Table d'analyse LL(1)

PCL - Travail de la grammaire

1 Algorithmes de calcul

1.1 Calcul des premiers

Rappel 1.1 : Premier

Pour tout mot $\omega \in (N \cup T)^*$, on appelle premiers de ω et on note $Premier(\omega)$, l'ensemble :

$$Premier(\omega) = \{a \in T \mid \exists\ \beta \in (N \cup T)^*\ ;\ \omega \xrightarrow{*} a\beta\}$$

Soit $\omega \in (N \cup T)^*$ un mot non vide. On a alors $\omega = \alpha_1\alpha_2 \dots \alpha_n \in (N \cup T)^*$. Pour identifier les premiers de ω on peut utiliser l'algorithme suivant :

Algorithme 1 : Calcul des "premiers" de ω

Entrées : $\omega = \alpha_1\alpha_2 \dots \alpha_n \in (N \cup T)^*$
Sorties : $Premier(\omega)$
si $\alpha_1 \in T$ **alors**
  $Premier(\omega) = \{\alpha_1\}$;
sinon
  $Premier(\omega) = \emptyset$;
  pour tous les $\alpha_1 \to \beta$ **faire**
    $Premier(\omega) \leftarrow Premier(\omega) \cup Premier(\beta)$;
  pour i *de 1 à* $n-1$ **faire**
    si $\alpha_1, \dots, \alpha_i \in Vide$ **alors**
      $Premier(\omega) \leftarrow Premier(\omega) \cup Premier(\alpha_{i+1})$;

1.2 Calcul des suivants

Rappel 1.2 : Suivant

Soit $G = (N, T, \to, X)$ une grammaire algébrique. Pour tout non-terminal $E \in N$, on appelle suivant de E et on note $Suivant(E)$ l'ensemble :

$$Suivant(E) = \{a \in T \mid S \to \alpha E a \beta\} \ \cup\ \{\$ \mid X \xrightarrow{*} \alpha E\}$$

Ici α et β représentent n'importe quel élément de $(N \cup T)^*$.

Soit $E \in N$ un non-terminal de G. Pour identifier les suivants de E on peut utiliser l'algorithme suivant :

Algorithme 2 : Calcul des "suivants" d'un non-terminal

Entrées : Non-terminal E
Sorties : $Suivant(E)$, l'ensemble des "suivants" de E
pour tous les $A \to \alpha E \beta$ *(cas direct)* **faire**
  $Suivant(E) \leftarrow Suivant(E) \cup Premier(\beta)$;
pour tous les $A \to \alpha E$ *ou* $A \to \alpha E \beta$ *tels que* $\beta \in Vide$ *(cas final)* **faire**
  $Suivant(E) \leftarrow Suivant(E) \cup Suivant(A)$;
si E *est l'axiome* **alors**
  $Suivant(E) \leftarrow Suivant(E) \cup \{\$\}$;

1.3 Calcul des symboles directeurs

Rappel 1.3 : Symboles directeurs

Soit $G = (N, T, \to, X)$ une grammaire algébrique. Pour toute règle de production $A \to \alpha$ dans G, on appelle symboles directeurs de $A \to \alpha$ et on note $SD(A \to \alpha)$ l'ensemble :

$$SD(A \to \alpha) = \begin{cases} Premier(\alpha) & \text{si } non\ Vide(\alpha) \\ Premier(\alpha) \cup Suivant(A) & \text{si } Vide(\alpha) \end{cases}$$

$Vide$ est la fonction qui vaut Vrai s'il existe une dérivation de α vers le mot vide.

Soit $A \to \alpha$ une règle de la grammaire G. Pour identifier les symboles directeurs de $A \to \alpha$, on peut utiliser l'algorithme suivant :

Algorithme 3 : Calcul des symboles directeurs d'une règle

Data : Règle $A \to \alpha$
Result : $SD(A \to \alpha)$
$SD(A \to \alpha) \leftarrow Premier(\alpha)$;
si $Vide(\alpha)$ **alors**
  $SD(A \to \alpha) \leftarrow SD(A \to \alpha) \cup Suivant(A)$;

2 Construction de la Table d'analyse LL(1)

Pour construire la table d'analyse LL(1) on peut utiliser l'algorithme suivant :

Algorithme 4 : Construction de la table d'analyse LL(1)

Entrées :
Grammaire algébrique $G = (N, T, \to, X)$ **Sorties :** Table d'analyse Tab
`/* Chaque ligne correspond à un non-terminal */`
`/* Chaque colonne correspond à un terminal */`
pour tous les $A \to \alpha$ **faire**
  pour tous les $d \in SD(A \to \alpha)$ **faire**
    $Tab(A, d) \leftarrow$ "$A \to \alpha$";

3 Symboles directeurs des règles de grammaire du canAda

Dans les sections suivantes, on a considéré que les unités lexicales reconnues par le lexer sont des terminaux.

3.1 Résultats du calcul des premiers

- Premier(<fichier>) = {with}

- Premier(<decl>) = {type | <ident> | procedure | function}

- Premier(<decl1>) = {; | is}

- Premier(<decl2>) = {access | record}

- Premier(<champs>) = {<ident>}

- Premier(<type>) = {<ident>}

- Premier(<params>) = {(}

- Premier(<param>) = {<ident>}

- Premier(<mode>) = {in}

- Premier(<mode1>) = {out}

- Premier(<expr>) = {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character }

- Premier(<expr1>) = {= | / | < | > | + | - | * | rem}

- Premier(<instr>) = {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character | return | begin | if | for | while }

- Premier(<instr1>) = {; | (}

- Premier(<operateur>) = {= | / | < | > | + | - | * | rem}

- Premier(<operateur1>) = {=}

- Premier(<acces>) = {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character}

- Premier(<acces1>) = {= | / | < | > | + | - | * | rem | .}

- Premier(<acces2>) = {(| = | / | < | > | + | - | * | rem | .}

3.2 Résultats du calcul des suivants

- Suivant(<fichier>) = {$}

- Suivant(<decl>) = {*}

- Suivant(<decl1>) = {*}

- Suivant(<decl2>) = {*}

- Suivant(<champs>) = {+}

- Suivant(<type>) = {; | (| is}

- Suivant(<params>) = {?}

- Suivant(<param>) = {;}

- Suivant(<mode>) = {?}

- Suivant(<mode1>) = {?}

- Suivant(<expr>) = {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem }

- Suivant(<expr1>) = {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem }

- Suivant(<instr>) = {+}

- Suivant(<instr1>) = {+}

- Suivant(<operateur>) = {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character }

- Suivant(<operateur1>) = {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character }

- Suivant(<acces>) = {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem | :}

- Suivant(<acces1>) = {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem | :}

- Suivant(<acces2>) = {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem | :}

3.3 Identification des symboles directeurs

- SD(<fichier> ::= with Ada.Text_IO; use Ada.Text_IO;
  procedure <ident> is <decl>*
  begin <instr>+ end <ident>? ; EOF) = {with}

- SD(<decl> ::= type <ident> <decl1>) = {type}

- SD(<decl> ::= <ident>,+ : <type>(:= <expr>)? ;) = {<ident>}

- SD(<decl> ::= procedure <ident> <params> ? is <decl>*
  begin <instr>+ end <ident>?;) = {procedure}

- SD(<decl> ::= function <ident> <params>? return <type> is <decl>*
  begin <instr>+ end <ident>?;) = {function}

- SD(<decl1> ::= ;) = {;}

- SD(<decl1> ::= is <decl2>) = {is}

- SD(<decl2> ::= access <ident> ;) = {access}

- SD(<decl2> ::= record <champs>+ end record;) = {record}

- SD(<champs> ::= <ident>,+ : <type>;) = {<ident>}

- SD(<type> ::= <ident>) = {<ident>}

- SD(<params> ::= (<param>;+)) = {(}

- SD(<param> ::= <ident>,+ : <mode>? <type>) = {<ident>}

- SD(<mode> ::= in <mode1>) = {in}

- SD(<mode1> ::= out) = {out}

- SD(<mode1> ::= ^) = {?}

- SD(<expr> ::= <entier><expr1>) = {<entier>}

- SD(<expr> ::= <caractere><expr1> true<expr1>) = {<caractere>}

- SD(<expr> ::= false<expr1>) = {false}

- SD(<expr> ::= null<expr1>) = {null}

- SD(<expr> ::= (<expr>)<expr1>) = {(}

- SD(<expr> ::= <acces><expr1>) = {<acces>}

- SD(<expr> ::= not<expr><expr1>) = {not}

- SD(<expr> ::= -<expr><expr1>) = {-}

- SD(<expr> ::= new<ident><expr1>) = {new}

- SD(<expr> ::= <ident>(<expr>,+)<expr1>) = {<ident>}

- SD(<expr> ::= character ' val (<expr>)<expr1>) = {character}

- SD(<expr1> ::= <operateur><expr><expr1>)
= {= | / | < | > | + | - | * | rem}

- SD(<expr1> ::= ^)
= {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem }

- SD(<instr> ::= <acces> := <expr> ;)
= {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character}

- SD(<instr> ::= <ident> <instr1>) = {<ident>}

- SD(<instr> ::= return <expr>? ;) = {return}

- SD(<instr> ::= begin <instr>+ end;) = {begin}

- SD(<instr> ::= if <expr> then <instr>+ (elsif <expr> then <instr>+)*
(else <instr>+)? end if;) = {if}

- SD(<instr> ::= for <ident> in reverse? <expr> .. <expr>
loop <instr>+ end loop;) = {for}

- SD(<instr> ::= while <expr> loop <instr>+ end loop;) = {while}

- SD(<instr1> ::= ;) = {;}

- SD(<instr1> ::= (<expr>,+);) = {(}

- SD(<operateur> ::= =) = {=}

- SD(<operateur> ::= / <operateur1>) = {/}

- SD(<operateur> ::= < <operateur1>) = {<}

- SD(<operateur> ::= > <operateur1>) = {>}

- SD(<operateur> ::= +) = {+}

- SD(<operateur> ::= -) = {-}

- SD(<operateur> ::= *) = {*}

- SD(<operateur> ::= rem) = {rem}

- SD(<operateur1> ::= =) = {=}

- SD(<operateur1> ::= ^)
= {<entier> | <caractere> | true | false | null | (| not | - | new | <ident> | character }

- SD(<acces> ::= <entier><expr1>.<ident><acces1>) = {<entier>}

- SD(<acces> ::= <caractere><expr1>.<ident><acces1>) = {<caractere>}

- SD(<acces> ::= true<expr1>.<ident><acces1>) = {true}

- SD(<acces> ::= false<expr1>.<ident><acces1>) = {false}

- SD(<acces> ::= null<expr1>.<ident><acces1>) = {null}

- SD(<acces> ::= (<expr>)<expr1>.<ident><acces1>) = {(}

- SD(<acces> ::= not<expr><expr1>.<ident><acces1>) = {not}

- SD(<acces> ::= -<expr><expr1>.<ident><acces1>) = {-}

- SD(<acces> ::= new<ident><expr1>.<ident><acces1>) = {new}

- SD(<acces> ::= <ident> <acces2>) = {<ident>}

- SD(<acces> ::= character ’ val (<expr>)<expr1>.<ident><acces1>) = {character}

- SD(<acces1> ::= <expr1>.<ident>.<acces1>)
= {= | / | < | > | + | - | * | rem | .}

- SD(<acces1> ::= ^)
= {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem | :}

- SD(<acces2> ::= (<expr>,+)<expr1>.<ident><acces1>) = {(}

- SD(<acces2> ::= <acces1>)
= {) | true | , | ; | ? | then | . | loop | = | / | < | > | + | - | * | rem | :}