Projet de compilation - Interpréteur SPF

Année académique 2024-2025, 1ère session

Modifié le 11/03/2025 – Alexandre Decan

L'objectif de ce projet de compilation est de réaliser un interpréteur pour le langage SPF. Le langage SPF (pour **S**imple **P**rogramme en **F**rançais) est un langage de programmation basique dans lequel les instructions et les mots-clés sont en français.

1 Langage SPF

Le langage SPF est un langage *faiblement et statiquement typé*, composé d'instructions, d'expressions et de structures de contrôle. Une instruction représente un *ordre* donné à l'interpréteur (par exemple, initialiser une variable ou afficher à l'écran), une expression représente une *valeur* (par exemple, le résultat de l'addition d'une variable et une constante), et les structures de contrôle définissent le comportement du programme (par exemple, un test conditionnel ou une boucle). Les instructions se terminent toujours par un point-virgule ("`;`"). Les lignes de code SPF débutant par un dièse ("`#`") sont ignorées par l'interpréteur, tout comme les éventuels espaces ou indentations situés en début ou en fin de ligne.

Les sous-sections suivantes décrivent en détail les possibilités offertes par le langage SPF.

1.1 Typage des variables

SPF supporte un nombre limité de *types de données* : les Booléens (`booléen`), les entiers (`entier`), les chaînes de caractères (`texte`) et les listes (`liste`). Toute expression et toute variable est de l'un de ces types.

Pour déclarer une variable, il est nécessaire de préciser son type. La valeur de la variable peut, optionnellement, être précisée lors de son initialisation. De façon générale, la déclaration d'une variable est de la forme `<type> <nom>;` ou `<type> <nom> = <expression>;`. L'exemple suivant montre la déclaration d'une variable `x` de type `entier`, sans initialiser sa valeur, ainsi que d'une variable `y` de type `texte` initialisée à `"Hello world!"`.

```
entier x;
```

```
texte y = "Hello world!";
```

Une variable ne peut être déclarée qu'une seule fois. Bien entendu, une variable non-déclarée ne peut pas être utilisée.[1] Le nom d'une variable est composé au minimum d'un caractère, ne peut contenir que des lettres (majuscules et minuscules) accentuées ou non, des chiffres ou un tiret bas ("_"). Le nom d'une variable ne peut pas débuter par un chiffre.

1.2 Expressions et opérations

Les expressions visent à être évaluées afin d'en calculer la valeur. Les expressions peuvent être littérales (par exemple, `vrai` ou `42`) ou le résultat d'une opération (par exemple, `1 + 1`).

Les expressions littérales suivantes sont disponibles dans SPF :

- `vrai` et `faux`, de type `booléen`;
- Une chaîne de caractères délimitée par des guillemets doubles, de type `texte`;
- Un nombre entier positif composé d'une suite de chiffres;
- Une liste de la forme `[x1, x2, ..., xn]` où chaque `xi` est une expression. La liste vide se note `[]`. L'expression `["Hello", "World"]` est un exemple d'une liste (type `liste`) contenant deux chaînes de caractères (type `texte`). Un exemple de liste contenant des expressions est `[1+1, 1+2, 1+3]`. La liste `[vrai, 1, "Hello"]` est un exemple de liste mixte.
- Une liste de la forme `[i:j]` où `i` et `j` sont deux expressions de type `entier`. Cette syntaxe produit une liste équivalente à `[i, i+1, i+2, ..., j-1, j]` (par exemple, `[2:4]` produit la liste `[2, 3, 4]`).

Les expressions (littérales ou non) peuvent être combinées afin de former d'autres expressions. Cette combinaison se fait au travers des *opérateurs* et du *parenthésage*. Si `expr` est un morceau de code SPF représentant une expression, alors `(expr)` est également une expression avec un type et une valeur identique. Les opérateurs supportés dépendent du type des expressions combinées et ne sont valides qu'avec les types adéquats. Par exemple, l'expression `x * y` n'est valide que si `x` et `y` sont deux variables de type `entier`.

Les opérations suivantes sont disponibles :

- Tous les types :
  - `x vaut y` s'évalue à `vrai` si `x` et `y` sont de valeur égale, à `faux` sinon. Seules des expressions de même type peuvent être comparées de la sorte. La syntaxe `x == y` est également supportée;
  - `x ne vaut pas y` s'évalue à `vrai` si `x` et `y` ont une valeur différente, `faux` sinon. Seules des expressions de même type peuvent être comparées de la sorte. La syntaxe `x != y` est également supportée;
- Type `booléen`:

[1] La gestion des erreurs sera abordée plus loin dans ce document.

  - `non` est un opérateur unaire inversant la valeur Booléenne (`non vrai` donne `faux`, par exemple);
  - `x et y` implémente le *et* logique;
  - `x ou y` implémente le *ou* logique;
- Type `entier` :
  - `-` est un opérateur unaire retournant l'opposé de son argument (`-1` retourne l'opposé de l'entier `1`);
  - `x + y` correspond à l'addition;
  - `x - y` correspond à la soustraction;
  - `x * y` correspond à la multiplication;
  - `x / y` correspond à la division entière;
  - `x <= y` (resp. `x < y`) s'évalue à `vrai` si `x` est (resp. strictement) inférieur à `y`, `faux` sinon;
  - `x >= y` (resp. `x > y`) s'évalue à `vrai` si `x` est (resp. strictement) supérieur à `y`, `faux` sinon;
- Type `texte` :
  - `x + y` correspond à la concaténation de deux chaînes de caractères;
  - `x[i]` où `x` est de type `texte` et `i` de type `entier` correspond à un `texte` composé du caractère situé à la position `i` (une expression de type `entier` dont la valeur est entre 1 et la taille de `x`);
  - `taille x` vaut la longueur du texte `x`;
- Type `liste` :
  - `x + y` correspond à la concaténation de deux listes;
  - `x[i]` correspond à l'élément situé à la position `i` (entre 1 et la taille de `x`);
  - `taille x` vaut la longueur de la liste `x`;

Parmi ces opérations, les opérateurs `non`, `-`, `[i]` et `taille` ont priorité sur les autres opérateurs. Les autres opérateurs s'évaluent de gauche à droite, à l'exception des opérateurs `+`, `-`, `*` et `/`, appliqués sur des entiers, qui doivent respecter la priorité arithmétique usuelle. Ces opérateurs ont priorité sur les opérateurs de comparaison, qui ont priorité sur les opérateurs logiques.[2]

1.3 Instructions prédéfinies

Le langage SPF propose quelques instructions prédéfinies afin de pouvoir manipuler les variables. Cela inclut notamment la déclaration d'une variable :

```
<type> <variable>;
<type> <variable> = <expression>;
```

Affecter une valeur à une variable déclarée se fait via l'instruction suivante :

```
<variable> = <expression>;
```

[2] En cas de doute, la priorité attendue est la même que pour les opérations équivalentes en Python, tel que décrit sur https://docs.python.org/3/reference/expressions.html#operator-precedence.

L'ajout d'un élément dans une liste se fait via l'instruction suivante :

```
ajouter <expression> dans <variable>;
```

Enfin, la dernière instruction proposée par SPF permet d'afficher le résultat d'une ou plusieurs expressions à l'écran, peu importe leur type. Il y a deux syntaxes permettant de faire cela :

```
afficher <expression>;
afficher <expression>, <expression>, ..., <expression>;
```

Dans le second cas, les valeurs des différentes expressions sont implicitement concaténées et séparées par un espace avant d'être affichées à l'écran. Notez que chaque appel à `afficher` génère une nouvelle ligne à l'écran.

Enfin, le rendu d'une expression affichée à l'écran dépend du type de l'expression. Pour les expressions de type `texte` ou `entier`, la valeur de l'expression est utilisée. Pour une expression de type `booléen`, les chaînes `vrai` et `faux` sont employées. Pour une liste, l'affichage se fait sous la forme `[x1, x2, x3, ...]` où chaque `xi` est rendu à l'écran en suivant les règles définies dans ce paragraphe. Par exemple, l'instruction `afficher [vrai, "vrai", 0, [1+1, "1+1"]];` provoque l'affichage `[vrai, vrai, 0, [2, 1+1]]`.

1.4 Tests conditionnels

Le langage SPF permet d'exprimer des tests conditionnels, en utilisant l'une des deux syntaxes ci-dessous.

```
si <expression> alors {
    ...
}

si <expression> alors {
    ...
} sinon {
    ...
}
```

Bien entendu, il est nécessaire que le type de `<expression>` soit `booléen`.

1.5 Boucles

Enfin, le langage SPF supporte aussi les boucles *tant que* et *pour chaque*. Une boucle *tant que* emploie la syntaxe suivante :

```
tant que <expression> faire {
    ...
}
```

Similairement au test conditionnel, il est attendu que le type de `<expression>` soit `booléen`.

La boucle *pour chaque* est un peu différente, car elle permet d'itérer sur chaque valeur d'une `liste` ou chaque caractère d'un `texte`. Cette valeur est stockée temporairement dans une variable dont le type est précisé lors de la déclaration de la boucle :

```
pour chaque <type> <variable> dans <expression> faire {
    ...
}
```

Par exemple, `pour chaque texte caractère dans "Hello world!" faire { afficher caractère; }` affiche chaque caractère sur une ligne séparée à l'écran.

Il est important de noter que la variable ainsi définie (`caractère` dans l'exemple) est **temporaire** et n'existe que dans le contexte de cette boucle. Une fois l'exécution de la boucle terminée, cette variable disparaît. Si une variable du même nom existait avant l'exécution de la boucle, elle reprend sa valeur précédente une fois la boucle effectuée. Le morceau de code suivant illustre cette situation et affichera `vrai`, `3`, `0`, `3`, `4`, `1` et `vrai` à l'écran après exécution :

```
booléen x = vrai;
afficher x;

pour chaque entier x dans [0:1] faire {
    pour chaque entier x dans [3:3+x] faire {
        afficher x;
    }
    afficher x;
}
afficher x;
```

2 Exemples de programmes SPF

Cette section fournit quelques exemples de code SPF. Ces exemples sont téléchargeables sur la plate-forme Moodle.

2.1 Recherche d'un maximum

Ce code SPF affiche la valeur maximale d'une liste d'entiers.

```
liste nombres = [1, 8, 3, 6];
entier maximum;

maximum = nombres[1];
pour chaque entier nombre dans nombres faire {
```

```
    si nombre > maximum alors {
        maximum = nombre;
    }
}

afficher "Le maximum dans", nombres, "est", maximum;
```

2.2 Factorielle d'un nombre

Ce code SPF calcule la factorielle d'un nombre.

```
entier nombre = 5;
entier factorielle;

factorielle = 1;
tant que nombre > 0 faire {
    factorielle = factorielle * nombre;
    nombre = nombre - 1;
}

afficher "La factorielle vaut", factorielle;
```

2.3 Monotonicité d'une liste

Le code SPF suivant affiche si une liste de nombres est monotone ou non. Une liste de nombres est monotone si les éléments de la liste sont rangés par ordre croissant.

```
liste nombres = [3, 3, 6, 7, 8];
booléen monotone = vrai;

pour chaque entier position dans [1:taille nombres - 1] faire {
    si nombres[position] > nombres[position + 1] alors {
        monotone = faux;
    }
}

si monotone alors {
    afficher "La liste", nombres, "est monotone";
} sinon {
    afficher "La liste", nombres, "n'est pas monotone";
}
```

2.4 Identifier les mots d'un texte

Le code SPF suivant décompose un texte en mots et enregistre chaque mot dans une liste. On considère que les mots d'un texte sont séparés par des espaces.

```
texte phrase = "Bonjour à tout le monde";
texte mot = "";
liste mots = [];

pour chaque texte caractère dans phrase faire {
    si caractère vaut " " alors {
        si taille mot > 0 alors {
            ajouter mot dans mots;
            mot = "";
        }
    } sinon {
        mot = mot + caractère;
    }
}

# Potentiel dernier mot
si taille mot > 0 alors {
    ajouter mot dans mots;
}

afficher mots;
```

3 Interpréteur SPF

3.1 Programme `spf.py`

Vous devez créer un programme, nommé `spf.py`, qui lit le code SPF d'un fichier, l'interprète et en affiche les résultats. Le programme `spf.py` prend en entrée un seul et unique paramètre positionnel : le chemin vers le fichier contenant le code SPF à devoir interpréter (par exemple, `python spf.py code.spf`). Comme le nom l'indique, `spf.py` doit être écrit en Python et doit pouvoir fonctionner sur n'importe quelle version de Python supérieure ou égale à la version 3.11.

Le programme `spf.py` accepte également les paramètres suivants :

- `--dump` ou `-d` provoque l'affichage de la *mémoire* du programme à l'issue de son exécution. Pour chaque variable `<nom>`, de type `<type>` et de valeur `<valeur>`, ce paramètre affiche `<type> <nom> = <valeur>` sur une nouvelle ligne.
- `--trace` ou `-t` provoque, durant l'exécution du programme, un affichage similaire des variables lors (1) de leur déclaration, (2) utilisation ou (3) modification, précédé de (1) `déclare`, (2) `accède` ou (3) `modifie`.

L'affichage généré par ces deux paramètres optionnels se fait sur la **sortie d'erreur** standard (`STDERR` et non `STDOUT`).

3.2 Librairie `lark`

Votre programme `spf.py` doit faire usage de la librairie *Lark*, une librairie offrant des fonctionnalités d'analyse lexicale et syntaxique en Python. Lark est disponible sur *PyPI* et peut donc être installé avec `pip install lark`. Sa documentation est disponible sur https://lark-parser.readthedocs.io/en/stable/.

La grammaire que vous développez pour interpréter le langage SPF doit être **sans conflit**, doit être **LALR(1)** et doit être contenue dans le fichier `spf.lark`.

3.3 Gestion des erreurs

Le programme `spf.py` doit s'assurer de gérer les erreurs qui peuvent survenir lors de son exécution. Toutes les erreurs en lien avec l'analyse et l'interprétation du programme SPF doivent être capturées par votre programme, aux endroits appropriés, et converties en des instances de sous-classes de `SPFException`.[3] Ces erreurs, avec la *traceback* associée, sont affichées dans le terminal à l'attention de l'utilisateur (comme les exceptions "classiques" de Python).

En particulier, il vous est demandé de générer les erreurs suivantes :

- `SPFSyntaxError` en cas d'erreur de syntaxe dans le programme SPF;
- `SPFUnknownVariable` lorsqu'une variable non-déclarée est utilisée;
- `SPFUninitializedVariable` lorsqu'une variable est utilisée mais n'a pas été initialisée;
- `SPFAlreadyDefined` lors de la déclaration d'une variable déjà déclarée (sauf itérateur dans une boucle *pour chaque*);
- `SPFIncompatibleType` quand une déclaration, opération, instruction ou expression possède un type incompatible;
- `SPFIndexError` quand une position invalide est utilisée pour accéder à un élément d'une liste ou d'un texte.

Le message affiché lorsqu'une exception survient doit mentionner la ligne concernée au sein du programme SPF (c'est-à-dire au sein du code passé en entrée de l'interpréteur, pas le code Python) si applicable, une courte description informative de l'erreur et, si applicable, le nom et/ou la valeur de la/des variable(s) impliquée(s).

4 Informations pratiques

Ce projet est à réaliser en groupe de **deux étudiants**. La composition des groupes doit être envoyée par e-mail à alexandre.decan@umons.ac.be pour le **vendredi 21 mars à 12h** au plus tard.

Le projet doit être remis via **Moodle** pour le **vendredi 16 mai à 18h** au plus tard. Vous devez remettre exactement les éléments suivants :

[3] Voir https://docs.python.org/3/tutorial/errors.html#user-defined-exceptions.

- Le fichier `spf.py` contenant votre interpréteur SPF;
- Le fichier `spf.lark` contenant la définition de votre grammaire;
- Un fichier `requirements.txt`[4] ou `pyproject.toml`[5] reprenant les dépendances (noms et versions) de votre programme;
- Tout module supplémentaire doit être placé dans un dossier `modules`;
- Un rapport de quelques pages au format PDF, nommé `rapport.pdf`, et reprenant les éléments suivants :
  - Une description brève du projet;
  - Une description de la grammaire implémentée et des *points sensibles* la concernant;
  - Une explication de votre approche pour gérer :
    - Les variables, leur déclaration, leur type et l'affichage via `--trace`;
    - Le test conditionnel de la forme *si/sinon*;
    - La boucle *tant que*;
    - La boucle *pour chaque*, incluant la gestion de la variable temporaire.
  - Une description brève des erreurs connues et des solutions envisagées;
  - Une brève présentation des difficultés rencontrées et des solutions implémentées/envisagées;
  - Les points éventuels pour lesquels vous avez fait appel à une IA[6] (ChatGPT, CoPilot, etc.);
  - La répartition du travail au sein du groupe.

En cas de problème lié à la répartition du travail au sein du groupe, les étudiants sont invités à m'en informer le plus rapidement possible.

Enfin, une **défense orale** sera organisée durant la session d'examen. Les modalités seront précisées sur Moodle.

5 Conseils

Il vous est fortement conseillé de créer le programme étape par étape afin de pouvoir déboguer plus facilement, en commençant par la gestion des variables et en terminant par les boucles. Essayez d'implémenter au plus tôt le support pour les paramètres `--dump` et `--trace`, ces derniers s'avèreront précieux pour identifier et corriger les erreurs.

Pensez au module `argparse` de Python,[7] plus pratique et bien plus propre que `sys.argv` pour gérer les paramètres d'une ligne de commande.

La librairie Lark possède plusieurs classes (`Visitor`, `Interpreter`, `Transformer`) facilitant son utilisation en fonction de la structure du langage à analyser. Dans

[4] Voir https://pip.pypa.io/en/stable/reference/requirements-file-format/

[5] Voir https://packaging.python.org/en/latest/guides/writing-pyproject-toml/

[6] Pour rappel, l'utilisation d'une IA est soumise au respect de la Charte IA de l'UMONS, voir https://web.umons.ac.be/app/uploads/sites/34/2024/05/CharteIA-UMONS.pdf

[7] Voir https://docs.python.org/3/library/argparse.html

le cadre de ce projet, nous vous conseillons de vous orienter vers la classe `Interpreter`.

Il est conseillé de créer une classe `Value` pour encoder le type et la valeur des variables et des résultats des expressions. La classe `Enum` en Python[8] vous permet de définir aisément les types supportés. Envisagez également une classe `Memory` pour gérer les variables. Cette classe peut, par exemple, proposer les méthodes `declare`, `get`, `set`, `typeof`, etc. Concevoir un *décorateur*[9] à appliquer sur les méthodes de votre sous-classe d'`Interpreter` peut vous aider à généraliser l'ajout d'un numéro de ligne lors de la gestion des erreurs, en interceptant certaines de vos exceptions et en les enrichissant à la volée.

L'évaluation de votre travail portera aussi bien sur le fond et la forme de votre rapport que sur l'aspect fonctionnel de votre programme. Pensez également à bien documenter votre code ! La qualité de l'architecture, la qualité et l'élégance du code sont également des facteurs intervenant dans la note finale. Il existe de nombreux outils en Python (*pep8*, *flake8*, *ruff*, *black*, etc.) pour vous aider à produire du code de qualité. N'hésitez pas à les utiliser !

Enfin, pensez à relire attentivement l'ensemble des consignes avant de remettre votre travail. Le respect des consignes intervient dans l'évaluation de votre travail.

Vous pouvez, bien entendu, me contacter alexandre.decan@umons.ac.be si vous avez des questions.

[8] Voir https://docs.python.org/3/library/enum.html
[9] Voir https://realpython.com/primer-on-python-decorators/