

# Bifton – Projet compilation

Samuel Mimram    Julien Cristau

ÉNS Lyon

10 mai 2004

## Analyse du code source

Analyse lexicale

Analyse syntaxique

## Représentations intermédiaires

Pascal

Assembleur

## Génération de l'assembleur

i386

Sparc

## Extensions / pour aller plus loin

- ▶ simple
- ▶ seul problème : gérer les sauts de ligne

- ▶ utilisation intensive des priorités et de l'associativité pour simplifier la grammaire

if\_statement:

```
| IF expression THEN statement  
  %prec NOELSE { If($2, $4, new_statement Noop) }  
| IF expression THEN statement  
  ELSE statement { If($2, $4, $6) };
```

- ▶ bornes des tableaux constantes
- ▶ tableaux de tableaux, etc.
- ▶ nous gardons la localisation des éléments dans le code source
- ▶ les var sont émulés par des pointeurs

## Vérifications :

- ▶ vérification de la cohérence : identifiants déjà définis, types, arité, bornes des tableaux
- ▶ chronologiques, sauf fonctions récursives
- ▶ erreurs précises (cause, localisation)
- ▶ pretty-printer
- ▶ les fonctions de la librairie standard sont prédéfinies

## Génération de la RI :

- ▶ suppression des fonctions inutiles
- ▶ précalcul des expressions constantes (y compris les if)
- ▶ structurée en blocs de base (meta\_instr : While, If, End)
- ▶ on garde l'information de profondeur lexicale

## La RI :

- ▶ des instructions de plus ou moins haut niveau (Fun / Push, Call)
- ▶ variable (registre) / variable\_def (type, profondeur, valeur, position) (→ attention aux tableaux)
- ▶ registres virtuels pour les flags

- ▶ choix de l'assembleur i386
- ▶ structure de la pile (prélude / postlude des procédures)
- ▶ le display

L'allocation des registres :

- ▶ réécriture des instructions de haut niveau :

$\text{Fun} \rightarrow \text{Push\_?} + \text{Call}$

- ▶  $\text{av} \Rightarrow \text{check\_avail}, \text{assert\_avail}, \text{used\_and\_replaced}$   
(attention aux tableaux)
- ▶ passe d'optimisations locales (mov inutiles, arithmétique élémentaire)

```
| Add_i(v1, v2, v3) ->
  if is_constant v2 then
    (
      assert (not (is_constant v3));
      let av', i = (alloc_instr !av (commute_instr instr)
                  needed_defs) in
        av := av'; i
    )
  else
    (
      check_avail v3 []
      (fun r3 ->
        used_and_replaced v2 r3;
        let r2 = get_var_reg v2 in
          set_available v1.var_def r2
        );
      [instr]
    )
```

## Génération de l'assembleur :

- ▶ Reg\_flag
- ▶ utilisation de l'adressage indirect : Pos\_base, Pos\_arr, Pos\_rec, Pos\_pointed, etc.
- ▶ `movl -12(%ebp), %eax`  $\rightarrow \alpha(r_1, r_2, \beta)$  pour  $\beta \in \{1, 2, 4, 8\}$ , multiplications avec des `sall` sinon
- ▶ deux types de « push »
- ▶ cas particuliers nombreux et pénibles (multiplication, division)
- ▶ gestion des portées lexicales (display)

## Une architecture tout à fait différente

- ▶ beaucoup plus de registres
- ▶ notion de fenêtre de registres : on ne peut accéder qu'aux registres de la procédure courante, les autres ne sont plus disponibles, mais seront restaurés au retour de la procédure
- ▶ un assembleur 3 adresses : aucun accès implicite à la mémoire, tous les load/store sont séparés des autres instructions
- ▶ un langage plus régulier avec beaucoup moins de cas particuliers

```
| Mult_i(v1, v2, v3)
| And(v1, v2, v3)
| Or(v1, v2, v3)
| Div_i(v1, v2, v3)
| Mod_i(v1, v2, v3)
| Add_i(v1, v2, v3)
| Sub_i(v1, v2, v3) ->
    assert_avail v2 [];
    assert_avail v3 [get_var_reg v2];
    ignore(alloc_reg v1 [get_var_reg v2; get_var_reg v3]);
    [instr]
```

Extensions :

- ▶ les pointeurs et `new`
- ▶ les fonctions
- ▶ optimisations diverses et variées

Pour aller plus loin :

- ▶ les fonctions
- ▶ BURS
- ▶ analyses data-flow (spill)