Rapport projet de Compilation Avancée

PALMISANO Quentin : 28705621

LOUIS Robin : 21116461

2021 - 2022

Table des matières

Structure générale du projet

Nous avons choisi d'utiliser le langage ocaml pour réaliser ce compilateur. Nous nous sommes inspirés du parser-ocamlyacc qui était proposé dans le TME1 que nous avions à ce moment-là tous les deux fait en ocaml. Ce compilateur se compose donc des 5 fichiers sources suivants :

- main.ml
- lexer.mll
- parser.mly
- ast.ml
- eval.ml

Le fichier main.ml se charge donc de récupérer le fichier à compiler passé en paramètre sur lequel on effectue l'analyse lexicale puis l'analyse syntaxique (lexer, parser, etc). Nous avons également implémenté un nettoyeur dans ce fichier main.ml, qui se charge après avoir généré le bytecode, de vérifier s' il comporte deux étiquettes successives et dans le cas échéant il supprime la première des deux. Nous avons un lexer.mll qui reconnaît tous les mots possibles de notre langage. Puis un parser.mly qui reconnaît la structure du langage source et renvoie l'arbre syntaxique correspondant dont les nœuds sont définis dans le fichier ast.ml.
Enfin le fichier eval.ml est chargé d'évaluer les différents nœuds de l'arbre syntaxique afin de produire le bytecode interprétable par la Mini-ZAM.

Schémas de compilation

Soit p l'environnement de compilation p est une (string, int) list dans laquelle sont stockées les noms des variables ainsi qu'un indice i. La i-ème valeur de la pile correspond à la valeur liée au nom de la variable associée à i dans l'environnement de compilation.

- Stat[if e then s1 else s2]p = Expr[e]p; BRANCHIFNOT L(n); Stat[s1]p; BRANCH S(m); LABEL L(n); Stat[s2]p; LABEL S(m)
- Expr[x]p,k = ACC i
- Expr[!x]p,k = CONST 0; PUSH; ACC n; GETVECTITEM
- Stat[x := e]p,k = CONST e; PUSH; CONST 0; ACC n; SETVECTITEM

- Stat[while e do s done]p,k = LABEL L(n); Expr[e]p,k; BRANCHIFNOT L(n+1); Stat[s]p,k; BRANCH L(n); LABEL L(n+1)
- Stat[if e then s1]p,k = Expr[e]p,k; BRANCHIFNOT S(n); Stat[s]p,k; BRANCH S(n); LABEL S(n)
- Stat[let x1 = e1 and · · · xn = en in s]p,k = Expr[e1]p,k; PUSH; Expr[x1]p,k; … Expr[en](x(n-1)::p),k+(n-1); PUSH; Expr[xn](x(n-1)::p),k+(n-1); Stat[s]p,k+(n-2); POP
- Stat[(s where x = e)]p,k = Stat[let x = e in s]p,k
- Stat[x[e1] := e2]p,k = Expr[e2]p,k; PUSH; Expr[e1]p,k+1; PUSH; Expr[s]p,k; SETVECTITEM
- Expr[{e1, · · · en}]p,k = Expr[e1]p,k; PUSH; … Expr[en]p,k+(n-1);MAKEBLOCK n
- Expr[x[e]]p,k = Expr[e]p,k; PUSH; PUSH; Expr[x]p,k; VECTLENGTH; PRIM >; BRANCHIFNOT fin; Expr[x]p,k; GETVECTITEM; LABEL fin
- Expr[(length e)]p,k = Expr[e]p,k; VECTLENGTH
- Expr[(cons e1 e2)]p,k = Expr[e2]p,k; PUSH; Expr[e1]p,k; MAKEBLOCK 2
- Expr[(hd e)]p,k = Expr[e]p,k; BRANCHIFNOT fin; GETFIELD 0; LABEL fin
- Expr[(tl e)]p,k = Expr[e]p,k; GETFIELD 1
- Expr[(empty e)]p,k = CONST 0; PUSH; Expr[e]p,k; PRIM =
- Expr[nil]p,k = CONST 0

Choix d'implantations

En ce qui concerne les incohérence dans le programme source tel que le print(not 42) ou le while 42 do print 0 done, nous avons choisi d'adapter notre grammaire afin qu'elle ne reconnaisse pas ces deux implémentations, elle retourne donc une erreur du parser.
Nous avons divisé les expressions en expressions et conditions afin de pouvoir faire des not ou des while uniquement sur des types boolean et non sur des entiers.

Extensions traitées

Nous avons traités les Extensions suivantes :

- les commentaires
- les règles de priorités
- le if unilatère (if e then s)
- les liaisons locales multiples (let x1 = e1 and · · · xn = en in s)
- une variante syntaxique pour les liaisons locales (s where x = e)
- la structure de tableau
- la structure de liste

Nous avons implémenté dans le lexer un token commentaire qui récupère les commentaire composé de texte entre sur une ou plusieurs ligne de la forme /*ceci est un commentaire*/ est qui ne les interprète tout simplement pas tout comme les espace tabulation et retour à la ligne.
Nous avons gérer toutes les règles de priorités dans le parser grâce à des %left sur nos tokens dans le lexer.
Nous avons implémenté le if then comme une instruction supplémentaire de la même manière que notre if then else.
En ce qui concerne les liaisons locales multiples, nous avons adapté le let des instructions de base pour que celui-ci interprète un let simple ou un let multiple à la fois.

Pour le where, il s'agit simplement d'un appel à let.