Ensta Bretagne
2, rue François Verny
29806 Brest cedex
France
Tel +33 (0)2 98 34 88 00

Polycopié

2A

28 avril 2020

Introduction à la compilation

Jean-Christophe Le Lann
`lelannje@ensta-bretagne.fr`

Table des matières

Chapitre 1

Introduction

> The purpose of (scientific) computing is insights, not numbers
>
> Richard Hamming

1.1 Le code source : une donnée très spéciale

La compilation est une discipline centrale en Informatique : un compilateur est un programme particulier qui traite une donnée très spéciale : un *code source*. Traditionnellement, le compilateur assure la traduction de ce source en un code binaire, qui peut être exécuté par une machine. Nous étudierons quelques variantes à cette définition, mais l'essentiel tient à cela.

Ce code source paraît d'emblée plus imposant que les traditionnelles données (scalaires, vecteurs, matrices,..) qui sont manipulées par les programmes les plus courants. L'énorme majorité des programme *ne sont pas* des compilateurs ! Un tel programme source semble exprimer bien plus que de simples données issues de la physique, des mathématiques etc : un code source, pensé et écrit par un humain (vous !) semble avoir un statut spécial, éloigné de la froideur d'une simple donnée. Un code source reflète notre façon de penser, et n'a de sens qu'à l'exécution (hormis pour les langages descriptifs). Un compilateur semble donc capable de le *comprendre* et de prendre part à notre réflexion. Une sorte d'"Intelligence artificielle", en quelque sorte. Pourtant, par une approche systématique qui consiste immanquablement à "diviser pour régner", le travail d'un compilateur n'a rien de magique, et il ne s'inscrit pas dans la discipline "IA" du moment. Il ne fait que transformer cette donnée qu'est le code source en une autre donnée qu'est l'assembleur.

Les techniques de construction d'un compilateur sont désormais bien connues. Nous allons nous atteler à les découvrir à travers des exercices de programmation. La construction d'un compilateur est une activité de programmation enrichissante car elle nécessite la mise en application d'un grand nombre de concepts simples. Mieux encore, elle donne une *satisfaction* immédiate : un compilateur manipule d'emblée, de manière mécanique, un grand nombre de données (lexèmes, classes), les organise de différentes manières (liste, arbres). La génération de code est l'aboutissement de cet enchaînement mécanique : la production de code assembleur complexe à partir du source initial en apparence simple peut donner le vertige !

1.2 Anatomie d'un compilateur. la *Big picture*

La figure suivante présente la *Big picture* d'un compilateur.

Ce schéma s'applique pour l'essentiel à tous les compilateurs : les vrais compilateurs comme GCC ou la suite Clang-LLVM présentent toutefois un nombre vertigineux de *passes* d'optimisations, que nous n'avons pas évoquées dans ce schéma. Ces passes s'appliquent le plus souvent sur une représentation intermédiaire (IR), qui sera introduite au chapitre 7.

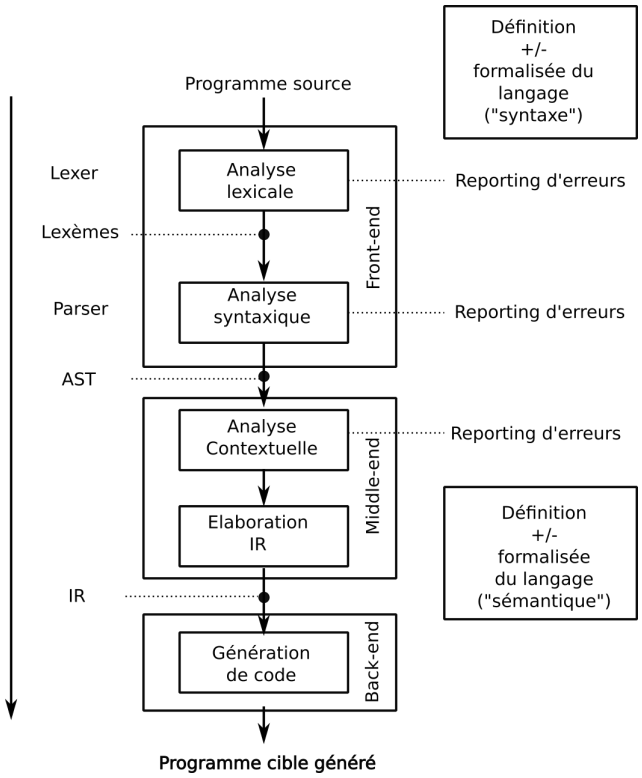


FIGURE 1.1: Anatomie d'un compilateur

1.3 Plan du cours

Nous suivrons dans ce cours la trame séquentielle du schéma précédent.

- Analyse lexicale : il s'agit de découper le texte source en unité lexicale ou *lexèmes*. Par analogie avec la language parlée, nous pouvons dire que nous sommes ici au niveau du *mot*.

- Analyse syntaxique : il s'agit ici de vérifier que la suite de lexèmes respecte la *grammaire* du langage source. Nous sommes au niveau de *la phrase* du langage.

- Arbre de syntaxe abstraite : il s'agit d'une représentation *en mémoire* du programme source. Un grand nombre de détails ont été filtrés. Il ne reste qu'une syntaxe *pure*, *abstraite*, qui fait la joie des Informaticiens, qui la considère comme un *modèle*.

- Notion de visiteur : il s'agit d'un *design pattern* [3], couramment utilisé dans le domaine de la compilation, et qui permet de parcourir l'AST précédent.

- Analyse contextuelle : il s'agit ici de vérifier des erreurs sur la signification ou *sémantique* du langage. Les dernières erreurs sont détectées ici.

- Représentation intermédiaire : il s'agit ici de *diviser pour régner*. On cherche à rendre simples les expressions complexes, et à "retomber" sur des instructions simples et en nombre réduit.
- Génération de code : le but de notre compilateur.

1.4 Approche pédagogique

Le livret s'accompagne actuellement (2020) de

- 7 vidéos sur Youtube, indiquées sur le site Moodle du cours.
- d'un slideware que vous pouvez consulter sur Moodle.

Les exercices sont simples et vise à construire de manière incrémentale un petit compilateur, autour d'une grammaire d'un mini-langage appelée Mini-C. Un projet vous est également demandé, et servira pour la totalité de l'évaluation : il s'agit de concevoir un langage et son compilateur dans son intégralité. Profitez de ce projet pour joindre l'utile à l'agréable : vos loisirs ou aspirations peuvent vous aider à *imaginer* un langage utile. Cela peut être inspiré d'un sport, d'un jeu, d'une science (simulation ,visualisation etc). Le langage peut être un langage de programmation, ou un langage de description (circuits, etc). C'est également l'occasion, si vous le souhaitez d'approfondir un langage, ou d'en apprendre un nouveau. Cela peut concerner le langage de réalisation de votre compilateur, ou le langage d'entrée lui-même. Le site Moodle du cours recense notamment des *mini*-langages de langages courants (mini-java, etc).

Les figures imposées sont les suivantes :

- Lexer, Parser
- AST
- Visiteur de type pretty-printer.

1.5 Compilation et interprétation

L'immense majorité des langages de programmation s'appuient sur un tel compilateur. On lui oppose parfois la notion d'interprétation. Le cours actuel (2020) n'aborde pas le domaine de l'interprétation [1]. Sachez toutefois qu'un interpréteur ne génère pas de code, mais réalise l'exécution directement à partir d'une structure intermédiaire (l'AST), étudiée ici. Il est donc assez aisé de réaliser dans votre projet un tel interpréteur.

1.6 Conseils bibliographiques

Il existe un grand nombre de livres sur la compilation, mais je recommande en particulier [6] et [2]. Le premier (qui repose sur Java) est selon moi le plus proche de mon ambition de vous faire développer un compilateur dans son intégralité, et qui maintient un équilibre entre Génie Logiciel, Compilation et Architecture matérielle. Plusieurs blogs proposent également des approches intéressantes. Je recommande en pariculier

- Eli Bendersky, notamment pour ses propositions autour de Python.
- Vidar Hokstad (en Ruby)

[1] prévu pour 2021

- Peter Sovietov (`true-grue` sur github)
- Marcelo G. de Andrade (sur github)
- On pourra lire l'article de Ghuloum [4] (basé sur Lisp), qui a inspiré un grand nombre de blogs de qualité.

1.7 Pour aller plus loin

Enfin, sachez que la lecture du code de compilateurs (et interpréteurs) comme Python ou Ruby devient accessible selon moi à l'issu de ce cours, voire pendant votre apprentissage. En particulier l'accès à l'AST de Python et Ruby est très facile ! Bonne lecture.

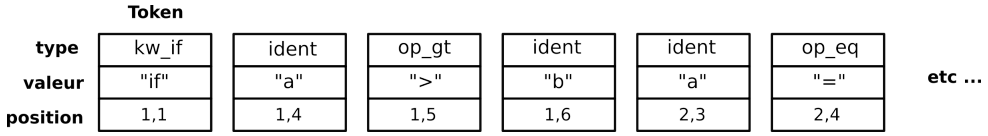
Chapitre 2

Analyse lexicale

Le but de l'analyse lexicale est de découper un programme source –un texte– en une suite d'élements insécables significatifs : les *lexèmes* (ou *tokens* en anglais). On peut garder l'analogie d'une découpe d'une phrase en mots. L'analyseur lexical ou lexer assure cette fonction, et sera en charge de fournir ces lexèmes à l'analyseur syntaxique qui opèrera par la suite.

Ainsi le texte source suivant peut être découpé de la sorte :

```
if a>b
  b=b-a;
else
  a=a-b;
end
```



2.1 Elements lexicaux

Les élements lexicaux d'un langage sont de 4 types :

- Les **mots clés** du langage : ce sont les mots "réservés" du langage, comme le `if` ou `begin`.
- Les **signes de ponctuation** : le point-virgule ou la parenthèse ouvrante `(` et fermante `)`, etc
- Les **identifiants** : `matrix_v2`
- Les **litéraux** : nombre réel `-12.455e+3`, chaîne de caractères `"enter a value"`, etc

2.2 Expressions régulières

L'outil central pour réaliser la découpe d'un texte en lexèmes est la notion d' "expression rationnelle", plus souvent appélée "expression régulière" ou "regexp". C'est Stephen Coole Kleene

qui a théorisé cette notion dans les années 40. Une expression régulière est une chaîne de caractères, qui décrit un *motif* régulier : ce motif est représentatif d'un ensemble –souvent vaste voire infini– de chaînes de caractères. Un exemple d'expression régulière est la regexp `/a+/`. Ce motif décrit l'ensemble des chaînes de caractères "a","aa","aaa" etc. Les barres de séparation indiquent que l'on a affaire à une Regexp de Ruby. Les regexps sont elles-mêmes un langage complet, qu'il est important de maîtriser. Les regexp ne servent pas qu'aux compilateurs, mais vous permettront de rapidement chercher dans un texte quelconque, formatté ou non, d'effectuer des remplacement, des statistiques relatives à ce texte (fréquence d'apparition de certain motifs), etc. Le tableau 2.1 recense quelques éléments du langage des expressions régulières.

TABLE 2.1: Langage d'expressions régulières

`.`	Matche [1] n'importe quel caractère (unique)
`a[0-9]`	**Séquence** : ici a suivi (obligatoirement) d'un (seul) chiffre.
`*`	**Répétition**. Matche l'élément précédent l'étoile, zéro fois ou plus.
`+`	**Répétition stricte**.Matche l'élément précédent l'étoile, une fois ou plus.
`|`	**Alternative**.Opérateur d'alternative ("ou")
`?`	**Option**. Match éventuellement l'élément précédent le point d'interrogation.
`[ ]`	Matche n'importe quel caractère (unique) figurant dans la liste entre crochets
`[^ ]`	Matche n'importe quel caractère sauf ceux mentionnés entre crochets
`^`	Matche le début d'une ligne
`\$`	Matche a fin d'une ligne

Quelques exemples :

TABLE 2.2: Exemples de chaînes décrites par des expressions régulières

ex1	ex2	ex3	regexp
a	aa	aaa	`/a+/`
1	3.14	-3.141	`/[+-]?[0-9]+(\.[0-9]+)?/`
Albert	albert	alBErt123	`/[a-zA-Z]+[0-9]*/`
Einstein	Bohr		`/Einstein|Bohr/`

On précise que le moteur d'expressions régulières (aussi appelées "Regexp") de Ruby s'appelle Oniguruma, et qu'il propose de nombreuses facilités d'écriture, plus ou moins classiques. On donne ci-dessous quelques exemples.

2.3 Analyse lexicale : première approche

A l'aide des expressions régulières, on peut tenter une première approche de l'analyse lexicale.

Approche naïve Le code Ruby suivant illustre une une première automatisation de la découpe d'un code en lexème : une chaine de caractère est analysée petit à petit, et réduite à chaque itération. On précise que dans ce code Ruby, le symbole `\$&` fait référence à la dernière expression "matchée" par l'expression régulière.

FIGURE 2.1: Première écriture d'un lexeur primitif

```
src="void main(){a=42}"
while src.size>0
  case src
  when /void|main|\(|\)|\{|\}|[a-z]+|\=|[0-9]+|\s+/
    src.delete_prefix!($&)
    puts "token : #{$&}"
  else
    raise "unknown token : #{src[0..-1]}"
  end
end
```

Emission de lexèmes Fort de l'expérience précédente, on peut désormais structurer quelque peu notre approche : le code suivant présente une méthode `lex` qui prend en argument un texte source, et le convertit en un tableau de lexèmes.

```
Token=Struct.new(:kind,:val)

def lex src
  tokens=[]
  while src.size>0
    case src
    when /\A\s+/
      tokens << Token.new(:space,$&)
    when /\Avoid\b/
      tokens << Token.new(:void,$&)
    when /\A\{/
      tokens << Token.new(:lbrace,$&)
    when /\A\}/
      tokens << Token.new(:rbrace,$&)
    when /\A\(/
      tokens << Token.new(:lparen,$&)
    when /\A\)/
      tokens << Token.new(:rparen,$&)
    when /\A\=/
      tokens << Token.new(:eq,$&)
    when /\A[a-z]+\b/
      tokens << Token.new(:ident,$&)
    when /\A[0-9]+/
      tokens << Token.new(:int_lit,$&)
    else
      raise "lexical error : #{src[0..-1]}"
    end
    src.delete_prefix!($&)
  end
  return tokens
end

src="void main(){a=42 voidmain}"
```

```
pp lex(src)
```

Le code précédent revoie :

```
[#<struct Token kind=:void, val="void">,
 #<struct Token kind=:space, val=" ">,
 #<struct Token kind=:ident, val="main">,
 #<struct Token kind=:lparen, val="(">,
 #<struct Token kind=:rparen, val=")">,
 #<struct Token kind=:lbrace, val="{">,
 #<struct Token kind=:ident, val="a">,
 #<struct Token kind=:eq, val="=">,
 #<struct Token kind=:int_lit, val="42">,
 #<struct Token kind=:rbrace, val="}">]
```

On notera au passage l'expression régulière pour les mots clés et identifiants, qui se terminent ici par `\b` (*word boundary*) pour en signaler la fin. Sans cela, un identifiant appelé "voidmain" (en un seul mot) apparaîtrait comme la succession de 2 lexèmes.

Discussion Le code suivant, fonctionnel, reste quelque peu frustre, et on peut l'améliorer. Plusieurs pistes d'amélioration sont possibles : on peut par exemple chercher à capturer la position du lexème, en précisant le numéro de ligne et de colonne, qui seront utiles à l'utilisateur lorsqu'on lui remontera des erreurs. Ces erreurs pourront être lexicales, mais ces indications de positions nous serviront tout au long des *passes* du compilateur et notamment lors de l'analyse syntaxique, qui suivra l'analyse lexicale. Cette capture va rendre notre propre code un peu plus complexe. C'est un très bon exercice, que je laisse à votre sagacité. Je vous livre toutefois un ambryon de code Python en annexe, qui peut vous servir à démarrer cet exercice.

En général, tous les langages de programmation proposent certes des facilités dans la manipulation des expressions régulières –comme nous venons de l'illustrer avec Ruby–, mais également des bibliothèques bien plus abouties que notre dernier essai. Dans le cas de Ruby, il existe plusieurs outils complets d'analyse lexicale. Ce sera également le cas pour l'analyse syntaxique. On appelle ces outils des *lexers* ou *scanners*.

2.4 Outils de génération de lexers

2.4.1 StringScanner

StringScanner est un *gem* Ruby, qui permet de facilement réaliser les tâches esquissées dans notre dernière tentative.

```
s = StringScanner.new('This is an example string')
s.eos?               # -> false

p s.scan(/\w+/)      # -> "This"
p s.scan(/\w+/)      # -> nil
p s.scan(/\s+/)      # -> " "
p s.scan(/\s+/)      # -> nil
p s.scan(/\w+/)      # -> "is"
s.eos?               # -> false
```

```
p s.scan(/\s+/)      # -> " "
p s.scan(/\w+/)      # -> "an"
p s.scan(/\s+/)      # -> " "
p s.scan(/\w+/)      # -> "example"
p s.scan(/\s+/)      # -> " "
p s.scan(/\w+/)      # -> "string"
s.eos?               # -> true

p s.scan(/\s+/)      # -> nil
p s.scan(/\w+/)      # -> nil
```

L'exemple précédent n'illustre qu'une petite partie de ses capacités. Il possède d'autres méthodes :

- Avancer le pointeur de lecture : `getch,get_byte,scan,scan_until,skip,skip_until`
- Faire une recherche, sans consommer la chaîne (lookahead) : `check,check_until,exist?,match?,peek`
- Se localiser `beginning_of_line?,eos?, rest?, pos`
- etc

2.4.2 RLTK

RLTK (Ruby language Tool Kit) est un ensemble de classes Ruby, qui permettent de développer de nouveaux langages. Entre autres, il existe une classe distincte, dédiée à l'analyse lexicale. L'exemple suivant illustre le cas d'une analyse lexicale minimale d'expressions arithmétiques parenthésées. Attention ! Pour l'instant, on ne reconnait pas le caractère correcte des "phrases" (ici expression arithmétiques) elles-mêmes, mais seulement leurs constituants (les mots ou lexèmes).

```
class Calculator < RLTK::Lexer
  rule(/\+/) { :PLS }
  rule(/-/)  { :SUB }
  rule(/\*/) { :MUL }
  rule(/\//) { :DIV }

  rule(/\(/) { :LPAREN }
  rule(/\)/) { :RPAREN }

  rule(/[0-9]+/) { |t| [:NUM, t.to_i] }

  rule(/\s/)
end
```

Cet exemple de RLTK est intéressant, car la plupart des outils associés ressemblent beaucoup à cela : à travers des mots clés (ici `rule`), on vient expliquer (ici à travers des blocks `\{...\}`) quelle action le lexeur doit effectivement réaliser lorsque la règle peut s'appliquer (lorsque la phrase "matche"). En général, on cherche à émettre un *objet* lexème, mais on peut imaginer tout autre chose. Dans l'exemple précédent, l'auteur a choisi d'émettre tantôt un simple symbole Ruby (par exemple `:PLS`), tantôt un tableau `[:NUM,t.to_i]` [2].

[2] Nous verrons plus tard, que cette dernière manière de faire, concernant le tableau, est plus profonde que les choses ne le laisse penser...

Sachez que la plupart des outils de génération de Lexers sont généralement intimement associés aux outils de génération de Parser (que nous verrons au chapitre suivant) : les outils les plus connus sont *lex* et *yacc*. Leurs descendants ou clones s'appellent *flex* et *bison*. Ils sont généralement utilisés dans un environnement de langage *C*. Depuis les années 90, plusieurs outils leur ont succédé : notamment ANTRL, mais ils proposent de mélanger les deux fonctions des outils précédents. Par ailleurs le terme générique de ce type d'outils est "compilers compiler", qu'on peut traduire par "compilateur de compilateurs", mais il s'agit là d'un raccourci malheureux : ces outils ne génèrent "que" le lexer et le parser.

2.5 Analyse lexicale et automates.

On doit voir les expressions régulières comme un "langage" facilitant la description du lexer. Toutefois, le mécanisme de reconnaissance de motifs, propre aux expressions régulière est fondé sur *les machines à états finis (FSM)*. C'est ce mécanisme qui réalise concrètement, en machine, la reconnaissance.

Ce formalisme rigoureux permet en outre de faire bien mieux que nos premiers exemples : en effet, n'avez-vous pas pressenti une certaine inefficacité, lorsque nous avons cherché à appliquer les expressions régulières, les unes à la suite des autres, à la recherche de la bonne correspondance ? Qu'adviendrait-il par exemple si, dans notre `case`, notre programme d'analyse lexicale devait examiner *toutes* ces expressions, avant de tomber sur la dernière, cette dernière étant la "bonne" ? On verrait là s'écrouler la performance de notre analyse et par là-même de tout notre compilateur ? Il est quelque peu gommé par les performances de nos ordinateurs modernes (notre procédé fonctionne dans un temps très raisonnable), mais reste un problème important. En réalité, notre implémentation naïve nous permet de nous poser d'autres questions : est-ce que le motif le plus long est bien repéré ? Est-ce que l'on peut décrire n'importe quelle suite de lexèmes avec un tel mécanisme ? etc. Historiquement, ce problème a été pris au sérieux.

2.5.1 Automates d'états finis déterministes : DFA

Un automate d'état finis (FSM : finite state machine) s'appelle ainsi car il présente un nombre fini d'états (il existe des automates qui présentent un nombre infinis d'états dénombrables). Les arcs permettent de transiter d'un état à un autre (ces arcs seront appelés "transitions" de manière interchangeable), chacun des arcs portant le label d'un symbole du code source (lettres, chiffres, etc). Parmi les états, il existe un état particulier, qui indique l'état de départ, alors que d'autres états marquent la fin de la reconnaissance. La figure suivante présent quelques automates d'états finis. Nous numérotons les états afin de faciliter la discussion (cette numérotation servira également lors de l'implémentation du moteur d'expressions régulières). Dans tous les cas, l'état de départ est marqué comme "1".

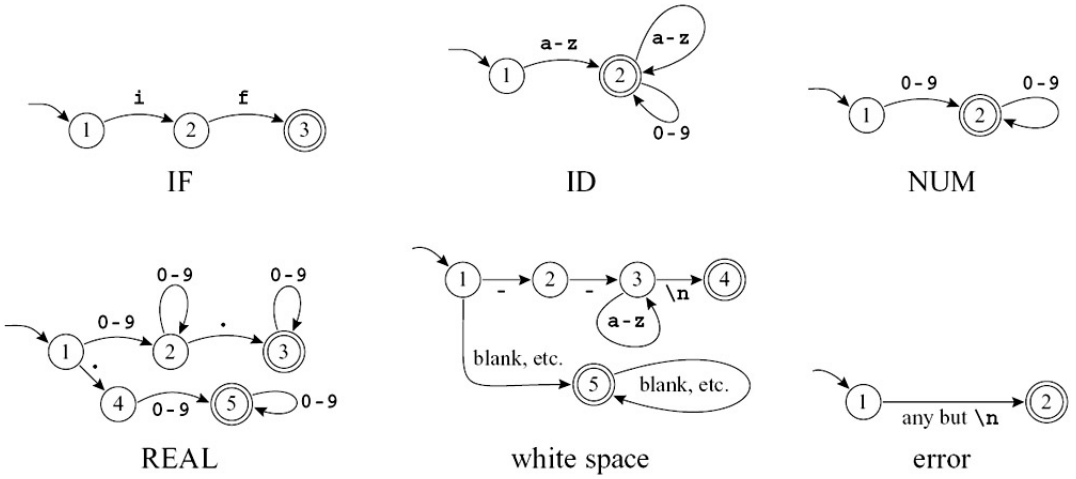


FIGURE 2.2: Six automates (extrait de [1]) représentant les 6 expressions régulières du tableau **??**

Observons le schéma suivant (extrait de [1]) : les états sont représentés par un cercle et les états finaux marqués par un double cercle. L'état de départ est quant à lui reconnaissable, car il possède une transition entrante, qui ne vient d'aucun état. Dans le schéma, on peut également constater que certaines transitions portent plusieurs caractères : il s'agit en réalité d'un "racourci graphique" pour regrouper plusieurs transitions partant d'un même état et aboutissant dans un même état. Par exemple, dans la FSM realtrive à la reconnaissance de 'ID', il existe en réalité 26 transitions entre les états 1 et 2, chacune labelisée par une lettre de l'alphabet.

Dans le cas d'un automate d'états finis *déterministe* (DFA), deux arcs partant d'un même état ne peuvent porter le même symbole. Un automate DFA accepte ou rejète une chaîne de notre code source de la manière suivante : partant de l'état de départ, l'automate nous permet, à partir de chaque caractère, de transiter vers un et un seul état. La chaîne de longueur n est *acceptée* si l'automate a réalisé n transitions et s'il parvient à un état marqué comme final. A l'inverse, si l'état d'arrivée n'est pas un état final, ou si, au cours des transitions, un arc n'a pas pu être suivi, l'automate rejète la phrase en entrée. On dit que *le langage reconnu par un automate est l'ensemble des chaînes de caractères qu'il accepte.*

`/if/`	`IF`
`/[a-z][a-z0-9]*/`	`ID`
`/[0-9]+/`	`NUM`
`/([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)/`	`REAL`
`/("--"[a-z]*"\n")|(" "|"\n"|"\t")+/`	no token, just white space
`/./`	`error`

TABLE 2.3: Expressions régulières associées aux 6 automates de la figure 2.2

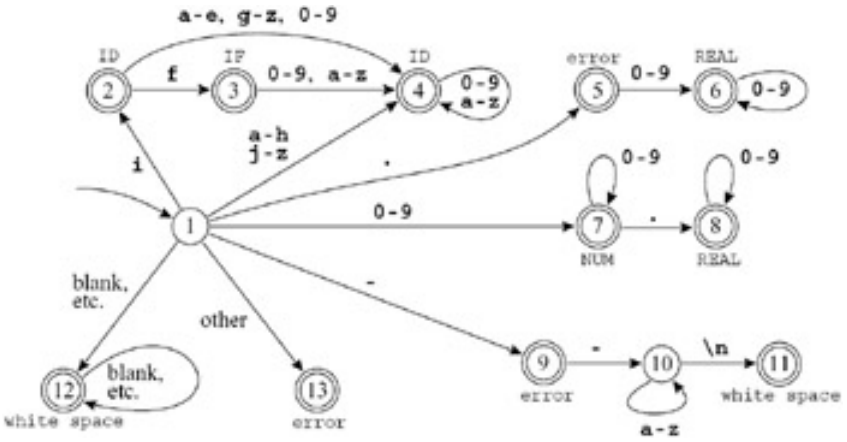


FIGURE 2.3: Automate résultat de la composition de 6 automates (extrait de [1])

La question qu'on doit se poser est désormais la suivante : comment composer ces six automates en une machine *unique*, qui puisse servir à reconnaître *tous* nos lexèmes ? Cette machine recherchée est en réalité l implémentation de notre lexer. Concernant le "comment ?", nous allons nous y pencher, mais on peut dès à présent dévoiler cette machine, présentée sur la figure 2.3. On peut peut observer que chaque état final est estampillé du type du lexème reconnu. L'état 2 a l'aspect de l'état 2 de la machine `IF` et de l'état 2 de la machine intinlinerubyID. L'état 2 étant final, l'état combiné est également final. L'état 3 ressemble à l'état 3 de la machine IF et l'état 2 de la machine ID : comme ces états sont tous deux finaux, nous utilisons une régle de priorité afin de les désambiguïser : nous plaçons le label IF sur l'état 3 car nous souhaitons que ce lexème soit reconnu comme un mot clé, et non un identifiant.

Cet automate peut s'encoder par une matrice de transition : un tableau à deux dimensions, indiçé par le numéro d'état et l' caractère d'entrée. On note la présence d'un état particulier (l'état 0), qui transite vers lui-même quelque soit la lettre en entrée. C'est une manière d'encoder l'absence d'un arc.

FIGURE 2.4: Code C du tableau de transition

```
int edges[][] = { /* ...012...-...e f g h i j... */
/* state 0 */    {0,0,...0,0,0...0...0,0,0,0,0,0...},
/* state 1 */    {0,0,...7,7,7...9...4,4,4,4,2,4...},
/* state 2 */    {0,0,...4,4,4...0...4,3,4,4,4,4...},
/* state 3 */    {0,0,...4,4,4...0...4,4,4,4,4,4...},
/* state 4 */    {0,0,...4,4,4...0...4,4,4,4,4,4...},
/* state 5 */    {0,0,...6,6,6...0...0,0,0,0,0,0...},
/* state 6 */    {0,0,...6,6,6...0...0,0,0,0,0,0...},
/* state 7 */    {0,0,...7,7,7...0...0,0,0,0,0,0...},
/* state 8 */    {0,0,...8,8,8...0...0,0,0,0,0,0...},
  et cetera
}
```

Il faudrait également un autre tableau, qui permet de mettre en correspondance le numéro d'état et les actions afférentes : l'état final 2 doit par exemple s'associer aux actions à réaliser pour émettre les actions liées à ID, etc.

2.5.2 Automates non déterministes : NDA

A l'inverse des automates déterministes vus précédemment, les NDA présentent potentiellement des transitions identiques, partant d'un même état. L'automate a donc "le choix" de transiter vers l'un des états destinations.

Mais revenons aux NDA. Ils offrent un pouvoir d'expression bien supérieur au DFA. Par exemple, il est compliqué de décrire certaines chaînes de manière générique dans l'univers des DFA. Par exemple, supposons que nous cherchions à construire une machine capable de reconnaître les chaînes de longueur n, dont le $n-3$-ième élément est un b. C'est très compliqué en DFA, mais grandement facilité par un NFA. Les NFA indiquent des *possibilités* et non des *certitudes*. Le schéma 2.5 (extrait de [5]) illustre notre exemple précédent : il existe deux possibilités dans le choix des transitions à partir de l'état 1.

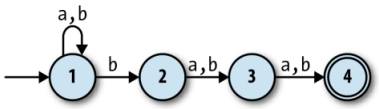


FIGURE 2.5: Automate non déterministe : il existe deux possibilités de transitions, sur la même condition b, pour l'état 1 (extrait de [5])

2.5.3 *free move*

Jusqu'ici, nous avions systématiquement un caractère (lettre, chiffre ou tout autre symbole) présent sur les arcs. Nous devons introduire un nouvel élément possible : il s'agit de ϵ ou *free move*. *epsilon* représente une chaîne vide. Il permet aux NDA de changer d'état sans consommer la moindre entrée. On pourra également représenter ϵ comme des arcs en pointillés. Il est grandement utile pour envisager la traduction systématique d'expressions régulières en tels NDA, mais également lors de la composition des automates.

2.5.4 Composition de NFA : algorithme de Thompson

On illustre ici deux sortes de compositions, telles que rencontrées dans les Regex. D'une part la *composition séquentielle* ("et"), d'autre part la *composition parallèle* ("ou" représenté par la barre verticale "
|" jusqu'ici). Cette composition va se faire par *induction* : on considère un régime courant, où l'on dispose de deux automates, et l'on construit un nouvel automate. Les figures suivantes (extraites de [5]) illustrent cette induction. La technique consiste, dans les deux cas, à utiliser une ou plusieurs transitions ϵ pour relier les états.

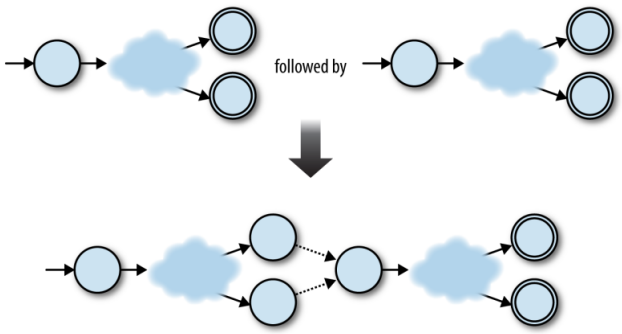


FIGURE 2.6: Automate résultat de la composition séquentielle (extrait de [5])

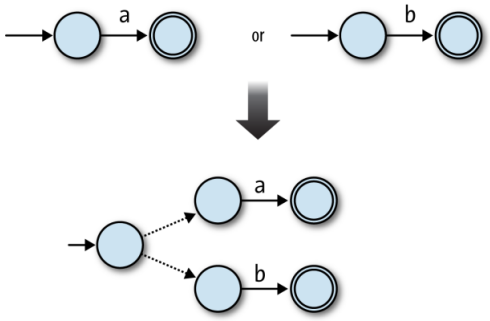


FIGURE 2.7: Automate résultat de la composition séquentielle (extrait de [5])

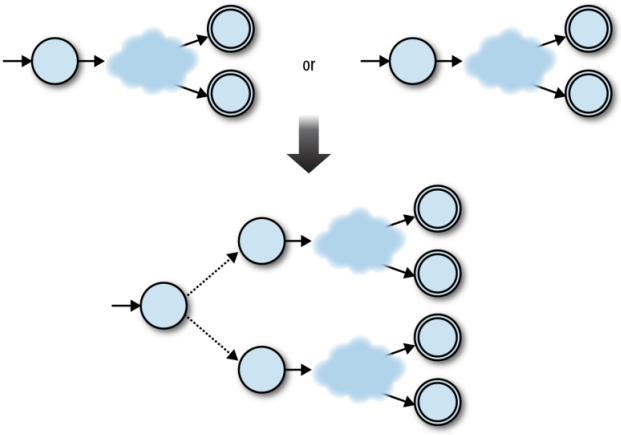


FIGURE 2.8: Automate résultat de la composition séquentielle (extrait de [5])

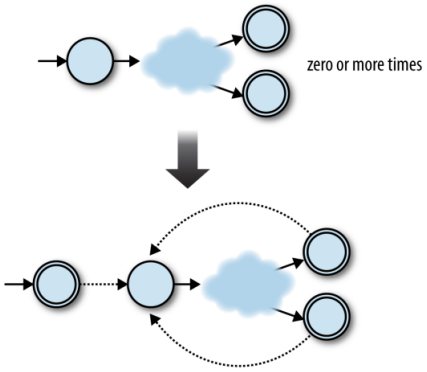


FIGURE 2.9: Automate résultat de la composition séquentielle (extrait de [5])

2.5.5 Equivalence NFA et DFA : algorithme de construction des sous-ensembles

La théorie des automates montre que pour tout automate de reconnaissance d'un langage, il existe un automate déterministe qui reconnaît exactement le même langage. C'est là un résultat important. L'algorithme pour y parvenir s'appelle "construction des sous-ensembles" (*subset construction*).

Comme expliqué par exemple dans [7], les états de l'automate déterministe correspondent à des *ensembles* d'états de l'automate initial. Commençons par l'état initial : cet état regroupe tous les états de départ de l'automate initial, augmenté des états atteignables par les transitions ϵ. Pour les états courant, le procédé est le suivant : sur un caractère c, et partant d'un état-ensemble $\{e_1, \ldots, e_n\}$, on observe les transitions étiquetées par c qui conduisent à des états $\{f_1, \ldots, f_m\}$, ainsi que les états $\{g_1, \ldots, g_p\}$ qui suivent par transitions ϵ. Dans l'automate DFA, on a alors une transition $\{e_1, \ldots, e_n\} \rightarrow_c \{f_1, \ldots, f_m, g_1, \ldots, g_p\}$

Exemple Cet exemple est emprunté à [7].Il vise à établir le DFA de l'expression régulière `/.*toto/`. On commence par établir l'automate NFA de départ 2.10. On a numéroté les états pour plus de facilité. Concernant l'état de départ, il est aisé d'établir qu'il est $\{1, 2, 4, 5\}$ et qu'il est non terminal.

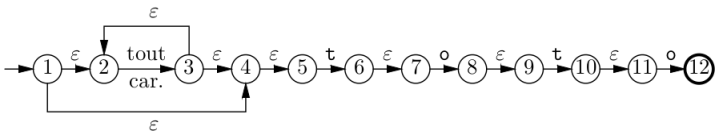


FIGURE 2.10: Automate de départ, représentant le NFA de l'expression régulière `/.*toto/` .

Concernant son état suivant, on peut constater qu'il n'existe que 2 cas possible : soit une transition sur t, soit une transition vers tout autre caractère.

- Sur t, 2 mène à 3, 5 mène à 6, 2 et 4 ne mènent à rien.
- A partir de 3, on atteint 2,4,5 par transitions ϵ.
- A partir de 6, on atteint 7.

En conséquence, on ajoute une transition $\{1, 2, 4, 5\} \rightarrow_t \{2, 3, 4, 5, 6, 7\}$.

Sur un caractère autre que t :

- 2 mène à 3,2
- 2,4,5 ne mènent à rien
- 3 mène à 2,4,5 par transitions ϵ

On ajoute donc une transition "pas t" entre $\{1, 2, 4, 5\} \rightarrow_t \{2, 3, 4, 5\}$, ce qui correspond au schéma de la figure 2.11.

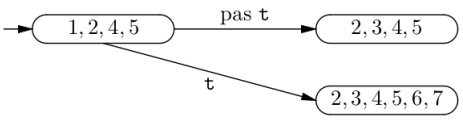


FIGURE 2.11: Automate DFA partiellement construit.

On répète le procédé pour tous les autres états de NFA initial. On trouve mécaniquement le DFA de la figure 2.12.

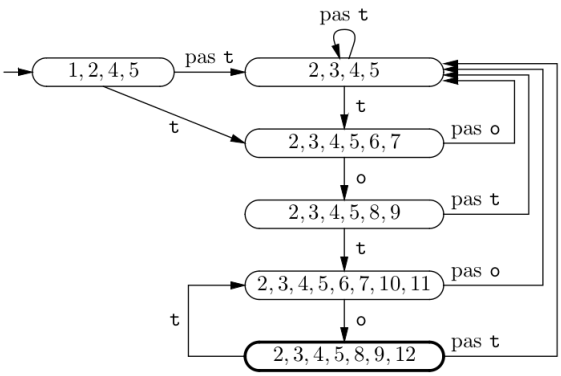


FIGURE 2.12: Automate final DFA, équivalent au NFA de l'expression régulière `/.*toto/`.

2.6 Conclusion

Ce premier chapitre nous a fait prendre connaissance des outils conceptuels qui nous permettent de réaliser une première *transformation* sur le code source de départ. Il s'agit effectivement plus d'une transformation que d'une *analyse*, puisqu'un ensemble de lexèmes à été produit, et que le code source, en tant que tel peut désormais être purement et simplement oublié. Parmi les concepts, on retient évidemment la notion centrale d'expressions régulières, dont l'expression requiert elle-même un langage dédié. Ce langage est lui-même *compilé*, puisque nous n'avons transformé en un automate déterministe. Dans le chapitre suivant, nous allons manipuler le flux de lexèmes générés et aborder l'analyse syntaxique.

Chapitre 3

Analyse syntaxique

Le but de l'analyse syntaxique est double.

1. En premier lieu, elle permet de s'assurer que le langage source est *conforme à la grammaire du langage*, et donc de remonter à l'utilisateur d'éventuelles erreurs liées à l'écriture de son programme. N'oublions pas que l'analyse lexicale se bornait à nous renvoyer un texte découpé ("saucissonné") en lexèmes,sans préoccupation aucune de leur juxtaposition. Un code source sera conforme à la grammaire si l'*enchaînement* des lexèmes, produit par l'analyse lexicale, trouve une *justification* dans les règles de la grammaire.

2. Mais l'analyse syntaxique va plus loin que cette seule vérification de conformité : en second lieu, elle est responsable de la transformation du texte en une structure de données particulièrement importante pour les phases aval du compilateur : cette structure de données est l'**arbre de syntaxe abstraite (AST)**, qui sera étudié au chapitre suivant. Le présent chapitre traite à l'inverse de la **syntaxe concrète**.

3.1 Notion de grammaire

Pour s'assurer qu'un programme source est conforme à la grammaire d'un langage, il est au préalable nécessaire de bien expliciter cette dernière. Historiquement, les premiers compilateurs (Fortran) souffraient d'un manque de formalisation de la grammaire : les grammaires étaient seulement exprimées en anglais. Elles se révélaient très peu pratiques, et sujettes à interprétation. C'est John Backus, qui a apporté au domaine un moyen conceptuel de décrire formellement la grammaire d'un langage de programmation : la première forme s'appelait "Backus Normal Form". Puis, Peter Naur s'est engagé dans une collaboration avec Backus. La BNF s'est légitimement rebaptisée "Backus-Naur form". John Backus a recu le prix Turing pour la création de Fortran et l'énoncé de la BNF, perçu comme un progrès formidable. C'est le langage Algol qui en a bénéficié en premier : Backus et Naur se sont rendus compte que leurs grammaires respectives ne "collaient" pas à propos d'Algol58. Cela a donné une formalisation de Algo60. Un peu plus tard, Niklaus Wirth, l'inventeur de Pascal (notamment) a proposé une amélioration de la BNF. Dans tous les cas, l'écriture de ces règles respecte elle-même une grammaire : une grammaire des grammaires, ou `Meta-grammaire`. Nous allons donc utiliser un langage pour décrtire un langage !

3.1.1 BNF : Backus-Naur form

La BNF utilise les notations suivantes :

- Les symboles non-terminaux sont inserés entre chevrons < et >
- Les règles sont écrites comme "X : := Y"

  - X est la partie gauche de la règle et ne peut être qu'un non-terminal.
  - Y est la partie droite de la règle et peut être :
    * un terminal (lexème)
    * un non-terminal (une autre règle)
    * la concaténation de terminaux et non-terminaux

    ...éventuellement séparés par un symbole d'alternative
    |

- la notation ϵ indique une chaîne de caractère *vide* (longueur 0).

Exemple 1 Cet exemple montre qu'il est possible d'utiliser le même non-terminal à droite et à gauche du symbole de définition.

⟨*S*⟩ : := 'a' <S> | 'a'

Exemple 2 La grammaire des expressions arithmétiques n'est pas triviale. Il est nécessaire de penser à l'associativité des opérateurs : on parlera de *précédence*. L'exemple (partiel) montre comment enchaîner les règles, de manière à reconnaître correctement les expressions, termes et facteurs.

⟨*expr*⟩ : := <term> '+' <expr>
| <term>

⟨*term*⟩ : := <factor> '*' <term>
| <factor>

⟨*factor*⟩ : := '(' <expr> ')'
| <const>

⟨*const*⟩ : := integer

Exemple 3 L'exemple suivant décrit la grammaire d'une suite d'instructions (for, assignation, bloc), telles qu'on les retrouve dans le langage C.

⟨*statement*⟩ : := <ident> '=' <expr>
| 'for' <ident> '=' <expr> 'to' <expr> 'do'
<statement>
| '' <stat-list> ''
| <empty> <stat-list> : := <statement>
';' <stat-list> | <statement>

Exemple 3 : BNF en BNF Il est également curieux de noter que cette meta-grammaire BNF peut s'exprimer en elle-même !

⟨*syntax*⟩ : := <rule>
| <rule> <syntax>

⟨*rule*⟩ : := <opt-whitespace> '< <rule-name> '>' <opt-whitespace> ' : :=' <opt-whitespace> <expression> <line-end>

⟨*opt-whitespace*⟩ : := ' ' <opt-whitespace>
| ""

⟨*expression*⟩ : := <list>
| <list> <opt-whitespace> '|' <opt-whitespace> <expression>

⟨*line-end*⟩ : := <opt-whitespace> <EOL>
| <line-end> <line-end>

⟨*list*⟩ : := <term>
| <term> <opt-whitespace> <list>

⟨*term*⟩ : := <literal>
| '<' <rule-name> '>'

⟨*literal*⟩ : := '" <text1> '"
| '' <text2> ''

⟨*text1*⟩ : := "
| <character1> <text1>

⟨*text2*⟩ : := "
| <character2> <text2>

⟨*character*⟩ : := <letter>
| <digit>
| <symbol> character1> : := <character>
| '' <character2> : := <character>
| '" <rule-name> : := <letter>
| <rule-name> <rule-char> <rule-char> : := <letter>
| <digit>
| '-'

3.1.2 EBNF : extented Backus-Naur form

L'extension proposée par Wirth vous paraîtra naturelle : il s'inspire en effet des expressions régulières (utilisées lors de l'analyse lexicale) pour enrichir le pouvoir d'expression de la BNF. Du moins, en apparence...En réalité, l'EBNF ne fait que **faciliter** l'écriture car tout ce qui s'exprime en EBNF peut effectivement s'écrire en BNF : BNF et EBNF possèdent le *même pouvoir d'expression*. Pour rappel, on retrouve :

- L'étoile de Kleen : $*$, qui indique 0 ou plus d'occurences.
- La croix de Kleen : $+$, qui indique 1 ou plus d'occurences.

- ? qui indique 0 ou 1 occurence. Il est fréquent que les symboles crochets [...] soient utilisés.
- L'utilisation des parenthèses pour marquer un groupe.
- etc...

On doit souligner que malgré la simplicité de ces meta-langages, aucune norme ne s'est imposée. On trouvera ici et là des variations syntaxiques (ce qui est presque un comble !).

Exemple 3 : Bien que nous parlons d'analyse syntaxique, cette technique de description peut s'appliquer à l'expression d'éléments a priori plutôt d'ordre lexical. On donne ici l'exemple des nombres flottants (pour un langage imaginaire)

⟨*real*⟩ : := ⟨*integer*⟩ ('.' ⟨*integer*⟩) ? ('e'[+-] ⟨*integer*⟩) ? ⟨*integer*⟩ : := ⟨*digit*⟩+

Exemple 4 : grammaire EBNF pour Lisp Le langage Lisp, dû à Mac Carthy, est un langage extrêmmement intéresant : sa syntaxe est épurée et repose uniquement sur l'imbrication des parenthèses. Nous reviendrons au chapitre suivant, pour mieux en percevoir la pureté, la pertinence et l'utilité dans le domaine de la compilation. A titre s'exemple, nous donnons le calcul d'un factorielle par une méthode récursive, écrite en Lisp.

```
(defun factorial (n)
  (if (= n 0) 1
    (* n (factorial (- n 1)))))
```

Sa grammaire tient en seulement quelques lignes d'EBNF. A contrario, des langages comme C ou C++ nécessitent de nombreuses pages !

```
s_expression ::= atomic_symbol
      | "(" s_expression "." s_expression ")"
      | list
list ::= "(" s_expression* ")"
atomic_symbol ::= letter atom_part
atom_part ::= empty
            | letter atom_part
            | number atom_part
letter ::= "a" | "b" | " ..." | "z"
number ::= "1" | "2" | " ..." | "9"
```

FIGURE 3.1: Grammaire EBNF du langage Lisp

3.1.3 Diagrammes de Conway

Conway a proposé une alternative graphique intéressante, afin de visualiser rapidement les enchaînements admis par un langage. On donne sur la figure 3.2 l'exemple d'une telle représentation : pour comprendre les possibilité d'imbrication des règles, il suffit de suivre les parcours du graphe, en commençant à gauche. Vous trouverez sur le wed des outils qui permettent de générer de telles représentations, et qui facilitent le partage d'information à propos d'un langage : en cours de création, tutoriels, etc.

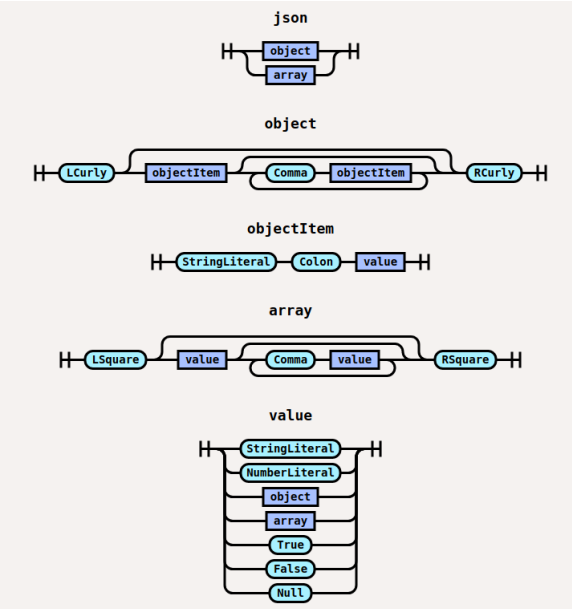


FIGURE 3.2: Diagramme de Conway pour le langage (de sérialisation de données) Json

3.1.4 Grammaires *Context-free*

Les grammaires dont nous parlons sont qualifiées d'*indépendantes du contexte*, ou *context-free*. On rappelle que toutes les règles dite *de production* que nous avons exprimées jusqu'ici sont de la forme $X \to \alpha$. Ces grammaires sont non-contextuelles (on dit aussi *hors-contexte* ou *algébriques*), car nous n'avons pas de précautions particulières à prendre lors de l'application de ces règles : ces règles s'appliquent toujours, de manière mécanique. Il n'y a pas de cas particulier, liè à un contexte d'utilisation. On pourrait en effet imaginer le contraire : telle production ne s'appliquerait que soumise à une condition particlière, liée au contexte, c'est-à-dire à l'endroit (ou au moment) où on souhaite l'appliquer. Par exemple, on peut imaginer que selon l'endroit du texte où l'on se trouve, un même symbole non terminal ne se "déclinerait" pas de la même manière. Wikipedia explique que par opposition est contextuelle une règle de la forme $c\ + X \to c + \alpha$, en raison de la partie gauche de la règle qui stipule un contexte pour X. Une telle règle signifie que X, dans le cas (contexte) où il est précédé du symbole terminal c c et du littéral +, peut être remplacé par α (et dans ce cas seulement).

Fort heureusement, dans la pratique, les grammaires sont généralement non-contextuelles. Les cas contraires existent bien entendu, et il faudra beaucoup de précautions pour les traiter. Dans VHDL par exemple, il existe une difficulté concernant la règle d'appel de procédure : dans VHDL une procédure qui n'a pas d'argument doit s'appeler sans parenthèses (ce qui peut se concevoir). La difficulté est alors d'analyser syntaxiquement un tel "nom", qui apparait par exemple dans le corps d'un processus : si dans le contexte, une procédure n'a pas été déclarée avec le même nom, alors il s'agit d'une erreur de syntaxe (une assignation sans partie droite par exemple), sinon, il s'agit bien dudit appel de procédure. Ici, rien dans la grammaire elle-même ne permet de décider du caractère correct ou incorrect du code source. Il faut effectivement se souvenir du contexte issu du passé de l'analyse, pour continuer l'analyse.

3.2 Analyse récursive descendante : analyse LL(k)

Il est temps de passer à la pratique ! On rappelle que l'analyse consiste à consommer les lexèmes, jusqu'au dernier, en respect des règles.La première méthode pour construire un *parseur* consiste à "partir par le haut", c'est-à-dire à envisager la règle "racine", qui explicite la forme globale que doit avoir le code source. On disposera d'une methode racine (par exemple **parse_root**), qui démarre le parsing.

Analyse lookahead $k = 1$ Par quel lexème doit commencer ce texte ? Cela peut-être très direct : par exemple, si un programme doit commencer par le mot clé "program", on doit s'assurer que notre source commence bien par ce lexème ! On doit donc se munir d'une méthode qui lit et vérifie que le premier lexème est bien du bon type. Cette méthode s'appelle généralement "expect(kind)" où le "kind" est le type du lexème. Si le bon lexème se présente en tête, cette méthode *consomme* le lexème et pointe sur le suivant dans le flux de lexèmes. Sinon, nous sommes manifestement tombés sur une erreur de syntaxe, qu'il faut s'empresser d'indiquer à l'utilisateur. Nous disposons de la ligne et de la colonne (pour l'instant 1,1 !) où l'on se situe et l'erreur sera facilement utilisable par notre programmeur.

En réalité trois méthodes nous seront utiles :

- **expect** : consomme un lexème d'un type attendu.
- **showNext** : retourne le lexème courant.
- **acceptIt** : consomme le lexème courant.

Le parser est qualifié de **prédictif** dans le sens où il ne fait que vérifier que le lexème suivant est bien celui auquel il s'attend.

Un code Ruby est proposé comme illustration de ces trois méthodes. Elles seront facilement adaptées à votre langage préféré, et incluses dans vos parsers.

```ruby
def expect expected_kind
  token=showNext
  if (actual_kind=token.kind)==expected_kind
    acceptIt
  else
    msg="Syntax error line #{token.pos.line}."
    msg+"expecting #{expected_kind}. Got #{actual_kind}."
    raise msg
  end
end

def showNext
  @tokens.first
end

def acceptIt
  @tokens.shift #consum first token
end
```

S'il n'y pas d'erreur on peut observer le lexème suivant, etc. Toutefois, très rapidement, la grammaire nous indiquera non pas un lexème particulier, mais un non-terminal **X** : il faudra alors simplement appeler une méthode **parse_X**, dans laquelle nous avons préalablement codé, sereinement, la règle afférente. Le procédure semble pouvoir se dérouler **récursivement** sans

encombre : soit on vérifie et consomme le lexème, soit on fait appel à une "sous-règle" codée dans une fonction (ou méthode). Cette récursivité d'appels de fonctions est qualifée de "mutuellement récursif".

Malheureusement, ce n'est pas si simple. Notre règle EBNF **Y** peut présenter des **alternatives** : **X_1 ou X_2 ou ... ou X_n**. Comment savoir quelle fonction de parsing lancer ? Il n'a pas d'autres possibilités que, dans la fonction **parse_Y**, de préalablement identifier le (ou les !) lexèmes qui doit (doivent) commencer forcément chacune des règles. Cette tâche est non-triviale car elle nécessite de lister (au préalable) ces **starters(X_i)** *récursivement*. Dès lors que ces starters sont connus (et différents les uns des autres), un pseudo code peut ressembler au code suivant.

```
def parse_y
  #....
  if showNext.included_in? STARTERS[x1]
    parse_x1()
  elsif showNext.included_in? STARTERS[x2]
    parse_x2()
  #....etc...
  elsif showNext.included_in? STARTERS[xn]
    parse_xn()
  end
  #...
end
```

Cette méthode dite **LL(1)** fonctionne dans bien des cas, et elle est simple à programmer. Elle peut se résumer en : on observe le type du lexème qui se présente, et on décide si d'une manière ou d'une autre, on peut le consommer. Cette méthode possède un autre avantage : en plus d'être intelligible et programmable par un humain, elle est très rapide.

Lookahead $k > 1$ Malheureusement, elle n'est parfois pas suffisante pour traiter certaines grammaires, y compris très courantes. Que se passe-t'il dans le cas de l'analyse d'instructions ("statements") de code C, lorsque le prochain lexème est un identifiant ? Dois-je lancer un méthode **parse_assignment**, une méthode **parse_func_call**, voire une méthode **parse_labelled_stmt** ? Là encore, la chose semble atteignable (et elle l'est !) : il suffit de regarder non pas seulement le token courant (identifier), mais également le lexème (encore) suivant. Nous avons basculé dans l'analyse dite **LL(2)**. Nos "starters" ne suffisent plus : il faudrait regarder les couples des deux prochains lexèmes...Evidemment, pour d'autres classes de grammaires, il faut aller encore plus loin et envisager une valeur de k très supérieure à 2. Les choses se compliquent...

Discussion Dans la pratique, la plupart des grammaires peuvent se formuler ou reformuler sous la forme **LL(1)**, et quelques règles nécessitent un lookahead supérieur (souvent 2). Au delà, le parsing doit être réalisé par un générateur, qui utilisera d'autres techniques que l'analyse descendante. Pendant longtemps, ce recours aux générateurs a été un passage obligé et systématique. L'histoire récente montre un certain revirement : les parsers des compilateurs majeurs comme GCC ou Clang ont été réécris à la main, et non pas générés. Parmi les raisons évoquées, on souligne notamment le reporting d'erreurs : il semble plus aisé de faire remonter à l'utilisateur des erreurs pertinentes. De plus, les outils envisagés arrivent avec leurs propres cohortes de difficultés, qui éloignent souvent le programmeur (de compilateur) d'un but relativement simple. Enfin, et c'est un argument "massue", la rapidité des codes écrits à la main ne se dément toujours pas.

J'ajoute une note personnelles : il existe une vraie satisfaction à écrire de tels parseurs (quoiqu'en dise certains détracteurs de l'approche manuelle, rarement des programmeurs) !

3.2.1 Parseur LL(1) d'expressions

Les expressions arithmétiques et logiques interviennent partout dans les langages de programmation classiques : conditions des structures de contrôle et calculs.Malgré leur simplicité apparente, il faut s'en médier !

On observera de près le code Ruby suivant, qui exprime les règles EBNF en LL(1). On

```
ADDITIV_OP  =[:add,:sub, :or]
def parse_expression
  parse_multiplicative
  while showNext.is_a?(ADDITIV_OP)
    acceptIt
    parse_multiplicative
  end
  return t1
end

MULTITIV_OP=[:mul,:div,:and]
def parse_multiplicative
  parse_term
  while showNext.is_a?(MULTITIV_OP)
    acceptIt
    parse_term
  end
end
```

Il est possible d'étendre ce parseur afin d'y inclure les comparaisons.

3.3 Analyse ascendante

Il existe d'autres manières de procéder, qui s'appuient par une analyse ascendante. Dans ce cas, l'analyseur reconnaît d'abord les constructions grammaticales de bas niveaux, pour progressivement remonter les règles de grammaires. Cette manière est moins intuitive que la précédente, mais couvre une plus grande gamme de types de grammaires : LR(k) parseurs et ses variantes (SLR, LALR, etc). De même que les méthodes descendantes LL(k), les parseurs LR(k) sont rapides car ils ne nécessitent pas de retour en arrière (backtracking).

La méthode fonctionne comme suit : en régime permanent, le parseur a accumulé un certain nombre de sous-arbres. Toutefois, ces sous-arbres ne sont pas encore reliés les uns aux autres. Il ne pourra le faire que lorsqu'il aura atteint la partie (droite) le lui permettant. Ces osous-arbres sont placés sur une pile.

Algorithme Shift-reduce Un parseur LR fonctionne de la gauche vers la droite, en appliquant deux actions appelés **shift** et **reduce**.

- L'action Shift fait avancer le pointeur d'un lexème vers la droite. Ce seul lexèmes devient un sous-arbre à part entière.
- L'action Reduce applique une règle de grammaire complète sur les sous-arbres précédemment établis et crée un nouvel arbre plus conséquent.

Ce procédé se répète jusqu'à consommation de l'intégralité du flux de lexèmes.

3.4 Outils de constructions de parseurs.

Il existe pléthores d'outils qui aident à la construction de parseurs. Dans le monde Ruby, on retrouve deux exemples emblématiques : Racc et Treetop. Le premier fait partie de la bibliothèque standard du langage (ce qui est un gage de pérénnité), tandis que le second, plus expérimental, fait partie de la classes de parseurs "PEG" (parsing expression grammar), très à la mode il y a une dizaine d'années.

Racc On donne ici la grammaire (simplifiée) d'un parseur Json. Comme pour l'analyse lexicale, il est possible (et souhaitable) d'ajouter des actions pour chacune des règles. Cet exemple ne les expose pas.

```
class JsonParser
  token STRING NUMBER TRUE FALSE NULL
  rule
    document
      : object
      | array
      ;
    object
      : '{' '}'
      | '{' pairs '}'
      ;
    pairs
      : pairs ',' pair
      | pair
      ;
    pair : string ':' value ;
    array
      : '[' ']'
      | '[' values ']'
      ;
    values
      : values ',' value
      | value
      ;
    value
      : string
      | NUMBER
      | object
      | array
      | TRUE
      | FALSE
      | NULL
      ;
    string : STRING ;
  end
end
```

Treetop les grammaires Treetop présentent d'emblée une apparence plus proche de l'analyse recursive descendante.

```
grammar SimpleHTML
  rule document
    (text / tag)* {
      def content
        elements.map{ |e| e.content }
      end
    }
  end

  rule text
    [^<]+ {
      def content
        [:text, text_value]
      end
    }
  end

  rule tag
    "<" [^>]+ ">" {
      def content
        [:tag, text_value]
      end
    }
  end
end
```

3.5 Classes de grammaires. Grammaires ambigües

Le schéma suivant (3.3, extrait de [**?**]) permet de situer le véritable pouvoir expressif des grammaires entre elles. Comme nous l'avons évoqué, on constate d'emblée que les grammaires de type LL(k) ne représentent qu'un sous-ensemble des grammaires envisageables. Comme nous l'avons exprimé ici, cela ne leur enlève en rien leur attrait, bien au contraire !

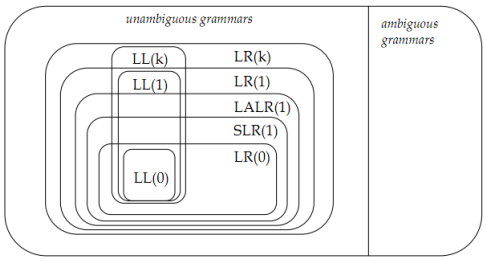


FIGURE 3.3: Classes de grammaires.

Le schéma présente également sur la droite un ensemble que nous n'avons pas abordé : celui des grammaires dites *ambigues*. Ce sont des grammaires dont la tentative d'application, en terme de reconnaissance d'un langage, conduit à plusieurs cas différents de structures. L'exemple le plus connu est celui du "dangling else", illustré (en partie) ci-dessous.

⟨*statement*⟩ : := 'if' <condition> 'then' <statement>
| 'if' <condition> 'then' <statement> 'else' <statement>

⟨*condition*⟩ : := <etc>

Le problème dans ce cas réside dans l'appartenance du bloc de code associé au "else" : selon comment on interprète la grammaire, ce bloc peut être relié au premier "if", ou au second ! Par exemple :

```
if a then if b then s else s2
```

peut être interprété comme :

```
if a then
  begin
   if b then s
  end
else s2
```

ou comme :

```
if a then
  begin
    if b then
      s
    else
     s2
  end
```

Malheureusement, les grammaires des langages parlés par les humains regorgent de telles ambiguités, liées aux regroupements de termes. Par exemple, dans la phrase *Avez-vous vu le collier du chien que Gustave a acheté hier ?*, il est impossible de savoir ce qui a été acheté hier : le chien, ou le collier ? Etc.

3.6 Conclusion et discussion

Dans ce chapitre consacré à l'analyse syntaxique, nous avons insisté sur l'apport fondamental de Backus, qui a proposé un premier moyen formel de décrire des grammaires. Comme on l'a suggéré, il existe malheuresement pléthore de manières de réaliser un outil effectif de parsing de ces grammaires. Ce tient notamment à la très grande variété de types de grammaires. Certaines grammaires sont très compliquées à analyser mécaniquement (et probablement humainement aussi). Il est de coutûme de dire que le langage C est très difficile à parser, et que concernant C++, c'est même impossible ! Dans tous les cas, l'étude des grammaires et des algorithmes reste, en 2020, un sujet de Recherche complexe. Différentes vagues et courants apparaissent ici et là, qui clament avoir résolu ce problème épineux et central.

Pour se sortir des difficultés pratiques de construction de parseur, on ne saurait trop recommander de recourir, autant que faire se peut, à des grammaires simples et bien pensées, qui se prêtent des analyses récursives descendantes. Mon expérience personnelle, sur des langages mainstream m'a progressivement amené à abandonner l'idée de génération totalement automatique, et à reposer sur une programmation adéquate de leurs parseurs. Je ne suis pas le seul à suivre cette tendance, que l'on retrouve dans GCC et beaucoup d'autres compilateur.

Chapitre 4

Arbre de syntaxe abstraite

4.1 Introduction

Ce chapitre constitue la suite du chapitre précédent intitulé "analyse syntaxique" : alors que ce dernier nous a permis de prendre connaissance de la notion de grammaire, et de nous initier à l'implémentation d'un parser, nous allons désormais nous intéresser à l'élaboration d'un structure de données fondamentale pour un compilateur : l'arbre de syntaxe abstraite ou AST. Cet AST sera embarqué dans nos parseurs.

La syntaxe abstraite est importante pour les Informaticiens : elle est à l'Informaticien ce que l'équation est au Mathématicien. C'est le moyen informatique de capturer des concepts, ainsi que les relations qu'ils entretiennent. Notamment, les raisonnements, preuves, explorations, transformations de programmes informatiques se feront sur la syntaxe abstraite, et non sur la syntaxe concrète.

4.2 Syntaxe programmative

L'idée centrale de la syntaxe abstraite est qu'il est possible de décrire les éléments de syntaxe à l'aide de la programmation.

4.2.1 A l'aide d'un langage interprété orienté-objet : Ruby

Par exemple, l'équation $y = a * x + b$ peut se décrire à l'aide d'objets savamment agencés, en langage de programmation orientés-objets.

```
class Equation
  attr_accessor :lhs,:rhs
  def initialize lhs,rhs
    @lhs,@rhs=lhs,rhs
  end
end

class Binary
  attr_accessor :lhs,:op,:rhs
  def initialize lhs,rhs
    @lhs,@rhs=lhs,op,rhs
  end
```

```
    end

    class Id
      attr_accessor :str
    end

    class Op
      attr_accessor :symb_str
    end

    ast=Equation.new(Id.new("y"),
          Binary.new(
            Binary.new(
              Id.new("a"),
              Op.new("*"),
              Id.new("x")
            ),
            Op.new("+"),
            Id.new("b")
          )
```

Notons qu'en Ruby, la description des classes peut se faire de manière encore plus concise :

```
Equation=Struct.new(:lhs,:rhs)
Binary=Struct.new(:lhs,:op,:rhs)
Id=Struct.new(:str)
Op=Struct.new(:symb_str)

ast=Equation.new(Id.new("y"),
          Binary.new(
            Binary.new(
              Id.new("a"),
              Op.new("*"),
              Id.new("x")
            ),
            Op.new("+"),
            Id.new("b")
          )
```

Toutefois, dans ce cours, nous ne retiendrons pas cette manière de procéder, car elle empêche la forme d'héritage traditionnelle (différentes techniques d'héritage par inclusion (*mixins* permettrait cela, mais cela sort du cadre de la compilation).

4.2.2 A l'aide de langages fonctionnels : Lisp et Ocaml

Les langages fonctionnels comme Caml ou Lisp permettent une manipulation encore plus naturelle de la syntaxe abstraite des langages. Par exemple, dans le cas de Lisp, l'exemple précédent devient (on omet la définition des fonctions de définition) :

```
(ast (eq y (+ (* a x ) b))
```

En Caml (ou OCaml), on dispose de types dit algébriques ou **types sommes**.

```
type op = ADD | SUB | MUL | DIV;;

type id = Id of string;;

type expression =
   Id of string
 | Binary of expression * op* expression;;

type ast = Equation of id * expression  ;;

Equation(Id("y"),Binary(Binary(Id("a"),MUL,Id("b")),ADD,Id("x")))
```

4.3 Construction de l'AST

Nous rappelons sur la figure suivante (fig. 4.1) le but poursuivi ici concernant le parseur : outre la vérification de la conformité d'un code source au regard de la grammaire, le parseur transforme le flux initial de lexèmes en notre AST. Nous continuons ainsi notre structuration progressive du code source, qui avait débuté par de simples caractères ASCII.

- On peut noter que les lexèmes (en rouge) se retrouvent dans les feuilles de l'AST.

- Notez également que certains caractères (comme les parenthèses, accolades et plus généralement les signes de ponctuation) n'apparaissent plus du tout dans l'AST. La construction de l'AST supprime donc certains éléments, qui sont "sous-entendus" par la structure-même de l'arbre. Par exemple, il est aisé de retrouver la liste d'arguments formels de la fonction précédente dans l'arbre, bien que les virgules initialement dans le source n'apparaissent plus dans l'AST.

- Les noeuds possèdent un nom générique, ainsi que les arcs reliants les noeuds.

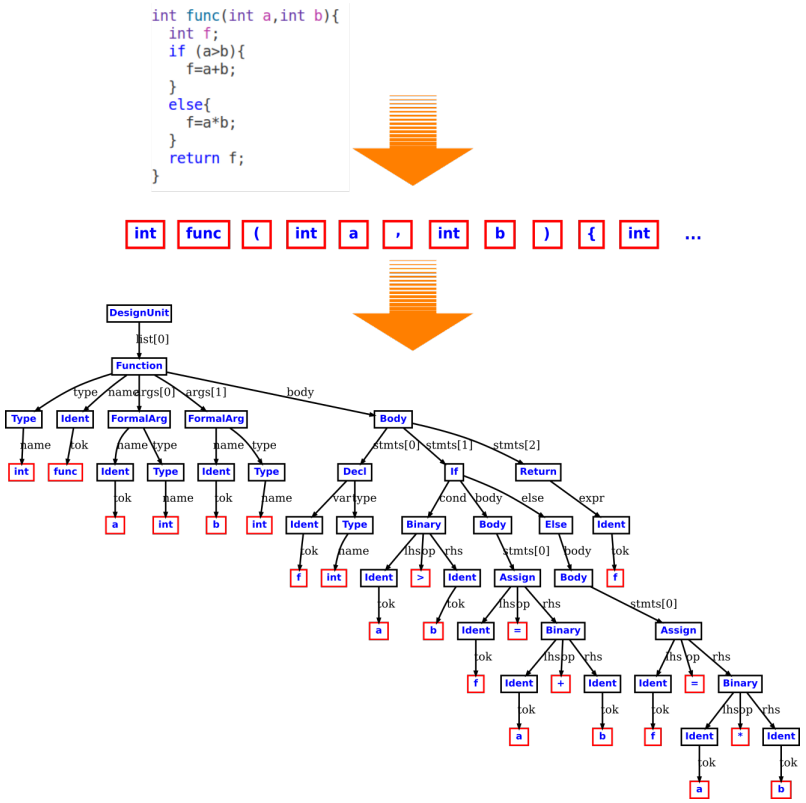


FIGURE 4.1: Des lexèmes à l'Arbre de Syntaxe Abstraite

4.3.1 Représentation objet des noeuds de l'AST

Chacun des noeuds l'arbre se structure avantageusement à l'aide du paradigme orienté-objet.

- Le nom générique de chaque noeud conduit à l'écriture d'une **classe**. On pourra instancier cette classe autant de fois que voulu, pour créer autant d'**objets** (ou "instances"). A ce titre, notre parseur peut générer énormément d'objets : énormément plus (et plus vite) que vos premiers programmes orientés objets.
- Le nom figurant sur les arcs reliant les noeuds sont en réalité le nom des **attributs** de l'objet, qu'on nomme également **variable d'instances**.
- Le **nommage** de ces classes et attributs est très important quant à l'*intelligbilité* de la syntaxe abstraite.

Diagramme de classe Le choix de ces objets conduit à un "paquet de classes", qui peut être avantageusement organisé de manière plus cohérente, sous la forme d'un **diagramme de classes** : il permet entre autre de préciser

- les **relations d'héritage** entre classes. Certaines classes abstaites mais structurantes pourront d'ailleurs être crées à ce titre : par exemple la notion de "statements" mérite probablement la création d'une classe spécifique, dont hériteront les classes "Return" ou "ForLoop", etc.
- les **relations de composition** (ou *containment*) : indiquées par un losange plein près de le classe "possédante" et d'une flèche près de la classe "possédée". Une multiplicité apparait sur l'arc séparant les deux classes. Par exemple "1..*" signifie : entre 1 et une infinité.

Métamodèle Un exemple d'un tel diagrame de classe (partiel) correspondant à l'AST de la figure 4.1 est donné sur le schéma 4.2. Il ne faudra pas confondre ce schéma avec l'AST lui-même. L'AST représente un modèle instancié, à partir du diagramme de classe, qui explicite les relations possibles entre les classes.

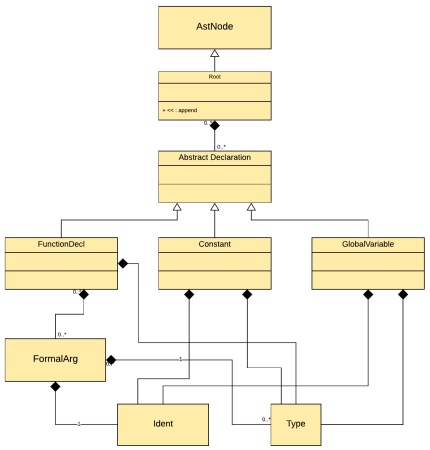


FIGURE 4.2: Metamodèle (partiel) de l'AST de la figure 4.1

L'exemple du nouveau diagramme suivant (fig 4.3 est également intéressant. Il représente la syntaxe abstraite d'un programme "inhabituel" : la description d'un immeuble. Il est inhabituel à plusieurs titres et notamment parce que nous somme habitués à penser en terme de langages exécutables, et qu'il s'agit ici d'un langage de description. Par ailleurs, il est intéressant de souligner que la présentation d'un tel diagramme ne semble en rien suggèrer la notion de "langage" informatique : en réalité, les informaticiens considèrent que l'implication est bidirectionnelle : un langage implique un diagramme de classes, mais également : un diagramme de classe implique un langage. Pour les informaticiens, un tel diagramme *est* la description d'un langage. Il n'est ici plus associé à une grammaire, comme proposé au chapitre précédent : nous sommes en face de sa syntaxe abstraite pure. Les Informaticiens de la modélisation appellent précisément cette syntaxe abstraite le "métamodèle".

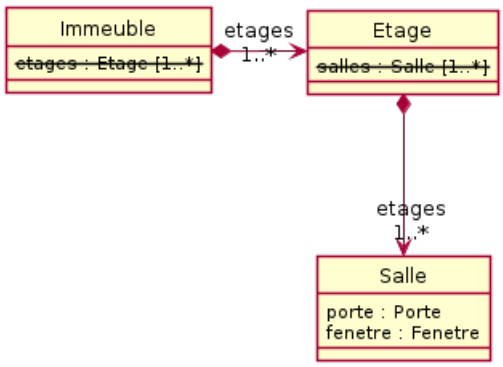


FIGURE 4.3: Syntaxe Abstraite et diagramme de classe : relation de composition. On notera que formellement, UML propose de ne plus indiquer les attributs (barrés ici) dans la classe elle-même, dès lors qu'ils sont indiqués sur les arcs reliant les classes.

4.4 Passage à la pratique : AST et parse tree

Principe Il est grand temps de passer à la pratique ! Si l'on prend une **grammaire** simple, composée d'un ensemble de règles Y_i, on peut considérer en première approche que :

- **parsing** : chacune des règles Y_i conduit à l'écriture d'une méthode $parseY_i$.
- **ast** : chaque méthode $parseY_i$ émet en retour un objet de classe Y_i.
- les appels récursifs de chaque méthode construisent des "sous-objets", qui viennent enrichir les attributs de chaque objet en cours de construction.

Parse tree ou AST ? En réalité, l'application stricte de cette manière de procéder ne conduit pas un AST, mais à un **parse tree**. La différence n'est parfois pas soulignée dans les livres d'introduction, mais elle me parait importante : comme on vient de l'expliquer, le parse tree est le reflet direct de la structure de la grammaire. Or, cette grammaire peut être formulée de manière plus on moins habile ou plus ou moins structurée : par exemple, le concepteur de la grammaire a eu recours à un très grand nombre de règles, afin de gagner en clarté. Certaines de ces règles ne sont donc que des *artifices de présentation de la grammaire*.

On dérogera donc aux principes précédents, dès lors que cela parait pertinent.

Exemple : parse tree vs AST Cet exemple est repris du site Stack Overflow, où la question se pose fréquemment. On propose ici unde description de la grammaire avec l'outil de génération de parser, appelé ANTLR. La grammaire décrit un ensemble d'assignations et d'expressions.

Le parse tree généré par le parseur (lui-même généré par ANTLR), ainsi qu'un AST sont représentés sur la figure suivante, à partir d'un code source constitué de 3 assignations très simples. L'application stricte de nos règles de construction précédente conduit à un parse tree très complexe. A droite, à l'inverse, on représente ce que l'on est en droit d'attendre d'un AST : c'est une version complète et la plus simple possible de ces mêmes expressions.

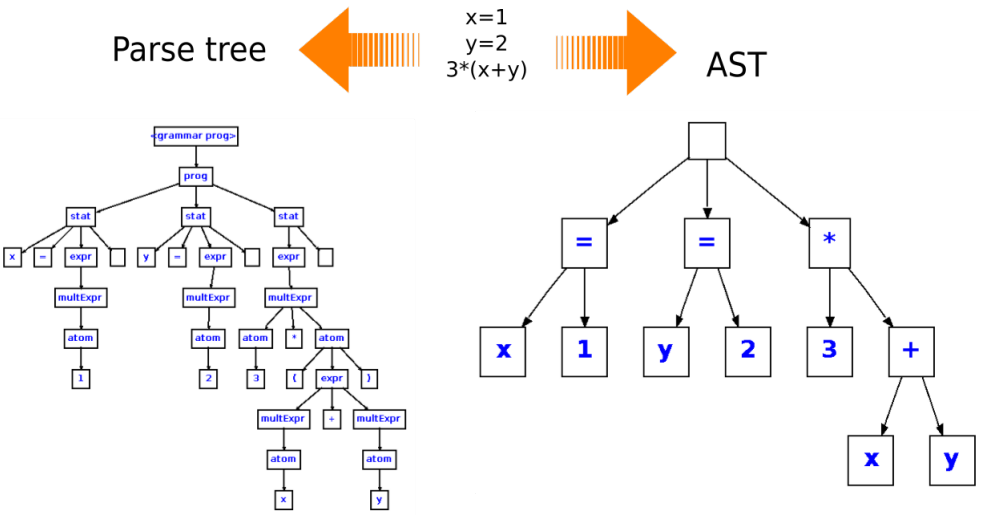


FIGURE 4.4: Parsee tree versus AST : l'AST conduit à une représentation plus simple et épurée, alors que le Parse Tree reflète strictement la structure de la grammaire, à travers les objets associés.

```
grammar Expr;
options
{
    output=AST;
    ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}

prog    :   ( stat )+ ;

stat    :   expr NEWLINE
        |   ID '=' expr NEWLINE
        |   NEWLINE
        ;

expr    :   multExpr (( '+'^ | '-'^ ) multExpr)*
        ;

multExpr
        :   atom ('*'^ atom)*
        ;

atom    :   INT
        |   ID
        |   '('! expr ')'!
        ;

ID      : ('a'..'z' | 'A'..'Z' )+ ;
INT     : '0'..'9'+ ;
NEWLINE : '\r'? '\n' ;
WS      : ( ' ' | '\t' )+ { skip(); } ;
```

4.5 Exemple en Ruby

Méthode de parsing Nous donnons ici deux exemples : l'un concerne le parsing d'instructions simples, et l'autre de code celui d'une boucle "while" : chacune des fonction émet la partie de l'arbre syntaxique correspondant. Dans le cas du "While", on constate notamment que la fonction retourne effectivement une instance de la classe "While", qui possède visiblement deux attributs, dont la valeur est passée dans le constructeur.

```ruby
def parse_stmt
  case showNext.kind
  when :ident
    ret=parse_assignment
  when :while
    ret=parse_while
  when :for
    ret=parse_for
  else
    line=showNext.pos.first
    raise "syntax error line #{line}"
  end
  return ret
end
```

```ruby
  def parse_while
    expect :while
    cond=expression()
    stmt=statement() #peut retourner un Body.
    return While.new(cond,stmt)
  end
```

Le code de la classe est également donné. On note la hiérarchie des classes (le symbole $<$ dénote l'héritage). Généralement, il est souhaitable de définir ces classes dans un fichier à part ou en tous les cas dans un espace de nommage adéquat : ici, en Ruby, nous avons défini ces classes dans un module Ruby.

Soulignons également que dans le cas d'un langage typé dynamiquement (c'est de le de Ruby, dont le typage dynamique est –au passage– fort), le type des attributs n'est évidemment pas déclaré [1]. Pour les plus curieux, je précise que "attr_accessor" est une méthode de classe (et non d'instance), qui crée dynamiquement des *accesseurs* : en l'occurence ici deux méthodes d'instance qui permettent l'accès en lecture ou en écriture des attributs.

Classe parseur Pour rendre la chose plus concrète, je vous laisse un exemple de code Ruby du parseur lui-même. Ce parseur est lui-même à insérer dans un objet "compilateur". Quelques remarques s'imposent :

- On retrouve les méthodes habituelles, vues au chapitre précédent, qui permettent l'inspection (showNext) ou la consommation des lexèmes (acceptIt, expect). On note la présence d'une nouvelle méthode "maybe", qui consomme si le bon lexème est présent, mais ne retourne

[1] Certaines bibliothèques Ruby permettent de renforcer ce typage lors de la déclaration, si cela est nécessaire.

```ruby
module MiniC
  class AstNode
    # skipped
  end

  class Stmt < AstNode
  end

  class CtrlStmt < Stmt
  end

  class For < CtrlStmt
    attr_accessor :init,:cond,:increment,:body
    def initialize init=[],cond=nil,increment=nil,body=nil
      @init,@cond,@increment,@body=init,cond,increment,body
    end
  end

  class While < CtrlStmt
    attr_accessor :cond,:body
    def initialize cond,body
      @cond,@body=cond,body
    end
  end

  # ...etc
end #module
```

pas d'erreur si ce lexème est absent. Cela sert évidemment à consommer (ou non) un lexème optionnel dans un grammaire. Il en existe beaucoup, par exemple, dans le langage VHDL !

- Ce parseur instancie un lexeur au lancement de la méthode "parse", et lui demande de retourner l'ensemble des lexèmes. Cette méthode dite de "slurping", qui dissocie totalement l'analyse lexicale et syntaxique en deux phases distinctes, a longtemps été décriée, car elle nécessite plus de mémoire : une autre méthode consisterait à demander les lexèmes au fur-et-à-mesure de l'analyse syntaxique. Techniquement, cela n'est pas beaucoup plus difficile. Je préfère personnellement la méthode proposée, car elle facilite notamment le debug, sur des machines qui n'ont a priori pas de fortes contraintes en terme de mémoire vive.

```
module MiniC
  class Parser
    attr_accessor :tokens

    def acceptIt
      tokens.shift
    end

    def maybe kind
      return acceptIt if showNext.is? kind
    end

    def expect kind
      if ((actual=tokens.shift).kind)!=kind
        puts "ERROR : Received '#{actual.val}' at #{actual.pos}"
        show_line(actual.pos)
        raise "Expecting '#{kind}'. "
      end
      return actual
    end

    def showNext(n=1)
      tokens[n-1] if tokens.any?
    end

    def lookahead(n=2)
      tokens[n] if tokens.any?
    end

    #............ parsing methods follow...........

end #module
```

4.6 Syntaxe abstraite versus syntaxe concrète

Sucre syntaxique On doit insister sur le fait que la grammaire des langages de programmation prend telle ou telle forme dans le seul but de rendre l'écriture d'un code plus aisée ou plus agréable :

```ruby
module MiniC
  class Parser
    #............ parsing methods ...........
    def parse code_str
      begin
        @tokens=Lexer.new.tokenize(str)
        @tokens=remove_comments()
        ast=design_unit()
      rescue Exception => e
        puts "PARSING ERROR : #{e}"
        puts "in C source at line/col #{showNext.pos}"
        puts e.backtrace
        abort
      end
    end

    def design_unit
      du=DesignUnit.new
      while tokens.any?
        case showNext.kind
        when :sharp
          case showNext(2).val
          when "include"
            du << include()
          when "define"
            du << define()
          end
        else
          du << declaration
          maybe :semicolon if tokens.any?
        end
      end
      du.list.flatten!
      return du
    end
    # other parsing methods
    # ....
   end
end #module
```

on parle souvent de langages qui présentent (ou non) un tel **sucre syntaxique**. Ce sucre vient "enrober" une substance plus centrale, qui est précisément la syntaxe abstraite. Plusieurs langages classiques pourraient par exemple se représenter avec la même syntaxe abstraite : les boucles "While" et les conditionnelles "If", les appels de fonctions etc diffèrent par leur syntaxe concrète, mais représentent fondamentalement les mêmes concepts.

Lisp et Smalltalk Les puristes aprécieront probablement une connexion directe entre la syntaxe concrète et la syntaxe abstraite. Les langages Lisp et Smalltalk sont frappants : ils réussissent la prouesse de présenter les mêmes concepts avancés que des langages plus récents (Java, Ruby, Python, etc), sans multiplier les éléments de syntaxe. Il est alors étonnant de constater qu'ils ne recoivent pas l'intérêt du plus grand nombre : la présence de sucre syntaxique semble avoir son importance dans l'adoption de tel ou tel langage, de la part d'un *utilisateur*. C'est une question de curseur et de dosage ! Gardez en tête toutefois que ces langages restent très actifs et attractifs ! Jettez-y un oeil ! Dans le chapitre précédent, la grammaire de Lisp a été décrite. Il est également de coutûme de dire que celle de Smalltalk tient sur carte postale ! Une telle carte postale est proposée sur la figure 4.5.

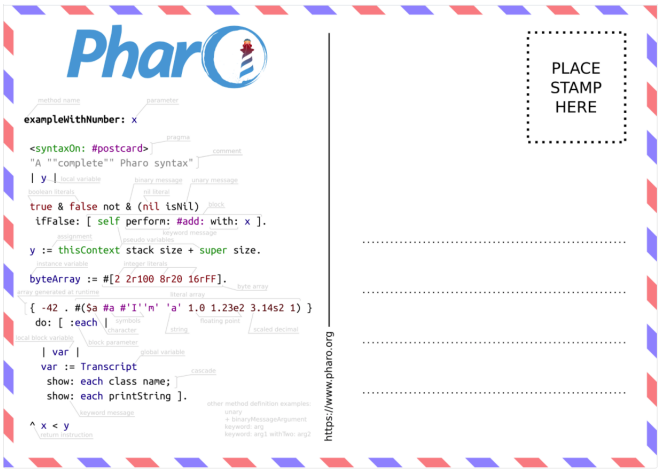


FIGURE 4.5: Comme pour LISP, la syntaxe de Smalltalk (version Pharo) tient sur une carte postale !

Hy : Python en Lisp La pureté de Lisp en terme de présentation de l'AST a poussé une communauté à proposer un nouveau dialecte de Lisp, *entièrement* basé sur Python : il s'agit de Hy (ou HyLang). Dans HyLang, on a débarrassé Python de sa syntaxe, pour ne retenir que son AST, présenté au programmeur sous forme de s-expressions. On peut ainsi programmer en Python, mais sans sa syntaxe concrète ! A priori, on peut réaliser cet exploit pour n'importe quel langage !

FIGURE 4.6: Hy (ou Hylang) est un Lisp basé sur l'AST de Python

4.7 Conclusion

Ce chapitre nous a permis de découvrir la seconde fonction d'un parseur : il est en charge de la création, en mémoire, d'une représentation du programme, sous une forme épurée, appelée Arbre de Syntax Abstraite. Cet objet est réalisé à l'aide d'une approche de programmation structurée : la programmation orientée objet nous a notamment facilité la chose. Nous verrons au chapitre suivant que ce même paradigme objet va nous permettre d'**exploiter** cette représentation en mémoire.

Chapitre 5

La pattern Visiteur

5.1 Introduction

L'élaboration de l'AST en mémoire constitue une phase importante dans la constitution d'un compilateur. Nous sommes désormais en mesure de l'exploiter, et de manipuler le langage. Nous devons nous munir des méthodes qui rendent possible cette manipulation. Le **pattern** (*motif*, en français) **visiteur** est à ce titre indispensable.

5.2 Principes du pattern Visiteur

Passes du compilateur Un véritable compilateur requiert la mise en place de plusieurs centaines d'algorithmes. Ils opèrent effectivement sur l'AST (ou sur une structure similaire que nous verrons plus tard). Le pattern visiteur va nous permettre dans tous les cas de nous reposer sur une technique éprouvée et robuste. A ce titre, soulignons que ce pattern fait partie d'une large palette de tels patterns ou, en français "motifs de conception" : rappelons qu'il s'agit de recettes éprouvées par plusieurs générations de programmeurs et rafinées au cours du temps. Ces patterns ont été recensés dans un livre désormais célèbre, écrit par 4 informaticiens : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Ils ont vite été surnommés "the gang of four". Le livre, publié en 1994, s'intitule "Design Patterns : Elements of Reusable Object-Oriented Software" [3].

Séparation algorithme/structure Le pattern consiste à séparer un algorithme de la structure sur laquelle il opère. Dans notre cas la structure est précisément un AST, mais on pourrait envisager d'autres structure (mais les informaticiens y verront probablement un langage, comme expliqué au chapitre précédent). On va pouvoir ajouter des traitements à la structure, sans la modifier intrinsèquement. Qu'est ce que cela signifie ? Cela signifie que malgré les nombreuses passes du compilateur, qui nécessitent a priori des calculs particuliers à chaque classe, les classes elles-memes **ne seront pas modifiées** !

Violation de l'encapsulation ? Le pattern visiteur –qui est issu de profondes réflexions semble parfois aller à l'encontre d'un principe orienté-objet fondamental : l'encapsulation. L'encapsulation signifie que l'objet est à même de réaliser un certain nombre de tâches, qui lui sont propres, de manière isolée. Chaque objet est seul responsable de délivrer ces services. L'encapsulation incite donc le programmeur à *enrichir* de la sorte *chaque objet*. Par exemple, à partir de notre AST, on peut chercher à réémettre le code source : c'est la notion de *pretty printer*. Il serait légitime que chacun de ces objets sache s'afficher ; toutes les classes devraient donc présenter une méthode "print". Si l'on cherche désormais

Toutefois, lorsque la structure est complexe et composite (ici un arbre constitué de noeuds de classes différentes), et que l'algorithme opère sur cet ensemble, le pattern trouve parfaitement sa légitimité.

5.3 Codage du Visiteur

Principes Les principes du codage d'un visiteur est le suivant :

- On adjoint une **unique méthode** `accept(visitor)` à chaque objet de notre AST. Il est possible de passer à cette méthodes des arguments, si on le souhaite : `accept(visitor,args)`. Mais pour simplifier la présentation, nous omettrons ces arguments dans un premier temps.
- On écrit une classe Visiteur, qui possède autant de méthode `visitK(y)` qu'il existe de classes `K` dans notre AST.
- En supposant que chaque classe `K` possède n attributs x_i, chaque méthode `visitK(y)` du visiteur se présente comme une succession d'appels $x_i.accept(self)$

Comportement Le comportement d'un traitement algorithmique par un visiteur peut sembler complexe : lors de la visite d'un objet k de classe `K`, la méthode $visitK$ du visiteur consiste à appeler les méthodes `accept` de chacun des attributs x_i de l'objet k. Ces méthodes `accept` appelées n'ont qu'un seul but : elles vont faire "rebondir" le flot de contrôle vers le visiteur qui les a appelées ! Mais ce "rebond" se fera désormais vers la méthode $visitX_i$: le principe d'appels imbriqués ("récursifs" ou plutôt "arborescents") est désormais initié.

Exemple (code non-optimisé) On modifie dans un premier temps notre (méta) modèle d'AST

```ruby
class Program < AstNode
  attr_accessor :name
  attr_accessor :body
  def accept(visitor)
    visitor.visitProgram(self)
  end
end

class Body < AstNode
  attr_accessor :stmts
  def accept(visitor)
    visitor.visitBody(self)
  end
end

class Stmt < AstNode
  def accept(visitor)
    visitor.visitStmt(self)
  end
end
# etc
```

Dans un second temps, on écrit le visiteur lui-même.

```ruby
class Visitor
  def visitProgram program
    program.name.accept(self)
    program.body.accept(self)
  end

  def visitBody body
    body.stmts.each{|stmt| stmt.accept(self)}
  end

  def visitStmt stmt
    # etc
  end
end
```

5.4 Amélioration par métaprogrammation

Nous venons de voir que le principe du visiteur consiste à déporter la contribution algorithmique de chaque objet K vers le visiteur : l'algorithme est ainsi centralisé dans une seule et même classe. Nous évitons ainsi la dispersion du code et favorisons la maintenance et l'intelligibilité de l'algorithme qui opère sur une structure pourtant hétérogène. La seule modification nécessaire à ce patterne a consisté à adjoindre une seule et unique méthode `accept` à chaque objet. Les différents traitements algorithmiques sur l'AST consisteront à sous-classer le Visiteur initial : il ne sera alors pas nécessaire de modifier à nouveau les classes. Quel que soit le nombre d'algorithmes envisagées, cette seule méthode `accept` suffira. On peut pourtant aller encore plus loin, dans le cas d'utilisation d'un langage interprété comme Python ou Ruby : il est en effet possible de se passer de l'écriture des méthodes `accept` pour chaque classe de l'AST. Cela se fait par **métaprogrammation**.

Principe Nous avons établi que chaque classe de notre AST hérite d'une classe primitive `AstNode`. Jusqu'ici nous avons omis de décrire la méthode `accept` de cette classe. Par métaprogrammation, il est possible de profiter de cette classe pour *décrire* les méthodes `accept` de toutes les autres classes. Il s'agit là d'une factorisation bien pratique !

```ruby
class AstNode
  def accept(visitor, arg=nil)
     name = self.class.name.split(/::/).last
     visitor.send("visit#{name}".to_sym, self ,arg) # Metaprograming !
  end
end
```

Au sein de la méthode `accept` de l'objet `AstNode` :

- On récupère le nom effectif de l'objet : par exemple le nom "Program"
- On construit le nom de la méthode voulue : par exemple "visitProgram"
- On demande au visiteur d'exécuter la méthode "visitProgram"

5.5 Applications du pattern Visiteur

Le pattern visiteur permet d'appliquer différents algorithmes sur la structure hétérogène de notre AST. Parmi les visiteurs utiles et généralement indispensables, on retrouve :

- **Visiteur simple** (DummyVisitor) : ce visiteur ne fait que parcourir la structure, sans réaliser le moindre traitement. Il permet au développeur du compilateur de s'assurer que l'ensemble des noeuds de l'AST est bien visité. On peut tout de même chercher à visualiser cette visite par un message "visitX" correctement indenté pour chaque classe X, en propageant une indentation à travers les arguments des méthodes.
- **PrettyPrinter** : ce visiteur cherche à ré-afficher le code source initial, ous une forme esthétiquement satisfaisante. La structure d'AST permet notamment d'afficher correctement les indendations.
- **Checker** (ou analyseur contextuel) : ce visiteur vérifie la cohérence *sémantique* de l'ensemble du programme. Notamment, il vérifie les points suivants :
  - déclarations des variables utilisées
  - non-duplication des déclarations
  - cohérence des types (affectations des expressions)
  - ...
- **Transformer** : ce visiteur réalise une copie intégrale de l'AST en un autre AST, qui contient les mêmes informations. Ce Transformer est utile à plusieurs titres : il permet notamment de définir des transformations locales, qui héritent de ce transformer.
- **Générateur de code** : pour peu que l'on dispose de schémas de traduction robustes sur l'AST, la visite peut aboutir à la génération d'un code d'un autre langage. Lorsque le code est de plus bas niveau, on pense bien entendu à l'assembleur. On peut également réaliser des transformation dites *source-à-source*.

5.6 Conclusion

Le pattern visiteur est un élément fondamental des compilateurs. Il permet de visiter l'AST et de lui appliquer différents algorithmes, sans modifier la définition des types de noeuds. Il s'applique à différentes passes des compilateurs.

Chapitre 6

Analyse contextuelle ou *sémantique*

6.1 Objectifs

On se souvient que les deux premières analyses qui ont été menées à bien jusqu'ici étaient :

- l'analyse lexicale : elle consiste à vérifier que les *mots* (lexèmes) rencontrés sont admis par le langage.
- l'analyse syntaxique : elle vérifie que l'agencement des lexèmes constitue de *phrases* correctes vis-à-vis des *règles* de le *grammaire* du langage).

L'analyse contextuelle, qui suit, est une passe du compilateur qui vise à établir si un programme a du un *sens* ou pas.

6.2 L'importance du contexte d'analyse

6.2.1 Le sens des phrases

Pour continuer l'analogie avec le Français, une phrase comme : "La sandale recommande Flaubert" est lexicalement et syntaxiquement correcte, mais n'a pas de sens. On pressent toutefois que nous rentrons ici dans des analyses plus délicates, et qu'il faudra prendre en compte un certain *contexte* d'analyse. Par exemple, si "la sandale" est un journal de bibliophiles, la phrase précédente peut retrouver un certain sens. Dans le cas qui nous préoccupe –l'analyse de programmes informatiques–, nous allons nous borner à vérifier que le programme source, transposé en AST, vérifie effectivement certaines règles bien établies.

6.2.2 Notion de LRM

Ces règles dépendent évidemment du langage : en général, pour les langages *mainstream*, ils s'accompagnent d'un texte *normatif*, qui énonce ces règles : c'est le *language reference manual* (LRM). Le LRM est un guide précieux pour le développeur du compilateur, mais est également recommandable pour les utilisateurs du langage. Le LRM utilise le langage courant (l'anglais le plus souvent) pour préciser ces règles. Ce LRM est parfois standardisé par des organismes spécifiques, comme l'IEEE, l'ANSI, l'ECMA, l'ISO ou l'ITU etc. Par exemple, le langage C a été standardisé et donc décrit par l'ANSI en 1989 (C89), puis par l'ISO (C99). La version actuelle (nous sommes en 2020) est celle de l'ISO/CEI 9899 :2011. Ruby, C++, VHDL et Verilog sont

également standardisés etc. D'autres langages mainstream ne sont pas forcément standardisé par ces organismes : c'est le cas de Python. Toutefois, le fait d'être standardisé ne signifie pas que le sens des programmes de tel ou tel langage est parfaitement décrit. Par exemple, il existe des *flous* sémantiques concernant l'ordre d'évaluation des arguments d'une fonction en C. Comment aller plus loin ?

6.2.3 Implémentation de référence

Comme pour Python, Ruby possède une implémentation de référence : le MRI ou "Matz Ruby Implementation". C'est également la version la plus utilisée. Les autres implémentations cherchent à reproduire le même comportement à l'exécution et accepter, lors de l'analyse, les mêmes programmes en entrée. L'existence d'implémentations alternatives ouvre généralement des pistes d'améliorations du langage : par exemple, l'implémentation JRuby (en Java) de Ruby permet actuellement d'explorer des alternatives concernant le parallélisme en Ruby (le modèle Ruby est basé sur des *green threads*, qui ne profitent pas du parallélisme des machines).

6.2.4 Sémantique formalisée

Certains langages vont encore plus loin. l'ISO Modula-2 par exemple, propose un modèle mathématique de son exécution. Cette pratique se développe dans le milieu académique, où différentes formes de sémantiques sont utilisées. Nous les citons ici uniquement, sans aller dans leur description, et encore moins dans leur pratique, qui dépasse largement le cadre de ce cours.

- sémantique opérationnelle : il s'agit de décrire les étapes de l'exécution de la machine sous-jacente.
- sémantique dénotationnelle (ou sémantique mathématique ou de Scott–Strachey) : il s'agit de construire des objets mathématiques qui décrivent le sens des éléments d'un programme, et qui autorisent leur composition. Le recours à des principes issus des langages fonctionnels est souvent de mise.
- sémantique axiomatique : à partir d'un ensemble d'assertions préalables, on décrit ici l'effet d'une instruction en terme d'assertions déduites.

Toutefois, cette pratique reste un sujet d'étude et aucune pratique généralisée ne semble se dessiner : il n'existe pas l'équivalent des générateurs de lexeurs ou de parseurs concernant l'analyse sémantique. La formalisation de telles sémantiques est pourtant fondamentale pour des domaines où l'assurance d'un bon fonctionnement est cruciale (safety-critical applications).

6.3 Exemples de vérifications dans différents langages

Revenons à notre compilateur. Nous disposons désormais d'un AST, qui représente un programme en mémoire. Quelles vérifications cherchons nous à réaliser ? Dans le cas de langages compilés statiquement et où les déclarations sont explicites on peut par exemple réaliser les vérifications suivantes :

- Vérifier que si une variable est utilisée, alors elle est bien déclarée au préalable.
- Vérifier qu'une variable n'est déclarée qu'une fois.
- Vérifier que dans une assignation, le partie droite et la partie gauche possède le même type.
- Vérifier que les indexes d'un tableau sont bien de type entier (positifs).
- Vérifier que lors d'un appel de fonction :

  - La fonction est bien déclarée
  - l'appel possède le bon nombre d'arguments
  - les types des arguments sont les bons
  - le type de retour est le bon
- etc

Dans d'autres types de langages, d'autres vérifications s'imposeront. Par exemple, dans le cas d'un langage de description matérielle comme VHDL, on pourra réaliser les vérifications suivantes. Outre les vérifications classiques précédentes :

- Vérifier qu'une architecture est bien associée à une entité connue.
- Vérifier qu'on ne cherche pas à écrire sur un port d'entrée.
- Vérifier qu'on ne cherche pas à lire sur un port de sortie.
- etc

6.4 Table des symboles

Malgré la variété des types d'analyses (vérifications) précédentes, il existe un concept indispensable à nos analyses. Il s'agit de la **table des symboles**. Cette table des symboles permet d'enregister les symboles lors de leur déclaration : telle variable `x` est déclarée en tant que `int`. Lors du parcours de l'AST (grâce à une visiteur spécifique), on enregistre de telles déclarations. Il sera alors par exemple possible, lors du parcours d'une expression de s'assurer qu'une variable rencontrée dans une feuille de l'AST correspondant que la variable est bien référencée. Il sera également possible de réaliser une propagation de type, pour calculer le type de l'expression elle-même. Parfois cette seule vérification de type conduit à l'écriture d'une passe spécifique appelée `TypeChecker`.

6.4.1 Table des symboles à portée simple

L'implémentation la plus basique d'une table des symboles repose simplement sur une structure de données disponible dans la plupart des langages de programmation : le tableau associatif, plus couramment appelé Hash ou dictionnaire. Le pseudo-code Ruby suivant illustre ce propos. Cet

```ruby
class SymbolTable < Hash
  def set symbol,info
    self[symbol]=info
  end
  def get symbol
    self[symbol]
  end
end

symtable=SymbolTable.new
symtable.set("x",VarDecl.new("x","int"))
symtable.get("x") #returns the VarDecl object
```

exemple ne fonctionne malheureusement que dans de rares cas. Supposons que nous analysions un code C, constitué de plusieurs déclarations de fonctions, et variables globales, etc. Au sein d'une fonction, nous pouvons également avoir des variables locales. Certains langages permettent même

de définir des fonctions *dans* des fonctions (ce n'est pas le cas du C, mais par exemple VHDL l'autorise). On voit donc que nos captures de déclarations dans la table des symboles précédente ne permet pas de distinguer ces différents *contextes*. Ces contextes sont essentiellement liés à la *portée des variables*, c'est-à-dire à leur *visibilité* à un certain endroit du code. En C, les variables globales sont par exemple visibles des fonctions déclarées. Il nous faut donc améliorer la table des symboles précédente, en prenant en compte ces imbrications de contextes.

6.4.2 Table des symboles à portées multiples

Pour gérer l'imbrication de contextes, on doit pouvoir :

- réaliser une déclaration de symbole dans un contexte précis (ou *scope*, en anglais) : on reposera localement sur une table de Hash.
- rechercher un symbole dans une hiérarchie de contextes : il faudra gérer ici une structure de données plus complexe. Par exemple, on peut utiliser une liste de Hash : on avance dans la liste au au fur-et-à-mesure que l'on "descend" dans un scope particulier. A l'inverse, plus l'on est proche de début de liste, plus le scope est général.

En général, les tables de symboles à portée multiple présente une interface proche de la suivante :

- **create_scope()** : créé un nouveau contexte de déclaration.
- **leave_scope()** : quitte le contexte courant.
- **set(symbol,info)** : déclare un symbole dans le context courant.
- **get(symbol)** : recherche la déclaration récurisivement dans la liste, en partant du contexte courant, et en remontant dans la liste, du particulier au général, ou plutôt du "local" au "global".

6.5 Cas particuliers

6.5.1 Symbole unique associé à différentes déclarations. Un exemple en C++

Le type de table des symboles précédente fonctionne dans la plupart des cas. Noter toutefois que certains langages rendent cette table inopérante, et il faudra alors réfléchir au cas par cas à des table des symboles adaptées. A titre d'exemple, le langage C++ autorise la déclaration d'une variable (disons "x") locale à une fonction, y compris si un paramètre de la fonction porte *également* le même nom ("x"). Dans ce cas, toute utilisation de "x" se réfèrera à la variable locale, et non au paramètre. L'analyse contextuelle devra donc accepter que dans un même contexte (celui de déclaration la fonction), on puisse avoir deux symboles identiques qui ne référencent pas la même déclaration concernant "x" : elle ne devra pas remonter de message d'erreur à l'utilisateur, concernant son usage multiple de "x". En terme d'implémentation, la table de Hash précédente ne fonctionnera pas et il faudra imaginer d'autres structures de données.

6.5.2 Portée statique et dynamique

Notre propos précédent (qui reste introductif) ne couvre que des langages dits *à portée statique*. On utilise l'adjectif *statique* lorsqu'il n'est pas nécessaire d'exécuter le code pour parvenir à ses fins. Le cas de la portée statique signifie qu'au moment de la compilation, nous possédons toute l'information, contenue dans l'AST, pour déterminer si un symbole est correctement utilisé. Les

livres utilisent parfois le vocable de *portée lexicale* comme synonyme de portée statique. Il existe d'autres langages où il faudra exécuter le code pour établir si une variable est effectivement bien déclarée et utilisée : dans ce cas, c'est la hiérarchie des *appels* qui attestera de la présence (ou non de la variable) dans un des contextes (pour le coup dynamique) des appelants. Noter que les languages orientés-objets utilisent un concept similaires appelé *dynamic dispatch*.

6.6 Conclusion

L'analyse contextuelle vient renforcer les analyses préalables réalisées sur le lexique et la grammaire du langage analysé. On cherche ici à déterminer si des symboles sont correctement utilisés. Pour cela, il est nécessaire de recourir à une structure de donnée nouvelle : la table des symboles, qui enregistre et restitue la présence et le type des symboles rencontrés, ainsi que la portée des déclarations. Ces vérifications successives permettent de filtrer différentes erreurs courantes bien répertoriées et d'envisager une génération de code dans de bonnes conditions.

Chapitre 7

Représentations intermédiaires

7.1 Introduction

A partir du seul AST produit par le parser, et un ensemble de vérifications réalisées dans l'analyse contextuelle, il est possible de générer du code pour une cible particulière. Par cible, on entend le jeu d'instruction d'un processeur. Il peut s'agir d'un jeu d'instructions d'un processeur réel (comme l'x86, ARM, Sparc, RISC-V) ou d'un processeur virtuel, comme la machine virtuelle Java (jvm). Toutefois, si ce saut entre l'AST et le code exécutable est parfaitement envisageable, on préfère généralement insérer une passe supplémentaire dans le flot de compilation. Il s'agit ici de passer par une **représentation intermédiaire** (entre l'AST et l'assembleur) qui nous apportera un ensemble de bénéfices.

7.2 Motivations

Une IR pour représenter plusieurs langages sources Parmi les bénéfices attendus, le recours à une IR permet d'envisager de l'utiliser comme représentation de *plusieurs* langages sources. Vous avez probablement remarqué qu'un grand nombre de langages présentent des constructions syntaxiques voisines voire identiques (boucles, conditionnelle, appels de fonctions, etc). Faire émerger une représentation intermédiaire permet à tous les constructeurs de compilateurs d'unir leurs efforts avant d'entrer dans les méandres de la génération de code.

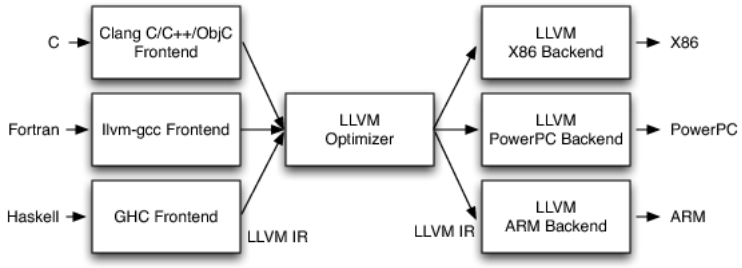


FIGURE 7.1: Une IR (ici LLVM) pour plusieurs langages sources et plusieurs cibles.

Une IR pour optimiser Il est par exemple possible d'envisager des opimisations *indépendantes de la cible*. Les développeurs auront tout intérêt à partager ces optimisations, favorables à plusieurs sources et plusieurs cibles.

Une IR pour plusieurs cibles Notre introduction précédente laissait entendre que notre but était de générer du code pour *une* cible. En réalité, si vous avez créé un superbe nouveau langage de programmation, il serait dommage de ne cibler qu'un seul type de machines, scindant d'emblée en deux vos futurs utilisateurs (ceux qui pourront en bénéficier,. . .et les autres). Etablir une unique IR permet de factoriser l'infrastructure qui permet la génération de code : plusieurs méthodes seront sensiblement identiques, quel que soit la cible.

7.3 Différents types d'IR

Il existe différents types de représentations intermédiaires. Pour être éligibles, elles doivent *a priori* remplir certains critères :

1. Une IR doit être plus simple que le langage source (ou les langages sources).
2. Une IR doit permettre la représentation de l'ensemble du langage source.
3. Une IR doit permettre certaines facilités en terme de génération de code.

7.3.1 Le C comme représentation intermédiaire

En premier lieu, on peut simplement citer le langage C, qui est utilisé comme IR de plusieurs compilateurs ! Cela peut paraître inattendu, mais le langage remplit assez facilement le critère 2 précédent : son caractère bas niveau à fait ses preuves (il est né de la création du systèmes d'exploitation UNIX). La manipulation directe de la mémoire (et des aspects matériels comme l'interaction avec des périphériques) attestent de sa malléabilité. On présente parfois le langage C comme un "macro assembleur". Par ailleurs, le critère 3 est également vérifié car les compilateurs C ont *déjà* été porté sur différentes cibles, ce qui facilite grandement la génération de code (elle est déjà faite) !

7.3.2 LLVM

Mais lorsqu'on parle d'IRs, on pense immanquablement à **LLVM**. LLVM, qui signifie *Low Level Virtual Machine*, est né en 2000 à l'Université de l' Illinois à Urbana–Champaign. Vikram Adve and Chris Lattner en sont ses principaux créateurs, mais ils ont été aidés par une communauté open source assez vaste. LLVM est l'IR d'un très grand nombre de langages, qui bénéficient de ses optimisations avancées. Comme langage supporté, on trouve bien entendu le C et le C++ : le front-end s'appelle alors Clang. Initialement, le projet visaient à étudier la compilation de langages dynamiques, mais LLVM est utilisé dans un grand nombre de contexte : par exemple, le simulateur open-source GHDL, développé par Tristan Gingold, génère du code intermédaire LLVM [1]. LLVM propose un langage textuel, dont on peut lire un aperçu ci-dessous.

```
define i32 @gcd(i32, i32) local_unnamed_addr #0 {
  %3 = icmp eq i32 %0, %1
  br i1 %3, label %14, label %4

; <label>:4:                                     ; preds = %2
  br label %5

; <label>:5:                                     ; preds = %4, %5
  %6 = phi i32 [ %12, %5 ], [ %1, %4 ]
  %7 = phi i32 [ %10, %5 ], [ %0, %4 ]
```

[1] GHDL propose également la génération d'autres IR comme le mcode, et du code x86.

```
  %8 = icmp slt i32 %6, %7
  %9 = select i1 %8, i32 %6, i32 0
  %10 = sub nsw i32 %7, %9
  %11 = select i1 %8, i32 0, i32 %7
  %12 = sub nsw i32 %6, %11
  %13 = icmp eq i32 %10, %12
  br i1 %13, label %14, label %5

; <label>:14:                                     ; preds = %5, %2
  %15 = phi i32 [ %0, %2 ], [ %10, %5 ]
  ret i32 %15
}
```

On peut schématiquement retenir les aspects suivants :

- Le code ressemble à de l'assembleur : le code se présente comme une suite de *basic-blocs*, estampillés par un *label*. Chaque basic blocs se termine soit par une instruction de branchement (saut conditionnel ou inconditionnel) ou d'une instruction de retour.
- A la différence de l'assembleur LLVM présente des variables en lieu et place des registres traditionnellement manipulés par l'assembleur : en ce sens, on peut dire que ces variables sont des registres en nombre *infini*.
- Les instruction sont simples : la plupart des instructions LLVM sont des assignations. La partie droite de l'assignation est une expression simple, souvent structurée autour de deux opérandes. Ces opérandes sont typées. On peut retrouver également aisément le type des variables assignées. On rappelle que le code précédent est le format textuel LLVM, mais qu'il est structuré en mémoire, et que ces types, par exemple, sont directement accessibles par les fonctions de manipulation de LLVM.

7.3.3 Gimple

Gimple est l'IR de la suite GCC. On se souvient que GCC est une *collection* de compilateurs. Ils ne partageaient pas initialement la même représentation des programmes. C'est chose faite avec Gimple, qui peut être vu comme proche de LLVM. Il possède tout comme lui une forme d'assignation unique (discutée dans la section suivante).

7.3.4 CIL

CIL (pour common intermediate language) est l'IR utilisée par la plateforme .NET de Microsoft.

7.4 Principle d'assignation unique : forme SSA

En observant de près le code LLVM précédent, on peut observer l'existence d'une instruction spéciale, qui n'existe pas dans les jeux d'instruction des processeurs : il s'agit de l'instruction **phi**. Cette instruction permet de réaliser le **principes d'assignation unique** : chaque variable n'est assignée qu'une fois ! Pourquoi cette curiosité ? Pour comprendre, il faut revenir à un code source traditionnel : une variable "x" peut bien entendu être affectée à différents endroits dans le code. Dans le but d'optimiser le code généré final, il est intéressant de connaître les *dépendances de données*. On cherche à établir ces dépendances par une analyse appelée *dataflow* et trouver l'ensemble des relations "qui modifie qui ?". En pratique, le fait qu'une même variable "x" puisse être modifiée par plusieurs assignations rend cette analyse très complexe. Il s'avère que la forme s'assignation unique (appelée en anglais Single Static Assignment ou **SSA**) permet de "suivre" la

modification d'une variable de manière plus aisée. La génération du LLVM va donc réaliser le *renommage* de la variable x en un ensemble de x_i. Il faut s'assurer, lorsqu'on *utilise* x, que l'on se réfère au "bon" x_i. L'instruction `phi` peut-être vue comme un multiplexeur logiciel qui permet, en certains points du programme (appelées points de jonction) de recombiner plusieurs sources x_i en une seule, et d'ainsi pouvoir "suivre" le flot de données à nouveau. Ce flot de données vient compléter le flot de contrôle réalisé par les branchements, rendant l'ensemble du programme analysable sous forme d'un graphe.

7.5 Décomposition des expressions complexes : code trois adresses.

Le code SSA précédent, et les IR en général, présentent des assignations dont la partie droite est extrêmement simple. Il ne s'agit plus que d'expressions binaires voire unaires. La décomposition d'expressions complexes présentes dans le source se fait à nouveau à l'aide d'un visiteur approprié. On présente toutefois ci-dessous un code simplifié, qui réalise le parcours de manière récursive, sans visiteur. Des variables intermédiaires préfixées par `tmp` sont générées lors du parcours.

```
Binary = Struct.new(:lhs,:op,:rhs)
Unary  = Struct.new(:op,:e)
Var    = Struct.new(:str)
Lit    = Struct.new(:str)

class Flattener
  def flatten expr
    case expr
    when Binary
      lhs=flatten expr.lhs
      rhs=flatten expr.rhs
      tmp=new_tmp()
      puts "#{tmp} = #{lhs} #{expr.op} #{rhs}"
      return tmp
    when Unary
      expr=flatten expr.e
      puts "#{tmp} = #{expr.op} #{expr}"
      return tmp
    when Var,Lit
      return expr.str
    end
  end

  def new_tmp
    @tmp||="tmp_0"
    @tmp=@tmp.succ
  end
end

discriminant=Parser.new.parse "b*b - 4*a*c"

Flattener.new.flatten(discriminant)

# display :
```

```
# tmp_1 = b * b
# tmp_2 = a * c
# tmp_3 = 4 * tmp_2
# tmp_4 = tmp_1 - tmp_3
```

Le type de code généré est souvent appelé *code trois adresses*, car trois arguments sont nécessaires à la réalisation de ces instructions : les deux opérandes et la destination. On utilise ce vocable y compris au niveau des expressions (sans parler d'assembleur).

7.6 Cas du langage Newage

Ma recherche... Dans le cadre de mes recherche dans le domaine des systèmes embarqués, je m'intéresse à la compilation de programmes parallèles, leur représentation et la génération de code sur des cibles hétérogènes telles qu'on les retrouve dans les System-on-Chip modernes. Parmi ces cibles, on retrouve notamment les FPGA du fait de leur versatilité et leur capacité à incarner cette hétérogénéité. Ces études m'ont mené à concevoir un langage appelé Newage. Newage possède les caractéristiques suivantes :

- description d'un système sous la forme d'un réseau de composants logiciels appelés *acteurs* qui communiquent par des canaux synchrones ou asynchrones (FIFO bornées).
- Le comportement de chaque acteur est pour l'essentiel séquentiel.
- La compilation se fait de manière incrémentale, petit-à-petit, en se rapprochant de la cible. Les résultats de chaque passe de compilation est accessible à l'utilisateur ; notamment à des fins d'analyse ou de refactoring manuel.
- Ces incréments passent par plusieurs paradigmes : code séquentiel, IR traditionnelle, machines à états finis, etc.

La représentation intermédiaire de Newage est présentée ci-dessous, sous forme graphique. J'ai explicité le graphe de flot de contrôle (CFG) d'un programme simple. Ce CFG est bien l'assemblage de plusieurs basic-blocs.

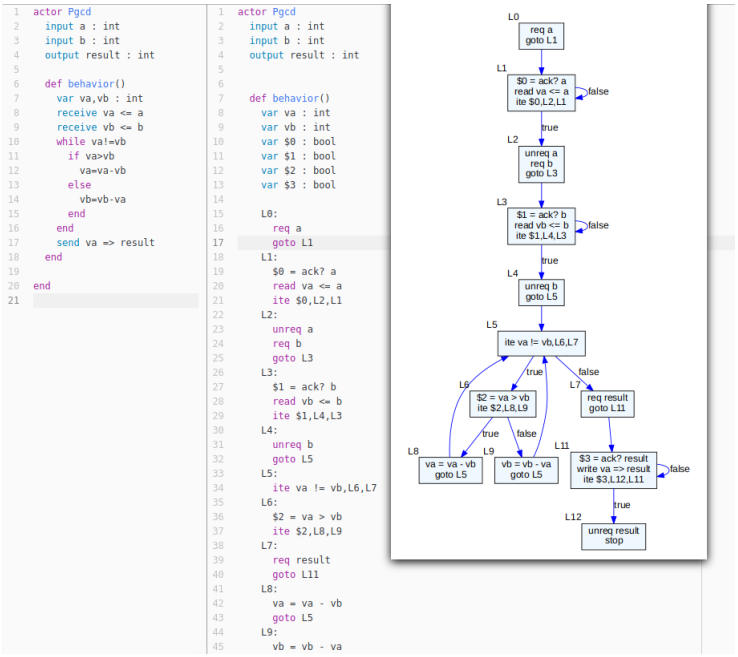


FIGURE 7.2: Source, IR et représentation du CFG dans le langage Newage

7.7 Conclusion

Ce chapitre nous a permis de prendre connaissance de la notion d'IR ou représentation intermédiaire. Elle agit comme langage intermédiaire, souvent unificateur, avant la génération de code. C'est également le lieu d'optimisations indépendantes de la cible.

Chapitre 8

Génération de code

8.1 Introduction

La génération de code est considérée comme le but ultime d'un compilateur. Elle intervient après un ensemble d'analyses préalables (déjà étudiées), qui renforcent la confiance qu'on peut avoir dans le code final généré. Ce code peut être de bas niveau comme dans le cas de l'assembleur, mais peut également se présenter sous forme d'un code de haut niveau, structuré, qui nécessite quelques précautions dans la mise en forme du code. Sur un autre plan, la génération de code doit fonctionner de la manière la plus mécanique et repose sur des règles de traduction simples et non ambigües. Ces règles s'appellent des *schémas de traduction*.

Dans ce chapitre, nous allons dans un premier temps rappeler un langage assembleur cible, déjà étudié en cours d'architecture des ordinateurs. Dans une second temps, nous étudierons quelques schéma de traduction. Enfin, on évoquera les techniques de génie logiciel assistant la génération de code elle-même.

8.2 Langage cible : rappel du mini-MIPS

Jeu d'instructions RISC Le langage Mini-MIPS étudié dans le cours d'architecture des ordinateurs est rappelé ici. Il se présente comme un petit ensemble d'instructions dont la plupart nécessitent trois opérandes, qui peuvent être des opérandes immédiats ou des registres. Un tel jeu d'instructions est du type RISC : *reduced instruction set*. Historiquement, on s'est rendu compte que les jeux d'instructions plus complexes (CISC) entraînaient des complexités matérielles ici et là, pénalisant l'ensemble du processeur. A l'inverse un tel jeu d'instruction RISC présente un *datapath* efficace en terme de fréquence de fonctionnement. Il nécessitera probablement de générer, pour un programme donné, un peu plus d'instructions à exécuter au final que dans le cas d'un processeur CISC, mais l'ensemble d'exécutera plus vite sur un RISC.

Notations : r nom de registre (`r 0`, `r 1`, ..., `r 31`)
o nom de registre ou constante entière (`12`, `-34`, ...)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1/o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme

Architecture Load/store Toute la difficulté du langage assembleur consiste à gérer les allers et venues entre la mémoire de données et les registres. On rappelle également que ces registres, en nombre également réduit (16 par exemple), sont proches de l'ALU qui réalise les opérations classiques (addition, multiplication etc). Ces registres sont constitués de bascule-D coûteuses en terme de surface. A l'inverse la mémoire de données, comme la mémoire d'instructions, est de type SRAM ou mieux encore de type DRAM, plus dense d'un facteur 6 environ (6 transistors pour stocker 1 bit dans le cas SRAM et 1 classiquement pour le DRAM). Les allers et venues permettent de manipuler de grandes quantités de données stockées en mémoire RAM. On rappelle que ces architectures sont qualifiées de "load-store", en référence aux deux intructions du même nom, chargées de réaliser ces transferts.

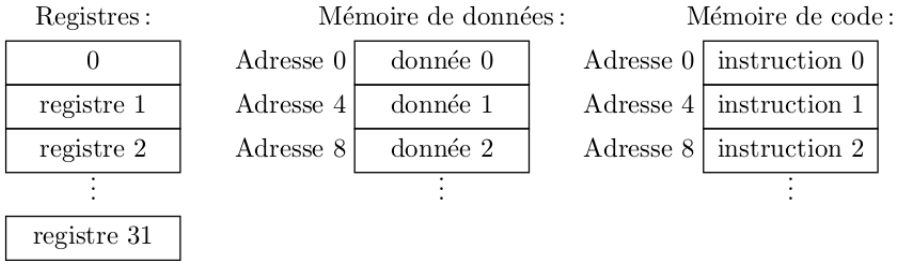


FIGURE 8.1: Modèle d'ISA load/store

8.3 Présentation de schémas de traduction

8.3.1 Cas des boucles *for*

Le cas de la boucle `for` présenté sur la figure suivante nous ramène immanquablement à une décomposition du contrôle et à une représentation intermédaire. Celle-ci peut être explicite comme vu au chapitre précédent, ou implicite : dans ce dernier cas, c'est la génération de code qui devra "contiendra" tout le savoir en matière de décomposition du contrôle. Sur le schéma emprunté à [2]

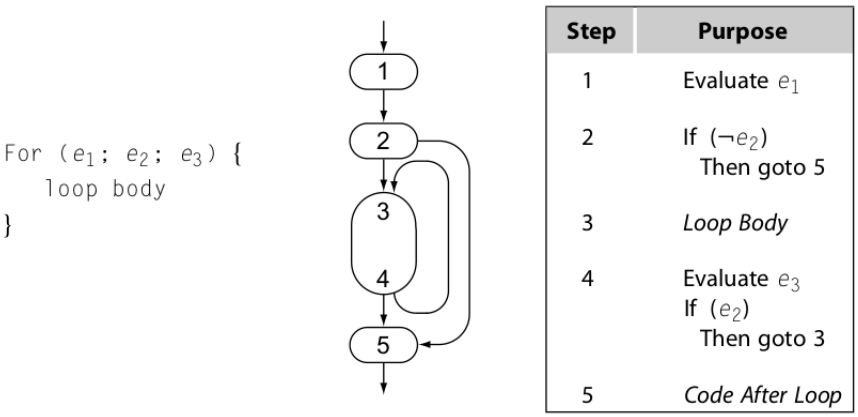


FIGURE 8.2: Schéma de traduction d'une boucle [2]

```
for (i=1; i<=100; i++) {
  loop body
}
next statement
```

```
        loadI   1       ⇒ ri      // Step 1
        loadI   100     ⇒ r1      // Step 2
        cmp_GT  ri, r1  ⇒ r2
        cbr     r2      → L2, L1
L1:     loop body                 // Step 3
        addI    ri, 1   ⇒ ri      // Step 4
        cmp_LE  ri, r1  ⇒ r3
        cbr     r3      → L1, L2
L2:     next statement            // Step 5
```

FIGURE 8.3: Boucle du langage C et code assembleur correspondant [2]

8.3.2 Cas des boucles *while*

```
while (x < y) {
  loop body
}
next statement
```

```
        cmp_LT rx, ry ⇒ r1    // Step 2
        cbr    r1     → L1, L2
L1:     loop body             // Step 3
        cmp_LT rx, ry ⇒ r2    // Step 4
        cbr    r2     → L1, L2
L2:     next statement        // Step 5
```

FIGURE 8.4: Boucle *while* du langage C et code assembleur correspondant [2]

8.3.3 Cas des appels de fonctions : la pile

Approche naïve Les appels de fonctions requièrent une attention particulière. On sait déjà que le code binaire associée à une fonction sera vu comme une suite d'instructions, localisées à partir d'une certaine adresse de la mémoire d'instructions. Dans le but d'implémenter un tel appel de fonction, on peut imaginer l'approche naïve suivante : au moment de l'appel, on dédie un registre particulier pour contenir l'adresse de retour à la fin de l'exécution de la fonction appelée. Dès lors que cette addresse de retour est bien stockée, on peut détourner le compteur programme vers l'addresse de la fonction. Cette adresse peut-être décidée et enregistrée dans la passe de génération de code de notre compilateur. Malheureusement, cette approche ne fonctionne pas : en effet, on oublie là que des fonctions peuvent appeler d"autres fonctions (voire elles-mêmes). Un seul registre dédié ne peut suffire popur contenir les adresses de retour...

Nécessité d'une pile ou *call stack* Il est donc nécessaire de recourir à une zone mémoire (et non des registres) pour garder trace des addresses de retour successives lors d'appels imbriqués. Cette zone d'appelle la *pile d'appel* ou an anglais *call stack* ou *runtime stack*. Lors de l'exécution, à chaque appel effectif d'une fonction `f`, cette pile va grandir : une zone dédiée à cet appel à `f` va être allouée. Cette petite zone, uniquement dédiée à cet appel à `f`, s'appelle **bloc d'activation** (*activation record* ou encore *stack frame*). Elle est le pendant "local" de la zone "globale" où l'on a stocké les variables globales. Cette zone va également nous permettre d'organiser d'autres éléments de l'appel, que nous avons négligés jusqu'ici : comment sont passés les arguments (paramètres) de l'appelant à l'appelé ? Comment les variables locales d'une fonction sont-elles stockées ? Le bloc d'activation va permettre d'y répondre. Le schéma suivant illustre la manière de procéder. On imagine que l'on a deux fonctions : `drawsquare`, qui appelle, elle-même, une seconde fonction `drawline`.

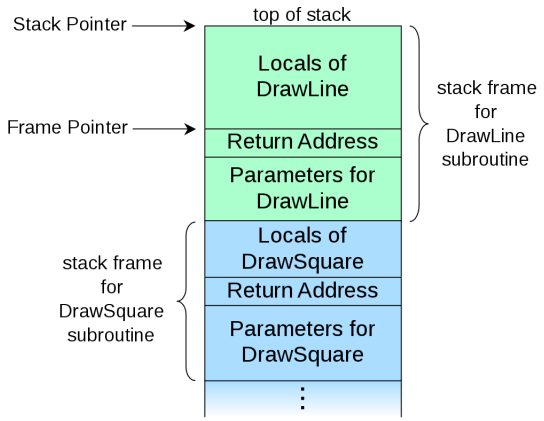


FIGURE 8.5: Principe de la pile d'appel d'un programme (source : Wikipedia)

- Deux registres dédiés vont permettre de suivre l'évolution des appels lors de l'exécution. Il s'agit du **pointeur de pile**, qui indique l'adresse du sommet de la pile, et du **pointeur de bloc** (le mot *frame pointer* est plus courant). Sur le schéma on peut constater que le frame pointer pointe "au milieu" du bloc s'activation. En réalité, il pointe sur l'adresse de retour de la fonction `f`. Au dessus de cette adresse, on retrouve les variables locales de `f` et en dessous ses arguments (parfois passés dans un ordre inverse de la présentation dans le code source).

- C'est au compilateur de générer le code assembleur qui va gérer correctement cette dynamique. Lorsqu'il rencontre une appel de `f`, il doit mesurer la taille du bloc d'activation de

`f`, et générer un code assembleur qui modifie le pointeur de pile de cette quantité. Il doit également faire pointer le stack pointer comme on vient de le voir. Enfin, chaque variable locale et chaque paramètre se voient affectées d'une adresse `relative` au frame pointer. Le compilateur doit bien évidemment passer les paramètres effectifs de `f` en les poussant sur cette pile (parfois certains jeux d'sintruction possèdent cette instruction `push`, mais notre processeur MiniMIPS n'en possédait pas).

- Dès lors que ce code assembleur de *préparation* s'est exécuté, l'appel se résume à détourner le compteur programme (PC) vers l'adresse où se situe le code de `f`. Ce code sera généré en connaissant toutes les adresses (relatives) précédentes, et notamment son adresse de retour.

8.4 Allocation dynamique : *heap* ou *tas*

Il existe une autre zone mémoire importante, qui permet de gérer les allocations *dynamiques* au sein des programmes. Il s'agit du *tas* (ou *heap* en anglais). C'est une structure totalement différente de la pile précédente, qui servait à empiler les contextes d'appels de sous-fonctions. Cette structure est représentée sur la figure suivante. On notera que les deux structures (heap et stack) croissent dans des directions qui rapprochent leurs adresses jusqu' à un point inéluctable où le système connaîtra un débordement et des erreurs bien connues du programmeur (stack overflow, échec d'allocation dynamique, etc).

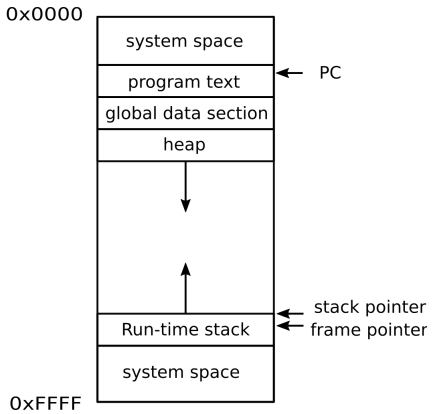


FIGURE 8.6: Pile et tas au sein de la mémoire

8.5 Techniques de génération de code

Dans cette section, nous présentons deux techniques courantes de génération de code. La second est plus exotique, mais est utilisée dans le domaine du Web et il nous a paru intéressant de la noter ici.

8.5.1 Par visiteur

La première des techniques est la plus courante. Elle repose sur le pattern utilisé par ailleurs dans les autres passes du compilateur : il s'agit à nouveau du pattern visiteur. Ce pattern nous a déjà permis de réaliser des actions appropriées dans certains noeuds de l'AST. Il s'agit ici pour nous, pour l'essentiel d'agglomérer les résultats des schémas de traduction, lors du parcours de l'arbre. A la fin de cette visite, on peut émettre un texte (ou un binaire), qui constitue le code attendu. Une des difficultés récurrente de génération de code à partir d'un AST très structuré est la présentation d'un code final correctement indenté. C'est loin d'être anecdotique, car la

qualité du code généré est souvent évalué d'abord par sa syntaxe. Ceci sera d'autant plus vrai, bien entendu, si notre code généré est lui-même un code structuré (comme du C).

La classe Code En Ruby, je propose une classe appelée "Code", qui simplifie cette agglomération. Une instance de Code permet d'accumuler, grâce à la méthode "«", des lignes de code. Ces lignes de code peuvent être du texte sous forme de chaînes de caractères, mais également d'autres instances de la classe Code. Chaque instance maintient une indentation, que l'on peut spécifier et modifier grâce aux accesseurs de la variable d'instance `indent`. Dès lors, toutes les codes accumulés le seront avec cette indentation. On peut enfin afficher le code final grâce à la méthode finalize ou décider de sauver ce code dans un fichier.

```
class Code

  attr_accessor :indent,:lines

  def initialize str=nil
    @lines=[]
    (@lines << str) if str
    @indent=0
  end

  def <<(thing)
    case code=thing
    when Code
      code.lines.each do |line|
        @lines << " "*@indent+line.to_s
      end
    when Array
      thing.each do |kode|
        @lines << kode
      end
    when NilClass
    else
      @lines << " "*@indent+thing.to_s
    end
  end

  def finalize
    return @lines.join("\n") if @lines.any?
    ""
  end

  def newline
    @lines << " "
  end

  def save_as filename
    File.open(filename,'w'){|f| f.puts(self.finalize)}
    return filename
  end

  def size
```

```
    @lines.size
  end

end
```

8.5.2 Par moteurs de template

Les moteurs de template sont très utilisées dans les technologies Web. Il s'agit de partir d'un texte qui servira de garbarit (template), mais que le moteur est capable de modifier grâce à la présence de balises préalablement placées par l'utilisateur aux endroits "stratégiques" : par exemple, une lettre type est (par définition) toujours la même et seul le nom et le sexe du destinataire doit être changé. L'utilisateur indiquera par balises où se trouvent ces éléments à modifier. Ce qui est intéressant, c'est la connexion directe et très pratique entre le *modèle* manipulé par l'outil (dans notre cas l'AST en mémoire) et ces balises. La connexion est absolument transparente ! La magie vient du fait de la possibilité de bénéficier d'une visibilité complète du contexte (ou "environnement", c'est-à-dire l'accès aux variables crées lors de l'exécution du modèle) complet lié au modèle, lors de l'application du template.

Exemple en Ruby : ERB ERB est de loin le plus utilisé pour générer des pages HTML et autres, dans le framework Ruby-On-Rails. Il permet donc la génération de pages dynamiques, adaptées au profil de chaque utilisateur qui s'est connecté à un site Web. Un exemple volontairement simple est présenté ici. Le lecteur intéressé pourra consulter un livre complet qui est consacré à ERB ("code generation in Action" []). Il s'agit pour nous ici de simplement lire les noms et prenoms d'informaticiens célèbres dans un fichier texte et de générer un code C, qui initialise un tableau de structures contenant ces noms. On peut multiplier ces exemples à foison : par exemple, on peut décrire un ensemble de classes et d'attributs dans un fichier texte, Json, Yaml ou XML et générer le squelette de code orienté objet pour différents langages (un template par langage). Ce dernier cas est intéressant car on n'écrira le template qu'une seule fois, et l'on pourra vite oublier les particularismes de tel ou tel langage concernant la déclaration alambiquées des constructeurs par exemple.

Dans le cas de génération de code C (cas évoqué des informaticien). Le code Ruby de génération de code est simplement :

```
require 'erb'

Scientist=Struct.new(:first_name,:name)

scientists=IO.readlines("names.txt").map{|line|
  line.gsub!(' ','')
  prenom,nom=line.split(',')
  Scientist.new(prenom,nom)
}
engine=ERB.new(IO.read("template.c"),0, '>')
puts c_code=engine.result()
File.open("result.c",'w'){|f| f.puts c_code}
```

Le template s'appelle ici "template.c" : on notera la récupération des données et de la date courante, etc. On peut reprocher au template l'entrelacement de code "normal" avec les balises elles-mêmes, qui viennent quelque peu obscurcir le code.

```
/* code generated  on <%=Time.now.strftime('%A %d,%B,%Y')%>*/
#define SIZE 100
struct{
```

```
  char * name;
  char * surname;
} computer_scientist_t[SIZE]={
  <%scientists.each do |scientist|%>
      {.surname = "<%=scientist.first_name%>",.name="<%=scientist.name.chomp%>"},
  <%end%>
};
```

A partir d'un fichier texte contenant quelques informaticiens, on obtient :

```
/* code generated  on Monday 27,April,2020*/
#define SIZE 100
struct{
  char * name;
  char * surname;
} computer_scientist_t[SIZE]={
        {.surname = "Alan",.name="Turing"},
        {.surname = "Edsger",.name="Dijkstra"},
        {.surname = "Grace",.name="Hopper"},
        {.surname = "John",.name="MacCarthy"},
        {.surname = "Lynn",.name="Conway"},
        {.surname = "Charles",.name="Babbage"},
  };
```

8.6 Conclusion

La génération de code repose sur l'application de règles de traduction du modèle de haut niveau en un modèle de plus bas niveau. Il s'agit le plus souvent d'assembleur, mais la génération de code structuré est également possible. On peut organiser l'infrastructure de génération de code autour du pattern visteur, appliqué sur l'AST ou une représentation intermédiaire (ayant elle-même son AST !). Il est également possible de recourir à des moteurs de template, employé dans le cadre des technologies Web.

Annexes

Chapitre 9

Python MiniC Lexer

On propose ici, à toutes fins utiles, un lexer embryonnaire pour le langage MiniC étudié en TD. Il est basé sur l'application successive d'expressions régulières. Bien que cela puisse être considéré comme maladroit au regard des techniques plus avancées (compilation d'un seul automate de reconnaissance, type Ragel), ce lexer fonctionne. Vous pourrez l'étendre et l'améliorer. On devra notamment gérer correctement la capture du numéro de lignes dans les lexèmes émis.

```
import re
import sys

class Token:
    def __init__(self,kind,val,pos):
        self.kind=kind
        self.val=val
        self.pos=pos

class Lexer:
    def __init__(self):
        self.TOKEN_DEF={
            "main"     : r"\Amain",
            "int"      : r"\Aint",
            "lparen"   : r"\A\(",
            "rparen"   : r"\A\)",
            "lbrace"   : r"\A\{",
            "rbrace"   : r"\A\}",
            "semico"   : r"\A\;",
            "ident"    : r"\A[a-zA-Z]+",
            "int_lit"  : r"\A\d+",
            "op_eq"    : r"\A\=",
            "op_add"   : r"\A\+",
        }
        self.source=""

    def open_file(self,filename):
        print("opening ",filename)
        f = open(filename, "r")
        self.source=f.read()

    def tokenize(self,filename):
```

```
        self.open_file(filename)
        tokens=[]
        current_pos=0
        while len(self.source) > 0:
            self.source=self.source.strip() # suppress leading spaces
            #print(self.source[0:10])
            for kind,regexp in self.TOKEN_DEF.items():
                match=re.search(regexp,self.source)
                if match:
                    pos_start,pos_end=match.start(),match.end()
                    #print("match for ",kind," at ",current_pos)
                    self.source=self.source[pos_end:]
                    current_pos+=pos_end
                    tokens.append(Token(kind,match.group(0),current_pos))
                    break
        return tokens


lexer=Lexer()
tokens=lexer.tokenize(sys.argv[1])

for tok in tokens:
    print("token ",tok.kind,"\t ",tok.val,"\t",tok.pos)
```

Bibliographie

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, USA, 2nd edition, 2003.

[2] Keith Cooper and Linda Torczon. *Engineering a Compiler : International Student Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[4] Abdulaziz Ghuloum. An incremental approach to compiler construction. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR. Citeseer*. Citeseer, 2006.

[5] Tom Stuart. *Understanding computation*. O'Reilly, USA, 1st edition, 2013.

[6] David Watt and Deryck Brown. *Programming Language Processors in Java : Compilers and Interpreters*. Pearson, 1st edition, 2000.

[7] Pierre Weiss and Xavier Le Roy. *Le langage Caml*. Dunod Université, France, 1st edition, 1992.