1. Présentation générale
2. Analyse lexicale
   1. Rappel sur les automates et Expressions régulières
   2. LEX
3. Analyse syntaxique
   1. Analyse ascendante
   2. Analyse descendante
   3. YACC
4. Analyse sémantique
   1. Diverses classes de contrôles
   2. Grammaires attribuées
   3. Analyse sémantique avec YACC
5. Arbre abstrait
6. Machine virtuelle

Partie frontale

(Dépend du langage source mais pas du langage cible)

L’analyseur lexical (AL) reconnait les lexèmes d’un programme et la classe (unité lexicale (token)) à laquelle chacun appartient.

Ex : Delta2 :=x\*3.14 + f(4 ) ;

L’AL découvre :

* Delta2 et nous indique que c’est un identificateur (IDF)
* Des espaces et les ignore
* := l’AL nous indique que c’est un opérateur d’affectation (OPAFF)

L’analyseur syntaxique (AS) vérifie qu’une suite de lexèmes (ou plutôt d’unités lexicales) a bien été générée en respectant une grammaire.

Partie finale

(Dépend du langage cible mais pas du langage source)

Programme source -> Analyseur Lexical -> fournit une unité lexicale -> Analyseur syntaxique

Demande une unité lexicale <-

1. Quelques petits problèmes :

X :=3+f( 4 );

Y := 5 \* 2;

Affectation -> IDF OPAFF expression point-virgule

Expression -> expression PLUS e1 (élément1)

1. Reconnaissance des identificateurs

If(a+b\*(3.14-c)==42\*g(4)){…}

Mot clé if -> Mot réservé

~~Identificateur if~~

Problème surprenant pouvant apparaitre si on conçoit mal un langage de programmation

Programme Fortran de la NASA. Pilotage de la sonde Mariner vers Vénus.

a) FORTRAN : AS TE RIX = 15

b) Déclaration implicite de variable.

I -> entier

J -> entier

I3 -> entier

ASTERIX -> réel

IASTERIX -> entier

c) boucle

Do 25 I=1,10 (boucle)

Do 15 I=1.10 DO15I=1.10 (affectation)

Int analyse\_lexicale(){

Boucle{

Reconnaitre IDF

Ou Reconnaitre Entier

Ou Reconnaitre REEL

Ou Reconnaitre <,>,<=,=>, !=, ==

Ou Reconnaitre :=

}

}

Zoom sur l’analyse syntaxique

La forme des règles de grammaire détermine la classe de la grammaire dans la hiérarchie de *Chomsky.*

Grammaire context-free (hors contexte) =

Faire l’AS de 2-3+1 c’est vérifier si cette phrase peut être générée par la grammaire G2

Arbre syntaxique

* La racine est étiquetée par l’axiome de la grammaire
* Chaque feuille est étiquetée par un terminal ou rien
* Chaque nœud est étiqueté par un non terminal ou rien
* Si étiquetant un nœud dont les fils sont alors est une règle de la grammaire

Une grammaire est ambigüe s’il est possible d’avoir plusieurs arbres syntaxiques.

Priorité des opérateurs

Associativité des opérateurs de même priorité

Concevoir une grammaire non ambigüe des expressions arithmétique qui permette :

* De respecter l’associativité à gauche
* De respecter les priorités classiques
* De permettre le parenthésage

Traduction dirigée par la syntaxe

Expression arithmétique infixée -> expression arithmétique postfixée

Forme postfixée

* Si est une constante :

Postfix (*e) = e*

* Si *e* est de forme *e1 op. e2*

Postfix(*e) = postfix(e1) postfix(e2) op.*

* Si *e* est de forme (e’)

*Postfix(e) = postfix(e’)*

On parcourt l’arbre en profondeur d’abord et dès qu’on atteint une action sémantique, on l’exécute.

1. Analyse lexicale

Analyseur syntaxique

Analyseur lexical

Table lexicographique

Sauvegarde des symboles = Table lexicographique (lexèmes)

Table lexico :

* Sauvegarde les lexèmes utiles
* Attribue un numéro lexicographique (entier) à chaque lexème stocké

lexème (Astérix) = 0

Table lexico = table de hashage

Chaque lexème va être associé un hash-code puis les lexèmes de même hash-code sont regroupés ensemble.

On va utiliser toute la table et les lexèmes de même hash-code seront chainés entre eux

asterix = x42+4 ;

obelix = asterix + b12 - c

table de hash-code table lexicographique

|  |  |  |  |
| --- | --- | --- | --- |
|  | Lexeme | Long | suivant |
| 0 | asterix | 7 | 2 |
| 1 | x42 | 3 | 3 |
| 2 | obelix | 6 | -1 |
| 3 | b12 | 3 | 4 |
| 4 | B21 | 3 | -1 |
|  | c | 1 | -1 |
| 500 |  |  |  |

Pcl

Rappel sur les langages réguliers

Opérateurs sur les langages :

* Union
* Concaténation
* Fermeture de Kleene
* Fermeture positive de Kleene

Expressions régulières :

1. ɛ est une expression régulière qui dénote le langage {ɛ}
2. ɛ, x est une expression régulière qui dénote le langage {}

Priorité (|) < priorité(.) < priorité (\*)

Rappel sur les automates

E = ensemble d’états

ɛ = alphabet

trans = fonction(taille) de transition :

état initial

F = ensemble des états finaux

Voir exemple papier

Simulation d’un AFD

Trouver l’expression régulière associée à un automate.

Ecrire la définition régulière associé à un automate

2) utiliser les règles de transformations

Construction directe d’un AFD à partir d’une regex

1. Construit l’arbre abstrait de

Voir fiche

1. Numéroter les feuilles
2. Calculer annulable, première position, dernière position

|  |  |  |  |
| --- | --- | --- | --- |
| Nœud n | Annulable(n) | Première position(n) | Derniereposition(n) |
| ɛ | Vrai | 0 | 0 |
|  | Faux | {x} | {x} |
| .    a1 a2 | Annulable(a1) et annulable(a2) | Si annulable(a1)  Alors  Sinon premierepos(a1) | Si annulable(a2)  Alors  Sinon (a2) |
| a1 a2 | Annulable(a1) ou annulable(a2) |  |  |
| X  a | vrai | Premierepos(a) | Dernierepos(a) |

1. Calculer position suivante

LEX : Un générateur d’analyseurs lexicaux.

Fichier source

Fichier cible

Forme générale d’un programme LEX : %{

Déclaration c

%}

Déclaration LEX

Obligatoire

%%

Expression régulière 1 {code c1}

Expression régulière n {code c2}

Expression régulière n {code cn}

%%

Déclarations de fonctions c (facultatif)

Les expressions régulières en LEX :

On y retrouve les symboles | () \*

Pas de symbole pour la concaténation.

Reconnaitre begin : b.e.g.i.n begin

Opérateur de classe []

Tout sauf :

Opérateur ? (option)

Opérateur { }

Chiffre [0-9]

Lettre [a-zA-Z]

%%

{lettre}({lettre]|{chiffre})\*

%%

[˽\t]+ {printf("˽") ;}

%%

a{1,3} ⬄ a|aa|aaa

a{5} ⬄ aaaaa

{lettre}{10}

Opérateur de contexte

1. $ (postfix)
2. ^ (préfix)

Exp1$ reconnait les lexèmes ayant exp1 comme modèle, à condition qu’ils soient en fin de ligne

^Exp1 reconnait les lexèmes ayant exp1 comme modèle, à condition qu’ils soient en début de ligne

Un analyseur lexical peut avoir un certain état.

1. Définir des états

%start e1 e2 e3

1. Faire passer l’analyseur dans un état e : BEGIN e dans le code c associé à une regex
2. Reconnaitre un lexème ayant exp1 comme modèle à condition que l’AL soit dans l’état e

<e>exp1

%start un\_debut deux\_debut trois\_debut

%%

^1 {BEGIN un\_debut ;}

^2 {BEGIN deux\_debut ;}

^3 {BEGIN trois\_debut ;}

<un\_debut> nieme {printf(« premiere ») ;}

<deux\_debut> nieme {printf(« deuxieme ») ;}

<trois\_debut> nieme {printf(« troisieme ») ;}

Nieme {ECHO ;}

\n {BEGIN 0 ;}

%%

Opération contexte droit : /

e1/e2

reconnait les lexèmes ayant e1 comme modèle à condition qu’immédiatement à sa droite se trouve un lexème ayant e2 comme modèle

**Analyse syntaxique ascendante**

Bloc -> début ld ;li fin

Ld -> d

Ld -> d ;ld

Li -> i

Li -> i ;li

Protophrase = phrase dans laquelle il peut encore se trouver des symboles non terminaux.

Un manche d’une protophrase lambda est une règle A -> B et une position dans lambda ou la chaine béta peut être trouves et placée par A pour produire la protophrase précédente dans une dérivation la plus à droite de lambda.

Les analyse syntaxique ascendante qui fonctionnent selon ce principe sont appelés analyseurs LR(k) :

Left to right scanning constructing a right most derivation inverse.

K= nombre de symbole à lire pout prendre une décision d’action.

3 classes principales d’analyseurs

SLR(1) simple LR(1)

LALR Look Ahead LR(1) 🡸YACC génère de tels analyseurs

LR(1)

YACC (Yet Another Compiler Compiler)

YACC est un générateur d’analyseur syntaxiques.