Projet Compilation: Création d'un nouvelle langage de programmation

Dictel

portabilité élevée et programmation memory-driven

Génie Logiciel

Réalisé Par

DOUSLIMI Yassir (GL 1)

EDDAGHAL Mohammed (GL 1)

EL BIACHE Houda (GL 1)

EL FAKHORI Fouad (GL 2)

EL ARFAOUI Ikrame (GL 1)

EL AZHAR Asmaa (GL 1)

Encadré Par

M. OULAD HAJ THAMI Rachid
Mr TABII Youness

Année Universitaire 2020 – 2021

Remerciement

Nous tenons à remercier avec une grande ferveur tous ceux qui nous ont soutenus directement ou indirectement pendant ce projet et sa réalisation.

Nous voulons également exprimer notre profonde gratitude à nos encadrants Mr OULAD HAJ THAMI Rachid et Mr TABII Youness pour leur aide en nous éclairant sur tout ce qui concerne les compilateurs et aussi leurs conseils pendant ce travail.

Enfin, nous devons remercier le corps professoral, car il a été à nos côtés pendant ces moments difficiles de la pandémie. Nous aurions aimé travailler sur ce projet directement avec nos professeurs, mais même dans ces circonstances, ils nous ont quand même soutenus.

Résume

Le but de ce projet est de concevoir un nouveau langage et d'explorer les façons dont il pourrait améliorer le paysage informatique. Ce document décrira l'évolution de la conception du langage Dictel et les différentes étapes que nous avons prises pour concevoir un mini-compilateur pour celui-ci. Le rapport abordera d'abord l'analyse et la conception, puis l'aspect réalisation du projet.

Abstract

The goal of this project is the conception of a new language and exploring the ways it could improve the computer science landscape. This document will describe the evolution of the conception of the Dictel language and the different steps we took to engineer a mini-compiler for it. The report will touch on analysis and conception first and then the realization aspect of the project.

Table de Matières

Chapitre 1

Introduction

Un langage de programmation est un langage formel comprenant un ensemble d'instructions qui produisent divers types de sortie. Ils sont utilisés dans l'informatique pour implémenter des algorithmes.

Des milliers de langages de programmation ont été créés et d'autres sont créés chaque année. ils peuvent être compilés, interprétés ou bien semi-interprétés. La majorité est écrite sous une forme impérative (c'est-à-dire comme une séquence d'opérations à effectuer) tandis que d'autres langages utilisent la forme déclarative (c'est-à-dire que le résultat souhaité est spécifié, pas comment y parvenir).

La description d'un langage de programmation est généralement divisée en trois composants: lexique(vocabulaire), syntaxe (forme) et sémantique (signification). Certains langages sont définis par un document de spécification (par exemple, le langage de programmation C est spécifié par une norme ISO) tandis que d'autres langages (comme Perl) ont une implémentation dominante qui est traitée comme une référence. Certains langages ont les deux, le langage de base défini par un standard et les extensions tirées de l'implémentation dominante étant communes.

Dans ce contexte, nous allons développer un compilateur du langage Dictel. On procédera d'abord par introduire ce langage, puis nous passerons à l'implémentation de l'analyseur lexical, puis nous aborderons l'analyseur syntaxique.

Chapitre 2

Analyse et Conception

Dans ce chapitre, nous présenterons notre analyse du langage souhaité qui nous conduira à une implémention qui répond a nos objectives.

2.1 Motivation et présentation du langage Dictel

De nombreux langages de programmation sont conçus d'une manière standard qui permet à l'utilisateur d'assumer la responsabilité de gérer ses variables manuellement. Dictel (le langage à réaliser) est un langage qui offrira des outils pour aider à faire les choses de manière plus sécurisée et concise. De plus, il n'existe pas de moyen standardisé pour traduire un langage à une autre. Dictel peut aussi fonctionner comme une couche intermédiaire afin que les programmes puissent devenir facilement portables puisque le langage ne stocke que des paires clé-valeur de variables qui composent le script. Cela peut également inclure des fonctions et bien sûr la fonction main.

2.2 Objectifs

Le langage que nous voulons créer doit répondre à ces buts suivantes:

- Créer un langage qui brouille les frontiéres entre les variables et les fonctions et d'autres outils de codage en introduisant une structure unique qui stocke et suit tous les composants d'un programme.
- Définir un cadre général qui peut améliorer la façon dont nous échangeons des algorithmes et des scripts en tirant parti d'un middleware que tout autre langage peut interpréter facilement.
- Amélioration de la façon dont nous stockons les variables en mémoire en ajoutant de nouveaux paradigmes qui gèrent le heap et le stack d'une maniere plus efficace.
- Mettre l'accent sur la réutilisabilité du code en mettant à niveau les blocs conditionnels et itératifs en pseudo-fonctions pouvant être appelées plusieurs fois.
- Simplifier la concurrence et le threading en ajoutant l'option pour tout bloc donné à s'exécuter de manière asynchrone dans son propre thread géré par le langage. Cela rendra nécessaire d'introduire un type de données responsable du transfert des données via les pipes. Le langage utilisera les pipes de C pour introduire une approche plus intuitive pour son utilisation.
- Le but le plus ambitieux de ce projet et le plus intéressant est de créer des scripts simplement en spécifiant quelques variables. Puisque le langage est créé par une structure de données unique, cette dernière serait capable de construire des instructions de manière autonome.

2.3 Lexique du langage Dictel

Dans tout langage informatique, les variables fournissent un moyen d'accéder aux données stockées en mémoire. Dictel ne permet pas d'accéder directement à la mémoire de l'ordinateur mais fournit plutôt une méthode conçu pour manipuler nos variables. Celle-ci contient plusieurs paramètres qui permettent de configurer exactement l'entité a créer.

2.3.1 Création d'un variable avec inference de type

```
add(monVariable, 45)
```

2.3.2 Création d'un variable sans initialisation (null par defaut)

```
add(monVariable)
```

2.3.3 Création d'un variable pour une seul utilisation avant d'etre liberer

```
add(monVariable, 45, use=1)
```

2.3.4 Création d'un variable avec un type pour eviter l'inference du type

```
add(monVariable, 45, type="int")
```

2.3.5 Création d'un variable avec un allocation dynamique(en octet)

```
add(monVariable, allocate=5*sizeof(int))
```

2.3.6 Création d'un variable immutable(constante)

```
add(monVariable, 45, state="immutable")
```

2.3.7 Création d'un variable non-nullable(raise error if monVariable is null)

```
add(monVariable, 45, state="nonNullable")
```

2.3.8 Création d'un variable de type tableau

```
add(monVariable, [45, 46, 47], type="int[]")
```

2.3.9 Création d'une fonction

```
add(maFonction, (monVariable) => {
    monVariable = monVariable + 1
})
```

2.3.10 Création d'une fonction asynchrone

```
add(maFonction, (monVariable) => {
    monVariable = monVariable + 1
}, run="asynchronous")
```

2.3.11 Création d'un pipe

```
add(monPipe, type="pipe")
```

2.3.12 Création d'un bloc conditionnel

```
add(monIf, (condition) ? {
    monVariable = monVariable + 1
} : {
    monVariable = monVariable - 1
}, kind="if")
```

2.3.13 Création d'un bloc itératif

```
add(monFor, (iterator) => {
    monVariable = monVariable + 1
}, kind="for")
```

2.3.14 Création d'un bloc itératif conditionnel

```
add(monWhile, (condition) => {
    monVariable = monVariable + 1
}, kind="for")
```

Remarque: On peut également utiliser le for et le if et le while traditionnel sans aucun problème

```
while(condition){
    for(int i=0; i<6; i++){
        if(condition){
            monVariable = monVariable + 1
        } else {
            monVariable = monVariable + 1
        }
    }
}
```

2.3.15 Opérateur ternaire

```
monVariable = true ? 1 : 0  // monVariable sera donc égale à 1
```

2.3.16 Création d'un range (interval)

```
add(monRange, 1..100)
```

2.3.17 Création d'un switch

```
add(monSwitch, (monNombre) => {
    1 -> monVariable = 1
    2 -> monVariable = 2
    3 -> monVariable = 3
    _ -> monVariable = 0  // valeur par défaut
}, kind="when")
```

2.3.18 Création d'une structure

```
add(maStructure, (monComposant1= 5, monComposant3= 6.5f) => {
    monComposant1 -> int
    monComposant2 -> string
    monComposant3 -> float
    monComposant4 -> int[]
}, kind="struct")
```

2.3.19 Modification d'un variable

```
monVariable = 500
```

2.3.20 Supprimer un variable de la mémoire

```
del(monVariable)
```

2.3.21 Voir les paramètres d'une variable

```
params(monVariable) // type, memoire, TTL, etc.
```

2.3.22 Afficher du texte sur la console

```
log("Hello World")
```

2.3.23 Lire la saisie d'un utilisateur

```
monNom = scan()
```

2.3.24 Afficher du texte sur la console avec interpolation

```
log("Hello ${monNom}")
```

2.4 Grammaire du langage Dictel (LL1)

On définit notre grammaire comme suit:

```
G = { T, NT, S, P } avec:
```

2.4.1 Terminaux

```
T = { 'acf','aco',',','+','-','*','/','=','!','<','>','<=','>=','?','void',
'char','short','int','float','long','double','signed','unsigned','string',
'pipe','type','immutable','unmmutable','null','if','for','when','while',
'struct','use','run','synchronous','asynchronous','sizeof','allocate',
'break','->','log','scan',':','..','return','params',')','(','add',
'erreur','eof','function','state','[',']','"','kind','$','=>','del' }
```

2.4.2 Non Terminaux

```
NT = {'ID', 'NUM', 'Chiffre', 'Lettre', 'PROGRAM', 'ADD', 'KIND,
VALUE', 'TYPE', 'STATE', 'ALLOCATE', 'USE', 'RUN', 'K', 'FUNCTION,
IFDEF', 'FORDEF', 'STRUCT', 'SWITCHDEF', 'ARRAY', 'PARMS', 'S', 'R', 'ALLOCATE,
TYPES', 'INSTS', 'INST', 'WRITE', 'BLOCIF', 'AFFEC', 'READ', 'BLOCFOR', 'BLOCWHILE,
WHENINST', 'RETURN', 'PARINST', 'DELETEINST', 'CONDITION', 'RELOP', 'EXPR,
ADDOP', 'TERM', 'MULOP', 'FACT'}
```

2.4.3 Les règles de production

```
Axiome : { PROGRAM }

# **Some tokens definitions :**

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

aco = '{'
acf = '}'

# ** La grammaire P **:

PROGRAM => ADD { ADD }
ADD => add ( ID KIND VALUE TYPE STATE ALLOCATE  USE RUN )
-----------------------------------------------------------
KIND  => epsilon | ,kind=" K
K => if", IFDEF |for ", FORDEF |when", SWITCHDEF | struct", STRUCT |function", FUNCTION

FUNCTION => (PARMS) => aco INSTS acf
IFDEF    => ( CONDITION ) ? aco INSTS acf  : aco INSTS acf
FORDEF   => (ID) => aco INSTS acf
STRUCT   => ( AFFEC { , AFFEC} ) => aco ID -> TYPES { ID -> TYPES } acof
SWITCHDEF => (ID) => aco INSTS acf

VALUE => ,VAL | epsilon
  VAL   =>  EXPR
      => [ARRAY]
      =>  NUM..NUM            //range

ARRAY =>  EXPR { , EXPR } | epsilon
PARMS => epsilon| ID { , ID }

-------------------------------------------
TYPE  => epsilon | , type = " TYPES {[]} "
```

```
STATE => epsilon | , state = " S "
S    => immutable|nonNullable|nullable

USE   => epsilon | , use = NUM
RUN   => epsilon | , run=" R "
R => asynchronous|synchronous
ALLOCATE => epsilon |, allocate = " chiffre * sizeof ( TYPES ) "
-----------------------------------------------------------------------------------------------
TYPES => void | char | short | int | long | float | double | signed | unsigned | string | pipe
-----------------------------------------------------------------------------------
INSTS =>  INST { INST }
INST  => epsilon | WRITE | BLOCIF | BLOCFOR | BLOCWHILE | AFFEC | ADD | RETURN | break |
WHENINST | PARINST | DELETEINST
-------------------------------------------------------
WRITE  => log( " { {ID|symbole|chiffre} $ aco ID acf {ID|symbole|chiffre}  } " | ID )

BLOCIF  => if ( CONDITION ) aco INSTS acf
=> CONDITION ? aco INSTS acf : aco INSTS acf
AFFEC   => ID = READ
READ => EXPR | scan()

BLOCFOR  for (AFFEC , CONDITION , INST) aco INSTS acof
BLOCWHILE =>  while ( CONDITION ) aco INSTS acof
WHENINST => NUM -> INST
RETURN => return EXPR
PARINST => params ( ID )
DELETEINST => del ( ID )
-----------------------------------------------------------------------------
CONDITION  => EXPR RELOP EXPR
RELOP => = | <> | < | > | <= | >=
EXPR => TERM { ADDOP TERM }
ADDOP => + | -
TERM => FACT { MULOP FACT }
MULOP => * | /
FACT => ID | NUM | ( EXPR )
-------------------------------------------------------------------
```

Chapitre 3

Conclusion et perspectives

Ce projet visait à créer un prototype de langage de programmation. Il est passé par de nombreuses étapes comme la conception et la mise en œuvre avec les analyseurs lexicaux et syntaxiques.

C'était un projet très fructueux et nous avons beaucoup appris sur le fonctionnement d'une langue et les différentes nuances qui résument sa création.

Nous espérons que ce ne sera pas la fin de cette langue mais un point de départ pour plus. Nous espérons que grâce à de futures améliorations impliquant la sémantique et la génération de p-code, ce langage deviendra vraiment ce que nous avons imaginé. Nous pensons sûrement que ce sera une énorme amélioration pour la communauté informatique.