TP2 de compilation
Fragments 0 et 1

Ce TP peut être fait dans le langage de votre choix.

Vous êtes invitez à abondamment utiliser la description des commandes de la machine.

Il n'y aura pas de correction car ce sont des parties du projet (mais vous pouvez poser des questions aux chargés de TPs).

Exercice 1 (Mon premier compilateur).

1. Compilez la mini-JS-machine.
2. Testez votre réponse (ou la correction) à la question 1 du TD4 sur la mini-JS-machine.
3. Récupérez votre code du TP1, vérifiez qu'il compile bien.
4. Rajouter une fonction qui prend l'AST d'une expression arithmétique, fait un parcours en profondeur gauche, et écrit, dans une string ou un fichier, la compilation de l'expression de départ ; pour ça, remarquez que tous les opérateur présent doivent simplement être écrits en notation postfix.
5. Modifiez le TP1 pour que la sortie soit le programme assembleur mini-JS de l'expression entrée, testez le à l'aide de quelques exemples et de la machine.

Exercice 2 (Taille du code et jumps (fragment 1)).

1. Reprenez votre code de l'exercice 1.
2. Midifiez votre struct/type/classe d'AST pour que chaque noeud puisse aussi acueillir un entier "taille" (vous pouvez aussi en faire une autre version avecla taille dedans).
3. Votre AST issu du TP1 (ou de la question précédente) n'a pas de "taille" sur les noeuds, modifiez la génération de l'AST de sorte à ajouter la taille dans chaque noeud (demanière linéaire s'il vous plais), sachant qu'une constante a une taille de 1 et une opération à comme taille $1 + t_1 + t_2$ où t_1 et t_2 sont les tailles de ses fils. Alternativement, vous pouvez calculer l'AST sans la taille et faire une passe sur l'arbre pour l'ajouter.
4. Remarquez que la "taille" ainsi calculé correspond à la taille du code généré. Utilisé cette remarque pour ajoutez l'opérateur ternaire d'expressions conditionnelles `_?_:_` . On rappel que celui-ci secode comme le `if_then_else` :

$$[\![\mathtt{e}_1 ? \mathtt{e}_2 : \mathtt{e}_3]\!] \quad := \quad [\![\mathtt{e}_1]\!] \qquad \mathtt{ConJmp}(t_2 + 1)[\![\mathtt{e}_2]\!] \qquad \mathtt{Jump}t_3[\![\mathtt{e}_3]\!]$$

où t_2 est la taile de $\mathtt{e}_2$ et t_3 celle de $\mathtt{e}_3$.

5. Normalement, vous devriez avoir un soucis : il faut générer une taille pour les noeuds `_?_:_`. Pour savoir quoi générer,il faut rgarder lataille du code ci-dessus : on voit que la taille est la somme des tailles des fils du noeud, plus 2.
6. Vérifiez que la priorité utilisée est correcte, pour ça, faites des tests en comparant avec JS. Rappel : pour tester JS, ouvrez une console dans votre Browser ; sur Windows/Linux et avec Firefox, il faut aller dans options → web developper → web console → `Ctrl +b`

Exercice 3 (commandes (fragment 1)).

1. Dans la grammaire utilisée pour générer le parseur, ajoutez le non-terminal `commande` définit uniquement (pour l'instant) comme une expression `<commande> := <expression>;`
2. Toujours dans la grammaire, ajoutez le non-terminal `programme` définit comme une séquence de `commande`s ; faites en le non-terminal principal.
3. Pour l'AST, il faut rajouter deux structures d'AST : l'une pour les ASTs des `commande`s, qui nesont pour l'instant que des *wrappers*[1] autours des ASTs des expressions, et l'autre pour les ASTs des `programme`s sont des listes d'ASTs de `commande`s.
4. Étendez le parseur à cette nouvelle grammaire.

Exercice 4 (Rendu minimal).

1. Dans le fichier de génération du lexeur, ajouter le token `IDENT` pour les variables.
2. Dans les expressions, ajoutez l'expression `<IDENT>` pour la lecture d'une variable, et la commande `<IDENT>=<expression>` pour l'assignement. Attention, l'assignement est prioritaire sur tous les autres opérateurs.
3. Modifiez l'AST en conséquence.
4. Étendez le parseur à cette nouvelle grammaire.
5. Ajoutez les tailles à vos nouveaux ASTs .

Exercice 5 (Fragment 1).

1. Transformez la commande `<IDENT>=<expression>` en expression, de sorte à pouvoir écrire `x = y = 3 ;` par exemple .
2. Ajoutez les booléens dans le lexeur.
3. Ajoutez la commande `{<programme>}`.
4. Ajoutez la commande `Si (<expression>) <commande> Sinon <commande>` :

$$[\![\texttt{Si (e}_1\texttt{) c}_2\texttt{ Sinon c}_3]\!] \quad := \quad [\![\texttt{e}_1]\!] \quad \texttt{ConJmp}(t_2+1)[\![\texttt{c}_2]\!] \quad \texttt{Jump}t_3[\![\texttt{c}_3]\!]$$

1. Un *wrappers* est une struct/type/objet ne contenant qu'un seul champ dont le type est le type "wrappé".

5. Étendez le parseur à cette nouvelle grammaire, n'oubliez pas de modifier les tailles correctement.
6. Ajoutez la commande `TantQue (<expression>) <commande>` :

$$[\![\texttt{TantQue (e) c}]\!] \quad := \quad [\![\texttt{e}]\!] \quad \texttt{ConJmp}(t_c + 1)[\![\texttt{c}]\!] \quad \texttt{Jump}(-t_c - t_e - 2)$$

7. Ajoutez les autres commandes.