ECOLE SUPERIEURE D'INGENIEUR DE RENNES

Projet de compilation

3ème délivrable

Olivier Ridoux

UNIVERSITÉ DE RENNES 1

Production de code exécutable

Troisième délivrable (1)

- Finaliser le *back-end*
  - produire un fichier **exécutable**
- Production d'un code exécutable
  - **traduire le code 3 adresses en du code exécutable**
  - **programmer le système langage (*runtime* ou libWH)**

Troisième délivrable (2)

- Pour une démo
  - lire des paramètres d'entrée
  - afficher les résultats

[shell] % whc f.wh

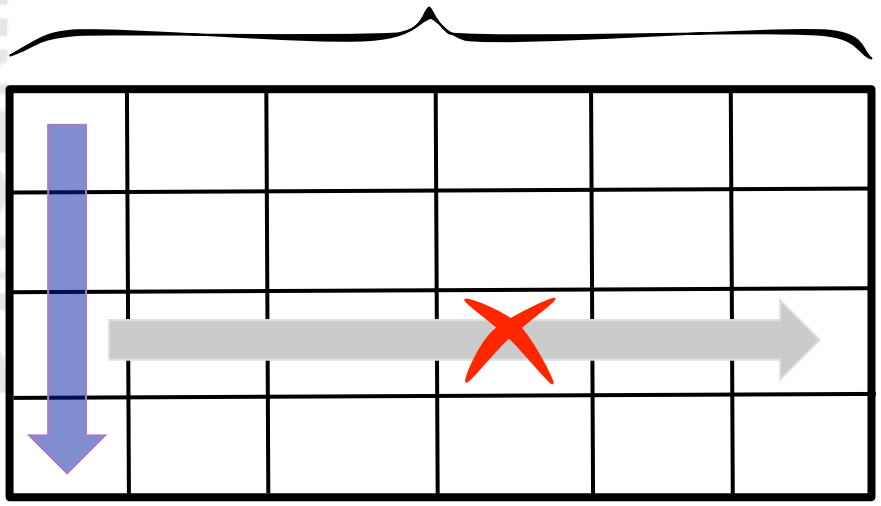
[shell] % f param$_1$ ... param$_n$

résultat

Stratégie générale d'agilité (1)

Toujours faire en sorte que du code exécutable puisse être produit et exécuté

Fonctionnalités : nop, nil, cons, if, while, ...



lire paramètres

lancer processus

run-time

afficher résultats

...

Stratégie générale d'agilité (2)

- Le plus petit programme !

  function dummy1:

  read X % nop % write X

- Le second plus petit programme

  function dummy2:

  read X % X := nil % write X

- Le suivant

  function dummy3:

  read X % Y := (cons nil X) % write X, Y

Stratégie générale d'agilité (3)

- Variantes

function dummy4:

read X, Y % nop % write Y, X

function dummy5:

read X, Y % X, Y := Y, X % write X, Y

function dummy6:

read X, Y, Z % X, Y, Z := Y, Z, X % write X, Y, Z

Stratégie générale d'agilité (4)

- Pour tester la boucle FOR

```
function busy1:
read X % for X do X := (incr X) od % write X
function busy2:
read X %
    for X do for X do X := (incr X) od od
% write X
function busy3:
read X %
    for X do for X do for X do X := (incr X) od od od
% write X
```

Stratégie générale d'agilité (5)

- Pour tester la gestion des symboles

function iseq:

read X, Y % Z := X =? Y % write Z

[unix] % iseq a b
nil

[unix] % iseq a a
(cons nil nil)

function iseqtoa:

read X % Z := X =? a % write Z

[unix] % iseqtoa b
nil

[unix] % iseqtoa a
(cons nil nil)

Produire du code exécutable

- Du code machine (MIPS)
  - du binaire
  - de l'assembleur
- Du C (Java, ...) structuré
- Du C (Java, ...) déstructuré
  - des appels de fonction
  - des macros
  - du *threaded code*

Production de code C (Java, ...)

- Traduire les instructions pseudo 3 @ en C (Java, ...)
  - penser global, agir local

- Ne pas oublier le prélude
  - essentiellement une fonction

main(argc, argv)

Produire le système langage (*runtime*, libWH)

- Définition des instructions (C)
- Lancement et finalisation du processus d'exécution
- Gestion de mémoire
- Fonctions de services
  - lecture des **paramètres** et écriture des **résultats**

  …

Définition des instructions

- Minimum
  - **la machine telle que vous l'avez définie**

...puis

  - **instructions de service**
    - *debug* (ex. trace d'exécution)
    - système (ex. prélude/postlude)
    - statistiques (ex. nb malloc, nb free)

Lancement et finalisation

- Minimum
  - espace **statique** d'allocation dynamique
  - lire les paramètres et les installer en **mémoire WHILE**
  - lancer l'exécution (**jump start**, **call engine**, ...)
  - afficher les résultats

...puis

  - espace **dynamique** d'allocation dynamique

...ou

  - affichage de **statistiques** d'exécution

Gestion de mémoire

- Minimum
  - allocation dynamique de **cons** et **nil**
  - contexte **statique** pour les variables

...puis

  - allocation de **symboles**
  - contextes **dynamiques** pour les appels

...puis

  - récupération de mémoire
  - comptage de références ?

Fonctions de services

- Minimum
  - allocation de mémoire
  - méthodes de termes
    - constructeur, comparateur, afficheur, ...
  - services systèmes
    - arrêt, ...

...puis

  - récupération de mémoire

...ou

  - affichage de statistiques d'exécution

Lire les paramètres (1)

- Analyser des notations de termes WHILE...

...construire leur représentation en mémoire

...initialiser les variables *read* avec

- Peut s'appuyer sur une fonction du système langage

Lire les paramètres (2)

- Faire un analyseur récursif
- Si erreur de syntaxe, *abort*
- Si erreur de cardinalité, continuer
  - pas assez de paramètres
    - remplacer les paramètres manquants par des **nil**
  - trop de paramètres
    - faire une liste des **argc - nbin** derniers paramètres
    - la passer dans le dernier paramètre

Lire les paramètres (3)

- Développement itératif
  - commencer par ne rien lire, et toujours construire des **nil**

  ...puis passer à des notations d'entier
    - ex. 2 ⟶ **(cons nil (cons nil nil))**
    - en C, penser à **scanf**

  ...puis **(cons nil nil)**

  ...puis lire des notations de termes WHILE

Afficher les résultats

- Parcourir la représentation des variables *write* récursivement en produisant une chaîne de caractères

write X, Y ⟶

X = …

Y = …

- Fonctions du système langage

Délivrables (1)

- 6-10 janvier
  - présentation du **schéma d'exécution** et de **l'architecture logicielle**
  - production de **code exécutable** incomplet
    - démo partielle (fonctionnalité réduite, mais **code exécutable**)
- 13 janvier
  - production de **code exécutable**
    - démo finale (fonctionnalité la moins réduite possible, mais **code exécutable**)

Délivrables (2)

- Fin de semestre (20 janvier)
  - un **rapport par projet** (environ 5 pages), de spécialiste à spécialiste (1 fichier PDF)
    - description générale, description de vos choix
    - qu'est-ce qui marche, qu'est-ce qui ne marche pas
  - une copie des documents techniques du projet (une archive qui ne contient que cela ! PDF !)
  - un **rapport individuel** (1 page), votre rôle dans le projet (1 fichier PDF par projet)

lire consignes sur page Moodle

Annexe pour génération de code machine

Production de code machine (1)

- Traduire les instructions 3 @ en séquences d'instructions machine
  - penser global, agir local
  - ne pas chercher à optimiser
- Ne pas oublier le prélude
  - bien lire la spécification !

Production de code machine (2)

- Binaire
  - bien lire la spécification !
  - vérifier la disponibilité d'une machine, même virtuelle

...ou assembleur

  - vérifier la disponibilité d'un assembleur et d'un émulateur

Macros vs. fonctions (1)

- Macros

  #define InstCons(X, Y, Z) { $code_{XYZ}$ }

  – $code_{XYZ}$ expansé (*inliné*) à chaque utilisation d'une macro

  – gourmand en mémoire de programme

  – efficace (?) en exécution

Macros vs. fonctions (2)

- Fonctions

  InstCons(WHterm * X, * Y, * Z) { $code_{XYZ}$ }

  – compact, mais...

  ...un appel par instruction

  – prologue de l'appel expansé à chaque instruction,

  ...pourtant quasi-identique à chaque fois

- On peut mélanger macros et fonctions

Threaded code (1)

- Objectif
  - obtenir le code exécutable le plus compact possible
  - intérêt pour hiérarchie de mémoire
- Factoriser les traitements dans des procédures

...mais aussi factoriser les séquences d'appel des procédures

Threaded code (2)

- Remplacer

code() { $inst_1$() ; $inst_2$() ; $inst_3$() }

...par

void (* code[])() = { $inst_1$, $inst_2$, $inst_3$ } ;

void pc = 0 ;

engine() { for (;;) (code[pc++])() ; }

Une fonction $inst_i$ (ex. goto) doit pouvoir modifier pc