Survol de notions de compilation

Chapitre 2

Sections 2.1, 2.2, 2.4

* Contenu *

- Description des langages de programmation
  - Grammaires hors-contexte
  - Ambiguïté
  - Associativité et priorité des opérateurs
- Analyse syntaxique
  - Analyse descendante

Survol

Les langages de programmation sont normalement décrits sous deux aspects:

- **leur syntaxe:** l'apparence des programmes et
- **leur sémantique:** la signification des programmes.

La syntaxe des langages peut être décrite clairement et simplement grâce aux grammaires hors-contexte. (Les expressions régulières contribueront aussi plus tard à la description de la syntaxe.)

Habituellement, la sémantique des langages ne peut pas être décrite à la fois clairement et simplement. La plupart du temps, la sémantique est donnée de façon informelle et est clarifiée à l'aide d'exemples.

Les grammaires hors-contexte ont un deuxième rôle. Elles peuvent aussi servir de cadre à l'élaboration d'une technique de compilation appelée traduction orientée-syntaxe.

Nous découvrons des rudiments de l'analyse syntaxique visant à reconnaître des expressions arithmétiques simples écrites en notation infixe (par ex., 9−5+2).

Définition de la syntaxe

Les grammaires hors-contexte sont utilisées pour décrire la syntaxe des langages de programmation.

Par exemple, si on dit qu'un énoncé '*if*' de C est de la forme:

> **if (** expression **)** statement **else** statement

alors on peut exprimer ce fait à l'aide d'une règle:

> *stmt* $\rightarrow$ **if (** *expr* **)** *stmt* **else** *stmt*

appelée *production* où *stmt* et *expr* sont des symboles *non-terminaux* et les autres symboles sont *terminaux*.

Une *grammaire hors-contexte* possède:

1. un ensemble de jetons, les terminaux;
2. un ensemble de non-terminaux;
3. un ensemble de productions, où chacune est une règle constituée d'un non-terminal à gauche d'une flèche et d'une suite de 0 ou plus symboles terminaux ou non-terminaux à droite; et
4. un non-terminal désigné comme étant le symbole de départ.

Exemple 2.1

La grammaire suivante permet de définir le langage des “suites de chiffres séparés par des signes d'addition ou de soustraction”, tels que 9−5+2, 3−1 et 7.

$$
\begin{array}{rcl}
\textit{list} & \rightarrow & \textit{list} + \textit{digit} \\
\textit{list} & \rightarrow & \textit{list} - \textit{digit} \\
\textit{list} & \rightarrow & \textit{digit} \\
\textit{digit} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

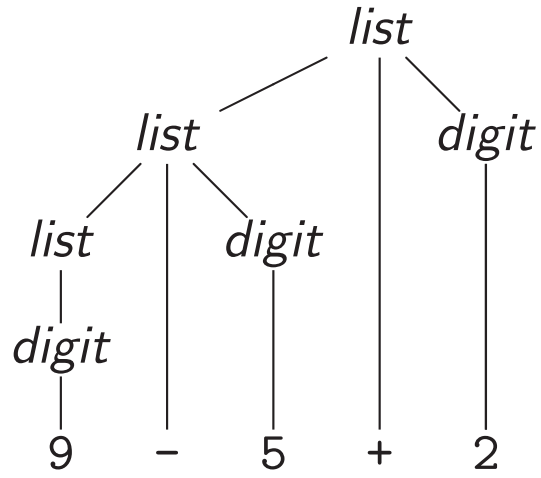
Le *langage* généré par la grammaire hors-contexte est l'ensemble des chaînes de terminaux obtenues par substitution répétitive d'un non-terminal par le membre droit d'une de ses productions et en ayant débuté par le symbole de départ.

Exemple 2.2

On peut justifier le fait que la chaîne 9−5+2 est une "*list*" par ce raisonnement:

1. 9 est une *list* car 9 est un *digit* (3ème production);
2. 9−5 est une *list* car 9 est une *list* et 5 est un *digit* (2ème production); et
3. 9−5+2 est une *list* car 9−5 est une *list* et 2 est un *digit* (1ère production).

L'arbre de dérivation suivant résume le raisonnement:



Autre exemple

De façon similaire, on peut décrire la syntaxe des blocs d'énoncés de Pascal. Ceux-ci sont une suite (possiblement vide) d'énoncés séparés par des points-virgules, le tout encadré par deux mots-clé.

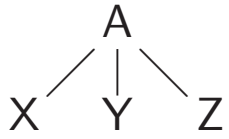
$$
\begin{array}{rcl}
\textit{block} & \rightarrow & \textbf{begin}\ \textit{opt\_stmts}\ \textbf{end} \\
\textit{opt\_stmts} & \rightarrow & \textit{stmt\_list} \mid \epsilon \\
\textit{stmt\_list} & \rightarrow & \textit{stmt\_list}\ \textbf{;}\ \textit{stmt} \mid \textit{stmt}
\end{array}
$$

La chaîne vide est dénotée par le méta-symbole ϵ.

Arbres de dérivation

Un arbre de dérivation indique de quelle façon une chaîne est dérivable à partir d'une grammaire hors-contexte.

S'il existe une production $A \to XYZ$, alors un arbre de dérivation peut contenir un noeud interne étiqueté A doté de trois enfants étiquetés X, Y et Z, dans l'ordre.



Étant donné une grammaire hors-contexte, un arbre de dérivation a les propriétés suivantes:

1. La racine est étiquetée avec le symbole de départ.
2. Chaque feuille est étiquetée avec un terminal.
3. Chaque noeud interne est étiqueté avec un non-terminal.
4. Si A est l'étiquette d'un noeud interne et que X_1, X_2, ..., X_n sont les étiquettes de ses enfants dans l'ordre, alors il existe une production $A \to X_1 X_2 \ldots X_n$. Exceptionnellement, un noeud interne peut posséder un seul enfant étiqueté avec ϵ s'il existe une production $A \to \epsilon$.

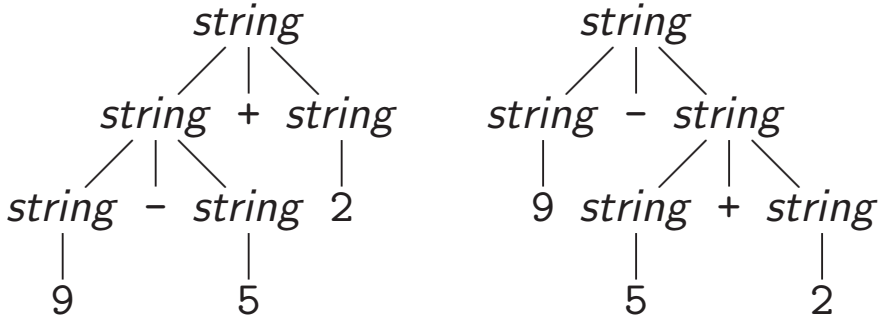
Ambiguïté

On ne peut pas toujours parler de *l'*arbre de dérivation d'une chaîne. Parfois, il en existe plus qu'un pour une même chaîne.

Par exemple, on peut aussi bien générer les listes de chiffres séparés par des opérateurs à l'aide de la grammaire suivante:

$$
\begin{array}{lcl}
\textit{string} & \rightarrow & \textit{string}\ \texttt{+}\ \textit{string} \\
\textit{string} & \rightarrow & \textit{string}\ \texttt{-}\ \textit{string} \\
\textit{string} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{array}
$$

Il existe *deux* arbres de dérivation pour la chaîne 9−5+2:



On dit qu'une telle grammaire est *ambiguë*.

Associativité des opérateurs

Par convention, les opérateurs arithmétiques `+`, `-`, `*` et `/` sont associatifs à gauche. Notre grammaire qui utilise *list* gère correctement l'associativité des additions et des soustractions.

Un opérateur qui est associatif à droite est l'opérateur d'affectation en C et en Java. Voici un fragment de grammaire qui gère correctement l'associativité de cet opérateur:

$$\begin{array}{rcl} \textit{right} & \rightarrow & \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{array}$$

Priorité des opérateurs

On pourrait facilement étendre notre grammaire basée sur *list* pour qu'elle comprenne les opérateurs de multiplication et de division.

Dans ce cas, comment interpréterait-on 9+5*2? Comme (9+5)*2 ou comme 9+(5*2)? On veut clairement la deuxième interprétation à cause de la priorité des opérateurs.

Voici une grammaire qui tient compte à la fois de l'associativité et de la priorité des opérateurs. Elle ajoute aussi les expressions entre parenthèses.

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \textit{expr} + \textit{term} \mid \textit{expr} - \textit{term} \mid \textit{term} \\
\textit{term} & \rightarrow & \textit{term} * \textit{factor} \mid \textit{term} \,/\, \textit{factor} \mid \textit{factor} \\
\textit{factor} & \rightarrow & \textbf{digit} \mid (\ \textit{expr}\ )
\end{array}
$$

Priorité et associativité en C

C-Operators in Order of Precedence

Operator	Description	
`()`	Function call	Left to right
`[]`	Array element	
`->`	Structure member pointer reference	
`.`	Class, structure or union member reference	
`sizeof`	Storage size in bytes of object / type	
`++`	Postfix Increment	
`--`	Postfix Decrement	
`++`	Prefix Increment	Right to left
`--`	Prefix Decrement	
`-`	Unary minus	
`+`	Unary plus	
`!`	Logical negation	
`~`	One's complement	
`&`	Address of	
`*`	Indirection	
`(type)`	Type conversion (cast)	Right to left
`*`	Multiplication	Left to right
`/`	Division	
`%`	Modulus	
`+`	Addition	Left to right
`-`	Subtraction	
`<<`	Bitwise left shift	Left to right
`>>`	Bitwise right shift	
`<`	Scalar less than	Left to right
`<=`	Scalar less than or equal to	
`>`	Scalar greater than	
`>=`	Scalar greater than or equal to	
`==`	Scalar equal to	Left to right
`!=`	Scalar not equal to	
`&`	Bitwise AND	Left to right
`^`	Bitwise exclusive OR	Left to right
`|`	Bitwise inclusive OR	Left to right
`&&`	Logical AND	Left to right
`||`	Logical inclusive OR	Left to right
`?:`	Conditional expression	Right to left
`=`	Assignment	Right to left
`+= -= *=`	Assignment	
`/= %= &=`	Assignment	
`^= |=`	Assignment	
`<<= >>=`	Assignment	
`,`	Comma	Left to right

Adapted from: http://www.mattmarsh.net/computing/download/cprec.doc

Source: http://www.ee.unb.ca/tervo/cmpe3221/c_precedence.pdf

Syntaxe des énoncés les plus communs

Voici une grammaire (ambiguë) qui génère des énoncés à la Pascal:

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{id} := \textit{expr} \\
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
\textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\
\textit{stmt} & \rightarrow & \textbf{while } \textit{expr} \textbf{ do } \textit{stmt} \\
\textit{stmt} & \rightarrow & \textbf{begin } \textit{opt\_stmts} \textbf{ end} \\
\textit{opt\_stmts} & \rightarrow & \textit{stmt\_list} \mid \epsilon \\
\textit{stmt\_list} & \rightarrow & \textit{stmt\_list} \textbf{ ; } \textit{stmt} \mid \textit{stmt}
\end{array}
$$

Analyse syntaxique (1)

L'analyse syntaxique:

- consiste à déterminer si une chaîne est générée par une grammaire hors-contexte;
- consiste du même coup à construire (explicitement ou non) un arbre de dérivation pour la chaîne;
- peut se faire à l'aide d'algorithmes généraux qui fonctionnent en temps $O(n^3)$ où n est le nombre de jetons, ce qui est trop coûteux pour le domaine de la compilation;
- peut se faire en temps linéaire à condition que la grammaire respecte certaines conditions.

Analyse syntaxique (2)

Caractéristiques des méthodes efficaces:

- les méthodes efficaces peuvent presque toutes être classées parmi les méthodes *ascendantes* ou *descendantes*;
- les méthodes *ascendantes* consistent à montrer comment, en faisant un assemblage à partir de la chaîne de jetons, on peut la réduire au symbole de départ;
- les méthodes *descendantes* consistent à montrer comment on peut effectuer des substitutions à partir du symbole de départ pour produire la chaîne de jetons (dérivation habituelle);
- les méthodes ascendantes sont plus complexes mais peuvent accommoder une variété plus grande de grammaires.

Analyse descendante

Cette méthode d'analyse consiste à déterminer, au fur et à mesure que la dérivation générant la chaîne est découverte, quelle est la prochaine production que l'on doit utiliser.

Par exemple, pour la grammaire suivante, il est facile de découvrir quelle est la bonne production à utiliser.

$$
\begin{array}{rcl}
\textit{type} & \rightarrow & \textit{simple} \\
 & | & \uparrow \textbf{id} \\
 & | & \textbf{array} \ [\ \textit{simple} \ ] \ \textbf{of} \ \textit{type} \\
\textit{simple} & \rightarrow & \textbf{integer} \\
 & | & \textbf{char} \\
 & | & \textbf{num dotdot num}
\end{array}
$$

L'analyse syntaxique descendante est *prédictive* lorsqu'on peut déterminer la bonne production à utiliser en n'inspectant qu'un nombre fini de jetons à l'avance (l'*horizon* ou le "*lookahead*"). Voir les fonctions FIRST et FOLLOW du chapitre 4.