
- Dans le cadre de programmes compilés vers du code pour une machine abstraite (bytecode Java ou Ocaml), le runtime comprendra un interprète de la machine abstraite.

Sommaire

1 Préambule

2 Introduction à la compilation
- Description d'un compilateur
- Exemple - analyse syntaxique
  - Principes
  - Mise en œuvre
- Sémantique des langages
- Exemple: Génération de code
- Compilateur versus interpréteur
- Techniques de construction de compilateurs
- Eléments d'assembleur

Exemple

- Compiler un langage ARITH d'expressions arithmétiques vers le code d'une machine à pile.

Le langage ARITH se compose d'une suite d'expressions:

```
set ident = expr
print expr
```

Les expressions peuvent comporter des variables locales.

Exemple de programme:

```
set x = 4
set xx = let y = 2 * x + 5 in - (y * y)
print x * xx
```

Analyse syntaxique

- La première phase d'analyse syntaxique, consiste à transformer la suite de caractères représentant le programme en un arbre de syntaxe abstraite qui ne conserve que les informations utiles pour le calcul.
- Pour cela, on passe par une phase intermédiaire, l'analyse lexicale qui construit une suite d'entités lexicales (token) qui servent d'alphabet d'entrée (symboles terminaux) pour la grammaire.

Description syntaxique du langage

<prog>	$::= \epsilon$ | <prog> <inst>
<inst>	$::=$ **set** *ident* = <expr>
	| **print** <expr>
<expr>	$::=$ *entier* | *ident* |
	| <expr> <binop> <expr> | <unop> <expr>
	| **let** *ident* = <expr> **in** <expr>
	| (<expr>)
<binop>	$::=$ + | - | * | /
<unop>	$::=$ -

Analyse syntaxique

La grammaire distingue les symboles terminaux qui forment l'alphabet d'entrée et les symboles non-terminaux qui sont des symboles intermédiaires pour la reconnaissance.

Exemple

- Terminaux: **set**, **let**, **in**, **print**, `=`, `+`, `-`, `/`, `*`, `(`, `)`, *ident*, *entier*
- Non-terminaux: `<prog>`, `<inst>`, `<expr>`, `<binop>`, `<unop>`.
- Règle de production: $X ::= m$ avec X un symbole non-terminal et m un mot utilisant à la fois des symboles terminaux et non-terminaux.
- Reconnaissance: Un mot m formé de terminaux est reconnu par la grammaire s'il existe un arbre de dérivation syntaxique dont les feuilles correspondent au mot m et les nœuds aux règles de production, la racine étant le symbole initial.

Analyse syntaxique sur un exemple

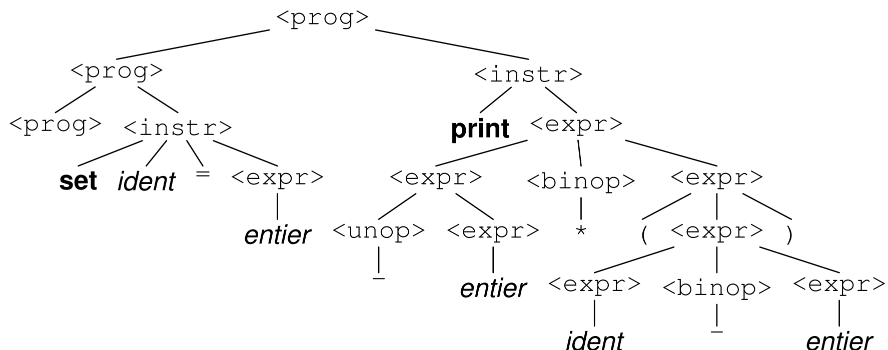
```
set foo = 45
print -3*(foo-2)
```

Suite de terminaux:

set *ident* = *entier* **print** - *entier* * (*ident* - *entier*)

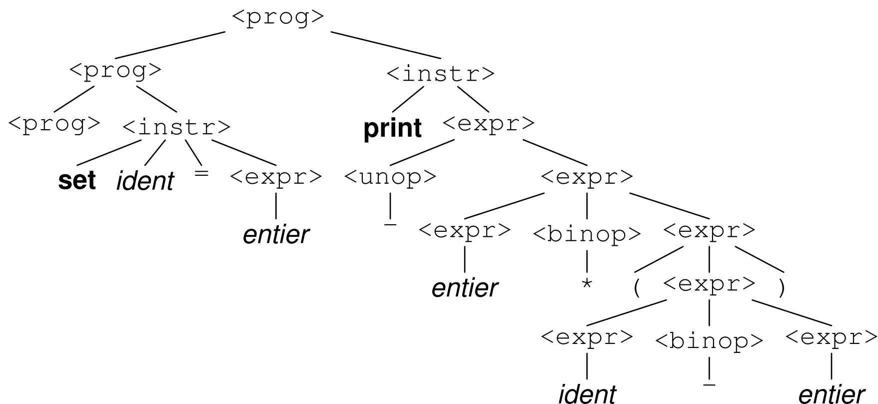
- Les espaces, retour à la ligne sont ignorés.
- Certaines suites de caractères, sont transformées en un “caractère” terminal (mot-clé, identifiant, constante entière).
- Certaines suites de caractères sans espaces sont transformées en plusieurs caractères terminaux.

Exemple: arbre de dérivation syntaxique



Autre dérivation

Arbre alternatif qui reconnait la même entrée (mais pas le même calcul).



- La grammaire est ambigüe (2 arbres pour la même entrée).
- Les règles de précédence permettent de choisir.

Les éléments à préciser

- Qu'est-ce qu'un identificateur ?
  (alpha-numérique, commence par une lettre)
- Qu'est-ce qu'un entier ?
  (suite de chiffres)
- Quels sont les mots clés ?
  (ne peuvent être utilisés comme identificateur)
- Comment se lèvent les ambiguités ?
  (règles usuelles pour les expressions arithmétiques)

Exercice

Construire la suite de tokens puis l'arbre de dérivation syntaxique pour le programme:

```
print let x = -5 in x * x + 2
```

Le programmes suivants ont des erreurs syntaxiques, dire pourquoi.

```
print tel x = -5 in x * x + 2
print (* x + 2
print x % 2
```

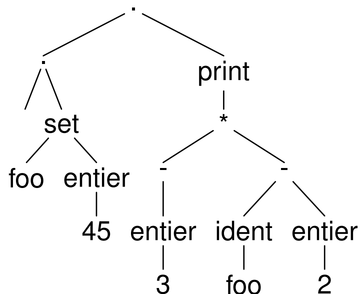
Choix de la structure d'arbre de syntaxe abstraite

- L'arbre de syntaxe abstraite est le résultat de la phase d'analyse syntaxique.
- Construction *simple* à partir de la grammaire (ie de l'arbre de dérivation syntaxique).
- Doit représenter l'information contenue dans le programme utile à l'analyse sémantique et la génération de code.
- Il retire de l'arbre de dérivation syntaxique ce qui ne sert pas au calcul.
- Il réintroduit les valeurs des terminaux comme les constantes entières ou les identifiants.

Arbre de syntaxe abstraite : exemples

- Les parenthèses sont utiles pour résoudre les ambiguités d'une écriture linéaire mais inutiles dans les arbres de syntaxe abstraites.
- Les commentaires sont inutiles pour engendrer le code.
- La localisation des constructions de programmes dans le fichier permet de mieux rendre compte des erreurs sémantiques.

Exemple d'arbre de syntaxe abstraite



Programmer l'analyse syntaxique

- Programmer directement l'analyse syntaxique de la chaîne de caractères jusqu'à l'arbre de syntaxe abstraite serait très fastidueux et sans doute peu efficace.
- On utilise des générateurs d'analyseurs qui permette de ne spécifier que les parties "utiles":
  - les entités lexicales;
  - les règles de grammaire;
  - les arbres de syntaxe abstraite;
- Les analyseurs obéissent à des règles syntaxiques spécifiques.
- Ils sont eux-mêmes compilés pour engendrer les fonctions d'analyse.
- Ils utilisent des méthodes puissantes à base d'automates.

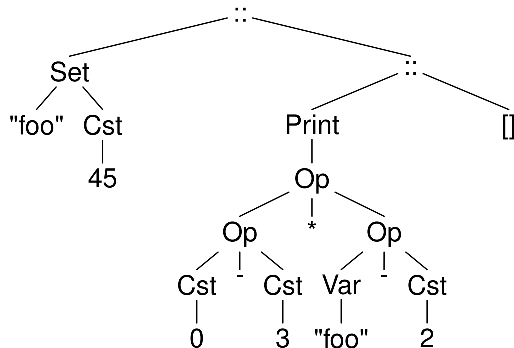
Structure des ast pour ARITH

Type de données ocaml pour représenter des structures arborescentes

```
type binop = Sum | Diff | Prod | Quot
type expr =
    Cst of int
  | Var of string
  | Op of binop*expr*expr
  | Letin of string*expr*expr
type instr =
    Set of string*expr
  | Print of expr
type prg = instr list
```

- Les principales classes syntaxiques correspondent à des types Ocaml.
- On a choisi de ne pas distinguer le moins unaire (codé par $0 - e$).

Exemple de l’ast dans la syntaxe OCAML



Principes

- Utilise des expressions régulières pour reconnaître des entités lexicales.
- Elle traite les entrées correspondantes et produit des tokens qui sont fournis à la grammaire (symboles terminaux).
- Les tokens produits doivent être déclarés (en général dans la grammaire).
- Certaines entités (identificateurs, constantes entières. . .) portent des valeurs qui seront utilisées pour construire l'ast.
- Le type ocaml des tokens

```
type token =
    CST of int | IDENT of string
  | SET | LET | IN | PRINT | EOF | LP | RP
  | PLUS | MINUS | TIMES | DIV | EQ
```

Exemple de fichier ocamllex : `lexer.mll`

Quelques abbréviations pour les expressions régulières:

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ident = letter (letter | digit)*
let integer = ['0'-'9']+
let space = [' ' '\t' '\n']
```

Les règles de reconnaissance

- Un analyseur a un nom associé à un ensembe de règles
- À une expression régulière sur les caractères ASCII correspond une action (production d'un token)

```
rule nexttoken = parse
  | '+'       { PLUS }
  | '-'        { MINUS }
  | '*'       { TIMES }
  | '/'       { DIV }
  | '='       { EQ }
  | '('        { LP }
  | ')'        { RP }
  | integer as x { CST (int_of_string x) }
```

Cas particuliers

Traitement des espaces, des mot-clés, fin de fichier et erreurs

```
rule nexttoken = parse
  | ...
  | space+  { nexttoken lexbuf }
  | ident   { id_or_kwd (lexeme lexbuf) }
  | eof     { EOF }
  | _       { raise (Lexing_error (lexeme lexbuf)) }
```

Entete de fichier

Déclare les fonctions utiles à l'opération d'analyse

```
  {
  open Lexing
  open Parser
  exception Lexing_error of string

  let kwd_tbl =
   ["let",LET;"in",IN; "set",SET;"print",PRINT]
  let id_or_kwd s =
   try List.assoc s kwd_tbl with _ -> IDENT(s)
}
```

Les outils de type lex

La commande

```
ocamllex lexer.mll
```

produit un fichier ml `lexer.ml` qui lui-même contient une fonction

```
val nexttoken : Lexing.lexbuf -> Parser.token
```

- Le module `Lexing` d'ocaml introduit des fonctions de manipulation lexicales.
- Des fonctions comme `lexeme` permettent d'accéder à la chaîne de caractères correspondant à l'expression régulière reconnue.
- Dans cet exemple, le type des tokens est défini dans le fichier de grammaire `Parser`.
- Un objet de type `Lexing.lexbuf` (buffer pour l'analyse lexicale, comportant des informations sur le positionnement dans le fichier) peut être construit à partir d'un fichier ou d'une chaîne de caractères.

Grammaire

- Une grammaire se définit par des symboles terminaux, non-terminaux et des règles de dérivation.
- Un *mot* est reconnu si on peut construire un arbre de dérivation syntaxique dont les feuilles sont les terminaux et les noeuds correspondent aux règles.
- Les tokens fournis par l'analyseur lexical correspondent aux terminaux de la grammaire.
- Les règles suivent les constructions syntaxiques du langage mais il faut tenir compte des ambiguités.
- Les actions associées aux règles permettent de construire les ast.

Exemple de fichier ocamlyacc

Fichier `parser.mly`
Le cœur du programme:

```
instr:
| SET IDENT EQ expr { Set($2,$4)}
| PRINT expr        { Print($2) }
;

expr:
| CST                           { Cst($1) }
| IDENT                         { Var($1) }
| expr PLUS expr                { Op(Sum,$1,$3) }
| expr MINUS expr               { Op(Diff,$1,$3) }
| expr TIMES expr               { Op(Prod,$1,$3) }
| expr DIV expr                 { Op(Quot,$1,$3) }
| MINUS expr %prec uminus       { Op(Diff,Cst(0),$2) }
| LET IDENT EQ expr IN expr { Letin($2,$4,$6) }
| LP expr RP                    { $2 }
;
```

Préambule

- Déclaration des tokens
- Précision des précédences

```
%token <int> CST
%token <string> IDENT
%token SET, LET, IN, PRINT
%token EOF
%token LP RP
%token PLUS MINUS TIMES DIV
%token EQ

/* priorités et associativités des tokens */
%nonassoc IN
%left MINUS PLUS
%left TIMES DIV
%nonassoc uminus
```

Points d’entrée

```
/* Point d'entrée de la grammaire */
%start prog

/* Type des valeurs retournées par l'analyseur syntaxique *
%type <Ast.prg> prog

%%
prog:
| instrs EOF {List.rev $1}
;
instrs:
| instr        { [$1] }
| instrs instr { $2::$1 }
;
```

Les outils de type Yacc

La commande

```
ocamlyacc parser.mly
```

produit un fichier ml `parser.ml` qui contient la déclaration du type des tokens ainsi que la fonction d'analyse:

```
val prog :
 (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Ast.prg
```

Pour construire un ast correspondant à un fichier `file`

```
let f = open_in file in
let buf = Lexing.from_channel f in
Parser.prog Lexer.nexttoken buf
```

A propos des outils d'analyse

- Les outils comme ocamllex et ocamlyacc font appel à des algorithmes complexes de calculs d'automates:
  - analyse lexicale : automates finis
  - analyse syntaxique : automates à pile

- Nous étudierons le fonctionnement de ces outils sans entrer dans le détail du calcul des tables.

Sommaire

1. Préambule

2. Introduction à la compilation
   - Description d'un compilateur
   - Exemple - analyse syntaxique
     - Principes
     - Mise en œuvre
   - Sémantique des langages
   - Exemple: Génération de code
   - Compilateur versus interpréteur
   - Techniques de construction de compilateurs
   - Eléments d'assembleur

Sémantique des langages

- La sémantique sert à préciser la nature des calculs représentés par un programme
- La plupart des langages courants ne disposent pas d'une sémantique formelle complète; la description informelle des manuels de référence peut être incomplète ou ambigüe (les compilateurs peuvent donc faire des choix multiples).
- Il y une grande diversité de sémantiques pour les langages de programmation (dénotationnelle, axiomatique, statique, ...).

Analyse de portée

- Un nom (on parle souvent de variable) est introduit dans le programme pour représenter une valeur qui est le résultat d'un calcul.
- Ce nom peut ensuite être réutilisé dans le programme.
- Le langage peut restreindre la partie du programme dans lequel un nom est visible, c'est le cas des variables locales ou bien de la construction **let** $x = e_1$ **in** e_2 (x n'est visible que dans e_2).
- Un même nom dans le programme peut être déclaré plusieurs fois, il représentera deux objets différents.

```
set x = 3 print x + let x = 2 * x in x + 5
```

- Les règles de portées définissent à quelle déclaration est associée chaque utilisation.
- On peut remplacer un nom x par un nom y à condition de renommer la déclaration et les utilisations correspondantes et que y ne soit pas déjà utilisé dans le programme.

```
set x1 = 3 print x1 + let x2 = 2 * x1 in x2 + 5
```

Analyse de portée - exemple

- Une phase d'analyse de portée vérifie:
  - tout nom utilisé a bien été déclaré
  - l'utilisation est bien faite dans la portée de la déclaration
- Elle peut engendrer un nom unique et effectuer la transformation du programme afin de simplifier les traitements ultérieurs.
- Création de nouveaux noms en ocaml en utilisant un “séparateur” et une référence privée. Création de nouveaux noms en utilisant un “séparateur” et une référence privée.

```
let new_name = let i = ref 0 in
  fun x -> incr i; x ^ "_" ^ (string_of_int !i)
```

Analyse de portée de ARITH

Une liste de visibilité `vis` les noms visibles associés à leur renommage.

```
exception ScopeError
let rec scope_expr vis = function
  | Cst n as x -> x
  | Op (o,e1,e2) ->
      let e1' = scope_expr vis e1
      and e2' = scope_expr vis e2
      in Op(o,e1',e2')
  | Var x -> (try let x' = List.assoc x vis in Var x'
              with Not_found -> raise ScopeError)
  | Letin(x,e1,e2) ->
      let e1' = scope_expr vis e1 and x' = new_name x
      in let e2' = scope_expr ((x,x')::vis) e2
      in Letin(x',e1',e2')
```

Analyse de portée de ARITH (2)

Extension aux programmes:

```
let rec scope_instrs vis = function
    [] -> []
  | Set(x,e)::prog ->
        let x' = new_name x
        and e' = scope_expr vis e in
        Set(x',e')::(scope_instrs ((x,x')::vis) prog)
  | Print(e)::prog ->
        Print(scope_expr vis e)::scope_instrs vis prog

let scope_prog = scope_instrs []
```

Sémantique opérationnelle à grand pas

- Elle explique le calcul effectué par un programme en fonction de l'environnement dans lequel il est exécuté.

$$\rho \vdash p \leadsto m$$

- Elle s'appuie sur un modèle des valeurs sémantiques (des objets mathématiques).
- Elle s'exprime sous forme de règles d'inférences qui permettent de définir la relation d'évaluation.

Exemple

Cas du langage ARITH:

- Les valeurs manipulées pour les expressions sont uniquement des entiers.
- Dans le langage de programmation, l'opération $+$ s'applique à deux expressions quelconques.
  Dans le modèle sémantique, on a des opérations pour effectuer les calculs correspondants sur les entiers.
- Un environnement associe des valeurs à des variables.

Les règles expliquent comment chaque programme produit une valeur:

- dans un environnement donné, une expression s'évalue vers un entier (si toutes les variables mentionnées sont définies dans l'environnement)
- un programme s'évalue également dans un environnement (initialement vide) et a pour valeur une liste des valeurs imprimées.

Exemple de règles

$$\frac{\rho \vdash e_1 \leadsto n_1 \quad \rho \vdash e_2 \leadsto n_2}{\rho \vdash \texttt{Op}(\texttt{PLUS}, e_1, e_2) \leadsto n_1 + n_2}$$

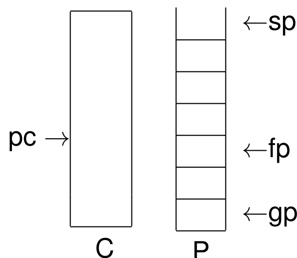
$$\frac{\rho \vdash e_1 \leadsto n_1 \quad \rho + (x, n_1) \vdash e_2 \leadsto n_2}{\rho \vdash \texttt{Letin}(x, e_1, e_2) \leadsto n_2}$$

$$\frac{\rho \vdash e \leadsto n \quad \rho + (x, n) \vdash p \leadsto l}{\rho \vdash (\texttt{Set}(x, e) :: p) \leadsto l}$$

$$\frac{\rho \vdash e \leadsto n \quad \rho \vdash p \leadsto l}{\rho \vdash (\texttt{Print}(e) :: p) \leadsto (n :: l)}$$

Une machine virtuelle à pile

L'organisation de la machine virtuelle est décrite par la figure suivante:



- zone de code `C`
- registre *pc* qui pointe sur l'instruction courante à exécuter.
- pile `P` permettant de stocker des valeurs entières.
- Registres permettent d'accéder à différentes parties de `P`:
  - *sp* (stack pointer) pointe sur le premier emplacement vide de la pile,
  - *fp* (frame pointer) repère l'adresse de base du tableau d'activation des fonctions
  - *gp* pointe sur la base de la pile à partir de laquelle sont stockées les variables globales.

Sémantique de la machine virtuelle

- la valeur correspond à la configuration de la machine
- les programmes sont des listes d'instructions
- on donne une sémantique opérationnelle à petit pas qui décrit la transformation de l'état lors de chaque instruction.

Règles sémantiques

Code	Description	*sp*	Condition
ADD	P[sp-2]:=P[sp-2]+P[sp-1]	sp-1	P[sp-2],P[sp-1] entiers
SUB	P[sp-2]:=P[sp-2]-P[sp-1]	sp-1	P[sp-2],P[sp-1] entiers
MUL	P[sp-2]:=P[sp-2]*P[sp-1]	sp-1	P[sp-2],P[sp-1] entiers
DIV	P[sp-2]:=P[sp-2]/P[sp-1]	sp-1	P[sp-2],P[sp-1] entiers
PUSHI n	P[sp]:=n	sp+1	n entier
PUSHN n	P[sp]:=0 ...P[sp+n-1]:=0	sp+n	n entier
PUSHG n	P[sp]:=P[gp+n]	sp+1	n entier
STOREG n	P[gp+n]:=P[sp-1]	sp-1	n entier
PUSHL n	P[sp]:=P[fp+n]	sp+1	n entier
STOREL n	P[fp+n]:=P[sp-1]	sp-1	n entier
WRITEI	imprime P[sp-1] sur l'écran	sp-1	P[sp-1] entier
START	Affecte la valeur de sp à fp	sp	
STOP	Arrête l'exécution du programme	sp	

Schéma de compilation

Avant d'écrire la fonction de compilation, il faut déterminer la correspondance entre les entités des deux langages.

- Le langage ARITH manipule des noms de variables
- La machine virtuelle stocke les données dans la pile et les repère par un décalage par rapport à un endroit donné sur la pile (par exemple repéré par `gp`).
- Chaque variable x de l'environnement est stockée à un endroit précis dans la pile.
  Le choix d'affectation est fait à l'analyse sémantique et conservé dans une table des symboles.

Calcul des décalages

- Suppose les noms uniques.
- Cas simple (sans fonction) qui permet d'allouer les variables statiquement.
- Exemple:

```
set x = 4
set xx = let y = 2 * x + 5 in - (y * y)
print x * xx
```

On peut stocker `x` à l'adresse 0 et les variables `xx` et `y` à l'adresse 1.

Représentation de la table des symboles

- La table des symboles peut s'implanter avec une table de hachage:

```
(* val tab : (string, int) Hashtbl.t
   val addsymb : string -> int -> unit
   val findsymb : string -> int *)
let tab = Hashtbl.create 17
let addsymb x n =  Hashtbl.add tab x n
let findsymb x = Hashtbl.find tab x
```

- Calcul du décalage maximum:

```
(* val maxsymb : unit -> int *)
let maxsymb () =
    Hashtbl.fold (fun _ n m -> max n m) tab 0
```

Calcul des décalages (2)

- Parcours d'une expression (*n* est le nombre de variables visibles)

```
(* val dec_expr : int -> expr -> unit *)
let rec dec_expr n = function
     Var _ | Cst _ -> ()
   | Op(o,e1,e2) -> dec_expr n e1; dec_expr n e2
   | Letin(x,e1,e2) ->
      dec_expr n e1; addsymb x n; dec_expr (n+1) e2
```

- Extension aux programmes:

```
(* val dec_instrs : int -> instr list -> unit *)
let rec dec_instrs n = function
    [] -> ()
  | Set(x,e)::prog -> dec_expr n e; addsymb x n;
                      dec_instrs (n+1) prog
  | Print(e)::prog -> dec_expr n e; dec_instrs n prog
(* val dec_prog : instr list -> unit *)
let dec_prog = dec_instrs 0
```

Schéma de compilation (2)

- Invariant à déterminer: où se retrouve la valeur du programme?
  - Valeur en sommet de pile
  - Correspondance entre les valeurs dans la pile et l'environnement pour toutes les variables visibles ($x \in \mathcal{V}$).
    $P \leftrightarrow_T \rho$ si $\forall x \in \mathcal{V}, P[T(x)] = \rho(x)$
- Formulation mathématique:

$$\frac{\rho \vdash e \leadsto n \quad P \leftrightarrow_T \rho \quad \texttt{compile}(T, e) = lc \quad (P, sp) + lc \leadsto (P', sp')}{P'[sp' - 1] = n}$$

- On garantit également que l'environnement n'a pas été altéré:

$$\forall x \in \mathcal{V}, P[T(x)] = P'[T(x)]$$

Sommaire

1. Préambule
2. Introduction à la compilation
   - Description d'un compilateur
   - Exemple - analyse syntaxique
     - Principes
     - Mise en œuvre
   - Sémantique des langages
   - Exemple: Génération de code
   - Compilateur versus interpréteur
   - Techniques de construction de compilateurs
   - Eléments d'assembleur

Génération de code pour ARITH

- Utilise un type concret `machine` pour les instructions de la machine.

```
type machine =
    Pushi of int | Pushn of int
  | Add | Sub | Mul | Div
  | Pushg of int | Storeg of int
  | Pushl of int | Storel of int
  | Writei | Start | Stop
```

- Correspondance des opérations sur les entiers:

```
(* val gen_op : binop -> machine *)
let gen_op = function
  Sum -> Add | Diff -> Sub | Prod -> Mul | Quot -> Div
```

Génération de code pour ARITH (2)

Traduction des expressions:

```
(* val gen_expr : expr -> machine list *)
let rec gen_expr = function
  | Cst n -> [Pushi n]
  | Var x -> [Pushg (findsymb x)]
  | Op(o,e1,e2) ->
      gen_expr e1 @ gen_expr e2 @ [gen_op o]
  | Letin(x,e1,e2) ->
      gen_expr e1 @ [Storeg (findsymb x)] @ gen_expr e2
```

Génération de code pour ARITH (3)

Extension aux programmes:

```
(* val gen_instrs : instr list -> machine list *)
let rec gen_instrs = function
 | [] -> []
 | Set(x,e)::prog ->
       gen_expr e @ [Storeg (findsymb x)]
                  @ gen_instrs prog
 | Print(e)::prog ->
       gen_expr e @ [Writei] @ gen_instrs prog

(* val gen_prog : instr list -> machine list *)
let gen_prog p =
    dec_prog p;
    let n = maxsymb () in
    Start::(Pushn n)::gen_instrs p@[Stop]
```

Sommaire

1. Préambule
2. Introduction à la compilation
   - Description d'un compilateur
   - Exemple - analyse syntaxique
     - Principes
     - Mise en œuvre
   - Sémantique des langages
   - Exemple: Génération de code
   - Compilateur versus interpréteur
   - Techniques de construction de compilateurs
   - Eléments d'assembleur

Interpréteur

- On peut évaluer les programmes d'un langage *L* en utilisant un autre langage de programmation dès que l'on connait les entrées du programme.
- Implantation plus ou moins sophistiquée des règles de sémantique.
- L'interpréteur ajoute un niveau supplémentaire d'exécution qui le rend en général moins efficace que le compilateur mais il est aussi plus facile à écrire (on peut utiliser toute la puissance du langage de programmation).
- L'interpréteur présuppose la connaissance des entrées du programmes, il ne permet pas de modulariser la construction des résultats.
- Le compilateur a juste besoin de savoir où il pourra trouver les valeurs d'entrées au moment de l'exécution.
- Il n'est pas forcément nécessaire de connaître précisément les valeurs des entrées pour faire des évaluations intéressantes. On peut également calculer avec des valeurs abstraites comme les types.

Interpréteur pour ARITH

- Un environnement stocke les valeurs de chaque variable par exemple sous la forme d'une table de hachage ou d'un tableau (indexé par les décalages)

```
(* val env : (string, int) Hashtbl.t
val addenv : string -> int -> unit
val findenv : string -> int
val clearenv : unit -> unit *)
let env = Hashtbl.create 17
let addenv x n =  Hashtbl.add env x n
let findenv x =  Hashtbl.find env x
let clearenv () =  Hashtbl.clear env
```

- Interprétation des opérations par des fonctions ocaml sur les entiers:

```
(* val interp_op : binop -> int -> int -> int *)
let interp_op = function
  Sum -> ( + ) | Diff -> ( - )
| Prod -> ( * ) | Quot -> ( / )
```

Interpréteur pour ARITH (2)

- Interprétation des expressions:

```
(* val interp_expr : expr -> int *)
let rec interp_expr = function
    Var x -> findenv x | Cst n -> n
  | Op(o,e1,e2) ->
        let n1 = interp_expr e1 and n2 = interp_expr e2
        in (interp_op o) n1 n2
  | Letin(x,e1,e2) -> let n1 = interp_expr e1
       in addenv x n1; interp_expr e2
```

Interpréteur pour ARITH (3)

- Interprétation des programmes:

```
(* val interp_instrs : instr list -> unit *)
let rec interp_instrs = function
    [] -> ()
  | Set(x,e)::prog ->
       let n = interp_expr e in
       addenv x n; interp_instrs prog
  | Print(e)::prog ->
       let n = interp_expr e in
       print_int n; print_newline();
       interp_instrs prog

(* val interp_prog : prog -> unit *)
let interp_prog p = clearenv (); interp_instrs p
```

Sommaire

1. Préambule
2. Introduction à la compilation
   - Description d'un compilateur
   - Exemple - analyse syntaxique
     - Principes
     - Mise en œuvre
   - Sémantique des langages
   - Exemple: Génération de code
   - Compilateur versus interpréteur
   - Techniques de construction de compilateurs
   - Eléments d'assembleur

Techniques de construction de compilateurs

- De nombreux compilateurs (Java, ocaml) sont écrits dans le langage lui-même
- On utilise un mécanisme d'auto-compilation (boot-strap)
  - un compilateur ou évaluateur élémentaire écrit dans un langage de bas niveau
  - des compilateurs de plus en plus avancés écrits dans le langage de haut niveau
- On peut également faire de la compilation croisée qui permet de créer des compilateurs pour des architectures multiples.

Détail du bootstrap

Voir aussi exercice de TD

Sommaire

1 Préambule

2 Introduction à la compilation
- Description d'un compilateur
- Exemple - analyse syntaxique
  - Principes
  - Mise en œuvre
- Sémantique des langages
- Exemple: Génération de code
- Compilateur versus interpréteur
- Techniques de construction de compilateurs
- Eléments d'assembleur

Assembleur MIPS

Dans le cours, on utilisera la génération de code MIPS (MIPS32) et un simulateur SPIM.

`http://pages.cs.wisc.edu/~larus/spim.html`

- MIPS est un assembleur de la famille RISC (Reduced Instruction Set) par opposition aux architectures CISC (Complex Instruction Set) comme le x86.
- Il est utilisé dans des processeurs embarqués, consoles de jeux ...
- Il a peu d'instructions assez régulières et un nombre important de registres .

Petit rappels d'architecture

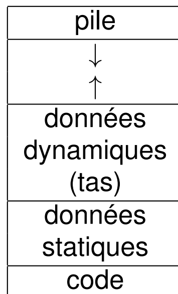
- Les registres se trouvent sur l'unité de calcul (CPU).
- La mémoire vive (RAM) contient les données et les instructions.
- Un octet (ou byte) se compose de 8 bits et peut se coder en hexadécimal à l'aide de deux caractères ([0-9,a-f]).
  $0x9b$ représente l'entier binaire 10011011
- Un mot se compose de 32 bits et donc de 4 octets.

Registres MIPS

- L'architecture MIPS s'appuie sur 32 registres entiers $0 à $31 ayant des rôles spécifiques.

Nom	Numéro	Rôle
$zero	$0	constante 0
$at	$1	réservé pour l'assembleur
$v0–$v1	$2–$3	utilisé pour les valeurs de retour des fonctions et l'évaluation des expressions
$a0–$a3	$4–$7	passage des arguments d'une fonction
$t0–$t7	$8–$15	temporaires
$s0–$s7	$16–$23	temporaires
$t8–$t9	$24–$25	temporaires
$k0–$k1	$26–$27	reservé pour l'OS
$gp	$28	pointeur global
$sp	$29	pointeur de pile (stack)
$fp	$30	pointeur de “frame” (tableau d'activation des fonctions
$ra	$31	adresse de retour

Organisation de la mémoire



- La pile permet de stocker des données dont on controle la durée de vie sur une base dernier entré - premier sorti.
- Le registre $*sp* pointe au sommet de la pile.
  Les adresses décroissent quand la pile augmente.
- Le pointeur $*gp* pointe au milieu de la zone des données statiques.

Instructions MIPS

- Exécution:
  - *$pc* contient l'adresse de l'instruction à exécuter
  - on lit 4 (ou 8) octets à cette adresse
  - on interprète ces bits comme une instruction et ses arguments
  - on exécute l'instruction
  - on modifie le registre *$pc* pour passer à l'instruction suivante (en général juste après sauf si instruction de saut)
- Trois sortes d'instructions
  - transfert entre registre et mémoire (strore/load)
  - instructions de calcul entre registres
  - instructions de saut
- Les instructions peuvent utiliser des constantes immédiates qui sont représentées sur 16 bits.

Quelques (pseudo-)instructions de calcul

- initialisation

`li`	\$r0, C	\$r0 $\leftarrow$C
`lui`	\$r0, C	\$r0 $\leftarrow 2^{16}$ C
`move`	\$r0, \$r1	\$r0 $\leftarrow$\$r1

- arithmétique entre registres:

`add`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$\$r1+\$r2
`addi`	\$r0, \$r1, C	\$r0 $\leftarrow$\$r1+C
`sub`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$\$r1-\$r2
`div`	\$r0, \$r1,\$r2	\$ro $\leftarrow$\$r1/\$r2
`div`	\$r1, \$r2	\$lo $\leftarrow$\$r1/\$r2; \$hi $\leftarrow$\$r1 mod \$r2
`mul`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$\$r1 $\times$ \$r2 (pas d'overflow)
`mult`	\$r1, \$r2	\$lo $\leftarrow$\$r1 $\times$ \$r2[low] ; \$hi $\leftarrow$\$r1 $\times$ \$r2[high]
`neg`	\$r0, \$r1	\$r0 $\leftarrow$-\$r1

Quelques (pseudo-)instructions de calcul (2)

- Test égalité et inégalité

`slt`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$1 si \$r1<\$r2 et \$r0 $\leftarrow$0 sinon
`slti`	\$r0, \$r1, C	\$r0 $\leftarrow$1 si \$r1< C et \$r0 $\leftarrow$0 sinon
`sle`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$1 si \$r1$\leq$\$r2 et \$r0 $\leftarrow$0 sinon
`seq`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$1 si \$r1=\$r2 et \$r0 $\leftarrow$0 sinon
`sne`	\$r0, \$r1, \$r2	\$r0 $\leftarrow$1 si \$r1$\neq$\$r2 et \$r0 $\leftarrow$0 sinon

- Opérations logiques (bit à bit) : `and`, `andi`, `not`, `or`, `sll`, `srl` ...
- Opérations sur les flottants : `add.s`, `add.d` ...

Chargement mémoire

- Une adresse (*adr*) a la forme générale $C(\$r0)$ et correspond à l'adresse dans le registre $r0 plus le décalage de la constante C sur 16 bits signés.
- Stocker une adresse :

`la`	$r0, adr	$r0 ←adr

- Lire en mémoire :

`lw`	$r0, adr	$r0 ←mem[adr], lit un mot
`lh`	$r0, adr	$r0 ←mem[adr], lit un demi-mot (16 bits)
`lb`	$r0, adr	$r0 ←mem[adr], lit un octet
`mfhi`	$r0	$r0 ←$hi
`mflo`	$r0	$r0 ←$lo

- Stocker en mémoire

`sw`	$r0, adr	mem[adr] ←$r0, stocke un mot
`sh`	$r0, adr	mem[adr] ←$r0, stocke un demi-mot (16 bits)
`sb`	$r0, adr	mem[adr] ←$r0, stocke un octet

Instructions de saut

- Architecture MIPS : branchement conditionnel avec déplacement relatif sur 16 bits
- Assembleur MIPS : branchement vers un label (calcul automatique du décalage).

`beq`	$r0, $r1, label	branchement conditionnel si $r0=$r1
`beqz`	$r0, label	branchement conditionnel si $r0=0
`bgt`	$r0, $r1, label	branchement conditionnel si $r0>$r1
`bgtz`	$r0, label	branchement conditionnel si $r0>0

Les versions `beqzal`, `bgtzal`... assurent de plus la sauvegarde de l'instruction suivante dans le registre *$ra*

- Saut inconditionnel dont la destination est stockée sur 26 bits

`j`	label	branchement inconditionnel
`jal`	label	branchement inconditionnel avec sauvegarde dans $ra de l'instruction suivante
`jr`	$r0	branchement inconditionnel à l'adresse dans $r0

aussi `jalr`...

Structure d'un programme assembleur

- Conventions lexicales:
  - Un commentaire commence par un `#` et se termine en fin de ligne
  - Les entiers s'écrivent en base 10 par défaut. On utilise le préfixe `0x` pour une écriture hexadécimale.
  - Les identifiants commencent par un caractère alphabétique et peuvent contenir des chiffres et le caractère _.
  - Les chaines de caractères sont délimitées par `"` et utilisent des caractères spéciaux \n, \t,\".
  - Les labels sont des identifiants terminés par le caractère `:`
- Directives:
  - `.data` les objets suivants sont stockés dans la partie données
  - `.text` les objets suivants sont stockés dans la partie code
  - `.globl` *symb* déclare la label comme global pouvant être référencé de l'extérieur
  - `.asciiz` *str* stocke la chaîne *str* terminée par le caractère null
  - aussi `.word`, `.byte`, `.float` ...
  - Interface avec SPIM : SPIM appelle le programme à l'adresse `main:` et stocke dans *$ra* l'adresse où revenir.

Appels systèmes

Une instruction spéciale `syscall` permet de faire des opérations de lecture/écriture

- Code instruction dans $v0
- Arguments dans $a0-$a3
- Résultat éventuel dans $v0

Exemple : pour afficher un résultat contenu dans un registre $t0

```
    .data
str:
    .asciiz "Answer = "
    .text
li $v0, 4              #code print_str
la $a0,str             # adresse string
syscall
li $v0, 1              #code print_int
move $a0,$t0           # constante à imprimer
syscall
```

Conclusion

- Survol rapide des principales étapes de la compilation
- Points abordés plus tard : représentation de structures de données, appels de fonctions, données dynamiques.
- Points non abordés : architecture avancées : pipe-line, parallélisme, mémoire . . .
- De nombreux niveaux de langages dont il faut maitriser la syntaxe, la sémantique . . .

Problème du bootstrap

- Ecrire le compilateur du langage source L dans le langage machine M dans le langage de haut-niveau L.
- Obtenir une version compilée du compilateur de L dans M dans le langage M (ie exécutable).

Solution du bootstrap

- On dispose d'une compilateur (ou interpréteur) C_0 d'un sous-ensemble L_0 du langage L vers M écrit dans le langage M.
- On écrit un compilateur naif C_1 de L vers M dans le langage L_0.
- On compile C_1 à l'aide C_0 et on obtient un compilateur (naif) D_0 de L vers M dans le langage M.
- On écrit un compilateur avancé C_2 de L vers M dans le langage L.
- On compile C_2 à l'aide D_0 et on obtient un compilateur (avancé) D_1 de L vers M dans le langage M.
- Le compilateur D_1 produit un bon code mais a été compilé par un programme naif donc n'est pas efficace.
- On recompile C_2 à l'aide D_1 et on obtient un compilateur (avancé et s'exécutant rapidement) D_2 de L vers M dans le langage M.

Retour