Projet 2: un compilateur

cours: Daniel Hirschkoff
TPs: Ghislain Charrier et Bogdan Pasca

Présentation du cours

Programmation, compilation

- cours orienté pratique
- quelques indications pour démarrer, puis se prendre en main
- évaluation: vous rendez un *compilateur*
  - un programme qui prend en argument un programme (le source), et renvoie un programme (l'objet)
  - programme que l'on compile à l'aide d'un compilateur...

Règles du jeu

- en binôme (au plus un monôme)
- travail clairement affecté au sein du binôme
- questionnaire
  - “projet 1 + . . .”

- *si vous ne savez pas encore si vous allez valider ce module, merci de l'indiquer sur la fiche*
- stratification des étudiants

Évaluation – étapes de la construction

- jalons dans le semestre
  - 15-21 février: vacances d'hiver
  - 12-18 avril: vacances de pâques
  - 21 mai: fin des cours (24-28: examens)
- quatre étapes
  1. analyses lexicale et syntaxique — 24 février
  2. représentation intermédiaire, structures à l'exécution — 29 mars
  3. *un* programme tourne — 26 avril
  4. le compilateur est prêt — 18 mai
- ces dates sont déplaçables (interférences avec d'autres cours)
  au plus tard 15 jours à l'avance

Quel compilateur

- par défaut: Pascal - $\rightsquigarrow$ RISC
  - dans quel langage? cf. transparent suivant
  - solution la plus raisonnable, où vous apprendrez ce qu'il faut apprendre

- sinon
  - Caml - $\rightsquigarrow$ C

    en Caml, pour apprendre le C
  - Haskell - $\rightsquigarrow$ G-machine

    . . . sujet à "censure" de ma part
- autres

  (ne) parlementons (pas)

Dans quel langage développer le compilateur

- trois choix OCaml C Java
- OCaml préconisé
  - pas de problèmes de pointeurs ou de `malloc`
- C ou Java
  - vous connaissez bien l'un de ces langages, ou alors
  - vous êtes plutôt à l'aise avec les bases de la compilation et vous voulez apprendre l'un de ces langages
- *autre*: parlementons

Références

- je vous conseille le trypitique d'Andrew Appel
  - disponibles à la bibliothèque
  - la première moitié du livre essentiellement
- google vous renseignera (notamment pour `bison`, `flex`, `ocamllex`, etc.)
- vos chargés de TPs et de cours aussi

Déroulement du cours

- cette semaine **pas de TP**
- quelques séances de cours (3, 4)
- les TPs démarrent un peu décalés
- puis rien que des TPs

allons-y

Les étapes de la compilation

code source $\rightsquigarrow$ code objet
(dans un fichier) (dans un autre fichier)

programme source

↓ analyse lexicale, analyse syntaxique

arbre de syntaxe abstraite

↓ analyses statiques (typage,. . .)

arbre décoré

↓ traductions, optimisations

code exécutable (langage machine)

lecture du fichier source: on récupère un *flot de caractères*

Analyse lexicale

chapitre 2 dans le livre d'A. Appel

Les deux étapes

- analyse lexicale

  flot de caractères (source) $\longrightarrow$ flot de *lexèmes*

  - lexème (*token*): "atome" du langage
  - typiquement:
    - mots-clefs (`let`, `begin`, `while`,...)
    - symboles réservés (`(`, `+`, `;;`, `;`,...)
    - identificateurs (`f`, `toto`, ...)

- ainsi `32*52+(let x = 5 in x*x)`

  $\longrightarrow$ `INT(32), MULT, INT(52), ADD, LPAREN, LET, ID("x"), EGAL, INT(5), IN, ID("x"), MULT, ID("x"), RPAREN`

  (`INT` et `ID` ont un *attribut*, entier et chaîne de caractères respectivement)

- analyse syntaxique

  flot de lexèmes $\longrightarrow$ *arbre de syntaxe abstraite*

  $\longrightarrow$
  ```
  Add( Mult(Int(32), Int(52)),
       Let("x", Int(5), Mult(Var("x"), Var("x"))) )
  ```

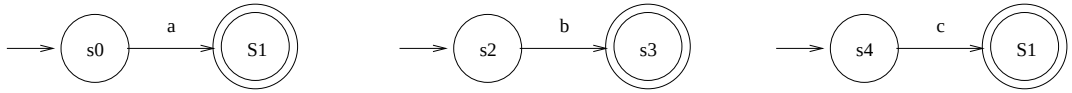
  - étape intermédiaire: arbre d'analyse syntaxique *(parse tree)*

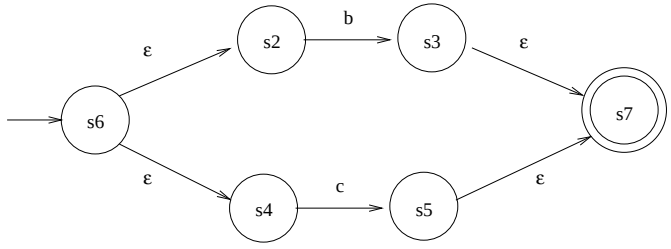
Analyse lexicale

chaque lexème est décrit par une *expression régulière*
principaux éléments (syntaxe de `ocamllex`):

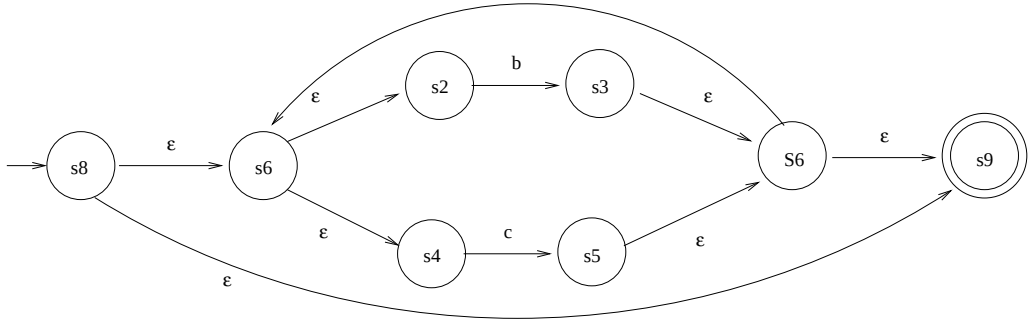
- caractère `'$'`, chaîne de caractères `"else"`
- intervalle `['0'-'9']` *(un chiffre)*
- disjonction `['\t' ' ']` *(tabulation ou espace)*
- juxtaposition `['A'-'Z']['a'-'z' 'A'-'Z']`
  (mot de 2 lettres commençant par une majuscule)
- répétitions: `+` signifie au moins 1, `*` zéro ou plus
  `['a'-'z']+['a'-'z' '0'-'9']*`
  (ça commence par une lettre puis des lettres ou des chiffres)
- (re)disjonction `a* | b*`

Expression régulière ↔ automate non déterministe

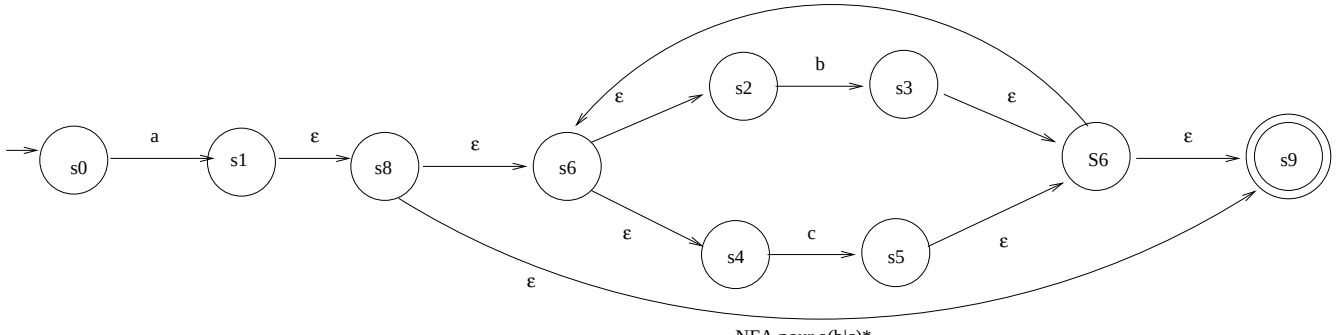




NFA pour b|c



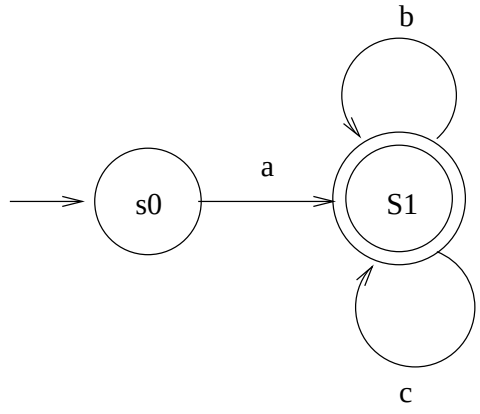
NFA pour (b|c)*



NFA pour a(b|c)*

Déterminisation, minimisation

- à partir de l'automate du transparent précédent, on dispose de procédures pour *déterminiser* l'automate (explosion du nombre d'états), puis le *minimiser*



- on aboutit à

© T. Risset, F. de Dinechin

- comment implémenter l'automate résultant?
  - une table (très creuse)

état	a	b	c	d	e
e1	-	e2	e3	-	-
e2	e4	-	-	-	-
e3	-	-	e3	-	-

  - éliminer les états: un plat de spaghetti, fait de `if` et de `goto`

L'automate en action

figures 2.4 page 22 et 2.5 page 24

Pour résumer

Outils logiciels pour faire tout ça

- en pratique, on utilise des **compilateurs**
  - d'expressions régulières
    vers un automate
  - de règles de grammaire
    vers un automate à pile
- des couples (analyseur lexical)/(analyseur syntaxique) sont monnaie courante
  - ici: ocamllex/ocamlyacc
  - en TP: une autre version

`ocamllex`

- on met tout ça dans un fichier `lexer.mll`, en associant un lexème à chaque expression régulière
  - quand plusieurs règles s'appliquent, c'est celle qui donne le lexème le plus long qui gagne, et s'il y a encore ambiguïté c'est la première règle dans l'ordre où elles sont écrites
  - DÉMO `lexer.mll` *(lexer = analyseur lexical)*
- un outil comme `ocamllex` transforme une liste d'expressions régulières en un *automate* déterministe, représenté par sa table de transitions
  résultat de la compilation: fichier `lexer.ml`
  - `ocamllex`, mais aussi (surtout) `lex, flex`

Forme d'un fichier Lex

- structure du fichier
  1. déclarations (C, Caml, ..)
     `#open Toto`
     c'est là qu'on fait référence à la définition des lexèmes
  2. définitions / abbréviations
     `digits [0-9]+`
  3. expressions régulières accompagnées d'actions
     - on associe un lexème à chaque expression régulière
     - certains lexèmes ont un *attribut*:
       on explique comment le calculer
       `| ['0'-'9']+ as k  { INT (int_of_string k) }`
- l'approche est très similaire en C:

```
[1-9][0-9]*   {
              yylval.iValue = atoi(yytext);
              return INTEGER;
              }

[ \t\n]+       ; /* ignore whitespace */
```

Analyse syntaxique

chapitre 3 dans le livre d'A. Appel

Analyse syntaxique

- l'analyse syntaxique se fonde sur une approche plus puissante: **règles de grammaire**
  - les règles de grammaire font intervenir les lexèmes et des "variables" (les non terminaux)
  - exemple de grammaire:

    $E \quad ::= \quad K \;\mid\; E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; \texttt{let}\ Id = E\ \texttt{in}\ E$

    - E: non terminal (il peut y en avoir plusieurs)
    - $K, \texttt{let}, Id, +, *, (,), \texttt{in}, =$: lexèmes
    - le $\mid$ se lit comme un "ou"

présentation alternative:

$E \to K \qquad E \to E + E \qquad E \to E * E \qquad E \to (E) \qquad E \to \texttt{let}\ Id = E\ \texttt{in}\ + E$

on parle de *grammaire hors contexte*

- analyse lexicale: on transforme un *flot* de lettres en un *flot* de "grosses lettres" (les lexèmes)
- analyseur syntaxique (ou *parser*): applique les règles de grammaire pour *reconnaître* une suite de lexèmes
  - on change la structure: un *flot* (de lexèmes) devient un *arbre*

Reconnaître une séquence de lexèmes

- l'idée est de construire un *arbre de dérivation* permettant de reconnaître la suite de lexèmes
  - grammaire: $P ::= P + P \mid P * P \mid K$ K un entier
  - soit le flot `32, +, 26, *, 2`
  - on peut faire deux dérivations:

$$\begin{array}{lcl} P & \to & P + P \\ & \to & K_{32} + P \\ & \to & K_{32} + P * P \\ & \to\to & K_{32} + K_{26} * K_2 \end{array}$$

ou alors

$$\begin{array}{lcl} P & \to & P * P \\ & \to & P * K_2 \\ & \to & P + P * K_2 \\ & \to\to & K_{32} + K_{26} * K_2 \end{array}$$

deux arbres différents: *ambiguïté*

  - aucune dérivation possible en revanche pour `9 * 1 + + 1`

Ce que fait le parser

$E \quad ::= \quad E + E \;\mid\; E * E \;\mid\; (E) \;\mid\; a \;\mid\; b \;\mid\; c$ `a+b*c`

pile	entrée	action
$	$a + b * c$	shift
$a	$+b * c$	reduce : $E \rightarrow a$
$E	$+b * c$	shift
$E+	$b * c$	shift
$E + b	$*c$	reduce : $E \rightarrow b$
$E + E	$*c$	shift **(très malin)**
$E + E*	c	shift
$E + E * c	$	reduce : $E \rightarrow c$
$E + E * E	$	reduce : $E \rightarrow E * E$
$E + E	$	reduce : $E \rightarrow E + E$
$E	$	accept

- construction d'une dérivation *"par la droite"*

Mettre au point une grammaire

- le langage de programmation, c'est la grammaire
- bugs=conflit 'shift/reduce': ambiguïté de la grammaire

DÉMO

```
12:  shift/reduce conflict (shift 8, reduce 4) on PLUS
12:  shift/reduce conflict (shift 9, reduce 4) on TIMES
state 12
    expr :  expr .  PLUS expr (4)
    expr :  expr PLUS expr .  (4)   le cas qui pose pb
    expr :  expr .  TIMES expr (5)

    PLUS shift 8   ce qui a été choisi
    TIMES shift 9   ce qui a été choisi
    RPAREN reduce 4
    EOL reduce 4
```

- lever l'ambiguïté:
  - modifier la grammaire (p.ex. en ajoutant des non terminaux)

    de
    S → if E then S else S
    S → if E then S

    à
    S → M *matched*
    S → U *unmatched*
    M → if E then M else M
    M → autres
    U → if E then S
    U → if E then M else U
  - introduire des précédences

    (dans l'exemple du transparent précédent, le $*$ a une précédence plus grande que celle de la règle qu'il veut réduire, et donc on fait shift)

ocamlyacc

- la version Caml de `yacc` (cf. aussi `bison`)
- générateur d'analyseurs syntaxiques
- on indique les lexèmes et les règles de grammaire
- $+$ des artefacts, pour gérer les questions de priorité des règles, signaler des erreurs de syntaxe de manière informative, etc.
- DÉMO `parser.mly`

Tout mettre ensemble

```
let lexbuf = Lexing.from_channel stdin
let parse () = Parser.main Lexer.token lexbuf

Parser.main :
    (Lexing.lexbuf -> Parser.token) -> Lexing.lexbuf -> Expr.expr = <fun>
```

- on prend l'entrée standard (`stdin`, le clavier)
- on fait l'analyse lexicale, `lexbuf` est un flot de lexèmes
- l'analyse syntaxique renvoie un `Expr.expr` (cf. `parser.mly`)
  l'application `parse ()` renvoie la prochaine expression reconnue sur la saisie au clavier
- DÉMO `main.ml`

- cf. la doc de Caml (*Lexer and parser generators*)

Constitution des binômes

SPIM

Présentation de SPIM

- SPIM simule une machine MIPS
  - n'est pas fidèle à la machine simulée en termes de performances
  - propose des *pseudo-instructions* (qui sont gérées de manière astucieuse par la vraie architecture, pas par le simulateur)
- on voit l'assembleur de SPIM comme un langage de programmation, avec son modèle d'exécution
- documentation de SPIM disponible (`spim.ps`)
  `xspim` pour cliquouiller
- le "moteur"
  - unité de calcul (CPU)
  - 32 registres, avec des conventions de nommage et d'utilisation: les respecter (si on était en vrai, il faudrait les respecter pour la compatibilité avec d'autres codes)

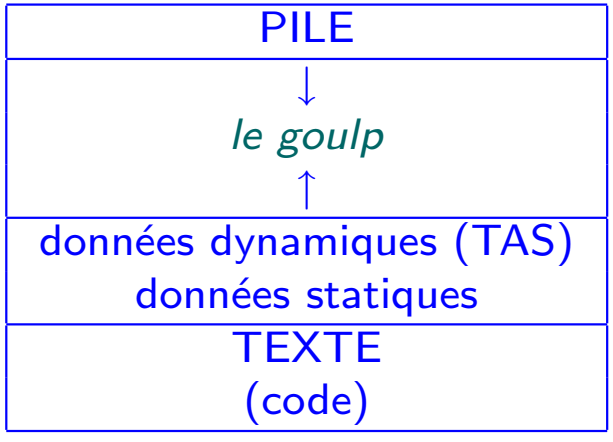
SPIM: ingrédients

- instructions en SPIM: un langage impératif

```
            bla
            bli     # commentaire captivant
machin:     blu
            blo
```

  tout réside quelque part en mémoire

- organisation de la mémoire



Instructions de SPIM (aperçu)

- opérations: elles se font uniquement sur les registres

`add` ri, rj, rk	la somme de rj et rk dans ri
`add` ri, rj, n	rj+n dans ri
`addu`	même chose en traitant ses arguments comme des entiers non signés
`div` ri, rj	divise ri par rj; quotient dans lo, reste dans hi
`seq` ri, rj, rk	ri=1 si rj=rk, 0 sinon
`seq` ri, rj, n	ri=1 si rj=n, 0 sinon

- sauter

`j` *lab*	saut à l'instruction labélisée par *lab*
`jal` *lab*	saute en *lab* après avoir noté en *ra* l'adresse de l'instruction suivante
`beq` ri, rj, *lab*	va en *lab* si ri=rj
`beq` ri, n, *lab*	va en *lab* si ri=n

Instructions de SPIM (suite de l'aperçu)

- interactions avec la mémoire

`lw` ri, add	met dans ri le mot situé en add
`la` ri, add	met dans ri l'*adresse* add
`li` ri, k	met k dans ri

formes possibles des adresses

`(rj)`	adresse contenue dans rj
`k(rj)`	k + adresse contenue dans rj
`lab`	adresse du label lab
`k`	l'entier k

`sw` ri, add	recopie le contenu de ri en add

SPIM – autres

- primitives prédéfinies: appels système

```
li $v0, 1      1 signifie print_int
li $a0, 32     32 est la valeur à imprimer
syscall
```

- directives
  - `.text` BLA mettre BLA dans la zone de texte
  - `.data` BLA mettre BLA dans la zone de données

```
        .data                                                   des données
str:    .asciiz "le plus difficile, ce n'est pas\n"             un label, des données
        .text                                                   du code
        li $v0, 4                                               4 signifie print_string
        li $a0, str
        syscall
```

Génération de code

Redescendre de l'arbre

- analyses lexicale et syntaxique: le *front end* d'un compilateur
- *back end*: génération de code

  à partir d'un arbre (pas trop l'arbre de syntaxe abstraite, mais le résultat de potentiellement beaucoup de transformations et d'analyses) on engendre du code *linéaire*

La pile

- la structuration de programmes en fonctions qui s'appellent entre elles rend naturelle l'utilisation d'une **pile**
- sur la pile: *enregistrements d'activation*
  - ce sont des enregistrements *(records)*
  - leur présence sur la pile correspond à la période durant laquelle l'appel de la fonction est actif
- retour aux expressions arithmétiques

```
type expr =
    Const of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | IfEq of expr*expr*expr*expr
                    (* if e1 = e2 then e3 else e4 *)
```

Évaluer les expressions arithmétiques sur la pile

une **machine abstraite à pile**: modèle d'exécution intermédiaire

(le modèle, pas l'exécution)

- pour évaluer `Add(Const 5, Const 7)`, on engendre

```
push 7
push 5
add
```

  - l'appel à `add`
    - dépile les arguments
    - fait le calcul
    - met le résultat sur le sommet de la pile
  - l'expression `Add(Const 5, Const 7)` est traduite en une suite d'instructions de la *machine abstraite*
    - la pile + un jeu simple d'instructions
    - abstraction: on ne rentre pas dans les détails sur l'architecture, sur comment est implémentée la pile, etc. *(portabilité, p.ex. Java)*

- le sommet de la pile est accédé très fréquemment

  p.ex. trois accès lors de l'exécution de `add`

Un peu plus de détails

- Raffinement de la machine abstraite
  - on dispose d'un *registre*, l'accumulateur, nommé `acc`
    - registre: accès rapide (il y en a généralement plus d'un!)
  - les calculs se font sur l'accumulateur
  - la pile n'est utilisée que pour stocker les arguments
- cela donne (nouveau jeu d'instructions)

```
acc <- 7
push acc
acc <- 5
acc <- acc + top    (* opération atomique *)
pop
```

Exemple un peu plus grand: `3 + (7 + 5)`

```
acc <- 3
push acc
 acc <- 7
 push acc
 acc <- 5
 acc <- acc + top
 pop
acc <- acc + top
pop
```

- *leitmotive*
  - on rend la pile dans l'état de propreté dans laquelle on l'a trouvée
  - le résultat est en `acc` à la fin
- but du jeu
  - jeu d'instructions simples
  - traitement uniforme (ici: évaluer une addition)
  - les instructions correspondent à des opérations élémentaires en machine

Raffinement encore: 'programmer' la pile soi-même

on atteint le niveau de SPIM

```
acc <- 7              li $a0, 7
push acc              sw $a0, 0($sp)
                      add $sp, $sp, -4
acc <- 5              li $a0, 5
acc <- acc + top      lw $t1, 4($sp)
                      add $a0, $a0, $t1
pop                   add $sp, $sp, 4
```

- `$sp` registre contenant l'adresse de la fin de la pile
- `li reg, imm` *met le registre `reg` à `imm`*
- `sw reg1, decalage(reg2)` *stocke le contenu de `reg1` a l'adresse (contenu de `reg2`)+`decalage`*
- `add reg1, reg2, imm` *ajoute `imm` au contenu de `reg2` et stocke dans `reg1`*
- `add reg1, reg2, reg3` *ajoute les contenus de `reg2` et `reg3` et stocke cela en `reg1`*
- `lw reg1, decalage(reg2)` *charge ce qui est stocké en l'adresse en `reg2` + `decalage` dans `reg1`*
- le résultat est en `$a0` à la fin

Le backend: de l'arbre au fichier objet

- `Const i` $\longrightarrow$

```
print_string "li $a0 "; print_int i;
print_string "\n";
```

- `Add (e1,e2)` $\longrightarrow$

```
engendre(e1);
print_string("sw $a0, 0($sp)\n");
print_string("add $sp, $sp, -4\n");
engendre(e2);
print_string("lw $t1, 4($sp)\n");
print_string("add $a0, $a0, $t1\n");
print_string("add $sp, $sp, 4\n");
```

Branchements (in)conditionnels

```
if e1=e2 then e3 else e4
```

```
                ‘‘engendre(e1)’’
                sw $a0, 0($sp)
                add $sp, $sp, -4
                ‘‘engendre(e2)’’
                lw $t1, 4($sp)
                add $sp $sp 4
                beq $a0, $t1, branche_vrai
branche_faux:   ‘‘engendre(e4)’’
                b fin_if
branche_vrai:   ‘‘engendre(e3)’’
fin_if:         ...
```

exercice: boucles

Les fonctions

- l'enregistrement d'activation d'un appel de fonction contient
  - les paramètres d'appel de la fonction nécessaires à l'exécution de l'appel
  - $+$ de quoi entrer et sortir dans la fonction
- deux morceaux de code à engendrer
  1. préparer la pile avant l'appel et lancer l'appel
  2. corps de la fonction: faire le calcul, et sortir de la fonction

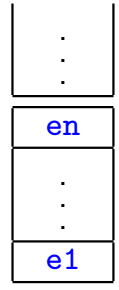
```
corps_f:   inst1
           inst2
           ⋮
```

Appel de la fonction: `f(e1,...,en)`

```
‘‘engendre(en)’’
sw $a0, 0($sp)
add $sp, $sp, -4
⋮
‘‘engendre(e1)’’
sw $a0, 0($sp)
add $sp, $sp, -4
jal code_f
blibli
```

on stocke sur la pile les arguments de la fonction

puis on y va (`code_f`)



- `jal`: saute à l'adresse donnée, et stocke l'adresse qui suit le `jal` dans un registre spécial, `$ra`
- lorsqu'on saute,
  - `$sp` pointe vers le début de la pile; celle-ci a grandi
  - `$ra` contient l'adresse de l'instruction a exécuter en retournant de l'appel
  - il faudra retourner en `blibli` lorsque `f` aura terminé son calcul

Au sein de la fonction `f(x1,...,xn)=corps`

```
code_f:   sw $ra, 0($sp)
          add $sp, $sp, -4
          ``engendrer(corps)''
          lw $ra, 4($sp)
          add $sp, $sp ``4*n+4''
          jr $ra
```

- on stocke ce que contient `$ra` dans la pile
  l'enregistrement fait donc `4*n+4` octets
- on calcule le corps de la fonction
- on remet l'adresse de retour dans `$ra`
- on fait décroître la pile
- on sort (`jr`: sauter à l'adresse contenue dans un registre)
  $\rightsquigarrow$ on exécutera `blibli`

- on ne voit pas apparaître `$a0`: il est dans `corps`
- optimisation: si `corps` ne fait pas d'appel de fonction, ne pas toucher à `$ra`
- tout ceci est donné à titre d'exemple
  en réalité, on fait plus compliqué: les arguments sont passés si possible dans des registres, les registres sont sauvegardés (par l'appelant, par l'appelé)

Quelques mots encore

- voilà notre premier “compilateur” : un front end, un back end
  - DÉMO
  - *on aurait pu tout faire en une passe. . .*
    - au lieu de `| expr PLUS expr { Add($1,$3) }`,
      mettre `| expr PLUS expr { $1 + $2 }`

    . . . *seulement tant que le langage est banal (`let f x = ...`)*
- il s’agit ici d’un exemple
  - autres jeux d’instructions
  - autres conventions d’appel pour les fonctions

    (si possible, tous les paramètres dans les registres)

  ⋮
- ce n’est pas fini: optimisations
  - *inlining*: `let f x = x+1, for i=1 to 2 do BLA`
  - *“jump around”*

  ⋮

Votre compilateur

- front-end: dépend du langage choisi
- back-end, génération de code:
  - SPIM pour Pascal-
  - C pour Caml- (programmer la pile en C)
  - à définir pour Haskell-

Entre le front-end et le back-end

Organisation

- les groupes sont constitués
- la semaine prochaine: vacances
- première deadline: le 26 février
  - analyse lexicale
  - analyse syntaxique
  - affichage de l'AST
  - davantage si vous le sentez, bien sûr (analyse statique: portée, typage, warnings...)
- changement de créneau à la rentrée:
  - ok pour tout le monde lundi 16h-18h?
  - mardi 13h30-15h30 quand il n'y a pas Sieste, et 16h-18h sinon? (G.Charrier ne peut pas mardi 16-18)

Table des symboles – analyse statique (chap. 5 livre A. Appel)

- tout au long de la phase de compilation, on manipule des structures de données auxiliaires
- *table des symboles*, implémentée comme une table de hachage des informations sur tous les identificateurs utilisés par le programme
  - nom de type
  - déclaration globale: type, localisation
  - fonction: son nom, le nom de son label d'entrée
    ses paramètres: leur nom, leur type, leur taille, leur localisation
    ses variables locales
    autres informations (récursive, terminale)

  à vous d'affiner/préciser
- table des symboles de départ, avec ce qui est prédéfini (constantes, etc.)
- analyse statique: vous avez fait Prog 1

  on range un symbole nouveau, on vérifie ensuite ses usages

Représentation intermédiaire (chap. 7 livre A. Appel)

- il est naïf d'engendrer du code en une passe `a = b+c`
  - choisir la bonne instruction assembleur
  - distribution des données entre registres et mémoire (p.ex., si on doit récupérer `c` en mémoire)
  - traduction de `read(a)`: `a` est un global ou une variable locale?
- 7.1 dans le livre d'A. Appel offre une bonne illustration:
  - constantes entières, labels
  - TEMP(t) un temporaire (ou *registre virtuel*)
  - BINOP(o,e1,e2) opération binaire
  - CALL(f,arglist) appel de fonction avec ses arguments
  - MOVE (avec MOVE(TEMP t,e) et MOVE(MEM e1, e2))
  - JUMP(e,labs) labs liste les labels où l'on peut sauter (selon l'évaluation de e)
  - SEQ(s1,s2) la séquence
- le but: (cf chap. 8 livre A. Appel)
  - traduire et séquentialiser l'arbre produit par l'analyse statique et obtenir un mix arborescent / linéaire
    - les branchements (if then else, boucles) sont explicites (gotos)
    - on cherche à manipuler des *blocs de base*, à l'intérieur desquels sont exécutées des séquences d'instructions

L'allocation de registres

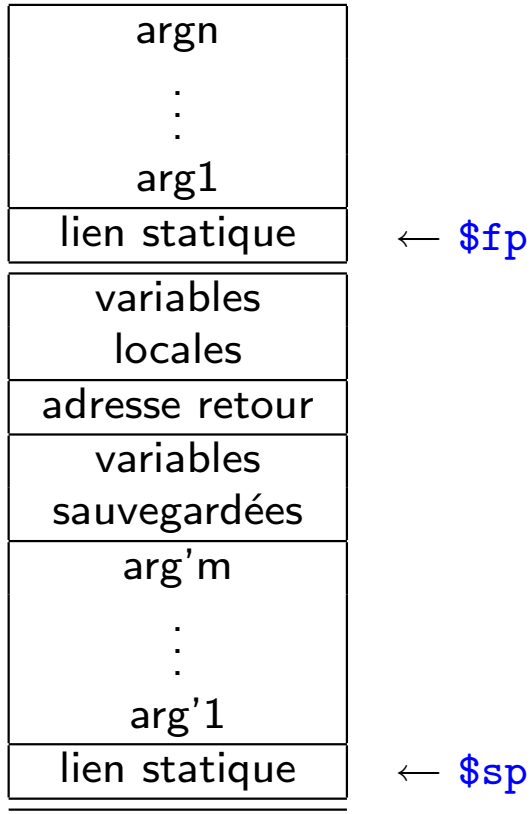
- on commence par faire comme s'il y avait une infinité de registres (temporaires, registres virtuels)
- on fait une analyse plus ou moins maline pour affecter des registres virtuels à des registres réels (chaps. 10& 11 livre A. Appel)
  la *durée de vie* des données est un élément central pour ce faire
- on réécrit: certains registres virtuels peuvent se retrouver en mémoire (sur la pile)
- NB: passage par référence: on met dans un temporaire l'*adresse* de l'argument
- la table des symboles joue un rôle d'*environnement* durant la phase de compilation
  elle renseigne en particulier, lors d'étapes ultérieures de compilation, sur l'*accès* à une variable:
  - temporaires pour les variables locales et les arguments
  - indices pour les variables globales
- au passage: une structure de données pour le code, pas un `print` directement (*post-optimisations* sur l'assembleur produit)

Code intermédiaire: fonctions

- les trois langages proposés (Pascal Caml Haskell) permettent de définir des fonctions à l'intérieur de fonctions

```
let f x =
  let g y = y*x in
..
```

- du coup, une forme plus sophistiquée pour les enregistrements d'activation (sur la pile)



Code intermédiaire: fonctions (2)

- $fp est utile pour accéder à l'environnement de la fonction appelante
  la fonction interne accède aux paramètres de la fonction externe grâce à $fp
  - *lien statique*: on passe dynamiquement ce lien à une fonction appelée, et celle-ci accède aux paramètres de la fonction appelante en calculant *statiquement* des décalages
  - on peut voir ça comme une implémentation des environnements à l'aide de tableaux dans la pile
  - cas particulier d'un appel récursif
- fonctions locales non retournées: on peut les allouer sur la pile, et chaîner les environnements
  - le fait qu'elles ne soient pas retournées fait que leur durée de vie dynamique n'excède pas leur portée lexicale
  - si on peut retourner une fonction? clôtures
- les fonctions locales peuvent être rendues globales par *lambda-lifting*

Temporaires et appels de fonctions

- au moment de l'appel d'une fonction, les arguments sont dans des temporaires de l'appelant; après l'appel, ils sont dans des temporaires de l'appelé (pas nécessairement les mêmes)
- ainsi, les arguments voyagent: lorsqu'une fonction f appelle une fonction g (avec un seul argument pour simplifier):
  - la valeur de l'argument est préparée dans un temporaire tf (vue de l'appelant) et transféré dans a0 avant l'appel à g.
  - à l'entrée de g la valeur de a0 est placée dans un temporaire tg (vue de l'appelé).
  - on a donc trois emplacements pour l'argument: tf (avant), a0 (pendant) et tg (après).
- ces mouvements sont nécessaires dans le cas général pour libérer les registres conventionnels pour d'autres appels de fonctions.
- mais ils seront éliminés, si possible, par l'allocateur de registres en choisissant le même registre a0 pour tf et tg.

Fonctions: prologue et épilogue

- typiquement,
  - Le *prologue* doit:
    1. Allouer k mots en pile.
    2. Sauver les registres *callee-saved*
    3. Placer les arguments (reçus à leur position conventionnelle, dans les registres puis dans le frame parent) dans les temporaires réservés à cet effet comme décrit dans le frame.
  - Symétriquement, l'*épilogue* doit:
    1. Placer le résultat éventuel dans le registre conventionnel v0
    2. Ressusciter les registres *callee-saved*
    3. Libérer les k mots en pile.
    4. Sauter à l'adresse de retour.
- l'allocation de registres a lieu vers la fin
  on retarde donc l'écriture des prologues et épilogues, ou alors on les écrit avec des constantes à définir (`taille_frame`)

Au sujet de Pascal-

- `procedure f(x:integer, var y:integer)`
  - procedure, ca veut dire fonction
  - `y` est passé *par référence*
    (`f` a le pouvoir de modifier la valeur de `y`: pas le sien, celui de l'appelant)
- on peut définir des types

```
const n=100;
type table = array[1..n] of integer;
var A: table;
```

- on peut passer des tableaux à des fonctions
- en cours: fonctions qui retournent des tableaux?