INFO0025 - Cours de compilateurs et systèmes d'écriture des compilateurs
Implémentation d'un compilateur de z-matrices.

Adrien Kunysz, Sam Kyritsoglou

16 mai 2006

Table des matières

1 Anatomie d'une z-matrice

Une z-matrice est une manière pratique de représenter une molécule en utilisant des coordonnées relatives. Par exemple, une molécule d'éthane dans la configuration éclipsée pourra être représentée par :

```
C
C  1  1.5173893
H  2  1.11721396  1  107.8021607
H  2  1.11205843  1  111.37226    3 -122.3562469
H  2  1.11783779  1  107.4581078  3  115.7104335
H  1  1.1169207   2  111.0770073  3    0
H  1  1.11712199  2  111.0150133  5    0.8782819
H  1  1.11702994  2  110.9935404  6  120.0975124
```

Chaque ligne représente un atome et commence avec le symbole de cet atome. Les six champs suivants sont en fait des groupes de deux nombres, le premier indiquant l'atome auquel on fait référence (« 1 » indiquant l'atome défini à la première ligne,...), le second déterminant un angle ou une distance par rapport à cette référence. On a donc zéro à trois groupes pour chaque définition d'atome :

- le premier groupe indique la distance entre l'atome défini et l'atome de référence ;
- le deuxième groupe détermine l'angle formé par les deux atomes de référence et l'atome défini ;
- le troisième indique l'angle dièdre entre le plan formé par les trois atomes de référence et celui défini par les deux premiers atomes de référence et le nouvel atome.

On remarque que le premier atome défini n'a pas besoin de référence puisqu'il est lui même la position de référence de tous les autres atomes. La position du deuxième atome ne nécessite quant à elle que d'être définie par sa distance au premier. De la même manière, le troisième atome n'a besoin que d'une distance et d'un angle pour être positionné sans ambiguïté tandis que l'on a besoin de connaître l'angle dièdre de tous les atomes à partir du quatrième.

Cette notation est assez pratique car elle permet de définir précisement et manuellement la position des atomes dans une molécule de proche en proche et redessiner la molécule quand on en connaît sa z-matrice est assez facile. De même, lors de l'étude de la dynamique vribrationnel des molécules, la z-matrice permet de reperrer directement quels sont les atomes ou groupes d'atomes en mouvement et quelles types de mouvement.

La notation z-matrice permet aussi d'utiliser des variables – qui sont en fait constantes – à la place de la valeur directe des angles et distances. Ces variables sont déclarées et définies après la z-matrice chacune sur une ligne contenant son nom et sa valeur. Par exemple, pour définir la variable `CC` qui contient la valeur 1,517 389 3, on écrira `CC 1.5173893`.

Notre compilateur prend donc en entrée une z-matrice et calcule les coordonnées cartésiennes de chaque atome. Si on reprend l'exemple de la molécule d'éthane, on obtiendra un résultat comme celui ci :

```
C     0           0           0
C     0           0           1.517389
H     1.063719    0           1.858956
```

```
H    -0.554226    -0.874797     1.922652
H    -0.462606     0.960776     1.852750
H     1.042196     0           -0.401670
H    -0.437943     0.946401    -0.400614
H    -0.522978    -0.902274    -0.400190
```

Dans le cadre de modélisation de molécules, les différentes relations permettant le calcul d'énergie utilisent les coordonnées cartésiennes. Dès lors, l'utilisation de ces dernières permet une utilisation directe des relations, sans transformation supplémentaire.

1.1 Modifications de la syntaxe

Pour rendre la syntaxe d'une z-matrice analysable par la méthode vue au cours, nous avons été contraints de la modifier un peu.

La première modification a été d'ignorer les retours à la ligne car leur prise en compte complique sensiblement l'élaboration de règles de grammaire cohérentes. La séparation entre deux définitions d'atomes ou de variables est donc signalée par une virgule. De même, pour séparer les définitions des atomes et celles des variables, nous avons choisi d'utiliser le point d'exclamation.

Une autre modification qui a dû être apportée concerne le regroupement des couples « atome de référence »-« distance/angle ». Pour éviter toute ambiguïté, nous les avons simplement encadrés par des parenthèses.

Enfin, nous avons exclu la possibilité d'avoir une molécule ne contenant qu'un seul atome.

La syntaxe z-matrice en notation BNF devient donc :

```
<zmat> ::= <molecule> | <molecule> <excl> <declarations>
<declarations> ::= <declaration> | <declaration> <comma> <declarations>
<declaration> ::= <var> <num>
<molecule> ::= <var> <comma> <atom1> |
               <var> <comma> <atom1> <comma> <atom2> |
               <var> <comma> <atom1> <comma> <atom2> <comma> <atoms>
<atoms> ::= <atom> | <atom> <comma> <atoms>
<atom1> ::= <var> <couple>
<atom2> ::= <var> <couple> <couple>
<atom>  ::= <var> <couple> <couple> <couple>
<couple> ::= <openpar> <num> <val> <closepar>
<val> ::= <num> | <var>
```

avec les symboles terminaux :

```
<num> ::= -?[0-9]+(.[0-9]+)?
<var> ::= [A-Za-z][A-Za-z0-9]*
<excl> ::= !
<comma> ::= ,
<openpar> ::= (
<closepar> ::= )
```

On remarque que syntaxiquement les numéros de référence des atomes sont des nombres réels. Il convient donc de vérifier que ce sont bien des naturels par après, ce qui est fait dans la fonction `translater.c:double2uint` (page 51, ligne 90).

2 Structure générale du compilateur

Le compilateur est composé de deux programmes principaux : l'analyseur de syntaxe qui s'occupe de générer la table SR et les structures de données qui seront utilisées par le compilateur et le compilateur lui même qui est chargé d'analyser et traduire les z-matrices.

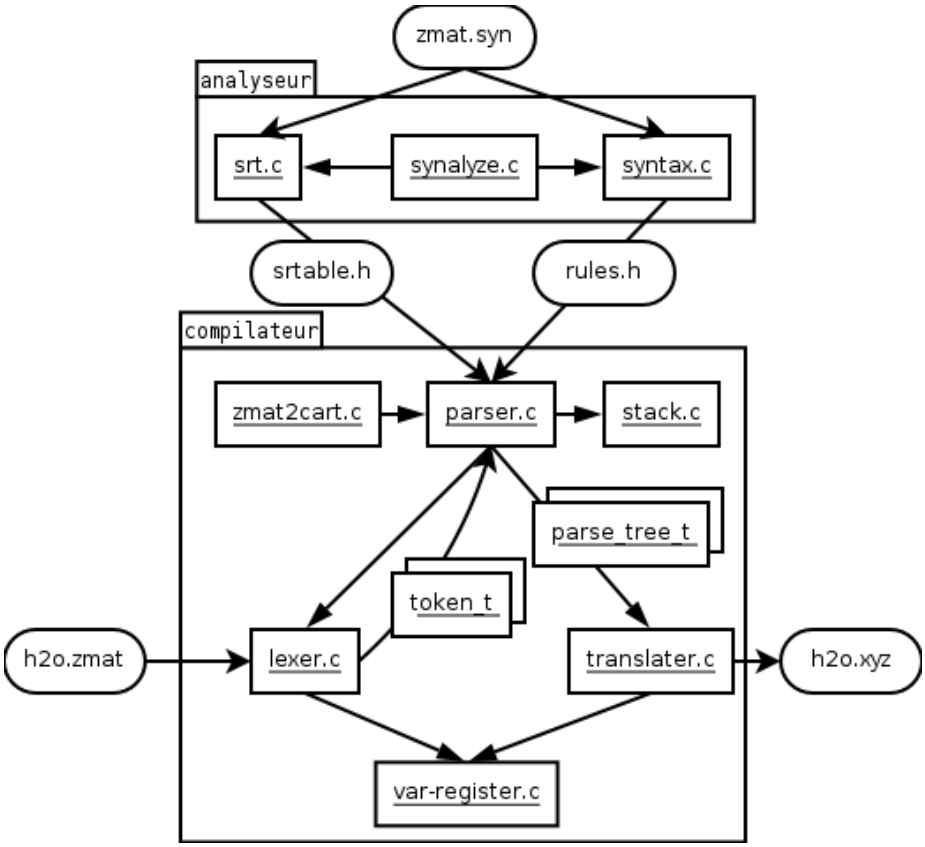

FIG. 1 – Architecture du compilateur.

2.1 L'analyseur de syntaxe

Pour que la syntaxe puisse être assez facilement modifiée, nous l'avons placée dans un fichier indépendant `syntaxes/zmat.syn` (page 21). Ce fichier contient la syntaxe sous forme de macros C qui sont inclus par les fichiers `srt.c` (page 33) et `syntax.c` (page 27). Lors de la compilation de ces fichiers, la syntaxe est automatiquement placée dans un tableau de structures pour être analysée par `srt.c:srt_build` (page 33, ligne 114) et `syntax.c:syntax_dumpc` (page 27, ligne 198) qui génèrent les fichiers `includes/srtable.h` et `includes/srt.h`. Ces fichiers contiennent respectivement la table SR et les règles syntaxiques sous forme de code source C qui sera inclus par le compilateur. Cette méthode rend l'analyseur complètement indépendant de la grammaire.

De plus, l'utilisation de macros pour définir les structures de données représentant les règles syntaxiques dans l'analyseur permet de réduire fortement les risques d'erreurs de programmation. Au lieu de devoir maintenir manuellement plusieurs structures interdépendantes, on définit entièrement chaque règle par un seul macro qui génère les structures nécessaires automatiquement.

En revanche, cette méthode gaspille assez bien de mémoire puisque la largeur du tableau de règles `syntax.c:rules` (page 27, ligne 37) correspond à la longueur de la plus longue règle. Cependant, ce gaspillage n'a lieu que dans l'analyseur de syntaxe et n'affecte pas le compilateur en lui même.

Les vérifications sur la syntaxe effectuées par `syntax.c:rules_check` (page 27, ligne 76) restent très basiques et il est nécessaire de vérifier manuellement qu'il n'existe pas de règle dont la partie droite est suffixe d'une autre. Les conflits de *shift/reduce* sont par contre bien détectés par `srt.c:srt_set` (page 33, ligne 47).

2.2 Le compilateur

Le compilateur est composé du *lexer* qui traduit le flux en entrée en symboles terminaux, du *parser* qui transforme cette suite de symboles terminaux en un arbre syntaxique et du traducteur qui est appellé par le parser après chaque réduction pour analyser l'arbre. Le lexer place les noms de variables qu'il trouve dans un registre de variables qui est utilisé ensuite par le traducteur.

Contrairement à ce que peut faire croire le schéma, le parser n'utilise pas directement la table SR et le tableau de règles syntaxiques. Il y accède *via* les fonctions *inline* `includes/srt.h:srt_get` (page 32, ligne 19) et `includes/syntax.h:get_rules_ending_with` (page 22, ligne 118) qui effectuent des vérifications supplémentaires sur les accès de manière à détecter plus facilement les erreurs de programmation. Pour éviter les ralentissements inhérents à ces vérifications, il est possible de les désactiver simplement à la compilation. De plus, l'attribut *inline* permet de faire en sorte que les appels à ces fonctions soient remplacés par les accés directs aux tableaux correspondants, ce qui augmente considérablement la vitesse d'accès aux éléments des tableaux. L'utilisation de la compilation conditionnelle grâce aux macros `RULES_FILE` et `SRT_FILE` permet de s'assurer que seules ces fonctions d'accès et leurs tableaux associés se retrouvent dans le programme final, de manière à économiser la mémoire gaspillée par le tableau de règles utilisé dans l'analyseur.

Le lexer tient à jour un compteur de lignes permettant de localiser les erreurs de syntaxe. Cependant, certaines erreurs ne sont détectées que plusieurs lignes plus tard. Cela signifie que lorsque le compilateur affiche un numéro de ligne pour une erreur, elle se trouve à cette ligne ou avant.

3 Structures de données et algorithmes utilisés

3.1 Registre de variables

Le registre de variable doit permettre d'associer des nombres à virgule flottante à des chaînes de caractères et de retrouver le nombre efficacement sur base du nom de la variable. Pour cela, nous avons décidé d'utiliser trois tableaux :

- `var-register.c:var_names` (page 38, ligne 21) qui contient les pointeurs vers les chaînes de caractères contenant les noms des variables ;

– `var-register.c:var_values` (page 38, ligne 26) qui contient les valeurs associées aux variables ;
– une table de hachage maintenue par la fonction standard `hsearch` dont les clés sont les noms des variables et les valeurs sont les index des variables dans les deux autres tableaux.

À chaque variable est associée un entier positif unique entre zéro et `MAX_VARS_COUNT`. Cet identifiant est la valeur associée au nom de la variable dans la table de hachage. Elle est aussi l'index de la variable dans les tableaux `var_names` et `var_values`.

Lorsque le lexer génère une variable sur base des caractères d'entrée, il appelle `var-register.c:register_var` (page 38, ligne 28) qui l'ajoute au registre si elle n'y est pas encore et retourne son identifiant. Par la suite, on se référera toujours à la variable en utilisant son identifiant et non son nom. Le traducteur associe par la suite une valeur à cette variable *via* `var-register.c:register_setval` (page 38, ligne 72) et la récupère le moment venu en utilisant `var-register.c:register_id2val` (page 38, ligne 65).

Initialement nous avons utilisé une *skiplist*[1] que nous avons implémentée nous même à la place de la table de hachage. Lors des tests avec de grosses z-matrices, nous nous sommes vite aperçus que le plus clair du temps d'exécution du compilateur était passé dans les accès à cette *skiplist*. Plutôt que d'essayer de réinventer la roue à nouveau, nous avons préféré utiliser les fonctions de table de hachage d'Unix. Après cette modification, les temps d'accès au registre de variables se sont vus considérablement diminués. Il est à noter que c'est la seule partie de notre code qui est un peu moins portable. Le reste se voulant entièrement conforme au standard ISO C99.

3.2 Types de symboles

Les types de symboles `includes/syntax.h:symbol_t` (page 22, ligne 29) sont une énumération des classes syntaxiques et symboles terminaux de la syntaxe. Les symboles sont numérotés de `0` à `S_LAST`. Les symboles les plus proches de zéro sont les symboles terminaux (de `TK_FIRST` à `TK_LAST`). Les classes syntaxiques sont situées après, de `S_FIRST` à `S_LAST`. Le symbole `S_DV` est un alias pour la variable distinguée, `S_ZMAT` dans notre cas. Un symbole spécial `S_INVALID` qui n'est présent dans aucune règle valide se trouve après `S_LAST`. En pratique, l'énumération dans le cas de notre syntaxe pourrait être représentée comme ceci :

```
typedef enum {
        TK_NUM,    // 0
        TK_VAR,
        /* ... */
        TK_CLOSEPAR,
        S_ZMAT,
        S_DECLARATIONS,
        /* ... */
```

[1] Une *skiplist* est une structure de donnée à équilibrage probabiliste permettant d'associer une clé à une valeur de manière presque aussi efficace qu'un arbre AVL mais plus facile à implémenter. Voir par exemple « *Skip Lists : A Probabilistic Alternative to Balanced Trees* » par William Pugh.

```
        S_VAL,
10      S_INVALID,
        TK_FIRST = 0,
12      S_DV = S_ZMAT,
        TK_LAST = S_DV - 1,
14      S_FIRST = S_DV,
        S_LAST = S_INVALID - 1
16  } symbol_t;
```

Comme tout ce qui est dépendant de la syntaxe, cette énumération est générée automatiquement par le préprocesseur C sur base du fichier `syntaxes/zmat.syn` (page 21).

3.3 Pile

La pile du parser est simplement une liste liée de tableaux. En plus des opérations classiques d'une pile (*push*, *pop*, *peek*), celle-ci a la particularité de permettre de retrouver le n^{e} élément sans le retirer de la pile *via* `stack.c:stack_peek` (page 17, ligne 133) et de retirer les n derniers éléments grâce à `stack.c:stack_discard` (page 17, ligne 92).

Pour éviter de devoir parcourir plusieurs tableaux lors d'une recherche de règle pendant une tentative de réduction, la taille des tableaux est fixée à deux fois la taille de la plus grande règle syntaxique.

3.4 Arbre syntaxique

L'arbre syntaxique est simplement composé de nœuds `includes/parser.h:parse_tree_t` (page 45, ligne 15) contenant deux pointeurs de nœuds – un fils et un frère – et un symbole de type `includes/lexer.h:token_t` (page 40, ligne 15). Un symbole est défini par un type et une valeur. La valeur n'a un sens que pour les nœuds terminaux dont le type est `TK_VAR` ou `TK_NUM`. Dans le premier cas c'est un identifiant de variable et dans le second un réel.

On remarque que l'on aurait sans doute pu gagner de la mémoire au prix d'une complexité accrue du code en exploitant le fait que seuls les nœuds terminaux (qui n'ont donc pas de fils) ont une valeur.

3.5 Table de règles pour l'analyse

Le tableau `syntax.c:rules` (page 27, ligne 37) des règles syntaxiques utilisées pour l'analyse de la syntaxe est généré par le préprocesseur C. C'est un tableau à deux dimensions : chaque élément du premier niveau contient le tableau des parties droites des règles dont la partie gauche est l'index (avec un décalage de S_DV). Par exemple, la règle

```
<zmat> ::= <molecule> | <molecule> <excl> <declarations>
```

sera représentée par un tableau de `includes/syntax.h:right_part_t` (page 22, ligne 62) comme ceci :

```
right_part_t zmatrules[] = {
2       { .len = 1, .right = { S_MOLECULE }},
        { .len = 3, .right = { S_MOLECULE, TK_EXCL, S_DECLARATIONS }},
4       { .len = 0, .right = { S_INVALID }}
};
```

qui se trouvera à l'index `S_ZMAT - S_DV` du tableau `rules` de premier niveau. Chaque tableau de second niveau est terminé par une règle invalide.

3.6 Table de règles pour le *parsing*

Le tableau de règles syntaxiques utilisé dans le parser n'est pas organisé de la même manière que celui de l'analyseur car il ne doit pas répondre aux mêmes besoins. Dans le cas du parser, on doit pouvoir retrouver la plus longue règle correspondant à une suite de symboles donnée (le haut de la pile). Pour cela nous avons choisi de regrouper les règles par leur dernier symbole et de les trier dans ces groupes selon leur longueur. En pratique, dans le cas de la syntaxe z-matrice, l'élément `S_COUPLE` du tableau `rules_ending_with` contiendra un pointeur vers un tableau de pointeurs de `includes/syntax.h:rule_t` (page 22, ligne 48) défini comme ceci :

```
// la partie droite de la regle la plus longue se terminant par S_COUPLE
static const symbol_t r_end_rS_COUPLE0[4] = {
        TK_VAR, S_COUPLE, S_COUPLE, S_COUPLE
};
// la regle la plus longue se terminant par S_COUPLE
static const rule_t r_end_S_COUPLE0 = {
        .left = S_ATOM,
        .len = 4,
        .right = r_end_rS_COUPLE0
};
// la partie droite de la 2eme regle la plus longue se terminant par S_COUPLE
static const symbol_t r_end_rS_COUPLE1[3] = {
        TK_VAR, S_COUPLE, S_COUPLE
};
// la 2eme regle la plus longue se terminant par S_COUPLE
static const rule_t r_end_S_COUPLE1 = {
        .left = S_ATOM2,
        .len = 3,
        .right = r_end_rS_COUPLE1
};
// la partie droite de la regle la plus courte se terminant par S_COUPLE
static const symbol_t r_end_rS_COUPLE2[2] = {
        TK_VAR, S_COUPLE
};
// la regle la plus courte se terminant par S_COUPLE
static const rule_t r_end_S_COUPLE2 = {
        .left = S_ATOM1,
        .len = 2,
        .right = r_end_rS_COUPLE2
};
// le tableau des regles se terminant par S_COUPLE vers lequel
// on mettra un pointeur dans rules_ending_with[S_COUPLE]
static const rule_t *r_ends_S_COUPLE[4] = {
        &r_end_S_COUPLE0, &r_end_S_COUPLE1, &r_end_S_COUPLE2, NULL
};
```

La recherche de la règle la plus longue correspondant au sommet de la pile s'effectuera alors simplement en parcourant le tableau associé au dernier symbole de la pile. Le tableau étant trié selon la longueur des règles, la première concordance sera la bonne. Cet algorithme, implémenté dans `parser.c:reduce` (page 46, ligne 69), est loin d'être optimal dans le cas général mais pour des règles syntaxiques qui sont en petit nombre et de longueurs réduites, il convient tout à fait.

3.7 Calcul des coordonnées

La conversion des coordonnées relatives vers les coordonnées cartésiennes qui a lieu dans `translater.c:rel2cart` (page 51, ligne 294) doit être effectuée dans l'ordre de déclaration de la z-matrice. En effet, le calcul des coordonnées d'un atome nécessite la connaissance des coordonnées de trois atomes auxquels il fait référence.

La méthode de calcul dépend de la position de l'atome dans la z-matrice :

- le premier atome est défini à l'orgine des axes ;
- la position du second se situe sur l'axe z, à la distance définie par rapport au premier atome ;
- le troisième atome se situe dans le plan xz. Dès lors, ces coordonnées sont calculées par projection sur les axes x et z de la distance par rapport à l'atome de référence, augmentées respectivement des coordonnées en x et en z de ce dernier.

Au delà du troisième atome, la méthode de calcul est identique pour tous les atomes. On peut résumer ce problème à trouver la position d'un point A, tout en connaissant la position de trois autres points, B, C et D, la distance $\|AB\|$, l'angle $\widehat{ABC}$ et l'angle formé par les plans déterminés par BCD et ABC.

On peut résoudre ce problème de la manière suivante, telle que représenté sur la figure 2. Soient le vecteur normal au plan BCD $\overrightarrow{V_1} = \overrightarrow{CB} \wedge \overrightarrow{CD}$ et $\overrightarrow{V_2} = \overrightarrow{V_1} \wedge \overrightarrow{CB}$ parallèle au plan BCD et perpendiculaire à $\overrightarrow{CB}$. Remarquons qu'après normalisation, les vecteurs $\overrightarrow{V_1}$, $\overrightarrow{V_2}$ et $\overrightarrow{CB}$ forment un système de trois vecteurs orthogonales et de normes unitaires.

Soient $\overrightarrow{V_3}$ la somme des projection de $\overrightarrow{V_1}$ et $\overrightarrow{V_2}$ sur la droite CB et $\overrightarrow{V_4}$ le résultat de l'inverse de la différence des projection de $\overrightarrow{V_3}$ et $\overrightarrow{CB}$ sur la droite AB. Ces deux opérations permettent un alignement du vecteur $\overrightarrow{V_4}$ sur le segment AB. On obtient alors les coordonnées de A par la somme des coordonnées de B et de $\|AB\| \cdot \overrightarrow{V_4}$.

Cet algorithme de transformation de coordonnées est largement basé sur le programme Cart de David Lauvergnat[2].

4 Mesures de performances et améliorations possibles

Le *profilage* du compilateur nous a permis de détecter et améliorer un certain nombre de points critiques au niveau des performances mais toutes les optimisations possibles n'ont pas été réalisées aussi bien par manque de temps que par souci de garder un code assez simple.

[2] http ://www.lcp.u-psud.fr/Pageperso/lauvergnat/cart/cart.html

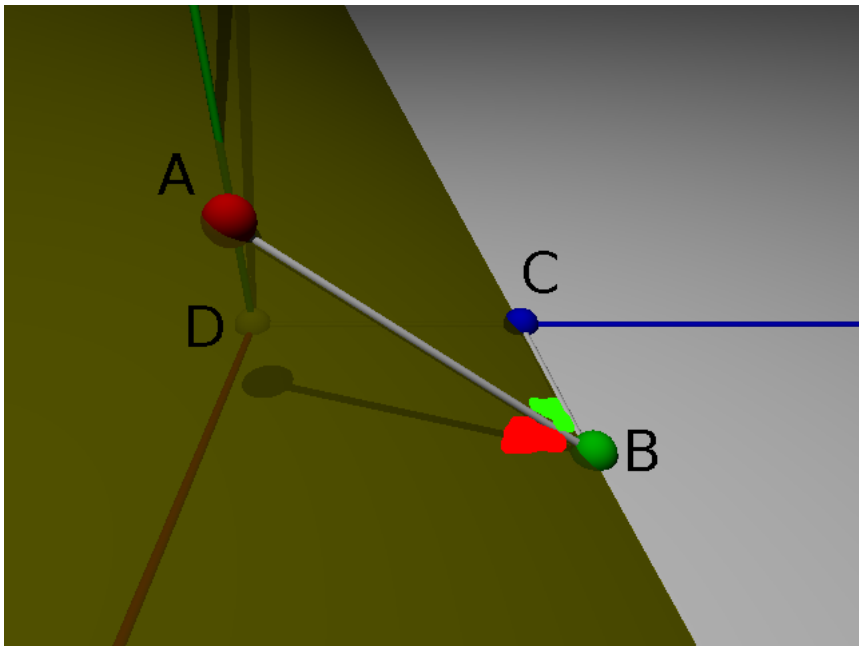

FIG. 2 – L'angle dièdre entre les deux plans en rouge, l'angle $\widehat{ABC}$ en vert.

4.1 Complexité algorithmique

L'optimisation manquante la plus évidente se trouve sans doute dans la recherche de règle correspondant au sommet de la pile dans `parser.c:reduce` (page 46, ligne 69). L'algorithme parcourt toutes les règles se terminant par le dernier élément de la pile de la plus longue à la plus courte jusqu'à trouver une correspondance sans jamais exploiter les données qu'il aurait pu récolter auparavant. Les mesures indiquent que c'est effectivement un des endroits dans lequel le compilateur passe le plus de temps. Cependant, cette perte de temps n'est pas si significative que l'on pourrait le croire et l'amélioration de cette partie nécessiterait de compliquer assez fortement le code.

En pratique, `gprof` indique que lorsque le nombre d'atomes augmente dans la z-matrice, le temps passé dans la traduction augmente plus vite que celui passé à rechercher la règle correspondant à une réduction. La fonction la plus intéressante à optimiser serait donc `translater.c:rel2cart` (page 51, ligne 294). Améliorer l'algorithme semble difficile mais il serait sans doute possible de gagner un peu de temps avec une réécriture de la fonction en assembleur ou en Fortran (qui a la réputation d'être très rapide pour ce type d'application).

Un profilage ligne par ligne montre qu'il est sans doute possible d'accélérer `lexer.c:lex_next` (page 41, ligne 120) en utilisant notre propre tampon de lecture ainsi que `parser.c:new_ptree` (page 46, ligne 28) et `parser.c:free_tree_down` (page 46, ligne 177) en utilisant un tableau de `parse_tree_t` au lieu d'appeller systématiquement `malloc` et `free` pour gérer les nœuds de l'arbre. Cependant, ces modifications compliqueraient significativement le code pour un gain de performances minime.

Nous n'avons pas particulièrement cherché à optimiser l'analyseur de syntaxe

temps (%)	appels	nom	temps (%)	appels	nom
18.75	1	print_atoms	21.64	1	print_atoms
15.83	1007234	reduce	14.14	4861374	reduce
12.92	1710848	lex_next	12.06	8292058	lex_next
11.53	1	parse	11.30	1	parse
11.25	1007234	on_reduce	10.35	4861374	on_reduce
4.17	103624	register_setval	4.82	13153432	stack_push
4.03	1007234	stack_discard	4.00	14584120	stack_peek
3.89	3021700	stack_peek	3.79	430694	register_setval
3.61	2718082	stack_push	3.53	4861374	stack_discard
3.33	3008452	xmalloc	2.90	1499994	register_id2val
3.19	407245	free_tree_real	2.79	14492890	xmalloc
2.22	1199976	free_tree_down	2.74	5999976	free_tree_down
2.08	503618	register_var	2.03	1861385	free_tree_real
1.53	299994	register_id2val	1.90	2430688	register_var
0.83	2904824	xfree_real	1.26	14062192	xfree_real
0.83	100000	register_id2str	0.76	500000	register_id2str

TAB. 1 – Les seize fonctions les plus consommatrices de temps pour des z-matrices de 100 000 (à gauche) et 500 000 atomes (à droite). Il est à noter que beaucoup d'appels de fonctions ont été remplacés par leur corps par le compilateur C (notamment, `rel2cart` est inclus dans `print_atoms`).

car il n'a normalement besoin que d'être lancé une seule fois et dans notre cas la taille de la grammaire est suffisament petite pour que même un algorithme plus que linéaire la traite en un temps raisonnable. Néanmoins, notre analyseur de syntaxe devrait quand même se comporter correctement même avec des grammaires très grandes (si on exclut la consommation mémoire excessive) bien que nous ne l'ayons pas testé dans ce cas de figure.

4.2 Complexité spatiale

Pour limiter l'utilisation de la mémoire, le traducteur libère les nœuds de l'arbre syntaxique dès qu'il n'en n'a plus besoin. Ainsi, l'arbre n'est jamais complet en mémoire. Cependant, ce système ne va pas aussi loin qu'il le pourrait. Idéalement, il ne devrait jamais y avoir plus de nœuds en mémoire qu'il n'en faut pour représenter un atome. Cela nécessite d'effectuer des modifications assez importantes au niveau de la pile et du parser. Ce dernier deviendrait aussi sans doute moins indépendant de la syntaxe. N'ayant pas trouvé de solution satisfaisante à ce problème, nous avons gardé un compilateur qui nécessite autant de nœuds qu'il y a de variables définies.

La taille des trois tableaux utilisés dans `var-register.c` (page 38) est fixée à `MAX_VARS_COUNT` à la compilation. Cela signifie qu'il y a un nombre maximum de variables qui peuvent être utilisées et que la consommation mémoire du registre de variables est fixée. En pratique cela ne pose pas vraiment de problème mais il est important d'en tenir compte si l'on désire utiliser notre compilateur dans des environnements à la quantité de mémoire extrêmement réduite ou lorsque l'on manipule des molécules démesurément grandes. Ce problème potentiel pourrait être réglé en remplaçant la table de hachage par une structure plus dynamique

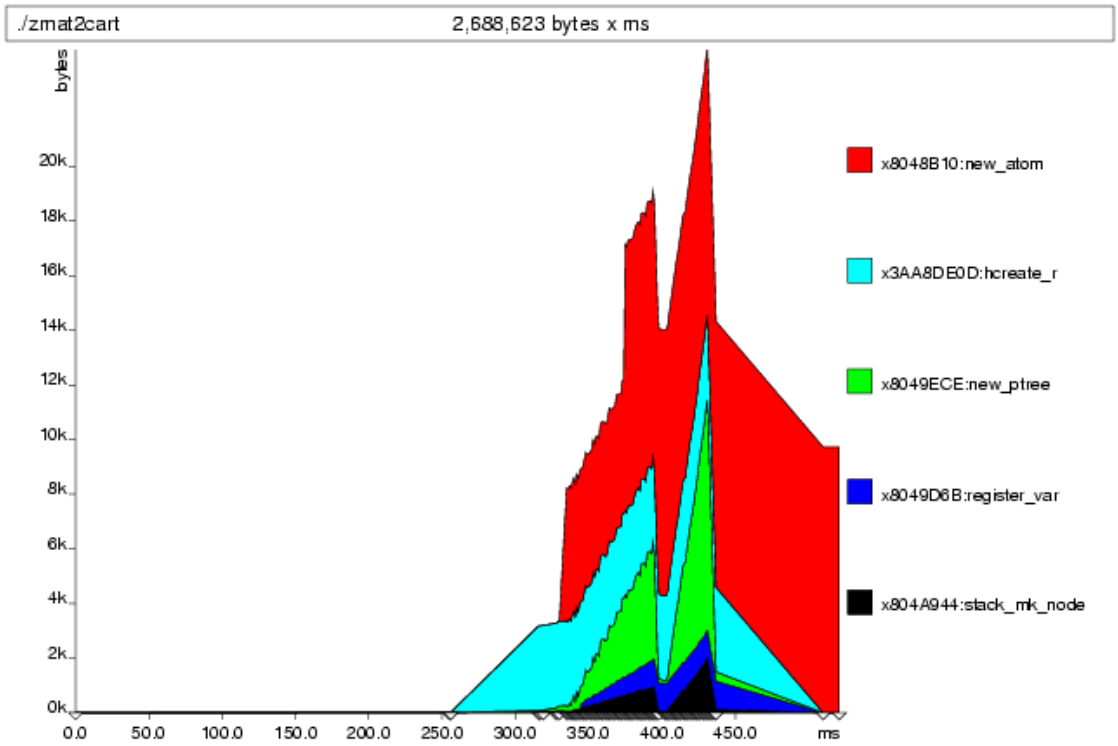

FIG. 3 – Utilisation du tas par les cinq fonctions les plus consommatrices de mémoire pour une z-matrice de 100 atomes quand `MAX_VARS_COUNT` vaut 256 et `ATOMS_ARRAY_BLK_SZ` vaut 64. On peut clairement distinguer l'analyse des atomes (avant 400 ms) de celle des variables (second pic). La mesure de temps affichée n'est pas représentative du temps réel que prend le programme quand il n'est pas profilé.

telle qu'un arbre binaire et en redimensionnant les deux autres tableaux à la volée comme c'est le cas pour le tableau d'atomes utilisé dans le traducteur dont l'espace est alloué par blocs de `ATOMS_ARRAY_BLK_SZ` éléments.

La taille occupée par le tableau de règles utilisé dans l'analyseur de syntaxe est proportionnelle à

```
(S_LAST - S_DV + 1)*MAX_RULES_PER_CLASS*MAX_RULE_LEN
```

ce qui signifie que selon la syntaxe, il peut rapidement prendre beaucoup de place pour rien puisque la plupart des éléments ne seront jamais utilisés. Nous n'avons cependant pas jugé essentiel de réduire cette occupation mémoire. En effet, ce tableau n'est utilisé que lors de l'analyse de la syntaxe qui n'a lieu qu'une seule fois. Le compilateur contient lui une version beaucoup plus compacte des règles syntaxiques qu'il est par contre impossible de créer uniquement sur base de macros.

A Code source du compilateur

Les fichiers ont été rassemblés par catégorie :
- les premiers contiennent les fonctions génériques qui ne sont pas directement liées au compilateur ;
- le deuxième groupe rassemble les fichiers formant l'analyseur de syntaxe ;
- suivent les fichiers utilisés par le compilateur proprement dit ;
- les derniers fichiers contiennent tout le code non essentiel comme les programmes de test,...

Listings

A.1 Code générique

Listing 1 – includes/misc.h

```
#ifndef _MISC_H_
#define _MISC_H_

#include <stddef.h>    // size_t
#include <stdlib.h>    // EXIT_FAILURE, exit()
#include <stdio.h>     // fprintf(), stderr
#include <math.h>      // islessequal(), isgreaterequal()

/**
 * Like realloc(), malloc() and calloc() but err() on error.
 */
void *xrealloc(void *ptr, size_t size);
void *xmalloc(size_t size);
void *xcalloc(size_t size);
void xfree_real(void *ptr);

/**
 * Detect some memory bugs more easily.
 */
#define xfree(p) do { xfree_real(p); p = NULL; } while (0)

#define err(...) do { fprintf(stderr, __VA_ARGS__);   \
                      exit(EXIT_FAILURE);             \
                    } while (0)

#ifdef NDEBUG
#  define debug(...)
#else
#  define debug(...) fprintf(stderr, __VA_ARGS__)
#endif

#define MIN(x,y) (((x) > (y)) ? (y) : (x))

/**
 * Compute the number of element in the array.
 * As seen in ISO/IEC 9899 section 6.5.3.4 § 6.
 */
#define countof(t) (sizeof(t)/sizeof(t[0]))

/**
 * #define FOO 42
 * stringify(FOO) => "FOO"
 */
#define stringify(x) #x

/**
```

```
 * #define FOO 42
 * xstringify(FOO) => "42"
 */
#define xstringify(x) stringify(x)

/**
 * Converts a char representing an integer to said integer.
 * Will only work for ASCII chars.
 */
unsigned int char2int(const char c);

/**
 * Floating point comparison.
 */
#define iseq(x, y)  (islessequal((x), (y)) && isgreaterequal((x), (y)))

#endif
```

Listing 2 – misc.c

```
#include <assert.h>
#include "misc.h"

void *xrealloc(void *ptr, size_t size)
{
        void *retval = realloc(ptr, size);
        if (NULL == retval && 0 != size)
                err("realloc failed\n");
        return retval;
}

void *xmalloc(size_t size)
{
        return xrealloc(NULL, size);
}

void xfree_real(void *ptr)
{
        free(ptr);
}

void *xcalloc(size_t size)
{
        void *retval = calloc(size, 1);
        if (NULL == retval && 0 != size)
                err("calloc failed\n");
        return retval;
}

unsigned int char2int(const char c)
{
```

```
32          assert(c >= '0' && c <= '9');
            return (unsigned int)(c - '0');
34  }
```

Listing 3 – includes/stack.h

```
    #ifndef _STACK_H_
2   #define _STACK_H_

4   #include <stdbool.h> // boolean type
    #include "syntax.h"  // MAX_RULE_LEN
6
    #define STACK_DATA_BLOCK_SZ (2*MAX_RULE_LEN)
8
    /**
10   * Structure of node of the stack.
     */
12  typedef struct stack_node_t {
            void *data[STACK_DATA_BLOCK_SZ];
14          unsigned int position;
            struct stack_node_t *next;
16  } stack_node_t;

18  /**
     * Structure of the head of the stack
20   */
    typedef struct stack_head_t {
22          unsigned int nb_elem;
            stack_node_t *next;
24  } stack_head_t;

26  /**
     * Create the head of the stack and return a pointer to this head
28   */
    stack_head_t *stack_mk(void);
30
    /**
32   * Add the element e to the top of the stack
     */
34  void stack_push(stack_head_t* head, void *e);

36  /**
     * return and remove the element at the top of the stack,
38   * return NULL is stack is empty.
     */
40  void* stack_pop(stack_head_t* head);

42  /**
     * Free all elements of the stack and the stack itsself.
44   */
    void stack_remove(stack_head_t* head);
```

```
46
   /**
48  * Free only the stack (head and node(s)).
    */
50 void stack_free(stack_head_t* head);

52 /**
    * Returns the ith element from the top of the stack without removing it.
54  * i must be less than the number of elements in the stack.
    *       stack_head_t *s = stack_mk();
56  *       stack_push(s, foo);
    *       stack_push(s, bar);
58  *       stack_push(s, baz);
    *       assert(stack_peek(s, 2) == foo);
60  *       assert(stack_peel(s, 0) == baz);
    *       stack_peek(s, 3);  // bug
62  */
   void *stack_peek(const stack_head_t* head, unsigned int i);
64
   /**
66  * Discard the topmost i elements from the stack.
    * It's like calling stack_pop(head) i times but faster.
68  */
   void stack_discard(stack_head_t* head, unsigned int i);
70
   #endif
```

Listing 4 – stack.c

```
   #include <assert.h>  // assert()
2  #include <stdio.h>  // printf() for debug

4  #include "stack.h"
   #include "misc.h" // xmalloc(), xfree()
6
   /**
8   * Alloc space for a node,
    * initialize all the element of the array to NULL.
10  * Returns the pointeur to the node.
    */
12 static stack_node_t *stack_mk_node(stack_node_t *next)
   {
14         stack_node_t *tmp;
           tmp = xmalloc(sizeof(*tmp));
16         tmp->next = next;
           tmp->position = 0;
18         for (int i = 0; i < STACK_DATA_BLOCK_SZ; i++){
                   tmp->data[i] = NULL;
20         }
           return tmp;
22 }
```

```
24 
   stack_head_t *stack_mk(void)
   {
26         stack_head_t *tmp;
           tmp = xmalloc(sizeof(*tmp));
28         tmp->nb_elem = 0;
           tmp->next = NULL;
30         return tmp;
   }
32 
   /**
34  * return true is the stack is empty
    */
36 static bool is_empty(stack_head_t *head)
   {
38         assert(NULL != head);
           return (0 == head->nb_elem);
40 }

42 void stack_push(stack_head_t* head, void *e)
   {
44         assert(NULL != head);
   //      debug("in push\t nb_elem : %u\n",head->nb_elem);
46         if (NULL == head->next){
   //              debug("in push\t no node\n \t create one\n");
48                 head->next = stack_mk_node(NULL);
           } else if (STACK_DATA_BLOCK_SZ == head->next->position){
50 //              debug("in push\t next node is full"
   //                    " and postion = %u\n",head->next->position);
52                 head->next = stack_mk_node(head->next);
           }
54         ++(head->nb_elem);
           head->next->data[head->next->position] = e;
56         (head->next->position)++;
   }
58 /**
    * Remove first stack node and change the pointer to the next stack node.
60  */
   static void stack_rm_node(stack_head_t *head)
62 {
           assert(NULL != head);
64         assert(NULL != head->next);
           stack_node_t *tmp = head->next;
66         head->next = head->next->next;
           xfree(tmp);
68 }

70 void* stack_pop(stack_head_t* head)
   {
72         assert(NULL != head);
```

```
74          if (is_empty(head)){
                    return NULL;
76          }

78  //      debug("in pop\t nb_elem : %u\n",head->nb_elem);

80          void *e = head->next->data[--(head->next->position)];

82          head->next->data[head->next->position] = NULL;

84  //      debug("in pop\t head->next->position = %u\n",head->next->position);
            if (0 == head->next->position){
86                  stack_rm_node(head);
            }
88          --head->nb_elem;
            return e;
90  }

92  void stack_discard(stack_head_t* head, unsigned int i)
    {
94          assert(NULL != head);
            assert(i < head->nb_elem);
96
            head->nb_elem -= i;
98          for (unsigned int j = i; j != 0 ; --j) {
                    head->next->data[(--head->next->position)] = NULL;
100                 if (0 == head->next->position){
                            stack_rm_node(head);
102                 }
            }
104 }

106 void stack_free(stack_head_t *head)
    {
108         assert(NULL != head);

110         while (NULL != head->next) {
                    stack_rm_node(head);
112         }
            xfree(head);
114 }

116 void stack_remove(stack_head_t* head)
    {
118         assert(NULL != head);

120         while (NULL != head->next) {
                    for (unsigned int i = 0 ; i < head->next->position; ++i) {
122 //                          debug("in stack_remove\t for : pos = %u\n",i);
```

```
                        if (NULL != head->next->data[i])
//                              debug("in stack_remove\t for :"
//                                    " free data[%u]\n",i);
                                xfree(head->next->data[i]);
                }
                stack_rm_node(head);
        }
        xfree(head);
}

void *stack_peek(const stack_head_t* head, unsigned int i)
{
        assert(NULL != head);
        assert(NULL != head->next);
        assert(i < head->nb_elem);

        stack_node_t *tmp = head->next;
        while(i >= tmp->position) {
                i -= tmp->position;
                tmp = tmp->next;
        }
        return tmp->data[tmp->position - i - 1];
}
```

A.2 Analyseur syntaxique

Listing 5 – syntaxes/zmat.syn

```
// This is the file containing the syntaxic rules for the zmat language.
// It will be #included with some tricks by syntax.h and syntax.c
// to generate the internal representation of the syntax that
// will be analyzed by srt_build() and syntax_dumpc().
// It means you can use C99 comments but you shouldn't #define anything
// excepted MAX_RULE_LEN and MAX_RULES_PER_CLASS without #undefining it after.
// The order of the macros calls and #defines within this file doesn't matter.
// TODO: write a BNF to syn converter or something

/**
 * The maximum length of a rule. If you set it too low,
 * you may see warnings when compiling syntax-analyzer.o depending on
 * your C compiler (GCC 4 will complain) and the generated
 * shift-reduce table will be corrupted (if synalize doesn't crash).
 * If you set it too high synalyze will just take more memory but
 * the final compiler won't be affected.
 * If your longest rule looks like this:
 *      <foo> ::= <bar> <baz> <quux> <titi> <toto>
 * then MAX_RULE_LEN must be set to 5 (or higher).
 */
#define MAX_RULE_LEN   7

/**
 * Maximum number of rules for a class. As for MAX_RULE_LEN
 * you better set it too high than too low.
 * If you have a rule like this:
 *      <foo> ::= <bar> | <baz> | <quux> <titi> | <toto>
 * then MAX_RULES_PER_CLASS must be at least 4.
 */
#define MAX_RULES_PER_CLASS  4

/**
 * Each token must be defined with DEF_TK(sym) with
 * sym the name that will be used to refer to it.
 * That name can contain only ASCII alphanumerics characters.
 */
DEF_TK(VAR)
DEF_TK(NUM)
DEF_TK(OPEN_PAR)
DEF_TK(CLOSE_PAR)
DEF_TK(COMMA)
DEF_TK(EXCL)

/**
 * The distinguished variable representing a complete zmat.
 * Obviously DEF_DV(sym, rules) must be called only once.
```

```
 * See the DEF_SC() description for details.
 */
DEF_DV(ZMAT, { 1, { SC(MOLECULE) }},
             { 3, { SC(MOLECULE), TK(EXCL), SC(DECLARATIONS) }})

/**
 * Each syntaxic class must be defined with DEF_SC(sym, rules) with
 * sym the name that will be used to refer to it and rules the rules
 * defining it.
 * The rules argument can be split into as many rules as you need for
 * the given class (up to MAX_RULES_PER_CLASS) since DEF_SC(sym, rules) is
 * really DEF_SC(sym, ...). A rule has two parts. First is its length (that
 * must be less or equal to MAX_RULE_LEN), second is an array of SC() and TK()
 * defining the rule itself.
 * So if you have a rule like this:
 *      <foo> ::= <bar> | <baz> <quux> <titi>
 * with <quux> a token, the corresponding DEF_SC() should look like this:
 *      DEF_SC(FOO, { 1, { SC(BAR) }},
 *                  { 3, { SC(BAZ), TK(QUUX), SC(TITI) }})
 * SC(foo) is used to refer to the foo syntaxic class and TK(foo) refers
 * to the foo token. Notice you don't need to DEF_*() a token or a syntaxic
 * class before refering to it.
 */
DEF_SC(DECLARATIONS,{ 1, { SC(DECLARATION) }},
                    { 3, { SC(DECLARATION), TK(COMMA), SC(DECLARATIONS) }})
DEF_SC(DECLARATION, { 2, { TK(VAR), TK(NUM) }})
DEF_SC(MOLECULE, /* { 1, { TK(VAR) }}, */
                 { 3, { TK(VAR), TK(COMMA), SC(ATOM1) }},
                 { 5, { TK(VAR), TK(COMMA), SC(ATOM1), TK(COMMA), SC(ATOM2) }},
                 { 7, { TK(VAR), TK(COMMA), SC(ATOM1), TK(COMMA), SC(ATOM2),
                        TK(COMMA), SC(ATOMS) }})
DEF_SC(ATOM1, { 2, { TK(VAR), SC(COUPLE) }})
DEF_SC(ATOM2, { 3, { TK(VAR), SC(COUPLE), SC(COUPLE) }})
DEF_SC(ATOM,  { 4, { TK(VAR), SC(COUPLE), SC(COUPLE), SC(COUPLE) }})
DEF_SC(ATOMS, { 1, { SC(ATOM) }},
              { 3, { SC(ATOM), TK(COMMA), SC(ATOMS) }})
DEF_SC(COUPLE, { 4, { TK(OPEN_PAR), TK(NUM), SC(VAL), TK(CLOSE_PAR) }})
DEF_SC(VAL, { 1, { TK(NUM) }}, { 1, { TK(VAR) }})
```

Listing 6 – includes/syntax.h

```
#ifndef _SYNTAX_H_
#define _SYNTAX_H_

#include <stddef.h>   // size_t
#include <stdbool.h>  // bool
#include <stdio.h>    // FILE
#include <assert.h>   // assert(3)

#include "misc.h"  // xstringify()

```

```
// Notice at lot of things here are declared depending
// on the RULES_FILE macro. If RULES_FILE is defined, only the
// types and functions needed by the parser will be available.
// If RULES_FILE is undefined, only the ones needed by the syntax
// analyser will be available.

#define SC(sym) S_ ## sym
#define TK(sym) TK_ ## sym

/**
 * A symbol is a syntaxic class or a token.
 * Tokens are first, starting at zero, ending at TK_LAST.
 * TK_INPUT is inserted at the end of tokens (it's TK_LAST).
 * Syntaxic classes are second, starting at S_DV (which
 * is an alias for the distinguished variable), ending at
 * S_LAST.
 * S_INVALID is a "NULL" value that indicate an error.
 */
typedef enum {
        #define DEF_TK(name)      TK(name),
        #define DEF_SC(sym, ...) SC(sym),
        #define DEF_DV(sym, ...) DEF_TK(INPUT) SC(sym), S_DV = SC(sym),
        #include SYNTAX_FILE
        #undef DEF_TK
        #undef DEF_SC
        #undef DEF_DV
        S_INVALID,
        S_LAST = S_INVALID - 1,
        S_FIRST = S_DV,
        TK_FIRST = 0,
        TK_LAST = S_DV - 1
} symbol_t;

#undef SC
#undef TK

#ifdef RULES_FILE  // compiling parser
typedef struct {
        const size_t len;       // length of the right[] array
        const symbol_t left;
        const symbol_t *right;
} rule_t;
#else  // compiling syntax analyzer
/**
 * The right part of a syntaxic class definition.
 * If
 *      <foo> ::= <bar> <baz> | <quux>
 * then there will be two corresponding right_part_t
 *      right_part_t foo0 = { 2, { S_BAR, S_BAZ }};
 *      right_part_t foo1 = { 1, { S_QUUX }};
```

```
 */
typedef struct {
        /**
         * The length of the right part of the rule.
         * That's the number of symbols in it.
         * len < MAX_RULE_LEN
         */
        const size_t len;  // FIXME: use a shorter type
        const symbol_t right[MAX_RULE_LEN];
} right_part_t;

typedef struct {
        symbol_t left;
        const right_part_t *right;
} rule_t;
#endif

/**
 * Tells wether a symbol is a valid symbol_t.
 */
static inline bool is_symbol(const symbol_t sym)
{
        return sym <= S_LAST /* && sym >= TK_FIRST */;
}

/**
 * Tells wether a symbol is a valid token.
 */
static inline bool is_token(const symbol_t sym)
{
        return sym <= TK_LAST /* && sym >= TK_FIRST */;
}

/**
 * Tells wether a symbol is a valid non-token symbol.
 */
static inline bool is_synclass(const symbol_t sym)
{
        return sym <= S_LAST && sym > TK_LAST;
}

/**
 * Returns a human-readable representation of the symbol.
 */
static inline const char *sym2str(const symbol_t sym)
{
        assert(is_symbol(sym));
        extern const char *symstrs[S_LAST+2];
        return symstrs[sym];
}
```

```
112 
113 #ifdef RULES_FILE  // compiling parser
114 /**
115  * Returns an array of pointers to all the rules ending with sym
116  * reverse sorted by length (longest rules are first).
117  * Last element of the array is NULL.
118  */
119 static inline const rule_t **get_rules_ending_with(const symbol_t sym)
120 {
121         extern const rule_t **rules_ending_with[S_LAST - TK_FIRST + 1];
122         assert(is_symbol(sym));
123         return rules_ending_with[sym];
124 }
125 #else  // the rest of the file is only for the syntax analyzer
126 /**
127  * Returns the number of rules corresponding to the
128  * given syntaxic class.
129  * If
130  *      <foo> ::= <bar> <baz> | <quux> | <toto>
131  * then
132  *      rules_count(S_FOO) == 3
133  */
134 size_t rules_count(const symbol_t sym);
135 
136 /**
137  * If
138  *      <foo> ::= <bar> <baz> <quux> | <titi> <toto>
139  * then
140  *      last_class(getrule(S_FOO, 1)) == S_TOTO
141  */
142 static inline symbol_t last_sym(const right_part_t *rule)
143 {
144         assert(NULL != rule);
145         return rule->right[rule->len - 1];
146 }
147 
148 /**
149  * If
150  *      <foo> ::= <bar> <baz> <quux> | <titi> <toto>
151  * then
152  *      first_class(getrule(S_FOO, 0)) == S_BAR
153  */
154 static inline symbol_t first_sym(const right_part_t *rule)
155 {
156         assert(NULL != rule);
157         return rule->right[0];
158 }
159 
160 /**
```

```
160  * Iterate the variable rule over the right_part_t
     * rules of sym. You don't have to declare the rule variable.
162  * Don't assume this macro expands to a single statement.
     * DON'T DO THIS:
164  *      FOR_EACH_RULE(S_DV, r)
     *              do_stuff(r);
166  * Write this instead:
     *      FOR_EACH_RULE(S_DV, r) {
168  *              do_stuff(r);
     *      }
170  * The type of r is const right_part_t *.
     * Beware of the i local variable.
172  */
     #define FOR_EACH_RULE(sym, rule)
     \
174          const right_part_t *rule = NULL;
     \
             for (unsigned int i = 0;
     \
176                  (rule = get_rule(sym, i)) && 0 != rule->len;
     \
                     i++)
178
     /**
180  * Returns the numth rule for the given
     * syntaxic class (starting at zero).
182  * num must be <= rules_count(sym)
     * If
184  *      <foo> ::= <bar> <baz> | <quux> | <toto>
     * then
186  *      getrule(S_FOO, 0)
     * will return a right_part_t pointer to a structure like this:
188  *      { 2, { S_BAR, S_BAZ }}
     * If
190  *      num == rules_count(sym)
     * then the returned struct is an invalid rule like this:
192  *      { 0, { S_INVALID }}
     */
194  const right_part_t *get_rule(const symbol_t sym, const unsigned int num);

196  /**
     * Check rules for consistency.
198  * Will abort() on error.
     * Successful termination of this function doesn't necessarily
200  * indicate that the rules are consistent.
     */
202  void rules_check(void);

204  /**
     * Print the syntaxic rules as an array of pointers to arrays
```

```
206  * of pointers to rule_t. Each rule_t* array contains all the
     * rules ending with a given symbol. The last element of the
208  * array being NULL. Within these arrays, rules are sorted by
     * length in descending order.
210  * The rule_t** array which is named "rules_ending_with" indexes
     * rule_t* arrays by their ending symbol. If a symbol is not the
212  * last element of any rule, its value in the array will be NULL.
     *
214  * In practice you should get something like this:
     *      static const rule_t r_end_TK_BAR0 = {
216  *              .left = S_FOO,
     *              .len = 3,
218  *              .right = { TK_BAR, S_BAZ, TK_BAR }
     *      };
220  *      static const rule_t r_end_TK_BAR1 = {
     *              .left = S_QUUX,
222  *              .len = 2,
     *              .right = { S_TOTO, TK_BAR }
224  *      }
     *      static const rule_t *r_ends_TK_BAR[3] = { &r_end_TK_BAR0, &r_end_TK_BAR
226  *
     *      static const rule_t **rules_ending_with[] = {
228  *              [TK_BAR] = r_ends_TK_BAR,
     *              [S_BAZ] = NULL,
230  *              ...
     *      };
232  */
    void syntax_dumpc(FILE *output);
234 #endif  // RULES_FILE

236 #endif  // _SYNTAX_H_
```

Listing 7 – syntax.c

```
    #include <assert.h>   // assert(3)
2   #include <stddef.h>   // NULL, size_t
    #include <stdlib.h>   // abort(3), qsort(3)
4   #include "syntax.h"

6   /**
     * Used by sym2str() for both syntax analyzer and parser.
8    */
    const char *symstrs[S_LAST+2] = {
10          #define SC(sym) S_ ## sym
            #define TK(sym) TK_ ## sym
12          #undef MAX_RULE_LEN
            #undef MAX_RULES_PER_CLASS
14          #define DEF_TK(name) xstringify(TK(name)),
            #define DEF_SC(sym, ...) xstringify(SC(sym)),
16          #define DEF_DV(sym, ...) DEF_TK(INPUT) DEF_SC(sym, __VA_ARGS__)
            #include SYNTAX_FILE
```

```
18          #undef DEF_TK
            #undef DEF_SC
20          #undef DEF_DV
            #undef SC
22          #undef TK
};

24
#ifdef RULES_FILE  // compiling parser
26  #  include RULES_FILE
#else  // rest of the file is only for syntax analyzer
28  /**
     * All the syntaxic rules for the language indexed (with
30   * a offset of -S_DV) by their left part.
     * i.e.: rules[S_FOO-S_DV] is an array of right_part_t
32   * representing the foo syntaxic class.
     * Notice this array takes a lot of space (more than sr_table depending
34   * on the syntax) so we don't include it in the compiler, we only need
     * it to fill sr_table and build the rules_ending_with[] array.
36   */
static const right_part_t rules[S_LAST-S_DV+2][MAX_RULES_PER_CLASS+1] = {

38          #define SC(sym) S_ ## sym
            #define TK(sym) TK_ ## sym
40          #undef MAX_RULES_PER_CLASS
            #undef MAX_RULE_LEN
42          #define INVALID_RULE { 0, { S_INVALID }}
            #define DEF_TK(name)
44          #define DEF_SC(sym, ...) { __VA_ARGS__, INVALID_RULE },
            #define DEF_DV(sym, ...) DEF_SC(sym, __VA_ARGS__)
46          #include SYNTAX_FILE
            #undef DEF_TK
48          #undef DEF_SC
            #undef DEF_DV
50          { INVALID_RULE }
            #undef INVALID_RULE
52          #undef SC
            #undef TK
54  };

56  // TODO: move this into common part ?
#ifndef MAX_RULES_PER_CLASS
58  #  error "MAX_RULES_PER_CLASS wasn't defined"
#endif
60

size_t rules_count(const symbol_t sym)
62  {
        for (unsigned int i = 0; ; i++)
64          if (0 == rules[sym-S_DV][i].len)
                return i;
66      abort();
```

```
}

const right_part_t *get_rule(const symbol_t sym, const unsigned int num)
{
        assert(is_synclass(sym));
        assert(rules_count(sym) >= num);
        return &rules[sym-S_DV][num];
}

void rules_check(void)
{
        for (symbol_t s = S_FIRST; s <= S_LAST; s++) {
                FOR_EACH_RULE(s, r) {
                        assert(S_INVALID != r->right[0]);
                        assert(MAX_RULE_LEN >= r->len);
                        // TODO: more checks
                }
        }
}

/**
 * A rules comparator that can be passed to qsort() to sort rules
 * per length in descending order.
 */
static int rules_comparator(const void *x, const void *y)
{
        assert(NULL != x);
        assert(NULL != y);
        const rule_t *r1 = (const rule_t*)x;
        const rule_t *r2 = (const rule_t*)y;
        assert(NULL != r1->right);
        assert(NULL != r2->right);
        return r2->right->len - r1->right->len;
}

/**
 * Dump all rules ending with given symbol. Returns the number
 * of matching rules. The rules are reverse sorted by length
 * in the r_ends_*[] array. Longest rules will be first.
 *      dumpc_rules_ending_with(TK_NUM, whatever);
 * should fprintf something like this:
 *      static const symbol_t r_end_rTK_NUM0[2] = { TK_VAR, TK_NUM };
 *      static const rule_t r_end_TK_NUM0 = {
 *              .left = S_DECLARATION,
 *              .len = 2,
 *              .right = r_end_rTK_NUM0
 *      };
 *      static const symbol_t r_end_rTK_NUM1[1] = { TK_NUM };
 *      static const rule_t r_end_TK_NUM1 = {
 *              .left = S_VAL,
```

```
 *                .len = 1,
118  *                .right = r_end_rTK_NUM1
 *        };
120  *        static const rule_t *r_ends_TK_VAR[3] = { &r_end_TK_VAR0,
 *                                                  &r_end_TK_VAR1, NULL };
122  * Return value will be 2.
 * TODO: get rid of the ending NULL and store the length in a shorter field
124  */
static unsigned int dumpc_rules_ending_with(const symbol_t end, FILE *output)
126 {
        assert(is_symbol(end));
128     assert(NULL != output);

130     // Store all matching rules in the matches[] array, sort it,
        // print rules and the array of rules.
132     // Since we don't really know of much rules can have the same
        // end we use the max number of rules. It could eat a
134     // lot of memory however since we'll probably only use the firsts
        // elements, the OS' VM should be able to deal with it intelligently
136     // (well, I hope so).
        static rule_t matches[(S_LAST - S_FIRST + 1)*MAX_RULES_PER_CLASS];
138     unsigned int rnum = 0;  // !< number of matching rules

140     // get the rules and sort them
        matches[0].left = S_INVALID;
142     matches[0].right = NULL;
        for (symbol_t left = S_FIRST; left <= S_LAST; left++) {
144             FOR_EACH_RULE(left, r) {
                        if (r->right[r->len - 1] == end) {
146                             matches[rnum].left = left;
                                matches[rnum].right = r;
148                             rnum++;
                        }
150             }
        }
152     if (0 == rnum)
                return rnum;
154     qsort(matches, rnum, sizeof(matches[0]), rules_comparator);
        matches[rnum].left = S_INVALID;
156     matches[rnum].right = NULL;

158     // print the rules
        for (unsigned int i = 0; i < rnum; i++) {
160             const symbol_t left = matches[i].left;
                const right_part_t *right = matches[i].right;
162             assert(is_synclass(left) && NULL != right);

164             fprintf(output, "static const symbol_t r_end_r%s%u[%zu] = {",
                                sym2str(end), i, right->len);
166             for (unsigned int i = 0; i < right->len; i++)
```

```
                        fprintf(output, "%s, ", sym2str(right->right[i]));
168             fprintf(output, "};\n");
                fprintf(output, "static const rule_t r_end_%s%u = {"
170                             "\n\t.left = %s,"
                                "\n\t.len = %zu,"
172                             "\n\t.right = r_end_r%s%u\n};\n",
                        sym2str(end), i, sym2str(left), right->len,
174                     sym2str(end), i);
        }
176
        // print the rules pointer array
178     fprintf(output,
                "static const rule_t *r_ends_%s[%u] = { ",
180             sym2str(end), rnum + 1 /* for NULL */);
        for (unsigned int i = 0; i < rnum; i++)
182             fprintf(output, "&r_end_%s%u, ", sym2str(end), i);
        fprintf(output, "NULL };\n\n");
184
        return rnum;
186 }

188 /**
 * dumpc_rules_ending_with() for all symbols and
190 * index all these arrays by their ending symbol.
 *      const rule_t **rules_ending_with[] = {
192 *              [TK_VAR] = &r_ends_with_TK_VAR,
 *              [TK_NUM] = &r_ends_with_TK_NUM,
194 *              ...
 *      };
196 * If a symbol has no rule ending with it, its pointer will be NULL.
 */
198 void syntax_dumpc(FILE *output)
{
200     const size_t symbols_count = S_LAST - TK_FIRST + 1;
        // Number of rules ending with given index.
202     unsigned int nb_rules[symbols_count];

204     for (symbol_t end = TK_FIRST; end <= S_LAST; end++)
            nb_rules[end] = dumpc_rules_ending_with(end, output);
206
        fprintf(output,
208             "const rule_t **rules_ending_with[%u] = {", symbols_count);
        for (symbol_t end = TK_FIRST; end <= S_LAST; end++) {
210             fprintf(output, "\n\t[%s] = ", sym2str(end));
                if (0 == nb_rules[end])
212                     fprintf(output, "NULL, ");
                else
214                     fprintf(output, "r_ends_%s, ", sym2str(end));
        }
216     fprintf(output, "\n};\n");
```

```
}

#endif  // RULES_FILE
```

Listing 8 – includes/srt.h

```
#ifndef _SRT_H_
#define _SRT_H_

#include <stdio.h>  // FILE
#include "syntax.h"  // symbol_t

typedef enum {
        ACT_INVALID = 0,  // must be zero (see sr_table definition)
        ACT_SHIFT,
        ACT_REDUCE,
        ACT_FIRST = ACT_INVALID,
        ACT_LAST = ACT_REDUCE
} action_t;

/**
 * Access the shift-reduce table at given position.
 * y must be a token.
 */
static inline action_t srt_get(symbol_t x, symbol_t y)
{
        assert(is_symbol(x));
        assert(is_token(y));
        #ifdef SRT_FILE
        extern const action_t sr_table[S_LAST+1][TK_LAST+1];
        #else
        extern /* const */ action_t sr_table[S_LAST+1][TK_LAST+1];
        #endif
        return sr_table[x][y];
}

#ifndef SRT_FILE  // the rest of the file is only for the syntax analyzer

/**
 * Build the shift-reduce table based on the rules array from syntax.h.
 * This function is supposed to be called only once.
 */
void srt_build(void);

/**
 * Dump the shift-reduce table in a somewhat
 * readable format to the given stream.
 */
void srt_print(FILE *stream);

/**
```

```
 * Dump the shift-reduce table as C code.
 */
void srt_dumpc(FILE *stream);
#endif  // SRT_FILE

#endif  /* _SRT_H_ */
```

Listing 9 – srt.c

```
#include <stddef.h>  /* NULL              */
#include <assert.h>  /* assert()          */
#include <stdio.h>   /* fprintf(), FILE */

#include "misc.h"    /* countof()                 */
#include "syntax.h"  /* symbol_t, get_rule(),... */
#include "srt.h"

/**
 * If SRT_FILE is defined, the file to which it points to is
 * supposed to contain an initialized action_t[][] table
 * with name sr_table (as outputed by srt_dumpc()).
 * Something like this:
 *      const action_t sr_table[S_LAST+1][TK_LAST+1] = {
 *              { ACT_REDUCE, ACT_SHIFT, ACT_SHIFT, ... },
 *              { ACT_INVALID, ACT_INVALID, ACT_REDUCe, ... },
 *              ...
 *      };
 */
#ifdef SRT_FILE  // compiling parser
#  include SRT_FILE
#else  // rest of the file is for the syntax analyzer

/**
 * This is initialized to 0 which means ACT_INVALID must be zero.
 * This is initialized to 0 (assuming neither the compiler nor the loader
 * is buggy) because C99 says:
 * 6.2.2 § 5 "If the declaration of an identifier for an object has
 *           file scope and no storage-class specifier, its linkage is
 *           external."
 * 6.2.4 § 3 "An object whose identifier is declared with external or
 *           internal linkage [...] has static storage duration."
 * 6.7.8 § 10 "If an object that has static storage duration is not
 *            initialized explicitly, then [...]
 *            - if it has arithmetic type, it is initialized
 *              (positive or unsigned) zero;"
 *            - if it is an aggregate, every member is initialized
 *              (recursively) according to these rules;"
 */
action_t sr_table[S_LAST+1][TK_LAST+1];

/**
```

```
 * y must be a token.
 * You are not supposed to call srt_set(x, y, act) twice
 * with the same values for x and y but a different one for act.
 */
static inline void srt_set(symbol_t x, symbol_t y, action_t act)
{
        assert(is_symbol(x));
        assert(is_token(y));
        assert(ACT_INVALID == sr_table[x][y] || act == sr_table[x][y]);

        sr_table[x][y] = act;
}

static void reduce(symbol_t x, symbol_t y);

static void shift(symbol_t x, symbol_t y)
{
        if (is_token(y))
                srt_set(x, y, ACT_SHIFT);

        if (is_synclass(y)) {
                FOR_EACH_RULE(y, r) {
                        if (first_sym(r) != y)
                                shift(x, first_sym(r));
                }
        }

        if (is_synclass(x)) {
                FOR_EACH_RULE(x, r) {
                        reduce(last_sym(r), y);
                }
        }
}

static void reduce(symbol_t x, symbol_t y)
{
        if (is_token(y))
                srt_set(x, y, ACT_REDUCE);

        if (is_synclass(x)) {
                FOR_EACH_RULE(x, r) {
                        if (last_sym(r) != x)
                                reduce(last_sym(r), y);
                }
        }

        if (is_synclass(y)) {
                FOR_EACH_RULE(y, r) {
                        if (first_sym(r) != y)
```

```
92                                      reduce(x, first_sym(r));
                        }
94              }
        }
96
        /**
98       * Some sanity checks for SR table.
         * You are not supposed to call this before filling the table.
100      * Will abort() on error.
         * TODO: make more checks
102      */
        static void srt_check(void)
104     {
                static bool already_got_here = false;
106             assert(!already_got_here);
                already_got_here = true;
108
                for (symbol_t s = TK_FIRST; s <= S_LAST; s++)
110                     for (symbol_t t = TK_FIRST; t <= TK_LAST; t++)
                                assert(ACT_INVALID == srt_get(s, t));
112     }

114     void srt_build(void)
        {
116             #ifndef NDEBUG
                srt_check();
118             rules_check();
                #endif
120
                shift(TK_INPUT, S_DV);
122
                for (symbol_t s = S_FIRST; s <= S_LAST; s++) {
124                     FOR_EACH_RULE(s, r) {
                                for (unsigned int i = 1; i < r->len; i++)
126                                     shift(r->right[i-1], r->right[i]);
                        }
128             }

130             reduce(S_DV, TK_INPUT);
        }
132
        static char act2char(action_t act)
134     {
                switch (act) {
136                     case ACT_INVALID:
                                return ' ';
138                     case ACT_SHIFT:
                                return 'S';
140                     case ACT_REDUCE:
                                return 'R';
```

```
142         }
            abort();
144 }

146 static const char *act2actsym(action_t act)
    {
148         switch (act) {
                case ACT_INVALID:
150                     return "ACT_INVALID";
                case ACT_SHIFT:
152                     return "ACT_SHIFT";
                case ACT_REDUCE:
154                     return "ACT_REDUCE";
            }
156         abort();
    }
158
    void srt_print(FILE *stream)
160 {
            assert(NULL != stream);
162
            for (symbol_t i = TK_FIRST; i < S_LAST+1; i++) {
164             for (symbol_t j = TK_FIRST; j <= TK_LAST; j++)
                        fprintf(stream, "%c", act2char(srt_get(i, j)));
166             fprintf(stream, "|%s\n", sym2str(i));
            }
168
            for (symbol_t i = TK_FIRST; i < TK_LAST+1; i++)
170             fprintf(stream, "-");
            fprintf(stream, "+\n");
172         for (symbol_t i = TK_FIRST; i < TK_LAST+1; i++)
                fprintf(stream, "%u", (unsigned int)i);
174         fprintf(stream, "\n");

176         for (symbol_t i = TK_FIRST; i < TK_LAST+1; i++)
                fprintf(stream, "%u = %s%c", (unsigned int)i, sym2str(i),
178                                          i%2 ? '\n' : '\t');
            if (TK_LAST % 2 == 0)
180             fprintf(stream, "\n");
    }
182
    void srt_dumpc(FILE *stream)
184 {
            assert(NULL != stream);
186         fprintf(stream, "const action_t sr_table[S_LAST+1][TK_LAST+1] = {\n");
            for (symbol_t i = TK_FIRST; i <= S_LAST; i++) {
188             fprintf(stream, "{ ");
                for (symbol_t j = TK_FIRST; j < TK_LAST; j++)
190                     fprintf(stream, "%s, ", act2actsym(srt_get(i, j)));
                fprintf(stream, "%s }\n", act2actsym(srt_get(i, TK_LAST)));
```

```
192                     if (S_LAST != i)
                                fprintf(stream, ",");
194             }
                fprintf(stream, "};\n");
196     }

198     #endif  // SRT_FILE
```

Listing 10 – synalyze.c

```
    #include <stdlib.h>   // EXIT_SUCCESS
2   #include <stdio.h>    // fopen(3), stdout, FILE
    #include <string.h>   // strncmp(3)
4
    #include "misc.h"     // err()
6   #include "srt.h"      // srt_build(), srt_dumpc()
    #include "syntax.h"   // syntax_dumpc()
8
    #if defined SRT_FILE || defined RULES_FILE
10  #  error "Tried to analyze the syntax based on the syntax analyzer output."
    #endif
12
    /**
14   * This is the syntax analyzer that will produce the shift-reduce table
     * and the syntaxic rules data structures optimized for use by the parser.
16   * It will output them as C source files.
     */
18  int main(int argc, char **argv)
    {
20          if (2 == argc && 0 == strncmp("-h", argv[1], 2))
                    err("Usage: %s [srt_output_file] [syntax_output_file]\n",
22                      argv[0]);

24          FILE *srt_out = (2 > argc) ? stdout : fopen(argv[1], "w");
            FILE *syn_out = (3 > argc) ? stdout : fopen(argv[2], "w");
26
            srt_build();
28          fprintf(srt_out, "// autogenerated shift-reduce table, do not edit\n");
            srt_dumpc(srt_out);
30          fprintf(syn_out,
                    "// autogenerated syntax data structures, do not edit\n");
32          syntax_dumpc(syn_out);
            if (stdout == srt_out)
34                  srt_print(stdout);

36          return EXIT_SUCCESS;
    }
```

A.3 Compilateur

Listing 11 – includes/var-register.h

```
#ifndef _VAR_REGISTER_H_
#define _VAR_REGISTER_H_

/**
 * Maximum variables that can be used. This includes chemical symbol names.
 */
#ifndef MAX_VARS_COUNT
#  define MAX_VARS_COUNT  128
#endif

/**
 * Adds the given variable name to the register and returns its id.
 * The initial value associated to the variable will be a NaN.
 */
unsigned int register_var(char *name);

/**
 * Returns the value associated to the variable.
 * Returns a NaN if no value has been set.
 */
double register_id2val(unsigned int id);

/**
 * Set the variable value.
 * id must be a valid variable id as returned by register_var().
 * value can't be a NaN.
 * Will err() if variable is already set to something else than a NaN.
 */
void register_setval(unsigned int id, double value);

/**
 * Returns the string corresponding to the given variable id.
 * Returns NULL on error.
 */
const char *register_id2str(unsigned int id);

#endif
```

Listing 12 – var-register.c

```
#include <stddef.h>  /* NULL
*/
#include <string.h>  /* strcmp()
*/
#include <limits.h>  /* CHAR_BIT
*/
```

```
4  #include <assert.h>  /* assert()
   */
   #include <math.h>    /* nan(), isnan()
   */
6  #include <search.h>  /* hcreate(), hsearch(), ENTER, FIND */

8  #include "misc.h"           /* xmalloc()    */
   #include "lexer.h"          /* current_line */
10 #include "var-register.h"

12 /**
    * Lowest unused id.
14  */
   static unsigned int last_id;
16
   /**
18  * Array holding the variables names indexed by id.
    * Auto initialized to NULL.
20  */
   static char *var_names[MAX_VARS_COUNT];
22
   /**
24  * Array holding the variable values indexed by id.
    */
26 static double var_values[MAX_VARS_COUNT];

28 unsigned int register_var(char *name)
   {
30         assert(NULL != name);

32         if (0 == last_id && 0 == hcreate(MAX_VARS_COUNT))
                   err("Can't create hash table of %u elements.\n",
34                     MAX_VARS_COUNT);

36         ENTRY new_var = { .key = name, .data = NULL };
           ENTRY *old_var = hsearch(new_var, FIND);
38         if (NULL != old_var)
                   // casting uint to void* and back is supposed to be safe
40                 return (unsigned int)(old_var->data);

42         if (last_id == MAX_VARS_COUNT)
                   err("Too many variables. Limit is %u\n", MAX_VARS_COUNT);
44
           var_values[last_id] = nan("");
46         assert(NULL == var_names[last_id]);
           var_names[last_id]  = xmalloc((strlen(name)+1)*sizeof(*name));
48         strcpy(var_names[last_id], name);

50         new_var.key = var_names[last_id];
           new_var.data = (void*)(last_id);
```

```
52          old_var = hsearch(new_var, ENTER);
            assert(NULL != old_var);
54
            return last_id++;
56  }

58  const char *register_id2str(const unsigned int id)
    {
60          if (id >= last_id)
                    return NULL;
62          return var_names[id];
    }
64
    double register_id2val(unsigned int id)
66  {
            if (id >= last_id)
68                  return nan("");
            return var_values[id];
70  }

72  void register_setval(unsigned int id, double value)
    {
74          assert(id < last_id);
            assert(!isnan(value));
76          if (!isnan(var_values[id]))
                    err("line %u: variable value has already been set (%s)\n",
78                      current_line, register_id2str(id));
            var_values[id] = value;
80  }
```

Listing 13 – includes/lexer.h

```
    #ifndef _LEXER_H_
2   #define _LEXER_H_

4   #include <stdbool.h>
    #include "syntax.h"  // symbol_t, is_token()
6
    /**
8    * Maximum variable name length.
     */
10  #define MAX_VAR_LEN 128

12  /**
     * The type that holds a token.
14   */
    typedef struct {
16          symbol_t type;     //<! A token type from symbol_t (TK_*).
            union { //<! Only used for TK_VAR and TK_NUM
18                  unsigned int id;
                    double f;
```

```
20          } val;
    } token_t;
22
    /**
24   * Current input line number, starting at 1.
     * Somewhat like Perl's $. variable.
26   */
    extern unsigned int current_line;
28
    /**
30   * Initialize the lexer with the given stream.
     * Must be called before any call to lex_next_sym().
32   */
    void lex_init(FILE *stream);
34
    /**
36   * Puts the next token from the stream in retval.
     * retval must be allocated properlly with enough place to store a token_t.
38   * retval.type will be set to S_INVALID on error.
     * If the value of the type field of the token is TK_NUM,
40   * the val.f field contains the number.
     * If the value of the type field of the token is TK_VAR,
42   * the val.id field contains an id corresponding to the variable name
     * stored in the variables register. The associated value will be a NaN.
44   * In all other cases, the value of the val.* field is unspecified.
     * If the stream returns EOF, retval->type will be TK_INPUT.
46   */
    void lex_next(token_t *retval);
48
    /**
50   * Prints a human-readable representation of
     * the given token to the given stream.
52   */
    void print_token(FILE *stream, const token_t *tk);
54
    #endif
```

Listing 14 – lexer.c

```
    #include <ctype.h>   // isalnum(), isalpha(), isdigit()
2   #include <stdio.h>   // fgetc(), ungetc(), fscanf()
    #include <stdlib.h>  // atoi()
4   #include <assert.h>  // assert()

6   #include "misc.h"           // xstringify()
    #include "var-register.h"
8   #include "syntax.h"         // TK_*, sym2str()
    #include "lexer.h"
10
    // This is the lexer that supply everything to
12  // convert an input stream to tokens.
```

```
// TODO: more helpful error messages

/**
 * The input stream from which we read tokens.
 */
static FILE *in_stream = NULL;

/**
 * Current line number.
 */
unsigned int current_line = 0;

/**
 * ungetc(c, stream) with error checking.
 * Do not call this function twice without fgetc()ing.
 */
static inline void xungetc(char c, FILE *stream)
{
        const int eof = feof(stream);
        int res = ungetc(c, stream);
        // This is an assert() because it will fail only if the function
        // is misused (or the implementation is bugged) since we are
        // guaranteed a pushback of one character by C99 section 7.19.7.11 §
3.
        assert(EOF != res || eof);
}

void lex_init(FILE *stream)
{
        assert(NULL != stream);  // FIXME: use GCC's nonnull attribute instead
        in_stream = stream;
        current_line = 1;
}

/**
 * Try to parse a variable name beginning with first from the stream.
 * Returns the id of the variable.
 * Will err() if the variable name is longer than MAX_VAR_LEN.
 */
static unsigned int parse_var(const char first)
{
        assert(NULL != in_stream);
        assert(isalpha(first));

        /* static */ char var_name[MAX_VAR_LEN+1];
        var_name[0] = first;
        unsigned int i = 1;
        for (i = 1; i < MAX_VAR_LEN; i++) {
                var_name[i] = (char)fgetc(in_stream);
                if (!isalnum(var_name[i])) {
```

```
62                          xungetc(var_name[i], in_stream);
                            break;
64                  }
            }
66          var_name[i] = 0;

68          if (MAX_VAR_LEN == i) {
                    char tmp = (char)fgetc(in_stream);
70                  if (isalnum(tmp))
                            err("line %u: variable name is too long (> %d)\n",
72                              current_line, MAX_VAR_LEN);
                    xungetc(tmp, in_stream);
74          }

76          return register_var(var_name);
}
78 #if 0
// This is nicer than doing the parsing by hand but I'm not
80 // sure it's portable and safe.
// C99 says that when we use %42[] in fscanf(),
82 // we have to pass a wchar_t* but GCC wants a char*.
// If we pass a char* while sizeof(char) < sizeof(wchar_t) and fscanf()
84 // really uses wchar_t internally, we might  be in trouble.
// Therefore, we will keep parsing by hand while this is not sorted out.
86 static unsigned int parse_var(const char first)
{
88          #define VAR_CHARS "abcdefghijklmnopqrstuvwxyz" \
                              "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
90                            "0123456789"
            assert(NULL != in_stream);
92          xungetc(first, in_stream);
            char var_name[MAX_VAR_LEN+2];  // 1 for first, 1 for '\0'
94          var_name[0] = first;  // FIXME: need to convert ?
            if (1 != fscanf(in_stream,
96                          "%" xstringify(MAX_VAR_LEN) "[" VAR_CHARS "]",
                            &var_name[1]))
98                  err("line %u: unable to parse variable.\n", current_line);
            return register_var(var_name);
100 }
#endif
102

/**
104  * Parse the number beginning with first from the stream.
  * first must be [0-9] or '-' (minus).
106  * This is supposed to parse any number that matches -?[0-9]+(\.[0-9]+)?
  * Will err() on error.
108  */
static double parse_num(const char first)
110 {
            assert(NULL != in_stream);
```

```
112         xungetc(first, in_stream);

114         double retval = 0;
            if (1 != fscanf(in_stream, "%lf", &retval))
116                 err("line %u: can't parse number.\n", current_line);
            return retval;
118 }

120 void lex_next(token_t *retval)
    {
122         assert(NULL != in_stream);
            assert(NULL != retval);
124
            char current;
126         do {
                    current = (char)fgetc(in_stream);
128                 if ('\n' == current)
                            current_line++;
130         } while (isspace(current));

132         switch (current) {
                    case '(' :
134                         retval->type = TK_OPEN_PAR;
                            return;
136
                    case ')' :
138                         retval->type = TK_CLOSE_PAR;
                            return;
140
                    case ',' :
142                         retval->type = TK_COMMA;
                            return;
144
                    case '!' :
146                         retval->type = TK_EXCL;
                            return;
148
                    case '-': case '0': case '1': case '2': case '3':
150                 case '4': case '5': case '6': case '7': case '8':
                    case '9':
152                         retval->type = TK_NUM;
                            retval->val.f = parse_num(current);
154                         return;
                    case (char)EOF:
156                         retval->type = TK_INPUT;
                            return;
158         }

160         if (isalpha(current)) {
                    retval->type = TK_VAR;
```

```
162                     retval->val.id = parse_var(current);
                        return;
164             }

166             retval->type = S_INVALID;
        }
168
        void print_token(FILE *stream, const token_t *tk)
170     {
                assert(NULL != stream);  // FIXME: use GCC's nonnull attribute instead
172             assert(NULL != tk);      // FIXME: use GCC's nonnull attribute instead
                assert(is_token(tk->type));
174
                fprintf(stream, "%s", sym2str(tk->type));
176             if (TK_VAR == tk->type)
                        fprintf(stream, " (%u)", tk->val.id);
178             else if (TK_NUM == tk->type)
                        fprintf(stream, " (%f)", tk->val.f);
180     }
```

Listing 15 – includes/parser.h

```
        #ifndef _PARSER_H_
2       #define _PARSER_H_

4       #if !(defined RULES_FILE) || !(defined SRT_FILE)
        #  error "You need to run the syntax analyzer before compiling the parser."
6       #endif

8       #include <stdio.h>  // FILE
        #include "lexer.h"   // token_t
10      #include "syntax.h"  // symbol_t

12      /**
         * A parse tree node.
14       */
        typedef struct parse_tree_t_ {
16              struct parse_tree_t_ *sibling;
                struct parse_tree_t_ *heir;
18              token_t tok;  // when NULL != heir,
                              // only the "type" field is relevant
20      } parse_tree_t;

22      /**
         * Build a parse tree from the input stream.
24       * If callback is not NULL, it will be called after each
         * stack reduction with the new node in argument. The callback() function
26       * may modify tree->heir but changing either the tok or the sibling field
         * will most probably cause problems.
28       * Will err() if the input stream can't be parsed.
         */
```

```
30 parse_tree_t *parse(FILE *input, void (*callback)(parse_tree_t*));
32 /**
    * Print a parse tree in .dot Graphviz format.
34  */
   void tree2dot(const parse_tree_t *tree, FILE *output);
36
   /**
38  * Free the node and its heirs. Returns the first sibling.
    */
40 parse_tree_t *free_tree_down(parse_tree_t *tree);
42 /**
    * Free the node, its heirs and siblings.
44  */
   void free_tree_real(parse_tree_t *tree);
46 #define free_tree(tree) do { free_tree_real(tree); tree = NULL; } while (0)
48 #endif  // _PARSER_H_
```

Listing 16 – parser.c

```
   #include <assert.h>  // assert()
2  #include <stdlib.h>  // NULL
4  #include "misc.h"    // err(), xmalloc()
   #include "stack.h"
6  #include "syntax.h"  // symbol_t, rule_t, get_rules_ending_with(),...
   #include "srt.h"     // srt_get()
8  #include "parser.h"
10 #ifndef NDEBUG
   /**
12  * Current count of parse_tree_t nodes allocated by new_ptree().
    */
14 static unsigned int parse_nodes_count = 0;
16 /**
    * Maximum live parse_tree_t nodes.
18  */
   static unsigned int parse_nodes_count_max = 0;
20 #endif
22 /**
    * Create a new node with the given values.
24  * The sibling field will be NULL.
    * The tok.val field will have an undefined value.
26  * Will err() on error.
    */
28 static parse_tree_t *new_ptree(symbol_t type, parse_tree_t *heir)
```

```
{
        assert(is_symbol(type));
        assert(NULL == heir || is_synclass(type));
        // TODO: this will call xmalloc() far too often
        parse_tree_t *retval = xmalloc(sizeof(*retval));

        retval->heir = heir;
        retval->sibling = NULL;
        retval->tok.type = type;
        retval->tok.val.f = nan("");

        #ifndef NDEBUG
        if (++parse_nodes_count > parse_nodes_count_max)
                parse_nodes_count_max = parse_nodes_count;
        #endif
        return retval;
}

/**
 * Create a new node from a token. The token is copied, which means you can
 * do whatever you want with it once the function has returned without
 * affecting the node.
 * The heir and sibling fields will be NULL.
 * Will err() on error.
 */
static parse_tree_t *tok2ptree(token_t *token)
{
        assert(NULL != token);
        parse_tree_t *retval = new_ptree(token->type, NULL);
        retval->tok.val = token->val;
        return retval;
}

/**
 * Search for the longest rule matching the top of the stack and
 * replace that stack part by the left part of the rule.
 * err() if no reduction can be made.
 * TODO: profiling say we might gain significant speed by using
 *       a better pattern matching algorithm
 */
static void reduce(stack_head_t *stack, parse_tree_t *top)

{
        assert(NULL != stack);
        assert(stack_peek(stack, 0) == top);

        // reverse sorted by length, last one is NULL
        const rule_t **possible_matches = get_rules_ending_with(top->tok.type);
        const rule_t *rule = NULL;
        for (unsigned int i = 0; NULL != possible_matches[i]; i++) {
```

```
78                      rule = possible_matches[i];
                        assert(is_synclass(rule->left));
80                      assert(top->tok.type == rule->right[rule->len - 1]);

82                      parse_tree_t *symbols[rule->len];
                        symbols[0] = NULL;
84                      symbols[rule->len-1] = top;
                        // We'll fill the rest with the top of the stack,
86                      // overwriting the NULL last.
                        for (unsigned int i = 1; i < rule->len; i++) {
88                              int pos = rule->len - 1 - i;
                                symbols[pos] = stack_peek(stack, i);
90                              if (rule->right[pos] != symbols[pos]->tok.type)
                                        break;
92                      }
                        if (symbols[0] == NULL
94                       || symbols[0]->tok.type != rule->right[0])
                         // last symbol has been checked implicitely (and in assert())
96                              continue;

98                      // found it, create father and tell heirs they have siblings
                        parse_tree_t *new = new_ptree(rule->left, symbols[0]);
100                     for (unsigned int i = 0; i < rule->len - 1; i++) {
                                assert(NULL != symbols[i] && NULL != symbols[i+1]);
102                             symbols[i]->sibling = symbols[i+1];
                        }
104
                        stack_discard(stack, rule->len);
106                     stack_push(stack, new);
                        return;
108             }

110             err("line %u: no syntaxic rule matches input\n", current_line);
    }
112
    parse_tree_t *parse(FILE *in_stream, void (*callback)(parse_tree_t*))
114 {
                assert(NULL != in_stream);
116             token_t tmp = { .type = TK_INPUT };
                parse_tree_t *stack_top = tok2ptree(&tmp);  // FIXME: memleak ?
118             stack_head_t *stack = stack_mk();
                stack_push(stack, stack_top);
120
                lex_init(in_stream);
122             token_t current = { .type = S_INVALID };
                lex_next(&current);
124             // FIXME: indentation level too deep
                while (!(TK_INPUT == current.type && S_DV == stack_top->tok.type)) {
126                     if (S_INVALID == current.type)
                                err("line %u: invalid token\n", current_line);
```

```
128
                switch(srt_get(stack_top->tok.type, current.type)) {
130                     case ACT_INVALID:
                                err("line %u: invalid sequence: %s %s\n",
132                                     current_line,
                                        sym2str(stack_top->tok.type),
134                                     sym2str(current.type));
                        case ACT_SHIFT:
136                             debug("shifting: %s %s\n",
                                        sym2str(stack_top->tok.type),
138                                     sym2str(current.type));
                                stack_top = tok2ptree(&current);
140                             stack_push(stack, stack_top);
                                lex_next(&current);
142                             break;
                        case ACT_REDUCE:
144                             debug("reducing: %s %s\n",
                                        sym2str(stack_top->tok.type),
146                                     sym2str(current.type));
                                reduce(stack, stack_top);
148                             stack_top = stack_peek(stack, 0);
                                if (NULL != callback)
150                                     callback(stack_top);
                                break;
152                     default:
                                abort();
154                 }
            }
156
            parse_tree_t *bottom = stack_peek(stack, 1);
158         assert(NULL != bottom && TK_INPUT == bottom->tok.type);
            free_tree(bottom);
160         // FIXME: it seems there is a memleak in the stack
            stack_free(stack);
162
            debug("maximum parse tree nodes count was %u\n"
164               "current is %u\n", parse_nodes_count_max, parse_nodes_count);

166         return stack_top;
}
168
void free_tree_real(parse_tree_t *tree)
170 {
            assert(NULL != tree);
172         do {
                    tree = free_tree_down(tree);
174         } while (NULL != tree);
}
176
parse_tree_t *free_tree_down(parse_tree_t *tree)
```

```
178 {
        assert(NULL != tree);
180     parse_tree_t *retval = tree->sibling;
        if (NULL != tree->heir)
182             free_tree(tree->heir);
        assert(parse_nodes_count-- > 0);
184     xfree(tree);
        return retval;
186 }

188 /**
 * Print the given node, its siblings and heirs to output
190  * in .dot format parsable by graphviz.
 * id and rank are the id and rank that will be used for
192  * that node. Return value is the max used rank.
 * Nodes with the same rank will be placed on the same level.
194  */
static unsigned int node2dot(FILE *output, const parse_tree_t *node,
196                              const unsigned int rank, const unsigned int id)
{
198     assert(NULL != node);
        assert(NULL != output);
200
        fprintf(output, "NODE%02u%02u [label=\"%s\"];\n",
202                 rank, id, sym2str(node->tok.type));
        unsigned int next_rank = rank+1;
204
        if (NULL == node->sibling) {
206             fprintf(output, "{rank=same; ");
                for (unsigned int i = 0; i <= id; i++)
208                     fprintf(output, "NODE%02u%02u ", rank, i);
                fprintf(output, ";}\n");
210     }
        else {  // NULL != node->sibling
212             fprintf(output, "NODE%02u%02u -> NODE%02u%02u;\n",
                                rank, id, rank, id+1);
214             next_rank = node2dot(output, node->sibling, rank, id+1);
        }
216
        if (NULL != node->heir) {
218             fprintf(output, "NODE%02u%02u -> NODE%02u%02u;\n",
                                rank, id, next_rank, (unsigned int)0);
220             next_rank = node2dot(output, node->heir, next_rank, 0);
        }
222
        return next_rank;
224 }

226 void tree2dot(const parse_tree_t *tree, FILE *output)
    {
```

```
228         assert(NULL != tree);
            assert(NULL != output);
230
            fprintf(output, "digraph ptree {\n");
232         node2dot(output, tree, 0, 0);
            fprintf(output, "}\n");
234 }
```

Listing 17 – includes/translater.h

```
  #ifndef _TRANSLATER_H_
2 #define _TRANSLATER_H_

4 #include <stdio.h>    // FILE
  #include "parser.h"   // parse_tree_t
6
  /**
8  * Parser callback that should be called after each reduction
   * It will free_tree(tree->heir) when possible to save memory.
10 */
  void on_reduce(parse_tree_t *tree);
12
  /**
14 * Calculate atoms cartesian coordinates and print them.
   * parse(input, on_reduce) must have been called first.
16 */
  void print_atoms(FILE *output);
18
  #endif
```

Listing 18 – translater.c

```
  #include <assert.h>  // assert()
2 #include <stdio.h>   // fprintf()
  #include <stdlib.h>  // NULL, abort()
4 #include <math.h>    // nan(), isnan(), cos(), sin(), sqrt()

6 #include "misc.h"           // err(), xrealloc(), xmalloc(), iseq()
  #include "lexer.h"          // token_t, current_line
8 #include "var-register.h"   // register_id2str(), register_setval()
  #include "syntax.h"         // S_*, TK_*, sym2str()
10 #include "parser.h"        // parse_tree_t
  #include "translater.h"
12
  #define PI  3.14159265358979323846
14
  #define COS_D(arg) cos(arg * PI / 180) // cos using degres
16 #define SIN_D(arg) sin(arg * PI / 180) // sin using degres

18 typedef struct {
       unsigned int atom;  // atom index in the atoms[] array
```

```
20          token_t value;      // angle or distance (TK_VAR or TK_NUM)
} couple_t;
22
typedef struct {
24          /**
             * Chemical symbol for this atom.
26           */
            const char *name;
28
            /**
30           * The three couples from the syntaxic three. When there are
             * less than three couples in the syntaxic tree (for S_ATOM1, S_ATOM2
32           * and the first heir of S_MOLECULE), the atom field is set to 1
             * and the value field is set to 0.
34           * refs[0] is a distance
             * refs[1] and refs[2] are angles
36           */
            couple_t refs[3];
38
            /**
40           * Cartesian coordinates of the atom. Set to NaN when not known
             * (when a refs[i].value is a TK_VAR for which we don't
42           * known the value yet).
             * TODO: this could be a union since we use exclusively (x,y,z) or refs
44           */
            double x, y, z;
46  } atom_t;

48  /**
     * Use vector_length() to read the len field
50   * and set it to NaN after each modification of x, y or z.
     */
52  typedef struct {
            double x, y, z, len;
54  } vector_t;

56  /**
     * Number of known atoms + 1. The additional cell is for the first atom
58   * which we'll known only last while the other ones are known in order.
     * So we start filling the array from index position 1 and add the first
60   * atom at position 0 after. We could set atom[0] first (its coordinates
     * are always (0, 0, 0)) but I can't find a nice way to do that.
62   * After new_atom() has been called with the first atom (that's the last call
     * to new_atom()), atoms_count is the exact number of atoms (it's not
64   * incremented on that last call).
     */
66  static unsigned int atoms_count = 1;

68  /**
     * Number of elements in atoms[] that are currently allocated.
```

```
70  */
    static unsigned int atoms_array_size = 0;
72
    /**
74   * The atom_t array. Notice that atom at index n is the one refered by
     * n+1 according to the syntax. It means if you have a line like
76   *      H  2  1.11783779
     * the "2" is for atom[1].
78   */
    static atom_t *atoms = NULL;
80
    /**
82   * We will realloc() atoms[] by blocs of ATOMS_ARRAY_BLK_SZ elements.
     */
84  #define ATOMS_ARRAY_BLK_SZ  64

86  /**
     * Converts a double to unsigned int.
88   * Will err() if x is not really a natural.
     */
90  static inline unsigned int double2uint(double x)
    {
92          if (isnan(x) || !iseq(trunc(x), x))
                    err("line %u: value is not integer (%f)\n", current_line, x);
94          if (x < 0)
                    err("line %u: negative value (%f)\n", current_line, x);
96          return (unsigned int)x;
    }
98
    /**
100  * Returns the atom id specified in a S_COUPLE.
     */
102 static inline unsigned int ref_from_couple(const parse_tree_t *tree)
    {
104         assert(S_COUPLE     == tree->tok.type);
            assert(TK_OPEN_PAR == tree->heir->tok.type);
106         assert(TK_NUM       == tree->heir->sibling->tok.type);
            assert(S_VAL        == tree->heir->sibling->sibling->tok.type);
108         assert(TK_CLOSE_PAR== tree->heir->sibling->sibling->sibling->tok.type);
            unsigned int retval = double2uint(tree->heir->sibling->tok.val.f);
110         if (0 == retval)
                    err("line %u: reference to invalid atom 0\n", current_line);
112         return retval - 1;
    }
114
    /**
116  * Returns the value (TK_NUM or TK_VAR representing an angle or distance)
     * specified in a S_COUPLE.
118  */
    static inline token_t val_from_couple(const parse_tree_t *tree)
```

```
120 {
        assert(S_COUPLE     == tree->tok.type);
122     assert(TK_OPEN_PAR == tree->heir->tok.type);
        assert(TK_NUM       == tree->heir->sibling->tok.type);
124     assert(S_VAL        == tree->heir->sibling->sibling->tok.type);
        assert(TK_CLOSE_PAR== tree->heir->sibling->sibling->sibling->tok.type);
126     assert(TK_NUM       == tree->heir->sibling->sibling->heir->tok.type
            || TK_VAR       == tree->heir->sibling->sibling->heir->tok.type);
128     return tree->heir->sibling->sibling->heir->tok;
}
130
/**
132  * Add an atom to the atoms[] array.
 */
134 static void new_atom(const parse_tree_t *atom)
{
136     assert(NULL != atom);
        assert(S_MOLECULE == atom->tok.type || S_ATOM1 == atom->tok.type
138         || S_ATOM2    == atom->tok.type || S_ATOM  == atom->tok.type);

140     unsigned int id = S_MOLECULE == atom->tok.type ? 0 : atoms_count++;
        if (atoms_count > atoms_array_size) {
142         atoms_array_size += ATOMS_ARRAY_BLK_SZ;
            atoms = xrealloc(atoms, atoms_array_size*sizeof(*atoms));
144     }

146     assert(TK_VAR == atom->heir->tok.type);
        atoms[id].name = register_id2str(atom->heir->tok.val.id);
148
        const parse_tree_t *couples[3] = { NULL, NULL, NULL };
150     switch (atom->tok.type) {
            case S_ATOM:
152             couples[2] = atom->heir->sibling->sibling->sibling;
            case S_ATOM2:
154             couples[1] = atom->heir->sibling->sibling;
            case S_ATOM1:
156             couples[0] = atom->heir->sibling;
            case S_MOLECULE:
158             break;
            default:
160             abort();
        }
162     for (unsigned int i = 0; i < 3; i++) {
            if (NULL == couples[i]) {
164             atoms[id].refs[i].atom = 0;
                atoms[id].refs[i].value.type = TK_NUM;
166             atoms[id].refs[i].value.val.f = 0;
                continue;
168         }
            atoms[id].refs[i].value = val_from_couple(couples[i]);
```

```
170                     atoms[id].refs[i].atom  = ref_from_couple(couples[i]);
                        if (atoms[id].refs[i].atom >= id)
172                             err("line %u: forward reference to an undefined atom: "
                                "atom %s refers to atom %u\n", current_line,
174                                 atoms[id].name, atoms[id].refs[i].atom);
                }
176             atoms[id].x = atoms[id].y = atoms[id].z = 0 == id ? 0 : nan("");
                // Notice we may already calculate x, y and z now if all
178             // refs[].value are TK_NUM but since we'll probably have to
                // go through all atoms after anyway that's not really useful.
180             // If variables were defined before the molecule, we would have
                // been able to do everything in one pass.
182     }

184     void on_reduce(parse_tree_t *tree)
        {
186             assert(NULL != tree);

188             switch (tree->tok.type) {
                        case S_ATOM:
190                     case S_ATOM2:
                        case S_ATOM1:
192                     case S_MOLECULE:
                                new_atom(tree);
194                             assert(NULL != tree->heir);
                                free_tree(tree->heir);
196                             break;
                        case S_DECLARATION:
198                             assert(TK_VAR == tree->heir->tok.type);
                                assert(TK_NUM == tree->heir->sibling->tok.type);
200                             register_setval(tree->heir->tok.val.id,
                                                tree->heir->sibling->tok.val.f);
202                             assert(NULL != tree->heir);
                                free_tree(tree->heir);
204                             break;
                        case S_ATOMS:
206                     case S_DECLARATIONS:
                                free_tree(tree->heir);
208                             break;
                        case S_ZMAT:
210                     case S_COUPLE:
                        case S_VAL:
212                             break;
                        default:
214                             abort();
                }
216     }

218     /**
         * Set the len field of v and return it.
```

```
220  * If v->len is not NaN, do nothing.
     */
222 static inline double vector_length(vector_t *v)
    {
224         assert(NULL != v);
            if (!isnan(v->len))
226                 return v->len;
            v->len = sqrt(v->x * v->x +
228                       v->y * v->y +
                          v->z * v->z);
230         return v->len;
    }
232
    /**
234  * Makes retval the vector joining a to b.
     * retval = ab
236  * retval must be a proprelly allocated vector_t pointer.
     */
238 static void vector(const atom_t *a, const atom_t *b, vector_t *retval)
    {
240         assert(NULL != a);
            assert(NULL != b);
242         assert(NULL != retval);

244         retval->x = a->x - b->x;
            retval->y = a->y - b->y;
246         retval->z = a->z - b->z;

248         retval->len = nan("");
    }
250
    /**
252  * Makes retval the cross/vector/outer product of a and b.
     * $retval = a \times b$ (a x b)
254  * retval must be a proprelly allocated vector_t pointer.
     */
256 static void cross_product(const vector_t *a, const vector_t *b,
                              vector_t *retval)
258 {
            assert(NULL != a);
260         assert(NULL != b);
            assert(NULL != retval);
262
            retval->x = (a->y * b->z) - (a->z * b->y);
264         retval->y = (a->z * b->x) - (a->x * b->z);
            retval->z = (a->x * b->y) - (a->y * b->x);
266
            retval->len = nan("");
268 }
```

```
270 /**
 * Normalize the given vector (make its length 1).
272  */
static void normalize(vector_t *a)
274 {
        assert(NULL != a);
276
        // FIXME: check for division by zero
278     a->x /= vector_length(a);
        a->y /= a->len;  // Don't call vector_length() while
280     a->z /= a->len;  // x, y and z are inconsistents.
        #ifndef NDEBUG
282     a->len = nan("");
        assert(iseq(1, vector_length(a)));
284     #endif
        a->len = 1;
286 }

288 /**
 * Calculate the cartesian coordinates for the specified atom.
290  * id must be a valid atom id, all cartesian coordinates for atoms
 * whose id is less than id must have been calculated before and
292  * all variables used in the atom definition must have been defined.
 */
294 static void rel2cart(unsigned int id)
{
296     // Algorithm based on David Lauvergnat's Cart
        // <http://www.lcp.u-psud.fr/Pageperso/lauvergnat/cart/cart.html>.
298
        // TODO: is it the fastest way to do it in C ?
300
        assert(0 < id && id < atoms_count);
302     assert(isnan(atoms[id].x) && isnan(atoms[id].y) && isnan(atoms[id].z));
        #ifndef NDEBUG  // stupid GCC who can't remove empty loops
304     for (unsigned int i = 0; i < id; i++)
            assert(!isnan(atoms[i].x)
306             && !isnan(atoms[i].y)
                && !isnan(atoms[i].z));
308     #endif

310     struct {
            atom_t *atom;
312         double val;
        } refs[3];
314     for (unsigned int i = 0; i < 3; i++) {
            refs[i].val = TK_VAR == atoms[id].refs[i].value.type
316                 ? register_id2val(atoms[id].refs[i].value.val.id)
                    : atoms[id].refs[i].value.val.f;
318         assert(0 <= atoms[id].refs[i].atom
                    && atoms[id].refs[i].atom < atoms_count);
```

```
320                 refs[i].atom = &atoms[atoms[id].refs[i].atom];
        }
322
        if (1 == id) {
324             atoms[id].x = atoms[id].y = 0;
                atoms[id].z = refs[0].val;
326             return;
        }
328     if (2 == id) {
                atoms[id].x = refs[0].atom->x
330                         + refs[0].val * SIN_D(refs[1].val);
                atoms[id].y = 0;
332
                atoms[id].z = refs[0].atom->z +
334                           (0 == atoms[id].refs[0].atom ? 1 : -1)*
                           refs[0].val * COS_D(refs[1].val);
336             return;
        }
338
        vector_t v1, v2, v3, v4;
340
        vector(refs[1].atom, refs[2].atom, &v1);
342     vector(refs[1].atom, refs[0].atom, &v2);

344     cross_product(&v2, &v1, &v3);
        normalize(&v3);
346
        cross_product(&v3, &v2, &v4);
348     normalize(&v4);

350     const vector_t v5 = {
                .x = COS_D(refs[2].val) * v4.x + SIN_D(refs[2].val) * v3.x,
352             .y = COS_D(refs[2].val) * v4.y + SIN_D(refs[2].val) * v3.y,
                .z = COS_D(refs[2].val) * v4.z + SIN_D(refs[2].val) * v3.z
354     };

356     normalize(&v2);

358     const vector_t v6 = {
                .x = SIN_D(refs[1].val) * v5.x - COS_D(refs[1].val) * v2.x,
360             .y = SIN_D(refs[1].val) * v5.y - COS_D(refs[1].val) * v2.y,
                .z = SIN_D(refs[1].val) * v5.z - COS_D(refs[1].val) * v2.z
362     };

364     atoms[id].x = refs[0].atom->x + v6.x * refs[0].val;
        atoms[id].y = refs[0].atom->y + v6.y * refs[0].val;
366     atoms[id].z = refs[0].atom->z + v6.z * refs[0].val;
}
368
/**
```

```
370  * Calculate cartesian coordinates and print them to output.
     * TODO: find a nice way to control output format from outside this function
372  */
     void print_atoms(FILE *output)
374  {
             static const char atom_output_fmt[] = "%s\t%f\t%f\t%f\n";
376  //         "object { %s translate <%f, %f, %f> }\n"  // for povray
             for (unsigned int i = 0; i < atoms_count; i++) {
378                  if (isnan(atoms[i].x)) {
                             assert(isnan(atoms[i].y) && isnan(atoms[i].z));
380                          rel2cart(i);
                     }
382                  fprintf(output, atom_output_fmt,
                             atoms[i].name, atoms[i].x, atoms[i].y, atoms[i].z);
384          }
     }
```

Listing 19 – zmat2cart.c

```
     #include <stdio.h>   // stdin, stdout, fopen(3)
2    #include <stdlib.h>  // EXIT_SUCCESS, NULL
     #include <string.h>  // strcmp(3)
4
     #include "misc.h"           // err()
6    #include "parser.h"         // parse()
     #include "translater.h"     // print_atoms(), on_reduce()
8    #include "lexer.h"          // MAX_VAR_LEN
     #include "var-register.h"   // MAX_VARS_COUNT
10
     /**
12    * The z-matrice compiler that will output
      * molecules in cartesian coordinates format.
14    */
     int main(int argc, char **argv)
16   {
             if (2 == argc && 0 == strcmp(argv[1], "-v")) {
18                   printf("%s by Adrien Kunysz and Sam Kyritsoglou\n"
                             "built on %s"
     "\n"
20                           "MAX_VARS_COUNT = %u"
     "\n"
                             "MAX_VAR_LEN = %u"
     "\n",
22                           argv[0], __DATE__ " " __TIME__,
                             MAX_VARS_COUNT, MAX_VAR_LEN);
24                   return EXIT_SUCCESS;
             }
26
             FILE *input  = argc > 1 ? fopen(argv[1], "r") : stdin;
28           FILE *output = argc > 2 ? fopen(argv[2], "w") : stdout;
             if (NULL == input || NULL == output)
```

```
30                  err("Usage: %s [-v] [ZMAT_FILE] [OUTPUT_FILE]\n", argv[0]);
        parse(input, on_reduce);
32      print_atoms(output);
        return EXIT_SUCCESS;
34  }
```

A.4 Fichiers non essentiels

Listing 20 – Makefile

```
# The syntax file to use for the shift-reduce table generation.
SYNTAX_FILE = zmat.syn

# The file to which we'll write and
# from which we'll load the shift-reduce table.
# It will be in the includes/ directory.
SRT_FILE = srtable.h

# The file to which we'll write and
# from which we'll load the optimised syntaxic
# rules. It will be in the includes/ directory.
RULES_FILE = rules.h

# Add "-O3 -DNDEBUG" for production.
# Add "-pg" for profiling.
# Add "-DMAX_VARS_COUNT=42" to change max variables count (default is 1024).
CFLAGS += -g -Wall -std=c99 -Iincludes                    \
          -DSYNTAX_FILE='"../syntaxes/$(SYNTAX_FILE)"'

# the program to use to compile .dot files
DOT        = dot

# the dot output format to use
DOT_EXPORT = ps

###

# compiler specific stuff
ifeq (gcc, $(findstring gcc, $(CC)))
        CFLAGS += -W -pedantic -fshort-enums
endif
ifeq (icc, $(findstring icc, $(CC)))
        CFLAGS += -Wcheck -Wbrief -no-gcc -fshort-enums
endif

###

# default target
zmat2cart:

# extra rules
%.$(DOT_EXPORT): %.dot
        $(DOT) -T$(DOT_EXPORT) -o $@ $<

# FIXME: why doesn't this work ?
#%.dot: test/parser-test
```

```
%.dot: %
48          ./test/parser-test $< $@

50  # special targets for files shared between syntax analyzer and parser
    srt-parser.o: CFLAGS += -DSRT_FILE='"$(SRT_FILE)"'
52  srt-parser.o: srt.c includes/$(SRT_FILE)
            $(COMPILE.c) -o srt-parser.o srt.c
54  srt-analyzer.o: CFLAGS += -USRT_FILE -URULES_FILE
    srt-analyzer.o: srt.c
56          $(COMPILE.c) -o srt-analyzer.o srt.c

58  syntax-parser.o: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"'
    syntax-parser.o: syntax.c includes/$(RULES_FILE)
60          $(COMPILE.c) -o syntax-parser.o syntax.c
    syntax-analyzer.o: CFLAGS += -USRT_FILE -URULES_FILE
62  syntax-analyzer.o: syntax.c
            $(COMPILE.c) -o syntax-analyzer.o syntax.c
64
    # files generated by the syntax analyzer
66  includes/$(SRT_FILE): synalyze
            ./synalyze includes/$(SRT_FILE) includes/$(RULES_FILE)
68  includes/$(RULES_FILE): synalyze
            ./synalyze includes/$(SRT_FILE) includes/$(RULES_FILE)
70
    # the syntax analyzer
72  synalyze: CFLAGS += -USRT_FILE -URULES_FILE
    synalyze: srt-analyzer.o syntax-analyzer.o
74
    # the compiler
76  zmat2cart: LDFLAGS += -lm
    zmat2cart: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"' -DSRT_FILE='"$(SRT_FILE)"'
78  zmat2cart: translater.o misc.o var-register.o  \
               parser.o syntax-parser.o srt-parser.o lexer.o stack.o
80
    # #define or not RULES_FILE and SRT_FILE for files #including syntax.h or srt.h
82  parser.o: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"' -DSRT_FILE='"$(SRT_FILE)"'
    # not strictly needed for lexer.o but 'better safe than sorry
84  lexer.o: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"'

86  translater.o: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"' \
                            -DSRT_FILE='"$(SRT_FILE)"'
88  translater.o: includes/$(RULES_FILE) includes/$(SRT_FILE)

90  ### end of syntax analyzer/parser tricks

92  # test programs
    test/stack-test: stack.o misc.o
94
    test/lexer-test: LDFLAGS += -lm
96  test/lexer-test: lexer.o misc.o var-register.o syntax-parser.o
```

```
98  test/rel2cart-test: LDFLAGS += -lm
    test/rel2cart-test: translater.o misc.o var-register.o
100
    test/var-register-test: LDFLAGS += -lm
102 test/var-register-test: var-register.o misc.o

104 test/parser-test: LDFLAGS += -lm
    test/parser-test: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"' \
106                             -DSRT_FILE='"$(SRT_FILE)"'
    test/parser-test: parser.o stack.o misc.o syntax-parser.o \
108                   lexer.o srt-parser.o var-register.o

110 test/translater-test: LDFLAGS += -lm
    test/translater-test: CFLAGS += -DRULES_FILE='"$(RULES_FILE)"' \
112                                 -DSRT_FILE='"$(SRT_FILE)"'
    test/translater-test: parser.o stack.o misc.o syntax-parser.o \
114                       lexer.o srt-parser.o var-register.o translater.o

116 .PHONY: clean
    clean:
118         $(RM) *~ includes/*~ test/*~ *.o test/*-test core core.* \
                  includes/$(SRT_FILE) includes/$(RULES_FILE) synalyze
```

Listing 21 – test/stack-test.c

```
    #include <stdlib.h> // rand(), srand()
2   #include <time.h> // time()
    #include <stdio.h>  // printf()
4   #include "stack.h"  // push_stack(), pop_stack();
    #include <misc.h>  // xmalloc()
6
    #define DEFAULT_SIZE_TEST 4242
8
    typedef struct int_t {
10          int data;
    } int_t;
12
    static void mk_array_rand(stack_head_t *stack, unsigned int *array, unsigned in
14  {
            for (unsigned int i = 0; i < *nb_test; i++){
16                  array[i] = rand();
                    stack_push(stack, &array[i]);
18  //                  printf("stack-test\tmk_array_rand\t%u : %u\n",i, array[i]);
            }
20          printf("stack-test\tmk_array_rand\t done\n");
    }
22
    static bool test_array_rand(stack_head_t *stack, unsigned int *array, unsigned
24  {
            unsigned int *get;
```

```
26          bool error = false;
            for (unsigned int i = *nb_test-1; i > 0; i--) {
28                  get = stack_pop(stack);
                    if (*get != array[i]){
30                          error = true;
                            printf("stack-test\ttest_array_rand_peek\t%u : %u\n"
32                          "\t\t get  : %u\n",
                            i,array[i],*get);
34                  }
            }
36          return error;
}
38
static bool test_array_rand_peek(stack_head_t *stack, unsigned int *array, unsi
40 {
            unsigned int *get;
42          bool error = false;
            unsigned int rand_num;
44          for (unsigned int i = *nb_test-1; i > 0; i--) {
                    rand_num = rand()%*nb_test;
46                  get = stack_peek(stack,rand_num);
                    if (NULL == get) {
48                          err("stack-test\ttest_array_rand\t "
                            "get NULL with rand_num = %u\n",rand_num);
50                  }
                    if (*get != array[*nb_test - rand_num - 1]){
52                          error = true;
                            printf("stack-test\ttest_array_rand\t%u : %u\n"
54                          "\t\t get  : %u\n",
                            rand_num,array[*nb_test - rand_num - 1],*get);
56                  }
            }
58          return error;
}
60
int main(int argc, char *argv[])
62 {
            srand ( time(NULL) ); // init the seed
64
            unsigned int nb_test;
66          if (2 == argc)
                    nb_test = atoi(argv[1]);
68          else {
                    nb_test = DEFAULT_SIZE_TEST;
70                  printf("stack-test\tUse default number of add to the stack"
                           " : %u\n",nb_test);
72          }

74          stack_head_t *stack = stack_mk();
            unsigned int array_test[nb_test];
```

```
76
                mk_array_rand(stack,&array_test[0],&nb_test);
78              if (test_array_rand_peek(stack,&array_test[0],&nb_test))
                        printf("stack-test\tERROR during peek test\n");
80              else
                        printf("stack-test\tNo error during peek test\n");
82
//              debug("Test stack_peek()\n");
84  //          unsigned int rand_num;
//              unsigned int *get;
86  //          for (unsigned int i = 0; i < rand()%nb_test; ++i){
//                      rand_num = rand()%nb_test;
88  //                  debug("rand_num = %u and ", rand_num);
//                      get = stack_peek(stack,rand_num);
90  //                  if (NULL == get)
//                              err("Value return by stack_peek() get is NULL\n");
92  //
//                      if (*get == array_test[nb_test-rand_num - 1]){
94  //                          printf("\tstack-test\t%u : PEEK OK : %u\n",
//                              nb_test-rand_num - 1,*get);
96  //                  }
//                      else {
98  //                          printf("\tstack-test\t%u : PEEK KO : %u\n",
//                              nb_test-rand_num - 1,*get);
100 //                  }
//              }
102             if (test_array_rand(stack,&array_test[0],&nb_test))
                        printf("stack-test\tERROR during simple test\n");
104             else
                        printf("stack-test\tNo error during simple test\n");
106
                mk_array_rand(stack,&array_test[0],&nb_test);
108
                unsigned int discard_rand = rand()%nb_test;
110             printf("stack-test\tDo Discart (remove %u)\n",discard_rand);
                unsigned int nb_test_after = nb_test;
112             stack_discard(stack, discard_rand);
                nb_test_after -= discard_rand;
114
                if (test_array_rand(stack,&array_test[0],&nb_test_after))
116                     printf("stack-test\tERROR during discard test\n");
                else
118                     printf("stack-test\tNo error during discard test\n");

120             mk_array_rand(stack,&array_test[0],&nb_test);
                stack_free(stack);
122             printf("stack-test\tNo error during stack_free()\n");

124             stack = stack_mk();
                unsigned int *tmp;
```

```
126             for (unsigned int i = 0; i < nb_test; i++){
                        tmp = xmalloc(sizeof(*tmp));
128 //                      debug("malloc done\n");
                        *tmp = rand();
130 //                      debug("rand() done\n");
                        stack_push(stack, tmp);
132 //                      debug("stack-test\t%u :%u\n",i, *tmp);
                }
134 //              debug("stack-test\t stack_remove()\n");
                stack_remove(stack);
136             printf("stack-test\tNo error during stack_remove()\n");

138             return EXIT_SUCCESS;
    }
```

Listing 22 – test/var-register-test.c

```
    #include <stdlib.h>
2   #include <stdio.h>
    #include <math.h>
4   #include "misc.h"
    #include "var-register.h"
6
    /**
8    * Only to avoid linking with lexer.o.
     */
10  unsigned int current_line = 0;

12  int main(int argc, char **argv)
    {
14          unsigned int ids[argc];
            double vals[argc];
16          for (int i = 0; i < argc; i++) {
                    ids[i] = register_var(argv[i]);
18                  register_setval(ids[i], (vals[i] = 13.37 + i));
            }
20          for (int i = 0; i < argc; i++) {
                    unsigned int id = register_var(argv[i]);
22                  if (id != ids[i])
                            err("id mismatch (%s, %u, %u)\n", argv[i], ids[i], id);
24                  double val = register_id2val(ids[i]);
                    if (!iseq(val, 13.37 + i))
26                          err("value mismatch (%s, %f, %f)\n",
                                argv[i], vals[i], val);
28                  printf("variable ok (%s, %u, %f)\n", argv[i], id, val);
            }
30          return EXIT_SUCCESS;
    }
```

Listing 23 – test/lexer-test.c

```
#include <stdlib.h>   // EXIT_*
```

```c
#include <stdio.h>   // printf(), stdin, stdout, FILE
#include "misc.h"    // err()
#include "lexer.h"   // next_symbol(), token_t, print_token()

int main(int argc, char *argv[])
{
        FILE *stream = argc < 2 ? stdin : fopen(argv[1], "r");
        if (NULL == stream)
                err("Usage: %s [filename]\n", argv[0]);

        lex_init(stream);
        token_t tk = { .type = TK_INPUT };

        do {
                lex_next(&tk);
                print_token(stdout, &tk);
                printf("\n");
        } while (TK_INPUT != tk.type && S_INVALID != tk.type);

        if (S_INVALID == tk.type)
                err("Invalid token in input stream ?\n");

        return EXIT_SUCCESS;
}
```

Listing 24 – test/parser-test.c

```c
#include <stdio.h>   // fopen(3), stdin, stdout
#include <stdlib.h>  // EXIT_SUCCESS
#include <assert.h>  // assert(3)

#include "misc.h"    // err()
#include "parser.h"  // parse(), tree2dot()

void on_reduction(const parse_tree_t *node)
{
        assert(NULL != node);

        static unsigned int rcount = 0;
        fprintf(stderr, "reduced to %s (%u)\n",
                        sym2str(node->tok.type), ++rcount);
}

int main(int argc, char **argv)
{
        FILE *input  = argc >= 2 ? fopen(argv[1], "r") : stdin;
        FILE *output = argc >= 3 ? fopen(argv[2], "w") : stdout;
        if (NULL == input || NULL == output)
                err("Usage: %s [ZMAT_FILE] [OUTPUT_GRAPH]\n", argv[0]);
        parse_tree_t *ptree = parse(input, on_reduction);
        tree2dot(ptree, output);
```

```
    free_tree(ptree);
    return EXIT_SUCCESS;
}
```