Rapport de projet
Sujet NNP - MV

Chakiya Ahamada
Yasmine Bagtache
Thomas Gerbaud
Haydar Selmi
Laurent Lin

ENSSAT LANNION | Université de Rennes

Table des matières

Liste des figures

1 Introduction

Notre projet consiste à créer et programmer un langage de programmation nommé NilNovi, ce langage peut traiter en paradigme procédurale. Le projet est intégralement codé en Python et on a commencé avec une partie du code déjà présente qui offre un module d'analyseur syntaxique et un module de d'analyseur lexical.

Afin de finaliser le langage on a travaillé sur plusieurs modules du langage y compris l'ajout d'une table des identificateurs, la génération de code et la programmation des points de génération, la création d'une machine virtuelle (VM) et sans oublier l'ajout de message d'erreur de compilation pour des anomalies sémantiques par exemple.

On va introduire dans le rappot le déroulement de conception et de programmation de chaque module ainsi que les liens entre ces modules, c'est-à-dire une des problématique de ce projet était de bien choisir dans le bon ordre afin de ne pas ralentir le développement global, un exemple avant de pouvoir complètement tester la machine virtuelle il fallait que la partie génération de code aient été finalisée. C'est ainsi qu'on expliquera dans ce rapport notre organisation et méthodologie de travail ainsi que les habitudes qu'on a intégrées au sein de notre groupe.

2 Table des identificateurs

La table des identificateurs (TDI) est une technique permettant de faciliter et d'accélérer la compilation. Le but est de rassembler dans une et une seule stucture de données l'ensemble des identificateurs d'un programme informatique souvent une table. On stocke dans cette structure toutes les informations sur un identificateur, par exemple pour une entrée d'une variable on aura le nom de la variable ainsi que son adresse statique.

Dans les parties suivantes on explique notre implémentation de la TDI.

2.a Implémentation

La table des identificateurs est implémentée par un dictionnaire Python, nommé `identifierTable`.

2.b Structure

Ce dictionnaire contient les tables des identificateurs, sous forme de dictionnaires, propres à chaque contexte, indexées par le nom de ce contexte, c'est à dire :

- le nom de la procédure principale pour la première entrée, la table associée contenant la signature des fonction et procédures déclarées, ainsi que l'adresse statique et type des variables déclarées
- le nom d'une fonction ou procédure pour les autres, la table associée contenant l'adresse statique et type des variables locales et paramètres associés

Ces dictionnaires associés à chaque contexte sont indexées par le nom des identificateurs qu'ils décrivent.

2.c Exemple

Soit le code suivant :

```
procedure pp is
  function double(v : in integer) return integer is
  begin
```

```
    <corps>
  end;
  procedure affiche() is
    i,j:integer;
  begin
    <corps>
  end;
  a,b,c : integer;
begin
  <corps>
end.
```

La table des identificateurs aura alors une structure de la forme :

```
{
  "pp": { # table associée au contexte principal
    "double": {params: [(entier, in)], retour: entier},
    "affiche": {params: []}
    "a": {nom: 'a', adresse: 0, type: entier}
    "b": {nom: 'b', adresse: 1, type: entier},
    "c": {nom: 'c', adresse: 2, type: entier}
  },
  "double": { # table associée au contexte de la fonction "double"
    "v": {nom: 'v', adresse: 0, type: entier}
  },
  "affiche": { # table associée au contexte de la procédure "affiche"
    "i": {nom: 'i', adresse: 0, type: entier},
    "j": {nom: 'j', adresse: 1, type: entier}
  }
}
```

2.d Construction

On commence par initialiser la table ainsi que les variables globales qui serviront à changer le contexte et compter les adresses :

```
identifierTable = dict()
mainContext = ""
currentContext = ""
cptAdress = 0
```

On récupère le nom du programme principal et on ajoute l'entrée correspondante :

```
procedure <ident> is
```

```
mainContext= ident
identifierTable[ident] = dict()
```

Dès que l'on rencontre une déclaration de fonction ou procédure, on l'ajoute à la table principale, et on lui crée sa propre table en changeant de contexte : `fonction <ident>(<listeSpecifFormelle>) return <type> is`

```
identifierTable[mainContext][ident] = ...
currentContext= ident
identifierTable[currentContext] = dict()
```

Pour chaque paramètre ou variable déclaré ensuite on ajoute l'entrée dans la table associée au contexte et on incrémente l'adresse statique : `<ident> : <type>;`

```
identifierTable[currentContext][ident] = {adresse: cptAdress, type: type}
cptAdress += 1
```

Puis à la fin des déclaration on recalcule sur le contexte global et on réinitialise le compteur d'adresses : `begin`

```
currentContext = mainContext
cptAdress = 0
```

S'il y a d'autres déclaration de procédures ou fonctions on recommence, sinon on stocke les variables du programme principal dans `identifierTable[mainContext]` de la même manière que décrite précédemment.

3 Génération du code

La partie génération de code consiste à ajouter aux différentes unités lexicales du langage des instructions nécessaires pour l'exécution et la bonne interprétation par la machine virtuelle (VM).

3.a Implémentation

Le code généré est stocké dans une simple liste Python dont les éléments sont les instructions utiles pour la machine virtuelle sous forme de chaîne de caractères.

3.b Points de génération

- Pour chaque unité lexicale (UL) nous avons intégré des points de génération, c'est à dire des instructions supplémentaires qui s'avèrent nécessaires par exemple pour garder une adresse statique en mémoire.
- Les diagrammes syntaxiques contiennent des panneaux d'aiguillage ainsi que ces points de génération afin de mieux représenter les instructions supplémentaires.

  Afin de visualiser les diagrammes syntaxiques avec les points de génération désinnées sur ceux-ci vous pouvez y accéder par ce document points_de_génération.pdf [1]

3.b.i Début et fin d'un programme

debutProg() au début d'un programme et finProg() quand on arrive à la fin avec end.

3.b.ii Procédures et fonctions

On termine toujours une procédure (respectivement fonction) par un `retourProc()` (respectivement `retourFonct()`). Dans le cas d'une fonction il peut y avoir plusieurs `retourFonct()` dans la déclaration s'il y a des alternatives donnant sur un `return`.

S'il y a des procédures ou des fonctions dans le programme principal, on doit sauvegarder l'adresse à laquelle est le `tra(adresse)` pour sauter les déclarations et ne pas exécuter le code des procédures et des fonctions avant l'exécution du programme principal. L'adresse sera mise à jour à la fin des déclarations.

Adresse	Instruction
2	tra(n)
3	...
...	...
n-1	retourFonct() ou retourProc()
n	Début programme principal

Tableau 1: Exemple avec déclaration de procédures ou de fonctions.

Quand on reconnait un appel avec ident(...) on fait `reserverBloc()` pour le bloc que l'appel va engendré.
On garde l'adresse de la première instruction d'une procédure ou d'une fonction lors de la génération de leur code pour pouvoir y revenir avec un `traStat(adresseInstruction(p), n)` pour nom p et n paramètres.

Adresse	Instruction
x	Début procédure/fonction p(x_1, x_2, ..., x_n)
...	...
y	traStat(x, n)

Tableau 2: Exemple d'appel procédure ou fonction.

3.b.iii Variables et paramètres

Il y a toujours réservations des emplacements des variables. Le nombre de variables est compté au moment de l'ajout dans la table des identificateurs. `reserver(n)` pour n variables.

Si le contexte est pas le contexte principal, la variable est globale. Sinon la variable est locale sauf si l'ident est en mode INOUT.

Pour les variables globales, l'adresse est empilée avec `empiler(ad(ident))` avant `affectation()`, `get()` ou `valeurPile()`.
Pour les variables locales, l'adresse est empilée avec `empilerAd(ad(ident))`.
Pour les paramètres en mode INOUT, l'adresse est empilée avec `empilerParam(ad(ident))` mais pour `valeurPile()`, il y a une vérification à faire. Si c'est l'appel de la procédure ou la fonction, il n'y a pas de `valeurPile()`. Sinon c'est identique aux variables globales.

3.b.iv Expressions

- Si c'est une constante `empiler(entier)`
- Si c'est un booléen `empiler(0)` pour faux, `empiler(1)` pour vrai.
- Si c'est l'opposé d'un entier ou la négation d'une expression booléenne, le `moins()` ou le `not()` est après l'expression. L'instruction est stocké et ajouté à la génération après l'instruction de l'expression.
- S'il y a un opérateur (`add()`, `sous()`, `mult()`, `div()`, `et()`, `ou()`, `egal()`, `diff()`, `inf()`, `infeg()`, `sup()`, `supeg()`) entre deux expressions, `<E1> op <E2>`, l'instruction `op` est stocké et ajouté après l'instruction des deux expressions.

3.b.v Alternatives et boucles

L'adresse de l'instruction d'arrivée d'un `tze(ad)` ou d'un `tra(ad)` n'est pas forcément connue à l'avance.
Une adresse temporaire est donnée dans ce cas là, puis l'instruction est modifié après la génération du code.

```
...
begin
  if <Expression> then
    <Instructions>
  else
    <Instructions>
  end
end;
```

Adresse	Avant else	Après else	Après end
x	<Expression>	<Expression>	<Expression>
x+1	tze(temp)	tze(n)	tze(n)
...	<Instructions>	<Instructions>	<Instructions>
n-1	...	tra(temp)	tra(m)
n	...	<Instructions>	<Instructions>
...	...	...	...
m-1	...	...	Fin de <Instructions>
m	...	...	<Instructions>

Tableau 3: Exemple de if, else.

```
...
begin
  while <Expression> loop
    <Instructions>
  end
end;
```

Adresse	Avant	Après
x	ad1	ad1
...	<Expression>	<Expression>
n	tze(temp)	tze(m)
...	<Instructions>	<Instructions>
m-1	...	tra(ad1)
m	...	...

Tableau 4: Exemple de boucle.

4 Machine virtuelle

Comme tout language de programmation le code se traduit en instructions assembleur spécifique à chaque machine et à chaque architecture de processeur exemple il y a une différence entre intel assembleur et ARM assembleur. C'est pour cela que la partie machine virtuelle est utile dans le sens ou ça rend notre language plus portable. A la manière où Java génère du bytecode, NilNovi génère des instructions pour notre machine virtuelle qui le reconnaît.

4.a Architecture

La machine virtuelle implémente plusieurs compartiments :

- **Une pile d'instructions** : Consiste en une simple liste contenant les instructions générées à partir du module analyse syntaxique.
- **Une pile de données** : Contient les données des variables locales et globales des procédures.
- **Un compteur ordinal** : Pointe sur l'adresse de l'actuelle instruction.
- **Un Pointeur de pile** : Pointe sur l'adresse de l'actuelle donnée.

– **Un registre de base** : Bloc de liaison permettant de désigner la dernière adresse de début de cadre de pile de données, utile lors de l'uitlisation des variables locales d'une sous-procédures.

4.b Usage

On a crée un module python permettant de jouer le rôle de la VM, ensuite la manuel d'utulisation de la VM avec la commande vm.py. Usage :

```
vm.py [-h] [-d |-d -s] inputfile
```

Arguments obligatoires:

`inputfile` : nom du programme compilé

Options :

`-h, --help` : affiche le message d'aide

`-d, --debug` : affiche le nom de l'instruction et l'état de la pile à chaque étape

`-s, --step-by-step` : permet d'exécuter les instructions une à une (à utiliser en mode debug)

4.c Implémentation

4.c.i Structure

La machine comprend deux listes `pile` et `po` représentant respectivement la pile de données et la pile d'instruction. Les variables entières `co`, `ip` et `base` représentent quant à elle respectivement le compteur ordinal, le pointeur de pile et le registre de base.

4.c.ii Instructions

La machine virtuelle analyse ligne par ligne chaque instruction, jusqu'à tomber sur `finProg()` ou une instruction inconnue, ce qui fait terminer le programme.

Lorsque la machine virtuelle analyse les lignes des différentes instructions, la VM va manipuler les 2 piles d'instructions et de données.

4.c.iii Interprétation de l'instruction reserver(n)

Cette instruction permet de reserver n cases mémoire dans la pile.

```
# reserver
elif re.match(r"reserver\(\d+\)", instr):
  n = int(re.search(r"\d+",instr).group())

  for i in range(n):
    pile.append(None)

  ip = ip + n # n : nombre de blocs à réserver
  co = co + 1
```

On voit bien que le code ajoute dans la boucle n cases vides puis ensuite il saute le pointeur de pile de n cases et le compteur ordinal est incrémenté.

En principe ceci est le cas pour chaque instruction mais où les traitements sur les piles et les pointeurs différent.

4.c.iv Interprétation des instructions pour un appel de fonction avec reconstitution du contexte.

4.c.iv.i Interprétation de l'instruction reserverBloc()

Lors de l'appel d'une fonction on va commencer par allouer le futur bloc de liaison.

```
# reserverBloc
elif re.match(r"reserverBloc()", instr):
# Cette instruction permet, lors de l'appel d'une opération,
# de réserver les emplacements du futur bloc de liaison
# et d'initialiser la partie pointeur vers le bloc de liaison de l'appelant .
  pile.append(None)
  pile.append(None)
  ip = ip + 2
  pile[ip - 1] = base
  co = co + 1
```

Comme par défaut la taille de ce bloc est de 2 cases on place l'adresse du pointeur 2 cases plus haut.

Ensuite on va réserver l'emplacement mémoire des n paramètres et les compiler.

4.c.iv.ii Interprétation de l'instruction traStat

```
# traStat
elif re.match(r"traStat\(\d+,\d+\)", instr):
  params = re.findall(r"\d+", instr)
  ad = int(params[0])
  nbp = int(params[1])
  base = ip-nbp-1
  pile[base+1] = co + 1
  co = ad
```

Cette instruction va aller placer la base au début du bloc de liaison afin d'avoir les paramètres qui sont des variables locales.

4.c.iv.iii Interprétation de l'instruction de retourFonct()

En plus d'une procédure les fonctions ont une valeur de retour. Il suffit de complier la valeur de retour et de finir le programme avec l'instruction retourFonct()

```
# retourFonct
elif re.match(r"retourFonct()", instr):
  # Cette instruction est produite à la fin de la compilation d'une instruction return
dans
  # une fonction. Outre son rôle dans le retour, elle assure que la valeur en sommet
de pile
  # sera le résultat de l'appel.
  #ar = int(re.search(r'\d+',instr).group())
  n = ip - base
  v = pile[ip]
  ip = base
  co = pile[base + 1]
  base = pile[base]
  for i in range(n):
  pile.pop()
  pile[ip] = v
```

5 Vérification sémantique

5.a Liste des erreurs sémantiques à vérifier

Pour éviter des erreurs sémantiques, notre premier but a été de les identifier et d'ensuite implémenter une vérification dans les fonctions du fichier `anasyn.py`. Les voici :

- Variable utilisée mais non instanciée
- Affectation d'une variable avec un typage incorrect
- Boucle avec une condition incorrecte, soit qui n'est pas booléenne
- Utilisation des mots-clés `or` et `and` avec des termes non booléens
- Opération de calcul avec des termes qui ne sont pas entiers
- Utilisation d'une fonction/procédure avec le mauvais nombre de paramètres
- Utilisation d'une fonction/procédure avec des paramètres de type incorrect
- Utilisation d'une variable booléenne avec un `get` ou un `put`

5.b Ajout de fichier pour les vérifications sémantiques

Des fichiers ont étés ajoutés dans le squelette du projet, dans les dossiers nna et nnp afin de tester que les vérifications sémantiques sont bien effectives lors d'une erreur Dans le dossier nna :

- error7_Added.nno : Utilisation de termes non booléens avec les mots-clefs or et and
- error8_Added.nno : Opérations de calcul avec des termes qui ne sont pas entiers
- error9_Added.nno et error10_Added.nno : Utilisation de variable non instanciées

Dans le dossier nnp :

- error1_Added.nno : Utilisation d'une fonction avec des types de paramètres incorrects (emploi d'un booléen à la place d'un entier)
- error2_Added.nno : Utilisation d'une fonction avec des types de paramètres incorrects (emploi d'un bentier à la place d'un booléen)
- error3_Added.nno : Utilisation d'une fonction avec un nombre incorrect de paramètres
- error4_Added.nno et error5_Added.nno : Utilisation d'une fonction avec un typage de paramètres incorrects
- error6_Added.nno : Utilisation d'une procéduren avec un nombre incorrect de paramètres

5.c Affichage des erreurs sémantiques

Afin de faciliter la reconnaissance de ces erreurs dans le terminal pour un utilisateur, nous avons ajouter au logger la possibilité d'afficher des messages en couleur selon son type. Nous avons choisis d'utiliser un `logger.error` pour les erreurs sémantiques.

De plus, dès qu'un `logger.error` est produit, on lance une exception qui interrompe immédiatement le programme, en précisant le type d'erreur est à quelle ligne du fichier `anasyn.py` on peut se référer pour la comprendre et la résoudre.

5.c.i Exemple de vérification sémantique : Typage incorrect

On effectue un test avec le fichier `error3.nno` du dossier `nno`, voici son code :

```
procedure e is
  val: integer;
begin
  val:=true  //mess. erreur attendu ici
end.
```

Résultat obtenu :

```
cahamada@ChakiChaki:/mnt/c/Users/User/Documents/Cours/INFO2/projet_tlc/projet
-compilation-2022-2023$ python3 src/anasyn.py -d --show-ident-table tests/nna
/error3.nno
2023-05-31 11:33:59,679 - anasyn - DEBUG - Name of program : e
2023-05-31 11:33:59,680 - anasyn - DEBUG - Parsing declarations
2023-05-31 11:33:59,680 - anasyn - DEBUG - identifier found: val
2023-05-31 11:33:59,680 - anasyn - DEBUG - now parsing type...
2023-05-31 11:33:59,680 - anasyn - DEBUG - integer type
2023-05-31 11:33:59,680 - anasyn - DEBUG - End of declarations
2023-05-31 11:33:59,680 - anasyn - DEBUG - Parsing instructions
2023-05-31 11:33:59,680 - anasyn - DEBUG - parsing expression: true
2023-05-31 11:33:59,680 - anasyn - DEBUG - parsing exp1
2023-05-31 11:33:59,680 - anasyn - DEBUG - parsing exp2
2023-05-31 11:33:59,680 - anasyn - DEBUG - parsing exp3
2023-05-31 11:33:59,681 - anasyn - DEBUG - parsing exp4
2023-05-31 11:33:59,681 - anasyn - DEBUG - parsing prim
2023-05-31 11:33:59,681 - anasyn - DEBUG - parsing elemPrim: true
2023-05-31 11:33:59,681 - anasyn - DEBUG - boolean true value
2023-05-31 11:33:59,681 - anasyn - ERROR - Affectation of a variable with a w
rong type : int != bool
Compilation failed  "L 345 - Fonction 'instr' :  Affectation of a variable wi
th a wrong type : "
```

On peut voir l'affichage d'un message d'erreur avec une description ce qui a été tenté, et indiquant aussi que la compilation a échoué.

5.c.ii Exemple de vérification sémantique : Appel d'une fonction avec un nombre incorrect de page

On effectue un test avec le fichier `error3_Added.nno` du dossier `nnp`, voici son code :

```
procedure e is
  function triple(a: integer) return integer is
  begin
    return a*3
  end;

  b: boolean;
  c: integer;
begin
  b:=true;
  put(triple(b))
end.
```

Résultat obtenu :

```
2023-05-31 11:44:12,650 - anasyn - DEBUG - parsed procedure call
2023-05-31 11:44:12,651 - anasyn - DEBUG - Call to function: triple
2023-05-31 11:44:12,651 - anasyn - ERROR - Type of given parameter is incorrect
Compilation failed  'L - 658 - Difference for parameter n° 0 expected type : int given
type : bool'
cahamada@ChakiChaki:/mnt/c/Users/User/Documents/Cours/INFO2/projet_tlc/projet-compilat
```

6 Conclusion

Le projet de compilation nous a permis de mettre en pratique nos connaissances théoriques, notamment dans le domaine de la théorie des langages, en développant un compilateur pour le langage NilNovi, qui s'exécute sur une machine virtuelle. Cette expérience nous a également offert l'opportunité d'explorer la programmation orientée objet en Python, en l'appliquant concrètement dans ce projet.

En travaillant sur le développement du compilateur, nous avons pu mettre en œuvre divers concepts et techniques que nous avions étudiés en théorie des langages. Par exemple, nous avons appliqué les principes de la grammaire formelle. De plus, nous avons utilisé des outils tels que les analyseurs lexicaux et syntaxiques pour analyser et interpréter le code source, afin de le transformer en un format compréhensible par la machine virtuelle, sans oublier la vérification sémantique en notifiant l'utilisateur de ses propres erreurs.

En conclusion, le développement du compilateur pour le langage NilNovi a été une expérience enrichissante. Non seulement nous avons pu mettre en pratique nos connaissances théoriques en théorie des langages, mais nous avons également acquis une meilleure compréhension du fonctionnement des langages de programmation. De plus, nous avons consolidé nos compétences en programmation orientée objet grâce à l'implémentation concrète de différentes fonctionnalités en Python. Ce projet nous a non seulement permis d'élargir notre champ de compétences, mais il a également renforcé notre passion pour le développement de logiciels et notre désir d'explorer de nouveaux domaines de programmation.

Bibliographie

[1] “Fichier contenant l'ensemble des points de génération.” (https://gitlab.enssat.fr/tgerbaud/projet-compilation-2022-2023/-/blob/main/points_de_g%C3%A9n%C3%A9ration.pdf)