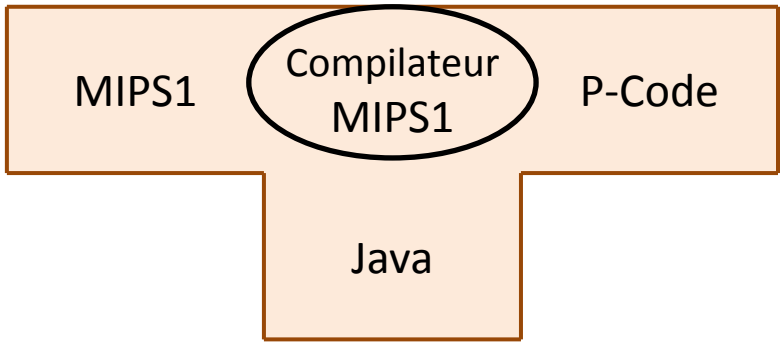
Chap. 1 : Analyseur lexical

Introduction

- On définit un petit langage qui s'inspire de Pascal.
- Une première version aura pour nom MIPS1 (Mini Pascal 1).
- Le but est la construction d'un compilateur qui génère du P-Code à partir d'un programme MIPS1.
- La figure suivante montre que le langage source est MIPS1, le langage cible est le P-Code et le langage hôte est Java.



Syntaxe du langage MIPS1

- PROGRAM ::= **program** ID **;** BLOCK **.**
- BLOCK ::= CONSTS VARS INSTS
- CONSTS ::= **const** ID **=** NUM **;** { ID **=** NUM **;** } | ε
- VARS ::= **var** ID { **,** ID } **;** | ε
- INSTS ::= **begin** INST { **;** INST } **end**
- INST ::= INSTS | AFFEC | SI | TANTQUE | ECRIRE | LIRE | ε
- AFFEC ::= ID **:=** EXPR
- SI ::= **if** COND **then** INST
- TANTQUE ::= **while** COND **do** INST
- ECRIRE ::= **write (** EXPR { **,** EXPR } **)**
- LIRE ::= **read (** ID { **,** ID } **)**
- COND ::= EXPR RELOP EXPR
- RELOP ::= **=** | **<>** | **<** | **>** | **<=** | **>=**
- EXPR ::= TERM { ADDOP TERM }
- ADDOP ::= **+** | **-**
- TERM ::= FACT { MULOP FACT }
- MULOP ::= ***** | **/**
- FACT ::= ID | NUM | **(** EXPR **)**

Règles lexicales

- ID ::= lettre {lettre | chiffre}
- NUM ::= chiffre {chiffre}
- Chiffre ::= **0**|..|**9**
- Lettre ::= **a**|**b**|..|**z**|**A**|..|**Z**

Méta-règles

Une série de règles définissent la forme d'un programme MIPS1.

- Un commentaire est une suite de caractères encadrés des parenthèses (* et *)
- Un séparateur est un caractère séparateur (espace blanc, tabulation, retour chariot) ou un commentaire
- Deux ID ou mots clés qui se suivent doivent être séparés par au moins un séparateur
- Des séparateurs peuvent être insérés partout, sauf à l'intérieur de terminaux
- Il n'y a pas de distinctions entre les majuscules et les minuscules.
- Toutes les variables sont implicitement déclarées de type entier.

Exemple de programme MIPS1

```
program test1;
const ta1=21; ta2=13;
var   x, y;
begin
      (* initialisation de x *)
      x:=ta1;
      read(y);
      while x<y do begin read(y); x:=x+y+ta2 end;
      (* affichage des resultats de x et y *)
      write(x);
      write(y)
 end.
```

Implémentation de l'analyseur lexical

Définition du type représentant les codes des symboles du langage :

- TYPE **TOKENS** = (ID_TOKEN, NUM_TOKEN, PLUS_TOKEN, MOINS_TOKEN, MUL_TOKEN, DIV_TOKEN, EG_TOKEN, DIFF_TOKEN, INF_TOKEN, SUP_TOKEN, INFEG_TOKEN, SUPEG_TOKEN, PARG_TOKEN, PARD_TOKEN, VIR_TOKEN, PVIR_TOKEN, PNT_TOKEN, AFFEC_TOKEN, BEGIN_TOKEN, END_TOKEN, IF_TOKEN, WHILE_TOKEN, THEN_TOKEN, DO_TOKEN, WRITE_TOKEN, READ_TOKEN, CONST_TOKEN, VAR_TOKEN, PROGRAM_TOKEN, ERR_TOKEN)

Implémentation de l'analyseur lexical

Définition du type représentant les symboles du langage :

- STRUCTURE **SYMBOLES** {

  TOKEN : **TOKENS** (*Contient le token du symbole *)

  NOM : **CHAINE[8]** (*Contient la forme textuelle du symbole*)

  }

Implémentation de l'analyseur lexical

Déclarations des variables :

- L'analyseur lexical est une **procédure** appelée par l'analyseur syntaxique.
- Chaque appel à la procédure d'analyse lexicale **SYMB_SUIV()** met à jour SYMB_COUR.
- SYMB_COUR : **SYMBOLES** (* contient les informations du dernier symbole lu *)
- CAR_COUR : **CARACTERE** (*contient le dernier caractère lu dans le flux d'entrée*)

Implémentation de l'analyseur lexical

Déclarations des procédures:

- **LIRE_CAR()** (* Lit le caractère courant dans le flux d'entrée, met à jour CAR_COUR *)
- **SYMB_SUIV()** (* procédure d'analyse lexicale, identifie le symbole courant dans le programme source et met à jour SYM_COUR*)
- **LIRE_MOT()** (* une sous-procédure de **SYMB_SUIV** qui reconnait les mots clés et les identificateurs *)
- **LIRE_NOMBRE()** (* une sous-procédure de **SYMB_SUIV** qui reconnait les nombres entiers *)

Implémentation de l'analyseur lexical

Algorithme d'analyse lexicale :

```
PROCEDURE LIRE_NOMBRE()
debut
   (*implémenter la règle lexicale*)
   (* NUM  ::=  chiffre {chiffre}  *)
fin

PROCEDURE LIRE_MOT()
debut
   (*implémenter la règle lexicale*)
   (* ID  ::=   lettre {lettre | chiffre} *)
   si (SYMB_COUR est un mot clé) alors
      (*affecter à SYMB_COUR le token du mot clé*)
   sinon
      (*affecter à SYMB_COUR ID_TOKEN *)
fin
```

NOM	TOKEN
program	PROGRAM_TOKEN
var	VAR_TOKEN
…	…

Implémentation de l'analyseur lexical

Algorithme d'analyse lexicale :

```
PROCEDURE SYMB_SUIV()
debut
   (*sauter les séparateurs*)
   (*traiter selon la catégorie*)
   si (CAR_COUR est une lettre) alors
      LIRE_MOT()
   si (CAR_COUR est un chiffre) alors
      LIRE_NOMBRE()
   cas  CAR_COUR  parmi
      '+' : SYMB_COUR.TOKEN ← PLUS_TOKEN
      . . .
    fin cas
fin
```

Implémentation de l'analyseur lexical

Gestion des erreurs lexicales :

- CODES_ERR est un type énuméré
- ERRORS est une structure composé des champs, COD_ERR et MES_ERR
- Mettre les erreurs dans un tableau TAB_ERREURS, le code de l'erreur est identique à son indice dans le tableau.

```
PROCEDURE ERREUR(code : CODES_ERR)
debut
   (*Afficher le message d'erreur correspondant au code
      et arrêter le programme*)
fin
```

COD_ERR	MES_ERR
CAR_INC_ERR	"caractère inconnu"
FIC_VIDE_ERR	"fichier vide"
. . .	. . .

Exemple de test de l'analyseur lexical

```
program test11;
const ta=21;
 var x,y;
 Begin
    x:=ta;
    read(y);
 end.
```

ANALYSEUR LEXICAL

PROGRAM_TOKEN
ID_TOKEN
PVIR_TOKEN
CONST_TOKEN
ID_TOKEN
EG_TOKEN
NUM_TOKEN
PVIR_TOKEN
VAR_TOKEN
ID_TOKEN
VIR_TOKEN
ID_TOKEN
PVIR_TOKEN
BEGIN_TOKEN
ID_TOKEN
AFF_TOKEN
ID_TOKEN
PVIR_TOKEN
READ_TOKEN
PARG_TOKEN
ID_TOKEN
PARD_TOKEN
PVIR_TOKEN
END_TOKEN
PNT_TOKEN
EOF_TOKEN

Travail à faire : Analyseur lexical

- Déclarer les tokens sous forme d'un ensemble énuméré : TOKENS.
- Déclarer le type des erreurs sous forme d'un ensemble énuméré : CODES_ERR.
- Déclarer la table des erreurs (code erreur, message associé).
- Déclarer le type symbole.
- Déclarer la table des mots clés.
- Déclarer les variables globales (symb_cour, car_cour, fich_sour).
- Ecrire une procédure qui initialise la table des mots clés.
- Ecrire la procédure ERREUR(code d'erreur) qui affiche le message d'erreur associé et stoppe l'exécution.
- Ecrire la procédure Codage_Lex qui reçoit en entrée la valeur d'un mot et retourne son token.
- Ecrire la procédure LIRE_CAR(), LIRE_MOT(), LIRE_NOMBRE, SYMB_SUIV(), etc.
- Tester votre analyseur lexical