TELECOM Nancy 2ème année
Octobre 2024 - Janvier 2025 - Avril 2025

Projet de compilation - Modules PCL1 et PCL2

L'objectif de ce projet est d'écrire un compilateur d'un langage de "haut niveau", en développant toutes les étapes qui le composent, depuis l'analyse lexicale jusqu'à la production de code assembleur ARM. Il s'agit d'un petit fragment du langage Python, langage que nous dénommerons Mini Python par la suite.
D'après le sujet du projet de compilation de l'X - Année 2023/2024.

Le sujet ci-dessous décrit la syntaxe et la sémantique du langage Mini Python ainsi que le travail à réaliser au cours des modules PCL1 et PCL2.

Réalisation du projet

L'écriture du compilateur complet concerne les deux modules PCL1 et PCL2. La partie PCL1 (octobre à mi-janvier) s'adressera à tous les élèves : il s'agit d'écrire les analyseurs lexicaux et syntaxiques, et d'implémenter la construction de l'arbre abstrait. La partie PCL2 (février à mi-mai) concernera le développement des contrôles sémantiques et la génération de code assembleur. Seuls les élèves des approfondissements IL et ISS sont concernés par ce second module PCL2.

Pour ce projet, les élèves des approfondissements IAMD, SLE et SIE ainsi que les élèves en mobilité au semestre 8 travailleront ensemble et formeront des groupes de 4. Les élèves des approfondissements IL et ISS (hors mobilité) sont concernés par les modules PCL1 et PCL2 et formeront des groupes de 4 sur ces 2 approfondissements uniquement.

Vous n'utiliserez aucun générateur d'analyseur lexical et syntaxique : vous aurez donc à développer par vous-même l'automate de reconnaissance des unités lexicales ainsi que l'analyseur syntaxique. La construction de l'analyseur syntaxique descendant à l'aide des "fonctions récursives" est conseillée.

Nous n'imposerons aucun langage pour le développement de votre compilateur, c'est à vous de choisir celui pour lequel l'ensemble des membres du groupe possède un maximum de compétences : C, Java, Python. Tous les membres des groupes de projet doivent travailler à part égale sur le projet.
Les groupes qui souhaitent utiliser un autre langage que ceux cités précédemment doivent contacter leur encadrant de TP.

Vous utiliserez un dépôt Git sur le Gitlab de TELECOM Nancy pour développer votre projet. On vous informera des modalités de création des dépôts.
Les dépots seront régulièrement consultés par les enseignants chargés de vous évaluer lors des séances de TP et lors de la soutenance finale de votre projet.
En cas de litige sur la participation active de chacun des membres du groupe au projet, le contenu de votre projet sur le dépôt sera examiné. Les notes peuvent être individualisées.

Première partie : module PCL1

PCL1 : séances TP 1 à 4 - définition complète de la grammaire du langage, conception des analyseurs lexical et syntaxique, construction de l'arbre abstrait et sa visualisation.

Les 4 séances de TP ont lieu les semaines 41, 42, 46 et 50. Vous aurez commencé l'implémentation de l'analyseur lexical à la semaine 42. En particulier, vous aurez défini sa structure et vous pourrez montrer quelques exmples d'unités lexicales reconnues. Votre automate devra lire le fichier source caractère par caractère.

Lors du TP de la semaine 46, vous aurez défini la grammaire, vous aurez commencé l'analyse syntaxique descendante et écrit des programmes de test. Vous aurez également réfléchi à la structure de l'arbre abstrait, à sa mise en oeuvre, et à sa visualisation.
Nous prendrons note de vos réalisations à cette date (semaine 46).

Votre compilateur doit signaler les erreurs lexicales, syntaxiques (et par la suite sémantiques - en PCL2) rencontrées. Lorsqu'une de ces erreurs est rencontrée, elle doit être signalée par un message le plus explicite possible, comprenant, dans la mesure du possible, un numéro de ligne.
Votre compilateur ne doit pas s'arrêter après une erreur lexicale ou syntaxique détectée, mais tenter de poursuivre l'analyse syntaxique.

Le TP de la semaine 50 est une séance d'évaluation intermédiaire où vous nous montrerez l'avancement de votre projet. Vous aurez testé votre compilateur sur des exemples variés de programmes corrects écrits en langage `Mini Python`, et sur des exemples comprenant des erreurs lexicales et syntaxiques.

Cette évaluation intermédiaire entrera dans la note finale du module PCL1.

La soutenance finale de cette partie PCL1 est fixée au mercredi 22 janvier 2025.

Vous montrerez lors de cette soutenance des exemples de programmes écrits en Mini Python et les arbres abstraits correspondants. Il est impératif que vous ayez prévu pour la soutenance des exemples de programmes permettant de tester votre projet. Ces exemples ne seront pas à écrire le jour de la démonstration.

Vous rendrez lors de cette soutenance un court **rapport d'activité** présentant le travail réalisé et les difficultés rencontrées. Vous incluerez dans ces documents la grammaire du langage, la structure de l'arbre abstrait et quelques jeux d'essais. Il n'est pas utile que ce rapport soit long, mais il doit être informatif.
N'oubliez pas les éléments de gestion de projet : répartition du travail et des tâches au sein du groupe et estimation du temps passé sur chaque étape de cette première partie du projet.

Vous remettrez ce rapport d'activité dans le casier de votre enseignant de TP pour le lundi 20 janvier 2025.

Seconde partie : module PCL2

Vous continuez votre projet en implémentant les contrôles sémantiques spécifiques à ce langage. Vous aurez réfléchi à la structure de la TDS pour la première séance de TP de PCL2.

Votre compilateur devra impérativement poursuivre l'analyse sémantique après avoir signalé une erreur sémantique.

Une fois cette étape de contrôles sémantiques réalisée, vous passerez à l'étape de génération de code. Pour cette dernière phase, vous veillerez à générer le code assembleur ARM de manière incrémentale, en commençant par les constructions "simples" du langage.

Des séances de TP sont prévues pour cette seconde partie PCL2. Nous reviendrons vers vous courant janvier à ce sujet.

A la fin du projet, donc à la fin du module PCL2, et pour le **mardi 20 mai 2025 - 18h**, vous rendrez un dossier qui complètera le précédent rapport d'activités de PCL1 et comprendra :

- les schémas de traduction du langage proposé vers le langage assembleur
- des *jeux d'essais* mettant en évidence le bon fonctionnement de votre compilateur, et ses limites éventuelles.
- une partie *gestion de projet*, à savoir une fiche d'évaluation de la répartition du travail sur cette seconde période avec la répartition des tâches au sein de votre binôme, l'estimation du temps passé sur chaque partie du projet, et le Gantt final.
- les divers CR de réunion que vous aurez rédigés.

La fin du projet est fixée au mercredi 21 mai 2025, date prévue pour les soutenances finales.

Lors de cette soutenance, on vous demandera de faire une démonstration de votre compilateur. Un planning vous sera proposé pour fixer l'ordre de passage des groupes.
Il est impératif que vous ayez prévu des exemples de programmes permettant de tester votre projet et ses limites : vous nous montrerez lors de votre soutenance votre "plus beau programme" écrit en langage Mini Python...

Aucun délai supplémentaire ne sera accordé pour la fin du projet.

Concernant l'évaluation de la génération de code, on vous fournira 4 "niveaux" de programmes écrits dans le langage du projet, ce qui vous permettra de vous situer dans cette phase de génération de code. Le niveau 2 est généralement celui requis pour obtenir une note de 10/20. Ceci n'est qu'une indication, car la note finale dépend aussi des évaluations intermédiaires et du dossier rendu.

Plagiat.
Bien entendu, il est interdit de "s'inspirer trop fortement" du code d'un autre groupe ; vous pouvez discuter entre-vous sur les structures de données à mettre en place, sur certains points techniques à mettre en oeuvre, etc... mais il est interdit de copier du code source sur vos camarades. Tout comme il est vivement déconseillé de demander à des "intelligences" d'écrire le compilateur à votre place... Si cela devait toutefois se produire, nous saurons en tenir compte dans nos évaluations.

1 Présentation du langage : aspects lexical et syntaxique

1.1 Conventions lexicales

Un commentaire débute par # et s'étend jusqu'à la fin de la ligne.
Les *identificateurs* d'un programme sont définis par l'expression régulière $\langle ident \rangle$ suivante :

$\langle digit \rangle$ ::= 0-9
$\langle alpha \rangle$ ::= a-z | A-Z
$\langle ident \rangle$ ::= ($\langle alpha \rangle$ | _) ($\langle alpha \rangle$ | _ | $\langle digit \rangle$)*

Les constantes entières suivent l'expression régulière suivante :

$\langle integer \rangle$ ::= 0 | 1-9 $\langle digit \rangle$*

Les mots clés du langage sont les suivants :

```
and   def   else   for     if        True   False
in    not   or     print   return    None
```

1.2 Syntaxe

Dans la spécification de la grammaire du langage, on utilisera les notations suivantes :

$\langle motif \rangle^*$: répétition du motif $\langle motif \rangle$ un nombre quelconque de fois ou aucune fois
$\langle motif \rangle_t^*$: comme précédemment, avec les occurences séparées par le terminal t
$\langle motif \rangle^+$: répétition du motif $\langle motif \rangle$ au moins une fois
$\langle motif \rangle_t^+$: comme précédemment, avec les occurences séparées par le terminal t
$\langle motif \rangle$? : utilisation optionnelle de $\langle motif \rangle$, c'est à dire 0 ou 1 fois

Les associativités et précédences des opérateurs sont données dans la table ci-dessous, de la plus faible à la plus forte priorité.

opérateur	associativité	priorité
or	gauche	plus faible
and	gauche	
not	–	
< <= > >= == !=	–	↓
+ −	gauche	
* // %	gauche	
− (*unaire*)	–	
[	–	plus forte

Remarque : une expression telle que $x < y < y$ ne sera pas autorisée par la syntaxe.
Les chaînes de caractères seront écrites entre guillemets (symbole "). Il y a deux séquences d'échappement : \" pour le caractère " et \n pour le caractère de passage à la ligne suivante.

En Python, les structures de blocs sont caractérisées et définies par l'indentation des lignes, c'est à dire par le nombre de caractères *espace* présents en début de ligne (on suppose qu'il n'y a pas de caractère de tabulation).
Pour repérer les blocs, la plupart du travail est effectué par l'analyseur lexical, qui produira les trois tokens suivants NEWLINE, BEGIN et END correspondant respectivement aux fins de lignes, à l'incrémentation et à la décrémentation de l'indentation.

L'algorithme à suivre est le suivant : l'analyseur lexical définit une pile d'entiers représentant les indentations successives. Cette pile est ordonnée, la plus grande valeur étant au sommet. Au départ, la pile contient une seule valeur, l'entier 0. Lorsque l'analyseur lexical rencontre un passage à la ligne suivante, il produit le token NEWLINE, puis "mesure" l'indentation de la ligne suivante, notée n et la compare avec la valeur de l'indentation m qui se trouve au sommet de la pile. Il y a 3 cas :

1. si $n = m$, rien à faire ;
2. si $n > m$, empiler n et produire un second token BEGIN ;
3. si $n < m$, dépiler jusqu'à trouver la valeur n, produire un token END pour chaque valeur dépilée et strictement supérieure à n. La valeur n, s'il y en a une, reste au sommet de la pile. Si aucune valeur dépilée n'est égale à n, émettre le message "indentation error".

La grammaire du langage Mini Python est donnée page 5.

Les sections suivantes des pages 6, 7 et 8 décrivent le typage et la sémantique du langage (extraits du sujet X-2023/2024).

⟨*file*⟩ ::= NEWLINE? ⟨*def*⟩* ⟨*stmt*⟩+ EOF
⟨*def*⟩ ::= def ⟨*ident*⟩ (⟨*ident*⟩*,) : ⟨*suite*⟩
⟨*suite*⟩ ::= ⟨*simple_stmt*⟩ NEWLINE
| NEWLINE BEGIN ⟨*stmt*⟩+ END
⟨*simple_stmt*⟩ ::= return ⟨*expr*⟩
| ⟨*ident*⟩ = ⟨*expr*⟩
| ⟨*expr*⟩ [⟨*expr*⟩] = ⟨*expr*⟩
| print (⟨*expr*⟩)
| ⟨*expr*⟩
⟨*stmt*⟩ ::= ⟨*simple_stmt*⟩ NEWLINE
| if ⟨*expr*⟩ : ⟨*suite*⟩
| if ⟨*expr*⟩ : ⟨*suite*⟩ else : ⟨*suite*⟩
| for ⟨*ident*⟩ in ⟨*expr*⟩ : ⟨*suite*⟩
⟨*expr*⟩ ::= ⟨*const*⟩
| ⟨*ident*⟩
| ⟨*expr*⟩ [⟨*expr*⟩]
| - ⟨*expr*⟩
| not ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| ⟨*ident*⟩ (⟨*expr*⟩*,)
| [⟨*expr*⟩*,]
| (⟨*expr*⟩)
⟨*binop*⟩ ::= + | - | * | // | % | <= | >= | > | < | != | ==
| and | or
⟨*const*⟩ ::= ⟨*integer*⟩ | ⟨*string*⟩ | True | False | None

2 Static Typing

Though Python is a dynamically-typed language, Mini Python is simple enough to allow us some checks at compile time. These checks are the following:

1. Any function used in an expression must be
   - either a function that was previously defined;
   - either the function we are currently defining (a recursive function);
   - one of the three built-in functions `len`, `list`, and `range`. (In Mini Python, `print` has built-in syntax and thus is not considered as a function.)

   In particular, there are no mutually recursive functions in Mini Python.
2. The names of the functions declared with `def` are pairwise distinct, and distinct from `len`, `list`, and `range`.
3. Function arity must be obeyed, *i.e.*, a function defined with n formal parameters must be called with exactly n actual parameters. Functions `len`, `list`, and `range` all have one parameter.
4. Built-in functions `list` and `range` are exclusively used in the compound expression `list(range(`e`))`.
5. The formal parameters of a function must be pairwise distinct.
6. The scope of variables is statically defined. A variable is either local to a function or global. A local variable x is introduced either as a function parameter or via an assignment $x = e$ anywhere in the function body. The scope of a local variable extends to the full body of the function. A global variable is introduced via an assignment in the toplevel code (the code outside of function definitions at the end of the program). Hint: It is convenient to see the toplevel code as the body of a `main` function. This way, global variables are simply variables that are local to function `main`.

   Note that it is not possible to shadow a variable in Mini Python. We do not (and could not) check at compile time that a variable is defined before being used.

3 Semantics

Any value in Mini Python has a *dynamic type*. This type is assigned to the value at creation time and it cannot be modified thereafter. There are five possible types: `none`, `bool`, `int`, `string`, and `list`. The semantics of an operation can vary according to the type of its actual parameters. In some cases, it can lead to a runtime failure. In this case, the code produced will display a message of the form

```
error: some message
```

and will terminate with exit code 1.

Values. The `none` type contains a single value, noted as `None`. In particular, this is the value returned by a function that reaches its return point without encountering a `return` statement. The `bool` type is the Boolean type, with two possible values are `False` and `True`. The `int` type is for signed 64-bit integers. Type `string` is for character strings. Finally, type `list` is for lists, which are heterogeneous and possibly empty. The elements of a list can be modified in place, with the statement x`[`e_1`] =` e_2, but the length of a list cannot be modified.

Built-in Function `print`. The `print` statement displays the value passed as a parameter, followed by a newline. The display format is as follows:

type	*affichage*
`none`	`None`
`bool`	`False`, `True`
`int`	in decimal, *e.g.* `42`
`list`	`[`e_1`, ..., `e_n`]`, *e.g.* `[1, 2, 3]`
`string`	without quotes, *e.g.* `abc`
	`\n` is printed as a newline
	`\"` is printed as `"`

Boolean Condition. The statement `if` and the operators `and`, `or`, and `not` all accept operands of any type, with the following semantics:

- `None`, `False`, the integer 0, the empty string, and the empty list are interpreted as false;
- any other value is interpreted as true.

Operators `and` and `or` are lazily-evaluated: the second operand is evaluated only if needed, *i.e.* if the value of the first operand does not determine the final value. The result of e_1 `and` e_2 (resp. e_1 `or` e_2) is either the value of e_1 or the value of e_2. For instance, `0 and [1,2]` evaluates to `0` and `1 and [1,2]` evaluates to `[1,2]`.

Iteration. The statement `for x in e: s` first evaluates expression `e`, which must be of type `list`. Then, for each value v in this list, in order, it assigns v to the variable `x` and executes the statement `s`.

Operators. Subtraction (`-`), negation (unary `-`), multiplication (`*`), division (`//`), and modulo (`%`) are only defined on type `int`, with signed 64-bits machine arithmetic[1]. Operator `+` is overloaded, with two parameters of the same type and the following semantics:

type of parameters	semantics
`int`	arithmetic addition
`string`	concatenation (in a new string)
`list`	concatenation (in a new list)
otherwise	failure

[1]Python's division is actually not machine's division, but we accept this difference.

Comparison Operators. The six comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`) always return a Boolean value. For any comparison operator, Boolean operands are interpreted as integer values (`False` being 0 and `True` being 1). Operators `==` and `!=` are defined for operands of any type, including two operands of different types. Operators `<`, `<=`, `>`, and `>=` are limited to the following cases:

type of operands	semantics
`bool` or `int`	arithmetic comparison
`string`	lexicographic comparison
`list`	lexicographic comparison

The comparison is structural: when comparing lists, the comparison operation is recursively applied to the list elements.

Built-in Functions. Function `len` is only defined on types `string` and `list`. It returns the length of the string and of the list, respectively. The expression `list(range(`e`))` is defined for an expression e that evaluates to some integer $n \geq 0$. It returns a list of integers $[0, 1, \ldots, n-1]$.

Differences w.r.t Python. There are some small runtime differences between Python and Mini Python, including (but not exhaustively):

	Python	Mini Python
arithmetic	unbounded	signed 64 bits
string display within lists	`['abc']`	`[abc]`
multiplying a string/list and an integer	defined	undefined
access $l[i]$ with $i < 0$	defined	undefined

Your compiler will not be tested on programs for which there is a distinct runtime behavior between Python and Mini Python.