0-0

Initiation à la compilation des langages de programmation
INF-445 cours 3

Ronan.Keryell`@enst-bretagne.fr`

—

Laboratoire Informatique & Télécommunications

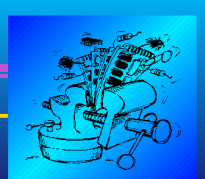
Département Informatique

École Nationale Supérieure des Télécommunications de Bretagne

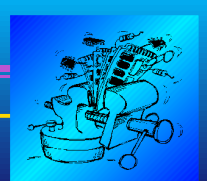
29 mai 2006

Version 1.5

- Si vous améliorez ces cours, merci de m'envoyer vos modifications ! :-)
- Transparents 100 % à base de logiciels libres (LaTeX,...)
- « Je suis contre les polys » (cf CdV) mais :
  - Cours « cliquable »
  - Dense :-(
  - Table des matières



- Intérêt des ordinateurs : la programmation !
- Problème : ordinateurs programmés en langage machine codé en binaire ☹
- Domaine ancien mais toujours vivant car indispensable !
  - Nouveaux langages, nouvelles machines
  - Grand écart entre architectures et langages (objets,...)
  - Preuve de programme, sûreté de fonctionnement
  - Optimisation (vitesse des programmes scientifiques, consommation électrique des dispositifs embarqués,...)

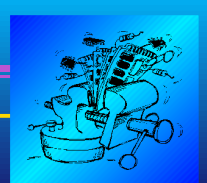


- Mais pourquoi apprendre ça ? ☹
  - Comprendre comment cela marche ! Nécessaire pour un ingénieur, un chercheur,... un curieux ! ☺
  - Informaticien compétent : comprend langages & matériel
  - Compilateur ≡ relie les 2 ⇝ comprendre compilation !
- Extension à d'autres domaines
  - Analyse fichiers de configuration
  - XML-isation de l'univers

  ⇝ Augmentation productivité si on connaît compilation
- Sujet difficile à faire passer en 2h40... ☹

- 30 % du prix d'une voiture haut de gamme est dans l'électronique
- 90 % de l'innovation dans les voitures sera dans l'électronique (Daimler-Chrisler, 2000)
- Plus d'électronique que dans tout le système Apollo
  - 75 processeurs
  - 150 moteurs électriques
  - 100 MLOC source
  - Besoin d'avoir une seule architecture pour tous les modèles du bas de gamme au haut de gamme
  - Évolutivité ⇝ potentiellement des FPGA (Daimler-Chrysler)
- Sûreté de fonctionnement
- Sécurité

- Programmer tout ça... ☺
- Optimiser ⇝ économie de matériel, coût−−, gain++



- Cours en français
  - http://lampwww.epfl.ch/courses/compilation01 du Laboratoire des Méthodes de Programmation (LAMP)
  - http://www.lri.fr/~paulin/COMPIL le cours plus théorique de Christine PAULIN-MOHRING
  - http://pauillac.inria.fr/~maranget/X/compil de Luc MARANGET
  - http://perso.ens-lyon.fr/tanguy.risset/cours/compil2004.html
  - http://www.lit.enstb.org/~keryell/cours/DEA/IAHP/html : module Informatique & Architecture Hautes Performances
- Regarder les cours de Berkeley en ligne

  http://www-inst.eecs.berkeley.edu/classes-cs.html
  - http://inst.eecs.berkeley.edu/~cs164 : Programming

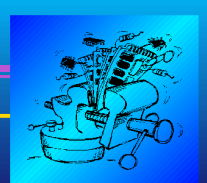
Languages and Compilers

- `http://inst.eecs.berkeley.edu/~graham` : Advanced Programming Language Implementation

- Livres
  - Le « dragon book ». « *Compilateurs : principes, techniques et outils* » Alfred AHO, Ravi SETHI, and Jeffrey D. ULLMAN. InterÉditions, 1991
  - « *The Compiler design handbook : optimizations and machine code generation* », éditeurs : Y. N. SRIKANT & Priti SHANKAR. CRC Press, 2003
  - « *Modern compiler implementation in JAVA* », Andrew W. APPEL. Cambridge University Press, 2002
  - « *Optimizing compilers for modern architectures : a dependence-based approach* », Randy ALLEN & Ken

KENNEDY. Morgan Kaufman Publishers, 2002

- « *Advanced compiler design and implementation* », Steven S. MUCHNICK. Morgan Kaufman Publishers, 1997



- Jusqu'aux années 1950 : que les instructions machines
- Développement du langage Fortran et de son compilateur par John Backus & Co chez IBM à partir de 1954 ⇝ ↗ vitesse d'écriture des programmes
- Machines langages des années 1970 : mettre le maximum en matériel pour système et langage. Trop complexe...
- Concept RISC : simplifier le matériel et laisser le compilateur optimiser
- Certains domaines sont encore dans l'assembleur : bases de systèmes d'exploitation, embarqué, DSP, E=M6 ☹

- L'utilisateur : veut un programme qui marche
- Le programmeur
  - Besoin de productivité
  - Outils
- Le langage (de haut niveau)
  - Impératif
  - Fonctionnel
  - Objet
  - Déclaratif
- Le processeur
  - Mémoire lente
  - Mémoire cache
  - Pipeline

- Unités fonctionnelles
- Instructions plus ou moins complexes
- Programmation de bas niveau en binaire

- Traduction langage de haut niveau
  - Compilateur
    - Traduit programme directement en instructions machine

      ```
      a = b + c;
      ```

    - Rapide une fois traduction faite

      ```
      fadd r3,r3,r1
      ```

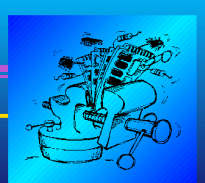

- Interpréteur
  - Traduit et exécute instruction par instruction
    1. Lit ligne
    2. Analyse ligne
    3. Décode instructions
    4. Exécute addition
       (a) Accède contenu case mémoire variable `b`
       (b) Accède contenu case mémoire variable `c`
       (c) Additionne 2 opérandes
    5. Écrit résultat case mémoire variable `a`
  - Programmation plus interactive
- Mélange des 2
  - Débogueur : permet modifier code dans le dos du programme compilé
  - Langage intermédiaire : Pcode (Pascal), JVM (Java),

PostScript (imprimante),...

- Écrit une fois pour toute un compilateur pour 1 langage intermédiaire : moyennement compliqué
- Écrit autant d'interpréteurs de ce langage que de cible : simple car souvent écrit dans un langage portable dont on a déjà un compilateur pour la cible ☺

- JIT
  - Compile langage intermédiaire vers cible avant exécution
  - Compilateur plus simple que pour langage général

- Analyse lexicale et sémantique
- Génération de code
- Optimisation



- Analyse des langages souvent divisée en 2 phases (couches)
  - Analyse lexicale : extrait symboles de base, mots, nombres, chaînes de caractères
  - Analyse sémantique : construit des phrases avec les mots précédents
- Langages simples : possible de faire les analyseurs à la main mais rapidement pénible... (`scanf`, `if ...`,...) ☹
- Utilisation d'outils de génération d'analyseurs à partir de grammaires des langages d'entrées ☺

Exemples avec `lex`/`flex`/`jlex`

- Prend un fichier `.l` (syntaxe du langage) et génère un `.c` (analyseur syntaxique correspondant) utilisable pour faire un compilateur
- Syntaxe à base d'expression rationnelles et concrètement réalisé sous forme d'automate
- Permet de détecter des erreurs syntaxiques
- Langage C (extrait du compilateur POMPC)

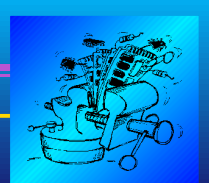
```
\"([^"\\\n]|\\.|\\\n)*\" {
        mystring = yytext;
        glineno = yylineno;
        return STRING;
}
```


```
(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))([eE][+-]?[0-9]+)?[fF] {
        mystring = yytext;
        return FLOAT;
}
(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))([eE][+-]?[0-9]+)? {
        mystring = yytext;
        return DOUBLE;
"||" return OROR;
"&&" return ANDAND;
"==" return EQUAL;
"!=" return DIFFERENT;
";" return SM;
"return" return RETURN;
```


Exemples avec `yacc/bison/jCup`

- Prend un fichier `.y` (grammaire du langage) et géne un `.c` (l'analyseur sémantique)
  - Construit des phrase à partir des mots fournis par l'analyseur syntaxiques
  - Si phrase détectée, appel à fonction utilisateur (par exemple pour construire une représentation abstraite du programme)
  - Un peu un comportement à la SAX
- Permet de détecter des erreurs syntaxiques et sémantiques
- Langage C (extrait du compilateur POMPC)

```
simple_statement :      RETURN me SM {
                    $$ = create_statement(RETURN,$2,0,0,0);
          }
```

```
| SM { $$ = create_statement(SM,0,0,0,0);}
| RETURN SM {$$ = create_statement(RETURN,0,0,0,0);}
| BREAK SM {$$ = create_statement(BREAK,0,0,0,0);}
| CONTINUE SM {$$ = create_statement(CONTINUE,0,0,0,0);}
| ce SM {
        exp *e;
        e = $1;
        $$ = create_statement(EXP_TERM,$1,0,0,0);
        }
| open_block end_block {
        $$ = $2;
        $$->st_modulename = $1->st_modulename;
        $$->st_lineno = $1->st_lineno;
}
| GOTO name SM { $$ = create_statement(GOTO,0,0,0,$2); }
| do_open statement WHILE LP ce RP SM {
```


```
                $$ = finish_statement($1,$2,$5);
        }

statement : simple_statement { $$ = $1; }
        | with_open statement {
                $$ = finish_statement($1,$2,0);
        }
        | while_open statement {
                $$ = finish_statement($1,$2,0);
        }
        | switch_open statement {
                $$ = finish_statement($1,$2,0);
        }
        | for_open statement {
                $$ = finish_statement($1,$2,0);
        }
```

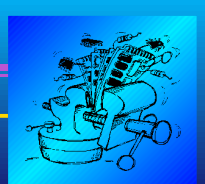

```
        | if_open statement  {
                $$ = finish_statement($1,$2,0);
        }
        | if_open statement_else else statement  {
                $$ = finish_statement($1,$2,$4);
        }
        | label statement {
                $$ = finish_statement($1,$2,0);
        }
        | error SM { $$ = 0;}

else : ELSE { set_active_statement();}
statement_else : simple_statement { $$ = $1;
}
        | while_open statement_else {
                $$ = finish_statement($1,$2,0);
```


```
}
| switch_open statement_else {
        $$ = finish_statement($1,$2,0);
}
| for_open statement_else {
        $$ = finish_statement($1,$2,0);
}
| if_open statement_else else statement_else {
        $$ = finish_statement($1,$2,0);
}
| label statement_else {
        $$ = finish_statement($1,$2,0);
}
```

- Il existe plein d'autres outils, y compris style à la DOM tel que sableCC

- Analyse lexicale et sémantique
- Génération de code
- Optimisation



- Besoin de faire vivre les structures de données du langage de haut niveau : initialisation, ramasse miette,...
- Fonctions des langages (`printf` ou `scanf` du C `stream` de C++,...)
- Organisation de la mémoire
  - Comment sont stockées les structures de données complexes du langage de haut niveau
  - Comment sont gérés les appels de fonction
- Gestion de la mémoire (`malloc()`, `alloca()`,...)
- Interface avec le système d'exploitation

- Générer du code machine équivalent au langage de haut niveau
- Traduction de choses complexes en des choses plus simples
- Utilisation des registres pour manipuler objets et pour stocker informations courantes
- Essayer de générer du code exécutable en parallèle sur les processeurs modernes
- Utilisation du graphe de dépendance pour extraire du parallélisme tout en respectant la légalité (causalité)

```
1  int fibonacci(int n) {
2    if (n <= 1)
3      return 1;
     else
5      return fibonacci( n - 1 )
6           + fibonacci( n - 2 );
7  }
```

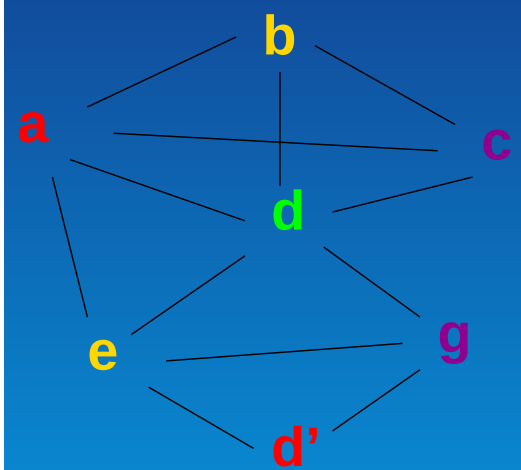
```
fibonacci:
        pushl   %esi
        xorl    %esi, %esi
        pushl   %ebx
        movl    12(%esp), %ebx
.L3:
        cmpl    $1, %ebx
        jle .L4
        leal    -1(%ebx), %eax
        subl    $2, %ebx
        pushl   %eax
        call    fibonacci
        addl    %eax, %esi
        popl    %eax
        jmp .L3
.L4:
        popl    %ebx
        leal    1(%esi), %eax
        popl    %esi
        ret
```

Optimisation *tail recursion*

Exemple de code avant et après compilation

- Unités de stockages les plus rapides de l'ordinateur... ☺
- ...Mais les plus rares... ☹
- Allouer des registres qu'aux ressources les plus utilisées
- Définir des zones de vie aux variables du programme de haut niveau
- Plusieurs variables peuvent partager le même registre si elles ne sont pas vivantes en même temps ☺⇝ Attention au debug... ☹

Source	*Interférences*	*Assembleur*
`a = b + c;`	a, b, c	r1 = r2 + r3
`d = b - c;`	a, b, c, d	r4 = r2 - r3
`e = t[a];`	a, d, e	r2 = t[r1]
`g = d + 3;`	d, g	r3 = r4 + 3
`d = g + 5;`	d', g	r1 = r3 + 5
`g = d + t[g] + e;`	d', g	r3 = r1 + t[r3] + r2



- Utilisation d'un algorithme de coloriage de graphe d'interférence pour allouer les registres (couleurs)
- Si pas assez de registres, stockage de toute manière en mémoire

- Analyse lexicale et sémantique
- Génération de code
- Optimisation

- Optimisations locales
  - Niveau d’une suite d’instructions machines
  - Diminution de l’usage de registres ou d’instructions
  - Simplifications algébriques
- Optimisations globales
  - Au sein d’une procédure entière
  - Utilisation du graphe de dépendance
  - Analyses sémantiques
- Optimisations interprocédurales
  - Tout le programme dans son ensemble
  - Par exemple projet PIPS

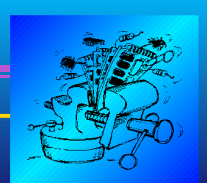
- Propagation de constantes
- Élimination de sous-expressions communes
- Évaluation partielle
- Simplifications algébriques

Compilation de langage ou XML ?

- Utilisation d'un parser pour un langage
  - Configurabilité maximale
  - Langage adapté au problème : programmation, lecture de données,...
  - Facile de changer la syntaxe
- XML
  - Définition d'une syntaxe unifiée
  - Plus besoin d'apprendre à utiliser un parser
  - On n'a plus besoin que d'un seul parser style DOM ou SAX un peu configurable via une DTD ou un Xschema
  - Fichier texte plus facile à éditer que des données binaires
  - Unification du format ⇝ ↗ portabilité
  - L'utilisateur doit s'adapter à XML ! ☹

- Peut-on vraiment avoir des langages de programmation en XML ? Oui... XSLT ☺

- Domaine « méta » : écrire des programmes qui manipulent des programmes
- Permet de comprendre beaucoup de domaines de l'informatique
- Facilite l'apprentissage des langages
- Domaine transverse avec plein des problèmes algorithmiques passionnants réutilisables ailleurs
- Permet de mieux comprendre comment exploiter les outils et les ordinateurs : optimisation, économies,...
- Permet d'améliorer les interfaces d'entrées de programmes
- Projets GABI, PHRASE... au Département Informatique



List of Slides