Rapport de projet

Réalisation d'un compilateur

Médéric HURIER
<mederic.hurier@etudiant.univ-nancy2.fr>

Omar EDDASSER
<omar.eddasser@etudiant.univ-nancy2.fr>

Hadrien TORRES
<hadrien.torres@etudiant.univ-nancy2.fr>

Romain SERGEANT
<romain.sergeant@etudiant.univ-nancy2.fr>

Étienne PARIZOT
<etienne.parizot@etudiant.univ-nancy2.fr>

Année 2011 – 2012

Table des matières

1. Conception....3
1.1. But du projet....3
1.2. Découpage....3
1.3. Grammaire....4
2. Réalisation....6
2.1. Structures....6
2.1.1. Table des symboles....6
2.1.2. Arbre syntaxique abstrait....7
2.2. Analyse lexicale et syntaxique....8
2.2.1. Utilisation de Flex....8
2.2.2. Utilisation de Bison....8
2.3. Génération de code....9
2.3.1. Construction de l’arbre d’analyse....9
2.3.2. Assembleur....10
3. Recette....12
3.1. Structure du projet....12
3.2. Utilisation du programme....12
3.3. Conclusions....12

1. Conception

1.1. But du projet

Notre objectif est de réaliser un compilateur capable de transformer des programmes du langage “miage” vers du code cible en beta-assembleur.
La langage “miage” est une version inspirée et simplifiée du langage C. Il comporte des fonctions de lecture, d’écriture ainsi que des conditions, des boucles et une arithmétique des entiers.
Le beta-assembleur est un outil pédagogique réalisé par l’institut du MIT. C’est un programme Java qui émule une architecture 32 bits et qui fournit un jeu de macro limité mais facile à appréhender.

Les finalités de ce module sont d’étudier :

- Les possibilités et les limites d’un compilateur
- les méthodes permettant d’écrire des programmes efficaces
- La gestion de gros projets
- La conception et la maintenance de programmes complexes

1.2. Découpage

Un compilateur peut être vu comme une succession d’étape transformant du code source de haut niveau en code cible de bas niveau. L’enchaînement est le suivant:

1. **Analyse lexicale** : découpe le code source en jeton et vérifie si ils appartiennent au vocabulaire du langage.
2. **Analyse syntaxique** : étudie la combinaison des mots pour tester la grammaire du programme.
3. **Analyse sémantique** : certifie la cohérence du programme en terme de types et de déclarations.
4. **Génération du code** : transforme les structures abstraites en code cible.

Chaque étape peut être réalisée par un programme indépendant. Ils sont ensuite réunis par des programmes intermédiaires en langage C qui initialisent les structures et coordonnent les actions.
Un fichier **Makefile** permet de définir les règles de création du compilateur, qui n’est au final qu’un fichier exécutable prenant des programmes en entrée et générant une chaîne de caractère (le code en langage cible).

Le nombre de passe donne le nombre de parcours de l’arbre d’analyse nécessaire à la génération du code. Pour notre projet, ce nombre est de 2.

1.3. Grammaire

La grammaire formelle est la base du langage de programmation. Elle permet de définir les règles syntaxiques, et par extension, les règles lexicales. Elle se compose:

- d'un ensemble fini de symboles terminaux
- d'un ensemble fini de symboles non-terminaux
- d'un ensemble de règles de production

Pour notre langage, nous avons défini la grammaire suivante:

- **Symboles terminaux**:
    - identificateur : [a-zA-Z_][a-zA-Z0-9_]*
    - commentaire multi-ligne : /* ... */
    - littéraux : [0-9]+
    - types : int, void
    - opérateurs arithmétiques : +, -, *, /
    - opérateurs de comparaison : =, !=, <, >
    - séparateurs d'instructions : ;
    - séparateurs d'expressions : (,), {, }, ","
    - mots-clés : if, else, while, return

- **Symboles non-terminaux** :
    - Symbole de départ : Programme
    - Portée globale : InstructionGlobale, DesInstructionGlobale, DeclarationFonction, DeclarationVariableGlobale, AffectationGlobale
    - Portée locale : InstructionLocale, DesInstructionLocale, DeclarationVariableLocale, AffectationLocale
    - Fonction : AppelFonction, Parametres, AutreParametre, Retour
    - Structures de contrôles : If, Else, While, Condition
    - Commun : Expression, AutreExpression, AutreIdentificateur, Facteur, Atome, Comparaison, Type

- **Règles de production** :
    - Programme : DesInstructionGlobale
    - InstructionGlobale : /* … */ | DeclarationVariableGlobale | DeclarationFonction | AffectationGlobale | If | While | AppelFonction;
    - DesIntructionGlobale : InstructionGlobale DesInstructionGlobale |
    - InstructionLocale : /* … */ | DeclarationVariableLocale | AffectationLocale | If | While | Retour | AppelFonction;
    - DesIntructionLocale : InstructionLocale DesInstructionLocale |
    - DeclarationFonction : Type <identificateur>(Paramètres) { DesInstructionLocale } | void <identificateur>(Paramètres) { DesInstructionLocale }
    - DeclarationVariableLocale : Type <identificateur>; | Type <identificateur> AutreIdentificateur; | Type AffectactionLocale
    - DeclarationVariableGlobale : Type <identificateur>; | Type <identificateur> AutreIdentificateur; | Type AffectationGlobale
    - AffectationLocale : <identificateur> AutreIdentificateur = Expression;
    - AffectationGlobale : <identificateur> AutreIdentificateur = Expression;
    - If : if (Condition) { DesInstructionLocale } Else

- Else : else { DesInstructionLocale }
- While : while (Condition) { DesInstructionLocale }
- Condition : Expression Comparaison Expression
- Expression : Facteur + Expression | Facteur - Expression | Facteur
- Facteur : Atome * Facteur | Atome / Facteur | Atome
- Atome : <litéral> | <identificateur> | AppelFonction | (Expression)
- AppelFonction : <identificateur>() | <identificateur>(Expression, AutreExpression)
- Parametres : Type <identificateur> AutreParametre
- AutreParametre : , Type <identificateur> AutreParametre |
- AutreExpression : , Expression AutreExpression |
- AutreParametre : ,<identificateur> Autreidentificateur |
- Retour : return Expression;
- Comparaison : < | > | == | !=
- Type : int

Parmi les règles de production, on observe certains schémas qui peuvent paraître répétitifs. Ils donnent en réalité des spécifications supplémentaires pour rendre le langage plus puissant:

- La séparation entre portée locale et globale permet d'isoler le contexte dans lequel l'expression est analysée. Elle limite également l'appel à certaines règles (comme la déclaration de fonction au niveau local).
- La priorité des expressions est forcée par une décomposition en 3 sous règles : Expression, Facteur et Atome. Elles permettent de rendre les opérations parenthésées plus prioritaires que les divisions et multiplications, et ces dernières plus prioritaires que les additions et les soustractions.
- L'emploi des règles commençant par "Autre..." comme AutreParametre ou AutreIdentificateur donne un moyen simple de chaîner les affectations, les paramètres et les expressions.

Cette grammaire n'inclus aucune récursivité gauche et n'est donc pas ambiguë.

2. Réalisation

2.1. Structures

2.1.1. Table des symboles

La table des symboles répertorie les identificateurs du programme et renseigne leurs types, leurs portées et leurs valeurs. Il s'agit d'une structure essentiel à la compilation, car elle permet manipuler les fonctions et les variables par référence tout au long du processus. Nous avons choisi pour notre implémentation une liste chaînée à double clé pour sa simplicité au vu du nombre d'entrée à gérer. La première clé donne le nom de l'identificateur et la seconde son rôle dans le contexte parmi "globale", "locale", "paramètre", "fonction".

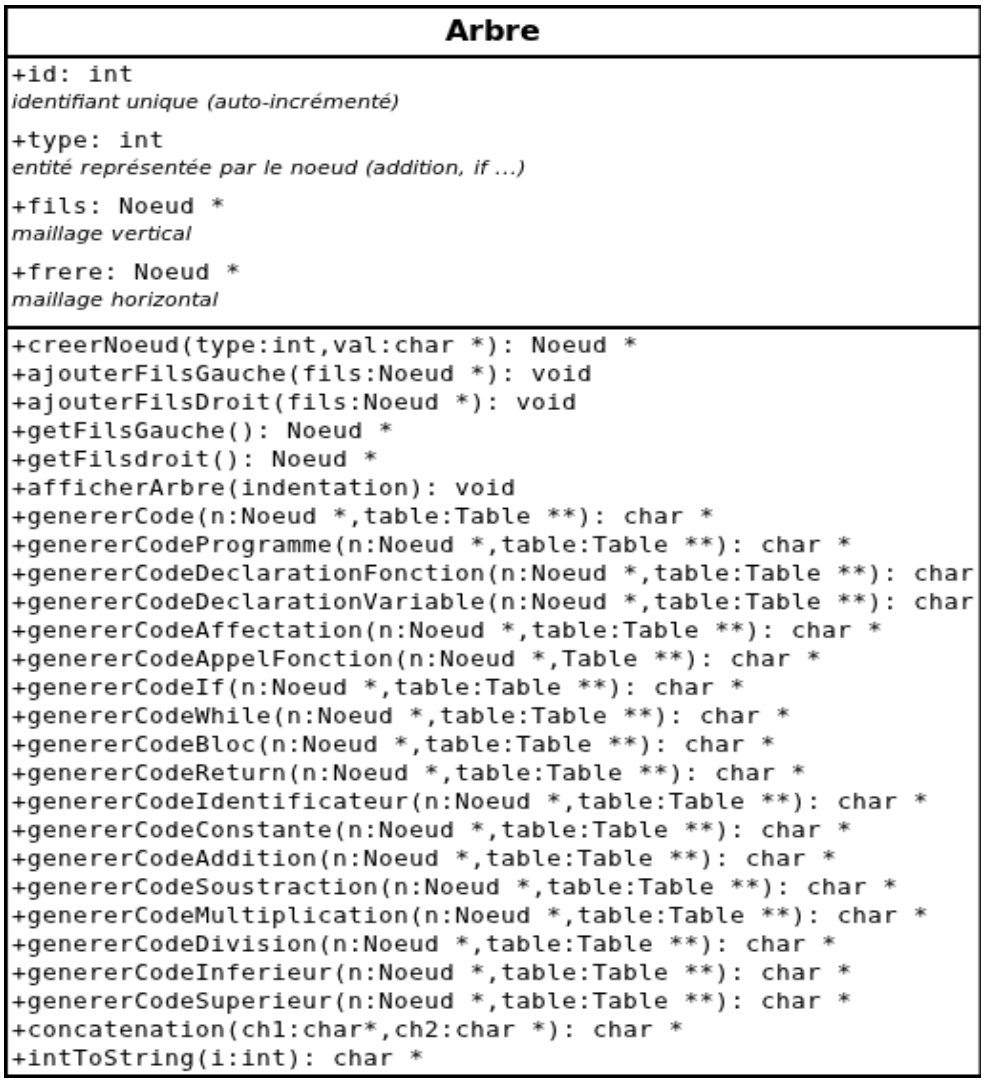
Ce diagramme donne la structure de la table et ses fonctions associées.

Dans notre première implémentation, nous n'avions que des fonctions génériques d'ajout, de suppression et de recherche. Nous avons ensuite complété la structure avec des méthodes spécifiques à la compilation, comme la recherche du type de variable.
Si le nouveau noeud à insérer existe déjà dans la table, sa valeur sera remplacée. Nous pouvons également renvoyé le nombre de paramètres d'une fonction ainsi que le nombre de variable locale en comptant les noeuds successifs de même type (clé 2). Cela s'avère très utile pour préciser l'espace à allouer au niveau assembleur.

2.1.2. Arbre syntaxique abstrait

L'arbre syntaxique est une bonne structure intermédiaire pour représenter un programme. Chaque feuille est un composant du programme. La hiérarchie horizontale (frères) et verticale (fils) de la structure permet de recomposer le programme ou d'en générer le code.

Pour notre implémentation, nous avons choisi le modèle très simple de ce diagramme:



L’arbre est composé de noeud, ayant chacun un identifiant unique, un type, des noeuds fils et des noeuds frères. Les noeuds fils sont distingués selon leur place à gauche ou à droite. C’est une information utile pour certaines directives du langage, comme les affectations.

Outre l’ajout de noeud et sa hiérarchie, la structure est également chargée de transformer chaque noeud en code du langage cible (chaîne de caractère). Il existe une fonction de génération par type de noeud, et elles font largement appel à la récursivité pour fractionner la tâche.
La fonction genererCode est la pièce maîtresse de l’arbre, chargée de renvoyer le code cible pour un noeud de façon totalement générique.

2.2. Analyse lexicale et syntaxique

2.2.1. Utilisation de Flex

Flex est un analyseur lexical Lex capable de détecter des motifs lexicaux à partir d’un fichier de données en entrée. Il reconnaît ces motifs à l’aide d’expressions rationnelles, et pour chaque occurrence, il exécute un code en langage C.

Dans notre compilateur, nous faisons un usage très simple de cet outil. Les chaînes de caractères reconnue sont associées à des symboles terminaux de notre programme, codés par des constantes. Ces symboles sont ici appelés jetons ou tokens.
A chaque occurrence d’une expression, la valeur de la constante est retournée dans le bloc de code correspondant. Exemple pour le signe “+”:

```
"+"             { ECHO; return(ADD); }
```

Ces valeurs sont ensuite analysées par un programme Bison.

2.2.2. Utilisation de Bison

Bison est un analyseur syntaxique chargée de comparer l’ordre des symboles avec la grammaire formelle du langage. Il est utilisé en complément de Flex pour détecter des erreurs de syntaxes dans le code source.

Comme pour Flex, il réalise des opérations en code C à chaque occurrence de règle de production rencontrée. Dans ce cas, les actions servent à construire l’arbre syntaxique abstrait. Exemple pour le noeud “Programme”:

```
Programme : DesInstructionGlobale {
        noeud_racine = creerNoeud( TYPE_NOEUD_PROGRAMME , "programme");
}
```

Une fois l’arbre construit, la génération de code peut commencer.

2.3. Génération de code

2.3.1. Construction de l’arbre d’analyse

L’arbre d’analyse peut être construit manuellement par un programmeur en créant des noeuds et en les associant pour former une hiérarchie. Il est cependant beaucoup plus puissant de le construire directement dans le fichier Yacc.

La construction se fait étape par étape, règle par règle par du code en C. Chaque règle crée un noeud et ses associations en fonction des symboles non-terminaux. L’arbre se construit ainsi par appels récursifs successifs.

Les arbres que nous construisons peuvent être affichés dans la sortie standard. Voici un exemple pour le fichier 01.miage:

```
Noeud 1 - 10 - programme
Fils :
--Noeud 2 - 20 - read
--Fils : /
--Noeud 3 - 20 - write
--Fils : /
--Noeud 4 - 20 - main
--Fils :
----Noeud 7 - 40 - affectation
----Fils :
------Noeud 6 - 110 - i
------Fils : /
------Noeud 5 - 120 - 0
------Fils :
```

Et pour le fichier 18.miage:

```
Noeud 1 - 10 - programme
Fils :
--Noeud 2 - 20 - read
--Fils : /
--Noeud 3 - 20 - write
--Fils : /
--Noeud 4 - 20 - f
--Fils :
----Noeud 8 - 60 - if
----Fils :
------Noeud 7 - 180 - 180
------Fils :
--------Noeud 5 - 110 - n
--------Fils : /
--------Noeud 6 - 120 - 0
--------Fils : /
------Noeud 9 - 80 - then
------Fils :
--------Noeud 12 - 90 - f
--------Fils :
----------Noeud 11 - 120 - 0
----------Fils : /
------Noeud 10 - 80 - else
------Fils : /
----Noeud 19 - 90 - f
----Fils :
------Noeud 18 - 130 - '+'
------Fils :
--------Noeud 13 - 110 - n
--------Fils : /
--------Noeud 17 - 50 - f
--------Fils :
----------Noeud 16 - 140 - '-'
----------Fils :
------------#Noeud 14 - 110 - n
```

```
------------Fils : /
------------Noeud 15 - 120 - 1
------------Fils :
```

2.3.2. Assembleur

Une fois l'arbre généré, nous appelons une fonction "genererCode" du fichier "arbre.c" pour nous renvoyer le code en langage cible.

Le programme assembleur est structuré de cette façon:

- inclusion des macros "beta.uasm"
- déclaration des variables globales en mémoire
- déclaration des fonctions
    - instruction locales
    - retour (optionnel)
- déclaration de la fonction main
    - met à la jour le bas et le sommet de la pile en fonction des variables déjà insérées au niveau global
    - instructions locales
- fin de la fonction main (remet les pointeurs et registre dans leurs états initials)

Plusieurs conventions ont été fixées pour les fonctions:

- L'appelant
    - empile les paramètres effectif de la fonction
    - empile un espace pour le résultat de la fonction
- L'appelé
    - empire la valeur de retour
    - empile le sommet de la pile
    - réserve les variables locales
    - effectue les calculs

Chaque structure de contrôle (if, while …) qui ne dispose pas de nom est fixée par un idice auto-incrémenté lors de la génération de l'arbre.

Le code ci-dessus est la génération du programme 01.miage:

```
.include ../lib/beta.uasm
.= 0

BR(main)
i: LONG(0)
#read:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)

fin_9:
        POP(BP)
        POP(LP)
        RTN()
write:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)

fin_write:
```

```
I       POP(LP)
        RTN()
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
CMOVE(0, R0)
PUSH(R0)
POP(R0)
ST(R0,1)

fin_main:
        POP(BP)
        POP(LP)
        HALT()
```

Et celui ci pour le fichier 16.miage:

```
.include ../lib/beta.uasm
.= 0

BR(main)
i: LONG(0)
j!read:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)

fin_9:
        POP(BP)
        POP(LP)
        RTN()
write:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)

fin_write:
I       POP(LP)
        RTN()
CMOVE(3, R0)
PUSH(R0)
POP(R0)
ST(R0,1)
main:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP,BP)
LD(i, R0)
PUSH(R0)
CMOVE(2, R0)
PUSH(R0)
POP(R1)
POP(R0)
CMPLT(R1, R0, R0)
POP(R0)
BF(R0, else1)
then1:
CMOVE(0, R0)
PUSH(R0)
POP(R0)
ST(R0,1)
BR(fsi1)
else#1:
CMOVE(1, R0)
PUSH(R0)
POP(R0)
ST(R0,1)
fsi1:
main:
        POP(LP)
        HALT()
```

3. Recette

3.1. Structure du projet

Le répertoire du projet reprend tout notre travail et les sources nécessaires pour construire le projet. Parmi ces fichiers:

- arbre.h, arbre.c : structure d'arbre syntaxique abstrait
- table.h, table.c : structure table des symboles
- constante.h : défini des constantes utilisées dans toute l'application
- mini.lex : fichier LEX d'analyse lexicale
- mini.y : fichier YACC d'analyse sémantique
- main.c : fichier de coordination des différentes actions
- test/ : jeu de donnée pour les tests
- docs/ : rapport et diagramme
- Makefile : permet de construire l'exécutable du compilateur

Pour utiliser le compilateur, les outils "Make", "GCC", "Flex" et "BISON" sont nécessaires. Il suffit de se placer dans un terminal et d'exécuter la commande:
`make`

3.2. Utilisation du programme

En résultat de l'étape précédente, un fichier "compilateur.exe" sera crée. Il s'utilise de manière très simple:
./compilateur.exe < test/01.miage

Cette commande permet de tester le fichier "01.miage". Des erreurs seront affichées en cas d'erreurs d'analyse dans la sortie standard.
Si tout se passe bien, un fichier en langage assembleur sera crée. Il ne fonctionne que dans l'environnement beta-assembleur comme précisé dans l'énoncé.

3.3. Conclusions

Ce projet a été en terme d'organisation et de difficulté l'un de plus exigeant de la formation. Il requiert des capacités conceptuels de haut niveau, en même temps que des connaissances sur la mécanique de bas niveau des langages.

L'élément le plus important de ce module est la grammaire du langage. C'est la première partie que le compilateur analyse, et c'est celle qui dicte la suite des actions. Concevoir une bonne grammaire a été une tâche longue, mais vitale pour gagner du temps pour la suite. Nous avons décidé de ne pas la négliger, et c'est un choix qui a payé.
En terme de répartition du travail, il est facile de découper le travail pour se répartir les tâches indépendantes entre elles. Cependant, il est plus difficile de réunir toutes les pièces pour finaliser le travail. Sur la fin du projet, nous n'avons pas trouvé de modèle idéal pour travailler et surtout de temps pour se voir. Le recours à des outils collaboratifs (Google Docs, Redmine, Mercurial …) a été d'une grande aide sur ce projet.

Au niveau technique, les programmes que nous avons utilisé touchent à des aspects très pointues de l'informatique : la création de langage de programmation. En tant que novice, il est difficile de d'appréhender ces concepts, mais les travaux pratiques réguliers et dirigés se sont avérés très utile. L'aide d'un tuteur également.
La partie technique la plus difficile a été la génération du code assembleur. Ce n'est pas tant le processus qui est compliqué, mais la syntaxe et les contraintes de l'assembleur. Le choix du beta-assembleur dans ce cas est justifié pour sa simplicité, mais rend plus difficile le travail autonome car il n'y a pas beaucoup d'aide en ligne.

Au final, ce projet a été intéressant sur le plan de l'analyse des langages de programmation et sur la compréhension de nos outils informatiques. Même si nous ne nous destinons pas à écrire un langage complet, il répond à la tendance moderne de conception de DSL (Domain Specific Langage) destiné à des non experts en informatique. L'analyse performante de motif dans un fichier texte est également une compétence très appréciée.