1 Lexique

1.1 “Atomes” du langage

Unit : `‘unit’`
Vrai : `‘vrai’`
Faux : `‘faux’`
Entier : `‘[0-9][0-9_]*’`
Flottant : `‘([0-9][0-9_]*)?\.[0-9_]*’`
Chaine : `‘"[\.]*"’`
IdfMin : `‘[a-z][a-zA-Z_]*’`

1.2 Opérateurs

Et : `‘et’`
Ou : `‘ou’`
Egale : `‘==’`
Different : `‘<>’`
Inf : `‘<’`
InfEgale : `‘<=’`
Sup : `‘>’`
SupEgale : `‘>=’`
Plus : `‘+’`
Moins : `‘-’`
Mul : `‘*’`
Div : `‘/’`
Modulo : `‘%’`

1.3 Genre (Types)

Pipe : `‘|’`
Appli : `‘->’`
FnGenre : `‘Fn’`
IdfMaj : `‘[A-Z][a-zA-Z_]*’`

1.4 Affectations

Aff : `‘=’`
Incr : `‘++’`
Decr : `‘--’`
AffPlus : `‘+=’`
AffMoins : `‘-=’`
AffMul : `‘*=’`
AffDiv : `‘/=’`
AffModulo : `‘%=’`

1.5 Séparateurs & Délimiteurs

ParO : `'('`

ParF : `')'`

AccolO : `'{'`

AccolF : `'}'`

Comment : `'#'`

Virgule : `','`

DeuxPoints : `':'`

FinLigne : `'[\CR\LF]+'`

1.6 Structure de contrôle

Fn : `'fn'`

Retour : `'retour'`

Si : `'si'`

Sinon : `'sinon'`

Tq : `'tq'`

Boucle : `'boucle'`

Remarques :

- En cas de 'conflit de caractère' (eg : '<' et '<=') le lexer reconnait l'élément le plus long (ie : '<=')
- Les mots-clés sont reconnus comme des identifiants dans un premier temps et distinguer a posteriori
- Le lexer produit un lexème **ChaineNonFermee** le cas échéant
- Le lexer produit un lexème **Erreur** pour tout caractère non reconnu

2 Grammaire

2.1 Expression

Expression → Disj

Disj → Conj DisjSuite
DisjSuite → '*Ou*' Disj
DisjSuite → 'ϵ'

Conj → Rel ConjSuite
ConjSuite → '*Et*' Conj
ConjSuite → 'ϵ'

RelOpde → Somme
Rel → RelOpde
Rel → RelOpde '*Egale*' RelOpde
Rel → RelOpde '*Different*' RelOpde
Rel → RelOpde '*Inf*' RelOpde
Rel → RelOpde '*InfEgale*' RelOpde
Rel → RelOpde '*Sup*' RelOpde
Rel → RelOpde '*SupEgale*' RelOpde

Somme → Terme SommeSuite
SommeSuite → '*Plus*' Somme
SommeSuite → '*Moins*' Somme
SommeSuite → 'ϵ'

Terme → Facteur TermeSuite
TermeSuite → '*Mul*' Terme
TermeSuite → '*Div*' Terme
TermeSuite → 'ϵ'

Facteur → Unaire Appel
Appel → '*ParO*' AppelArgs '*ParF*' Appel
Appel → '*ParO*' '*ParF*' Appel
Appel → 'ϵ'
AppelArgs → Expression AppelArgsSuite
AppelArgsSuite → '*Virgule*' AppelArgs
AppelArgsSuite → 'ϵ'

Unaire → '*Plus*' Unaire
Unaire → '*Moins*' Unaire
Unaire → Atome

Atome → '*ParO*' Expression '*ParF*'
Atome → '*Unit*'
Atome → '*Vrai*'
Atome → '*Faux*'
Atome → '*Entier*'
Atome → '*Flottant*'
Atome → '*Chaine*'
Atome → '*IdfMin*'
Atome → '*Fn*' FnArgs FnRetour Bloc

FnArgs → '*ParO*' FnArg FnArgsSuite '*ParF*'
FnArgs → '*ParO*' '*ParF*'
FnArgs → 'ϵ'
FnArg → '*IdfMin*' '*DeuxPoints*' Genre
FnArgSuite → '*Virgule*' FnArg FnArgSuite
FnArgSuite → 'ϵ'
FnRetour → '*Appli*' Genre
FnRetour → 'ϵ

2.2 Genre (Déclaration des types)

Genre → Union
Union → GenreTerme UnionSuite
UnionSuite → '*Pipe*' Union
UnionSuite → 'ϵ'
GenreTerme → GenreAtome
GenreAtome → '*IdfMaj Tous*'
GenreAtome → '*IdfMaj Unit*'
GenreAtome → '*IdfMaj Vrai*'
GenreAtome → '*IdfMaj Faux*'
GenreAtome → '*IdfMaj Booleen*'
GenreAtome → '*IdfMaj Entier*'
GenreAtome → '*IdfMaj Flottant*'
GenreAtome → '*IdfMaj Nombre*'
GenreAtome → '*IdfMaj Chaine*'
GenreAtome → '*ParO*' Genre '*ParF*'
GenreAtome → '*FnGenre*' GenreFnArgs GenreFnRetour
GenreFnArgs → '*ParO*' Genre GenreFnArgsSuite '*ParF*'
GenreFnArgs → '*ParO*' '*ParF*'
GenreFnArgs → 'ϵ'
GenreFnArgsSuite → '*Virgule*' Genre GenreFnArgsSuite
GenreFnRetour → '*Appli*' Genre

2.3 Instruction

Programme → Corps '*FinSequence*'
Bloc → '*AccolO*' Corps '*AccolF*'
Corps → Instruction CorpsSuite
CorpsSuite → Comment FinLigne Corps
Comment → '*Comment*' [Tous lèxemes sauf '*FinLigne*']+
CorpsSuite → 'ϵ'
Instruction → Retour
Instruction → Si
Instruction → Sinon
Instruction → Tq
Instruction → Boucle
Instruction → '*Idf*' IdfInstruction

Instruction → '*FinLigne*'

Retour → '*Retour*'

Retour → '*Retour*' Expression

Si → '*Si*' Expression Bloc

Sinon → '*Sinon*' Si

Sinon → '*Sinon*' Bloc

Tq → '*Tq*' Expression Bloc

Boucle → '*Boucle*' Bloc '*Tq*' Expression

IdfInstruction → Appel

IdfInstruction → '*Aff*' Expression

IdfInstruction → '*Incr*'

IdfInstruction → '*Decr*'

IdfInstruction → '*AffPlus*' Expression

IdfInstruction → '*AffMoins*' Expression

IdfInstruction → '*AffMul*' Expression

IdfInstruction → '*AffDiv*' Expression

IdfInstruction → '*AffModulo*' Expression

Remarques :

- Les fins de lignes à l'exception du cas où elles sont rencontrées en fin d'instruction sont gérés en dehors de la grammaire.
- La vérification qu'un sinon apparait à la suite d'un si n'est pas géré par la définition de la grammaire mais comme un cas particulier dans le code.

3 Sémantique

3.1 Affectations

La gestion des affectations par le langage est extrêmement laxiste, il est possible d'affecter un entier à une variable puis de lui affecter une chaine de caractères plus loin voire de lui assigner deux valeurs de types différents dans deux branches alternatives de code. Néanmoins l'analyseur sémantique garde trace du type de chaque variable au fur et à mesure du flot du programme et effectue une vérification précise de la compatibilité entre les types des variables et leurs utilisations.

3.2 Fonction de première classe et fermeture

Le langage supporte la définition de nouvelles fonctions. De plus, ces fonctions sont de première classe (manipulables comme des valeurs au même titre qu'un entier ou une chaîne de caractères) et supporte le principe de fermeture (par portée lexicale). Elles sont définies exclusivement anonymement (ce n'est qu'en étant assignée à une variable qu'elles obtiennent un nom). La fermeture des variables au moment de la définition de la fonction est réalisée par 'copie'. Autrement dit, les nouvelles affectations a posteriori de la variable enfermée n'affectent pas la fonction :

```
x = 1
f = fn -> Entier {
   retour x
}

x = 2
assert(f() == 1)
```

3.3 Système de Type

Les types sont définis récursivement à partir des types de base. Les types de base sont les types concrets présents à l'exécution :

Unit : Type singleton de la valeur unit

Vrai : Type singleton de la valeur vrai

Faux : Type singleton de la valeur faux

Entier : Type des nombres entiers

Flottant : Type des nombres réels representés avec virgule flottante

Chaine : Type des chaînes de caractères

À partir de ces types peut être construit de nouveaux types avec les deux constructions suivantes :

Union : Etant donné deux types A et B leur union (notée 'A|B') représente un nouveau type. Dans la correspondance de Curry-Howard, il represente le 'ou' logique ; ie. avec P(T) le prédicat associé à un type T, on a "$P(A|B) \iff P(A) \cup P(B)$".

Fonction : Ce type est nécessaire pour representer les fonctions qui sont de première classe. Il est notée Fn(Type_Arg1, ...) -> Type_Retour et encode le nombre d'arguments, leur types et celui du retour de la fonction.

Pour raisonner quant au types des variables le langage utilise la notion de généralisation. Un type A généralise un autre type B si toute valeur assignable au type B est assignable au type A. On remarque :

- $\forall$A, A généralise A
- $\forall$A,B, A|B généralise A et B
- Fn(A1, ..., An) -> R généralise Fn(B1, ..., Bn) -> S si et seulement si, $\forall$i, Bi généralise Ai et R généralise S. (Notons l'inversion pour le type des arguments. Si l'on assimile le type fonction à un type paramètrique, on retrouve le problème de la variance, i.e : Une fonction est covariante en son type de retour et contravariante en les types de ses arguments)

A partir de ce principe de généralisation est défini un nouveau type, le type Tous qui est le type qui généralise tout les autres types. (Ou de manière équivalente, le type qui est l'union de tous les autres types).

Les déclarations de types ne sont nécessaires que dans les déclarations de fonctions. Tout les autres types peuvent être inférés à partir des valeurs littérales et des types des fonctions.

Problème des fonctions récursives : Tel qu'est défini le langage les fonctions récursives posent problèmes. En effet, l'analyse sémantique est purement sequentielle et les fonctions n'ont un nom qu'après avoir été analysée (si elles sont assignées à une variable). Pour supporter les fonctions récursive l'analyseur est obligé de considérer le fait qu'une fonction est sur le point d'être affectée à une variable et de faire passer le nom de la variable en question à l'analyse de la fonction pour que cette dernière puissent reconnaitre les occurences d'appels recursifs. Une fonction peut ainsi être flaggée comme récursive et être considérée comme s'enfermant elle-même. Il faut noter que cette solution ne fonctionne pas pour les fonctions mutuellement récursives, par exemple :

```
pair = fn(n: Entier) -> Booleen {
   si n == 0 { retour vrai }
   retour impair(n - 1)
}

impair = fn(n: Entier) -> Booleen {
   si n == 0 { retour faux }
   retour pair(n - 1)
}
```

La première occurence de la variable 'impair' induit une erreur car elle est considérée non définie par le compilateur.

4 Langage Intermédiaire & "Machine Virtuelle"

Après l'analyse sémantique et avant l'éxecution, le programme est compilé dans un langage intermédiaire semblable à un langage d'assemblage ou à un bytecode. Ce langage intermédiaire est ensuite interpreté par une "machine virtuelle" fonctionnant sur une base de pile. (Plus exactement deux piles mais on aurait très bien pu se ramener à une unique pile).

En effet, on peut grossièrement assimiler le langage intermédiaire à une transformation en notation postfixée du programme source facilement interpretable par une machine à pile. Par exemple l'expression :

```
x = 5 * 6 + 3
```

transformée en un arbre syntaxique abstrait :

```
Affectation
   x
   Somme
      Produit
         5
         6
      3
```

peut ensuite être compilée sous la forme

```
empiler 5
empiler 6
faire Produit
empiler 3
faire Somme
depiler/stocker dans x
```

en assumant que chaque opération (et par extension chaque fonction) prend ses arguments en sommet de pile et y met à la place sa valeur de retour.

Le jeu d'instruction du langage intermédiaire/de la machine virtuelle est défini sur ce principe :

EPL <valeur littéral> : Met la <valeur littéral> au sommet de la pile d'appel

DPL : Dépile le sommet de la pile d'appel

CHR <indice variable> : Charge la variable à <indice variable> et sur la pile d'appel

STK <indice variable> : Stocker le sommet de la pile (en dépilant) à <indice variable>

PRM <nom primitive> : Met primitive <nom primitive> au sommet de la pile d'appel

CLT <nom de la routine> [~]!<nombre de variables à cloturer> : Définition d'une fonction mise au sommet de la pile d'appel

STI <decalage> : Saut inconditionnel

STV <decalage> : Saut si vrai dépilé de la pile d'appel

STF <decalage> : Saut si faux dépilé de la pile d'appel

APP : Appel de la fonction sur la pile

RET : Fin de la fonction en cours et retour à la fonction appelante

Pour representer le code des fonctions sous la forme de ce langage intermédiaire on utilise le concept de routine qui est la suite statique d'instructions d'une fonction. Les routines sont définies par un bloc commencant par le nom entre crochet suivi d'un nombre representant l'allocation nécessaire à la fonction.

Une fonction est ainsi la donnée dynamique constituée d'une réference vers une routine et d'un ensemble de variables enfermées. Elles sont créées par l'instruction **CLT** qui associe ces deux composantes. Le caractère '~' devant le nombre de variables à capturer indique que la fonction est recursive.

Les variables locales de chaque fonction sont representées par la deuxième pile fonctionnant par frame. Autrement dit, chaque fonction défini un nombre de variables locales dont elle a besoin (une allocation) et au moment de l'appel de la fonction la machine virtuelle déplace le curseur de la pile de la taille demandée et la fonction peut referencer ses variables en fonction d'indices determinés statiquement vers cette frame.

L'argument des instructions de saut est défini en terme de nombre d'instructions de décalage par rapport à l'instruction de saut en elle-même. Elles sont suffisantes pour representer les trois structures de contrôle et les deux opérateurs paresseux 'et' et 'ou' (uniquement STV aurait même suffit).

Le langage intermédiaire contient aussi des annotations informatives pour le nom des variables et les positions correspondantes du code source.

Example de compilation

```
[] 4
EPL 1 @1:14
STK 0 &fact_borne @1:12
EPL 1 @2:15
STK 1 &fact_defaut @2:13
EPL 5 @3:13
STK 2 &fact_test @3:11
CHR 0 &fact_borne @5:8
CHR 1 &fact_defaut @5:8
CLT fact0 ~2 @5:8
STK 3 &fact @5:6
CHR 2 &fact_test @10:16
CHR 3 &fact @10:11
APP @10:15
PRM ecrire_ln @10:10
APP @10:10
EPL unit @1:1
RET @1:1

[fact0] 4
```

```
STK 3 &fact0 @5:8
STK 2 &fact_defaut @5:8
STK 1 &fact_borne @5:8
STK 0 &n @5:8
CHR 0 &n @6:7
CHR 1 &fact_borne @6:12
PRM inferieur_egale @6:9
APP @6:9
STF 4 @6:4
CHR 2 &fact_defaut @6:32
RET @6:25
STI 1 @6:4
CHR 0 &n @7:11
CHR 0 &n @7:20
EPL 1 @7:24
PRM moins @7:22
APP @7:22
CHR 3 &fact @7:15
APP @7:19
PRM multiplier @7:13
APP @7:13
RET @7:4
```