Projet Architecture des Ordinateurs
Machine à Pile

Théophile Dano, Vincent Gouteux, Adrien Legros

2 février 2017

Première partie
Le compilateur assembleur

1 Approche générale

Dans la manière de concevoir ce petit compilateur, nous avons voulu utiliser les manières les plus populaires pour le faire. C'est-à-dire créer un compilateur en plusieurs passes, qui toutes ont leur propres entrées/sorties et qui travaillent à la chaîne.

L'idée est que le travail du compilateur est divisé en 4 phases :
— L'analyse lexical (le "lexing")
— L'analyse grammaticale (le "parsing")
— L'optimisation
— La génération de code

Nous avons décider d'ignorer la phase d'optimisation du code puisqu'elle n'est pas du tout requise pour ce projet, même si cela pourrait être intéressant.

1.1 Analyse Lexicale

Le rôle du *lexer* est de générer depuis une chaîne de caractères une liste de *jetons* ("*token*" dans la suite) pour identifier chacune des suites de caractères qu'il reconnait.

Le but du lexer est donc de découper ce qui lui est donné en une liste de tokens pour ensuite pouvoir en comprendre la structure.

Regardons sur un petit exemple. Disons que l'on veuille analyser une chaine de charactères qui représente une phrase en français, et que notre lexer ne reconnaisse que les *VERBE*s et les *NOM*s et que tout autre mot soit identifié par un token de type *MOT*.

Le chat mange la souris

Si nous devions exécuter notre lexer sur cette phrase test, il produirait la liste de tokens suivante :
— MOT : "Le"
— NOM : "chat"
— VERBE : "mange"
— MOT : "la"
— NOM : "souris"

A ce stade le lexer a fini son travail. Il a atteint la fin de l'entrée et n'a rencontré aucune erreur. Il passe donc la main au parser.

1.2 Analyse grammaticale

Grammaire L'enjeu d'un parser est, étant donné une chaine de tokens et leur contenu, vérifier qu'ils forment une structure conforme à une grammaire définie à l'avance. Ici il faut entendre le mot grammaire au sens classique du terme, c'est-à-dire une liste de règles définissant comment on peu organiser des mots d'un langage.

Si maintenant on introduit pour notre lexer le token ARCTICLE alors les mots "le" et "la" seront reconnus comme tels. Cela va nous aider à définir une grammaire extrêmement simple avec une unique règle PHRASE-SIMPLE.

Le lexing de notre première phrase se transforme alors en :
— ARTICLE : "Le"
— NOM : "chat"
— VERBE : "mange"
— ARTICLE : "la"
— NOM : "souris"

En suivant la notation EBNF (*Extended Backus-Naur Form*) pour les grammaires, on peut définir PHRASE-SIMPLE comme :

```
PHRASE_SIMPLE
   : ARTICLE NOM ARTICLE NOM
   ;
```

On remarque immédiatement que n'importe quelle phrase basée sur cette structure grammaticale sera reconnue et validée par le parser. (par exemple "*Les taupes creusent le sol*" est une entrée valide).

C'est aussi le rôle du parser de faire remonter les possibles erreurs de grammaires contenues dans le programme.

Représentation interne du code Il est évident que l'on ne veut pas juste vérifier que la structure du code soit correcte mais bien en faire quelque chose. Par exemple générer du code correspondant dans un autre langage, ou colorer les mots clés dans un éditeur etc.

Pour cela, il nous faut garder en mémoire chaque groupe de tokens qui a été reconnu et validé par le parser dans une structure de données appropriée. Les arbres sont pour cela bien adaptés puisqu'ils permettent d'organiser les informations de manière très pratique. On peut représenter une PHRASE-SIMPLE par un nœud ayant pour fils chacun des tokens repérés dans l'ordre.

De manière générale, on utilse un *A.S.T.* ("*Abstract Syntax Tree*" où Arbre de Syntaxe Abstraite).

1.3 Génération du code

Cette dernière passe d'un compilateur n'est généralement pas compliquée. En effet, une approche simple et efficace est de parcourir simplement l'arbre (AST) généré par le parser et pour chaque nœud de traverser ses fils récursivement.

Il y a évidement des problèmes qui se posent comme la gestion des identifiants (noms de variables etc.). Sont-ils utilisés dans un cadre correct ? Ont-ils bien étés définis avant ? Si oui, sont-ils du type attendu et ainsi de suite.

Ces problèmes se résolvent souvent facilement par l'inspection d'une table de symboles ("*Symtable*" en anglais). Cette table (souvent une *Hash Map*) répertorie chaque noms de symboles définis dans un programme et leur associe des informations connexes (type, mutabilité, définition, etc.).

2 Application au projet

Dans cette partie nous allons voir dans quelle mesure il est possible d'appliquer ces techniques au projet en C à l'aide de bibliothèques et d'un *parser-generator* : le tandem *Flex/Bison*, version retravaillée de *Lex/Yacc*.

2.1 Lexer – Flex

Pour la première passe de notre compilateur, le lexing, nous avons choisit d'utiliser Flex un générateur de lexer simple a prendre en main, souvent utilisé dans les cours d'introductions à la compilation et à l'analyse de textes divers.

Nous n'allons pas détailler le fonctionnement interne de Flex mais simplement analyser quelques portions du fichier **scanner.l**.

Dans la première partie, nous avons détaillé comment on pouvait définir certains types de tokens. Il s'agit ici (avec Flex) de le faire avec du code C. Regardons le plus simple des tokens du projet : le mot clé **pop**.

```
pop { return TOKEN_POP; }
```

La syntaxe utilisée n'est pas du tout pensée pour être du C pur. Seul les parties entre accolades doivent être du code C. Ici, le mot **pop** est utilisé pour désigner un *pattern* a reconnaitre et l'action associée est tout simplement de renvoyer au parser l'information que l'on est tombé sur un token de type POP.

Utiliser Flex est d'une trivialité sans pareil. Cependant, pour identifier un type particulier d'entrée qui n'est pas définit clairement au préalable, il faut utiliser un outil obscur au premier regard, mais qui

est très logique.

Les expressions régulières ("*regex*" de "*regular expressions*") sont très utilisées en informatique pour décrire des *patterns* et reconnaitre des formations syntaxiques particulières. On appelle cela plus couramment le *pattern matching*, dans le sens ou l'on essaie de reconnaitre des patterns particuliers.

Prenons l'exemple des étiquettes (labels) d'instructions dans notre petit langage assembleur. On ne sait pas donner une liste de mots étant à reconnaitre comme des étiquettes et donc on ne peux pas lister tous les labels possibles. Cependant, on connait la manière dont sont définis les labels. Ce sont des mots (combinaisons de caractères ASCII suivis de chiffres ou d'underscores) qui ne sont pas des mots-clés et qui sont suivis de deux points.

On peut alors définir ca comme suit à l'aide des expressions régulières et Flex :

```
[a-zA-Z_][a-zA-Z0-9_]*":" {
        yytext[strlen(yytext) - 1] = 0; // pour ignorer les ":"
        yylval.ident = strdup(yytext);

        return TOKEN_LBL;
}
```

Sans rentrer dans les détails des expression régulières, la règle utilisée ici est très simple. La règle [a-zA-Z_] précise que le premier caractère doit être une lettre peu importe sa casse, ou bien un underscore. Ensuite, [a-zA-Z0-9_]* permet de récupérer n'importe quel caractère alphanumérique ou underscore. L'étoile est utilisée pour indiquer que l'on accepte une répétition sans limite de cette règle. Enfin, " :" indique que le token doit se terminer par deux points.

Finalement, si la règle est "matchée", on exécute quelques actions et on retourne le token au parser qui saura quoi en faire.

2.2 Parser – Bison

Maintenant que notre fichier ou code source a été relu et découpé par le lexer, le parser peut commencer son travail. C'est-à-dire construire l'AST et vérifier que l'entrée soit conforme à la grammaire utilisée.

Ici, nous n'allons pas nous éterniser sur le pourquoi du comment des grammaires mais il est important de noter qu'il faut faire attention aux récursions des règles (une règle s'invoque elle même, par la droite ou la gauche par exemple) car cela peut mener à des boucles infinies et planter le programme.

Cela étant dit, regardons comment on peut mettre en place une grammaire et son parser avec Bison. Nous allons regarder qu'un seul exemple pour pouvoir ensuite parler de la génération de l'AST. Dans le fichier **grammar.y** on trouve une liste de règles de grammaire parmis lesquelles celle qui définit comment un programme assembleur doit être organisé.

```
program
  : %empty
  | stat NL program
  | TOKEN_LBL {
        tok = $1;
        if (yydolog) fprintf(yylog, "%3d: label -> %s\n", pc, $1);
        add_sym_entry(labels, $1, pc);
  } stat NL program
  | NL program
  ;
```

Comme pour Flex, les commandes en langage C sont définies entre accolades. Ici, NL correspond a un retour à la ligne, stat est n'importe quelle instruction assembleur valide, et TOKEN_LBL le token associé aux labels (vu plus haut).

La règle **%empty** signifie qu'un programme valide peut ne rien être (chaine vide). Sinon (la barre verticale signifie une alternative), un label suivi d'une instruction et d'un retour à la ligne est accepté, une instruction simple et un retour à la ligne de même.

On peut observer que la règle **program** est récursive (par la droite). C'est-à-dire qu'une suite de **program** est un **program** (eux mêmes constitués d'instruction et d'étiquettes).

Regardons maintenant l'action faite lorsque l'on rencontre un label. On récupère le token dans la variable **tok** grâce au symbole **$1** qui sera remplacé par Bison par le pointeur adapté. Ensuite on ajoute le label à la Symtable (table des symboles) en indiquant comme information connexe son adresse dans le code.

En jetant un rapide coup d'œil à la règle **stat**,

```
stat
  : TOKEN_POP INT {
      tok = $1;
      if (yydolog) fprintf(yylog, "%3d: pop %d\n", pc, $2);
      ast_tail->next = new_instr(POP, $2, pc++);
      ast_tail = ast_tail->next;
  | ...
  ;
}
```

On voit comment est gérée la génération de l'AST par le parser. Lorsqu'une instruction correcte est rencontrée, on génère un nœud dans l'arbre qui lui correspond. Ici, l'instruction **pop** est valide si le token POP est suivi du token INT (pour un nombre entier). Dans ce cas, on génère un simple nœud instruction avec son adresse dans le code et son argument (le nombre entier).

2.3 AST et génération de code

Dans cette dernière partie liée au compilateur, nous allons nous intéresser à l'AST et la génération de code pour la machine virtuelle (à pile) qui sera discutée dans la prochaine partie.

Pour cela, intéressons-nous à la structure utilisée pour les nœuds de l'AST.

```
struct node {
        int lno;
        int sym;
        union {
                int arg;
                char *name;
        };
        struct node *next;
};
```

On remarque immédiatement qu'il s'agit en réalité d'une liste chaînée et non d'un arbre (même si une liste chaînée est un arbre ...). En effet, comme le langage assembleur utilisé a un syntaxe très linéaire, il n'y a aucune structure de code qui soit assez complexe pour justifier l'utilisation de plusieurs fils par nœuds (si l'on avait pensé le langage en acceptant de simples expressions, on aurait du modifier cette structure).

Le membre **int** *lno* correspond à la ligne ou l'instruction apparait. **int** *sym* dénote du type de l'instruction (pop, push, call, etc.). L'**union** est utilisée pour donner l'information de soit l'argument numérique de l'instruction, ou l'identifiant (par exemple dans **jmp** *label*).

Construction de l'AST On utilise ensuite les trois fonctions déclarées dans le fichier **ast.h** :

```
struct node *new_ctrl(int sym, char *to, int lno);
struct node *new_instr(int sym, int arg, int lno);
struct node *new_ph(int sym, char *name, int lno);
```

qui servent respectivement à :

— Ajouter un nœud de contrôle (**jmp, call** ...)

— Ajouter un nœud pour une instruction classique (**pop** 10, ...)

— Ajouter un "*placeholder*" pour signaler la position d'un label (ignoré à l'étape de la génération de code, mais utilisé pour repérer un label dans le code).

Génération du code Pour générer le code associé à l'AST de notre programme, il nous faut le traverser de part en part.

A chaque nœud rencontré on augmente un **PC** fictif et on convertit les instructions en leur code, et on calcule leur argument si nécessaire. Prenons l'exemple d'un noeud de type **jmp** vers un label *main*.

Pour convertir cette instruction particulièrement, on doit récupérer l'adresse a laquelle on a rencontré le label (la valeur du PC fictif). Pour cela, on parcoure la Symtable (table des symboles) à la recherche du symbole *main*. Une fois trouvé on récupère la valeure de PC qui lui est associé dans la table et on calcule la différence entre ce PC et le courrant pour obtenir un *offset*.

Deuxième partie

La machine a pile

3 Charger le code – le loader

La machine virtuelle (à pile) de ce projet est donnée en entrée un fichier ou est écrit un code source en hexadécimal. Le but du *loader* est de charger ce code dans un tableau qu'il aura alloué et qui sera la zone de code utilisée par l'interpréteur.

Pour cela, le loader parcoure le fichier dans son entièreté une première fois et compte combien de ligne il y a. Pour chaque ligne trouvée il alloue 2 cases dans le tableau de code. Une pour l'instruction et une pour l'argument (même si zéro ou inutile pour l'instruction en question).

Lorsque le loader doit récupérer la valeur signée de l'argument, il doit la recomposer depuis les deux entiers séparés dans le fichier. Il décale d'abord de 8 bits vers la gauche la première valeur, et applique un OU bit à bit dessus avec la seconde. De cette manière il reconstitue un nombre signé de 16 bits a l'aide de deux nombres non signés de 8 bits.

Une fois le code chargé, le loader passe la main à l'interpréteur.

4 L'interpréteur de bytecode

Peut-être la partie au cœur de ce projet, l'interpréteur de bytecode est la ou toute la magie opère.

Pour rappel, la machine à pile possède (évidemment) une pile (le "*stack*") :

```
typedef struct {
        int size;
        int capacity;
        word *data;
} stack;
```

mais aussi un espace mémoire de 4000 mots de 16 bits et un registre PC. Le registre SP étant un pointeur vers la taille du stack.

```
typedef int16_t word; // memory.h

word mem[4000] = {0};
word pc = 0;
```

Les computed-gotos Tout l'intérêt de notre implémentation repose sur une extension C des compilateurs modernes (**clang, gcc**) qui permet d'extraire l'adresse d'un label avec une double esperluette [1].

Notre première implémentation utilisais un grand **switch** où l'on testais la valeur de l'instruction et exécutait un bout de code en fonction. Le programme fonctionnait très bien, mais cela posait un problème d'optimisation dans le sens ou un **switch** est compilé avec des instructions de vérification etc. et ne permet pas vraiment une prédiction de branches par le processeur.

Avec la technique des *computed goto*s et en compilant avec l'optimisation au maximum (-O3) le code généré est vraiment plus rapide. Pour ne pas élargir encore ce rapport voici un lien vers un article très intéressant qui en explique le fonctionnement [2].

Comme le code de l'instruction **halt** est 99, on doit créer une table de 100 pointeurs et remplir les codes non-utilisés avec un pointeur vers un label d'erreur.

1. https ://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html
2. http ://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables

Le *dispatch* est la structure de contrôle de cette approche. Elle permet de récupérer la prochaine instruction, son argument et renvoie le contrôle au label associé en allant le chercher dans la table. En pseudo code C on aurait :

```
void *table[]; // La table des labels

dispatch:
    op = code[pc++];
    arg = code[pc++];
    goto *table[op];

pop:
    mem[arg] = pop_stack();
    goto dispatch;
```

Implémentation Comment se matérialise cette implémentation dans le code ? Et bien on créé un label pour chaque instruction (ici **do_POP** par exemple) que l'on place dans la table des labels a l'indice correspondant a la valeur du code de l'instruction. On écrit ensuite le code associé à cette instruction à la suite et on retourne au *dispatch*.

Cette approche permet aussi de ne pas créer d'*overhead* (sur-utilisation) de la pile (celle du programme, pas du projet) en n'ayant pas d'appel de fonction à faire en permanence comme on pourrait trouver dans d'autres implémentations.

Troisième partie

Instructions additionnelles

Après avoir terminé le cahier des charges du projet et avoir écrit quelques programmes test, il nous a semblé intéressant d'ajouter quelques instructions à ce langage d'assemblage.

— **mswap**
Il s'agit d'une instruction très simple qui échange la valeur de deux cases mémoires dont les adresses sont sur la pile. Le code équivalent (sous forme de procédure) est :

```
mswap:  pop 3000
        pop 3001
        push 3000
        push 3001
        ipush
        ipush
        push 3001
        ipop
        push 3000
        ipop
        ret
```

Le haut de la pile contient les deux adresses à intervertir. On utilise ici les cases 3000 et 3001 comme des registres temporaires.

— **retz, retnz**
Ces deux instructions sont semblables a **ret** mais sont conditionnelles. retz est un retour de procédure si et seulement si le haut de la pile est zéro, retnz ne retourne que si le haut de la pile n'est pas zéro.

— **getc**
Cette instruction empile le prochain caractère tapé dans la console par l'utilisateur.

— **putc**
Similaire à **write**, **putc** permet d'afficher sur la console le charactère correspondant au haut de la pile (dépile la valeur).

Quatrième partie

Exemples d'exécution

Voici un programme qui implémente les fonctions **gets**, **puts** et **strlen** à l'aide des instructions getc et putc.

```
call main
halt

main: push# 0
      call gets
      push# 0
      call puts
      call ln
      push# 0
      call strlen
      pop 1000
      write 1000
      call ln
      ret

gets: pop 3000
_L1:  pop 3001   ; l'adresse courrante de la chaine de caracteres
      getc
      dup
      push# 10
      op 0
      jpc _L2    ; char == \n
      push 3001
      ipop
      push 3001
      push# 1
      op 11
      jmp _L1
_L2:  push# 0
      push 3001
      iPop
      pop 3001
      push 3000
      ret

puts: pop 3000   ; return address
_L3:  pop 3001   ; string address
      push 3001
      iPush
      dup
      jpc _L4    ; print it if it is not the null byte
      pop 3001
      push 3000 ; otherwise return to the caller
      ret
_L4:  putc
      push 3001
      push# 1
      op 11
      jmp _L3

ln:   push# 10
      putc
```

```
        ret

strlen: pop 3000   ; return address
        pop 3001   ; string address
        push# 0
        pop 3002
_L5:    push 3001
        ipush
        jpc _L6
        push 3002
        push 3000
        ret
_L6:    push 3001
        push# 1
        op 11
        pop 3001
        push 3002
        push# 1
        op 11
        pop 3002
        jmp _L5
```