TP compilation : ART

Le langage ART introduit la possibilité d'écrire d'une seule pièce des expressions arithmétiques complexes, ainsi que des expressions booléennes.

1 Description du langage ART

La principale nouveauté du langage ART par rapport au langage STK est la séparation de deux concepts :

— les *expressions*, qui calculent des valeurs,

— les *instructions*, qui réalisent des actions.

Parmi les expressions on isole de plus le concept d'*expression gauche*, désignant celles des expressions qui peuvent être utilisées comme cible d'une instruction d'affectation ou d'une instruction de saut (autrement dit, des expressions susceptibles de désigner des adresses).

1.1 Spécification

1.1.1 Expressions

La grammaire pour les expressions est la suivante :

```
expr   ::=  n
         |  b
         |  l_expr
         |  ( expr )
         |  unop expr
         |  expr binop expr

l_expr ::=  id
         |  * expr

unop   ::=  - | !

binop  ::=  + | - | * | / | % | == | != | < | <= | > | >= | && | ||
```

où *expr*, *l_expr*, *unop* et *binop* sont des symboles non-terminaux désignant respectivement les expressions, les expressions gauches, les opérateurs unaires et les opérateurs binaires, où `n`, `b` et `id` sont des symboles terminaux désignant respectivement une constante entière, une constante booléenne et une étiquette, et où les autres symboles sont terminaux. Les priorités entre les différents opérateurs devront être les mêmes qu'en mathématiques et que dans les langages usuels.

1.1.2 Instructions

Les instructions reprennent les instructions `nop`, `exit`, `print`, `jump`, `jump/when` et `write` du langage STK mais en leur fournissant explicitement leurs paramètres sous la forme d'expressions (ou d'expressions gauches) éventuellement placées entre parenthèses, selon la

grammaire suivante.

instr ::= `nop`
| `exit`
| `print (` *expr* `)`
| `jump` *l_expr*
| `jump` *l_expr* `when` *expr*
| *l_expr* `:=` *expr*

De plus, chacune de ces instructions doit être suivie d'un point-virgule.

1.1.3 Programmes

Un programme ART est un fichier texte comportant deux parties :

1. la première commence par la mention `.text` et contient une suite d'instructions terminées par un point-virgule et de déclarations d'étiquettes (terminées par un deux-points comme dans STK et ASM),
2. la deuxième commence par la mention `.data` et contient les données du programme, sous le même format que dans STK et ASM.

Des commentaires peuvent encore être présents dans chacune de ces deux parties, sous la même forme que précédemment.

1.1.4 À propos des expressions gauches

Les expressions gauches (*l_expr*) sont un cas particulier d'expressions désignant selon le contexte une adresse a ou la valeur stockée à cette adresse a en mémoire.

— Utilisée comme membre gauche d'une affectation ou comme destination d'un saut, une expression gauche est interprétée comme l'adresse a (et c'est à cette adresse que l'on écrit ou que l'on saute).
— Utilisée au sein d'une expression ordinaire, une expression gauche est interprétée comme la valeur stockée à l'adresse a en mémoire (autrement dit on fait une lecture à l'adresse a).

Le symbole * utilisée sous forme unaire ($*e$) a la même signification qu'en langage C. Ainsi, en supposant que e est un pointeur, $*e$ désigne la cellule mémoire pointée par e. Ainsi, en supposant que `a` soit une variable stockée à l'adresse `1024` et contenant la valeur `42`, `b` une variable stockée à l'adresse `1025` et contenant la valeur `1024` (c'est-à-dire l'adresse de `a`) et `c` une variable stockée à l'adresse `1026`.

— `c := a` donne 42 comme valeur à `c`
— `c := b` donne 1024 comme valeur à `c`
— `c := *b` donne 42 comme valeur à `c`
— `*b := 1` donne 1 comme valeur à `a`
— `*b := *b` est équivalent à `a := a`

1.2 Exemple

Voici un exemple de programme ART à gauche et sa traduction possible en un programme STK à droite.

```
.text
  jump test;
loop:
  print(a);
  print(10);
  a := a+1;
test:
  jump loop when a<123;
  exit;
.data
a: 97
```

```
.text
  test JUMP
loop:
  a READ PRINT
  10 PRINT
  a a READ 1 ADD WRITE
test:
  loop a READ 123 LT JUMPWHEN
  EXIT
.data
a: 97
```

2 Objectifs du TP

2.1 Contrat de base

Réaliser un compilateur de ART vers STK formé des trois fichiers suivants :

— `ARTLexer.mll` : un fichier ocamllex d'analyse lexicale, fournissant une fonction `ARTLexer.token` qui reconnaît le prochain lexème de l'entrée.

— `ARTParser.mly` : un fichier menhir d'analyse grammaticale et de traduction, fournissant une fonction `ARTParser.program` effectuant l'analyse grammaticale d'un programme ART entier et renvoyant une chaîne de caractères représentant le programme STK correspondant.

— `ARTCompiler.ml` : le fichier principal qui combine les deux fonctions précédentes. *Ce dernier fichier est fourni et n'a pas besoin d'être modifié.*

2.2 Extensions

2.2.1 Messages d'erreur

Modifier l'analyseur pour qu'il fournisse des messages d'erreur plus précis.

2.2.2 Optimiser l'utilisation des registres

Cette extension nécessite d'avoir réalisé l'extention « allocation de registres » du TP précédent.

Considérons une expression de la forme $e_1 + e_2$, et supposons que l'évaluation de e_1 nécessite r_1 registres et que l'évaluation de e_2 nécessite r_2 registres. En fonction de r_1 et de r_2, combien faut-il de registres pour évaluer l'expression complète ? À quelles conditions est-il plus économe de commencer par e_1 ? par e_2 ? (Bonus : trouver une expression avec n additions qui peut être évaluée avec seulement deux registes en faisant les bons choix d'ordre mais qui peut nécessiter $n+1$ registres avec les mauvais choix.)

Objectif de cette extension : ajuster la compilation des opérations arithmétiques commutatives (+ et $*$) pour choisir l'ordre d'évaluation des opérandes de sorte à minimiser le nombre de registres nécessaires. Il faudra pour cela modifier la fonction d'analyse grammaticale des expressions pour qu'elle ne renvoie pas seulement la chaîne de caractères représentant le programme STK généré, mais également le nombre de registres nécessaires. Le nombre de registres nécessaire doit être calculé avec la formule trouvée en réponse au paragraphe précédent.