Analyse sémantique

L'analyse sémantique, consiste en une validation que la sémantique du langage est correcte. Quelques examples:
- Variables lues avant initialisation
- Labels dupliqués dans un switch
- Réaffectation de constantes
- Visibilité des méthodes invoquées

Types de règles
- **Static rules**: vérifiées à la compilation
- **Dynamic rules**: vérifiées à l'éxécution

Analyse de nom

L'analyse de nom consiste en la validation des identifiants des variables. On valide aussi le **scope** des variables. Les erreurs classiques analysées ici sont:
- **Nested scope**: les variables définies dans les scopes enfant ne sont pas accessible par les parents

```
int x = 10;
{
    int y = 20;
    // x et y accessibles ici
}
// seul x est accessible ici
```

- **Shadowing**: une variable dans une portée interne a le même nom qu'une variable dans une portée externe. La variable interne "masque" la variable externe.
- **Duplicated Definitions**: une variable est définie deux fois avec le même nom → **erreur**

Pour gérer l'analyse de nom on utilise une **tables des symboles** qui peut stocker les infos suivantes:
- Nom de la variable
- Type
- Portée
- Adresse mémoire (pour la génération de code)

Systèmes de types

- **Primitifs**: `int, float, bool, char`
- **Composés**: `arrays, structs, classes`
- **Fonction**: types des params. et retour

Strong vs Weak typing
- **Strong**: conversions de types explicites requises
- **Weak**: conversions implicites fréquentes

Type checking

Literals :

$$\Gamma \vdash n : \text{int}$$
$$\Gamma \vdash \text{true} : \text{bool}$$

Variables :

$$(x : \tau) \in \Gamma \implies \Gamma \vdash x : \tau$$

Addition :

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

Augmenter l'environnement:

$$\text{LETIN}\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma[x \to T_1] \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

Règles de fonctions

Application :

$$\frac{\Gamma \vdash f : \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

Si f prend un T_1 et retourne un T_2 et que e est un T_1, alors $f(e)$ donne un T_2.

Abstraction :

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

Si en supposant que x est de type T_1, l'expression e a le type T_2, alors la fonction $\lambda x.e$ a le type $T_1 \to T_2$.

$\lambda x.e$: focntion anonyme x param, e corps de la fonction.

Arbres de dérivation

Un arbre de dérivation est la preuve visuelle qu'un programme respecte les règles de types, construite en empilant les règles.

$$\text{Lit}\ \frac{}{\Gamma \vdash i : \text{int}} \qquad \text{Ident}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{Add}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\text{Prod}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}}$$

$$\text{Prod}\ \frac{\text{Lit}\ \dfrac{}{\Gamma \vdash 2 : \text{int}} \qquad \text{Add}\ \dfrac{\text{Ident}\ \dfrac{\Gamma(z) = \text{int}}{\Gamma \vdash z : \text{int}} \quad \text{Lit}\ \dfrac{}{\Gamma \vdash 1 : \text{int}}}{\Gamma \vdash z + 1 : \text{int}}}{\Gamma \vdash 2 * (z + 1) : \text{int}}$$

Erreurs de typage courantes :
- Type mismatch : `int x = "hello";`
- Fonction appelée avec mauvais arguments
- Opération non supportée : `"hello" - 5`
- Retour de type incorrect

Inférence de type

Mécanisme pour déterminer les types d'expressions à la compilation **sans annotations de type explicites**.

```
-- Type inféré: id :: a -> a
id x = x
a = id True       -- a :: Bool
b = id "hello"    -- b :: String
```

Type Variables

Placeholders pour types pas encore connus : a, b, T

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

Type Constraints

Conditions imposées sur les types basées sur leur usage.

Formes :
1. **Equality Constraints** : Types identiques requis
2. **Function Type Constraints** : Contraintes sur paramètres et retour

Unification

Résout les contraintes de type en trouvant itérativement des substitutions pour les variables de type.

Algorithme Hindley–Milner

Algorithme largement utilisé pour l'inférence de type.

Étapes :
1. **Type Variables** : Assigner variables de type uniques
2. **Type Constraints** : Générer contraintes basées sur opérations
3. **Unification** : Trouver substitution cohérente
4. **Generalization** : Trouver le type le plus général

```
check :: Expr -> [Expr]
check expr = check' expr []
where
  check' :: Expr -> [String] ->
[Expr]
  check' (Cst _) _ = []
  check' (Var v) env = [Var v | v
`notElem` env]
  check' (Bin e1 _ e2) env = check'
e1 env ++ check' e2 env
  check' (Let x e1 e2) env = check'
e1 env ++ check' e2 (x : env)
```

Interpréteurs

Les interpreteurs se basent sur les étapes d'analyse lexicale, syntaxique et sémantique pour traduire du code source en actions exécutables.

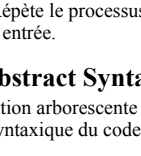
Les langages interprétés sont souvent **plus faciles à apprendre**, **plus rapide à utiliser**, **indépendant de la plateforme** et **plus facile à déboguer**. Cependant, ils peuvent être **plus lents** en termes de performance, **moins optimisés** et **moins adaptés** aux applications nécessitant une haute performance.

REPL

- **Read** : Lit l'entrée de l'utilisateur.
- **Eval** : Évalue l'expression ou la commande.
- **Print** : Affiche le résultat de l'évaluation.
- **Loop** : Répète le processus pour chaque nouvelle entrée.

AST (Abstract Syntax Tree)

Représentation arborescente abstraite de la structure syntaxique du code source.



Vaudrait l'expression: $2 + 3 * 4$, en Haskell nous pouvons écrire la fonction pour évaluer l'AST:

```
eval :: Expr -> Int
eval (Const n) = n
eval (Binary left op right) =
  case op of
    '+' -> eval left + eval right
    '-' -> eval left - eval right
    '*' -> eval left * eval right
    _ -> error "Unknown operator"
```

Statements et états

Un **statement** est une instruction qui effectue une action, comme l'affectation d'une variable ou une boucle.
- **Statements**: modifient l'état du programme.
- **Expressions**: produisent des valeurs.

Environnement

L'environnement est une structure de données qui mappe les variables à leurs valeurs actuelles dans le contexte d'exécution. Les fonctions d'évaluation utilisent l'environnement pour **accéder** et **modifier** les variables.

```
type Env = Map String Value
```

Compilateurs

Le compilateur traduit le code source écrit dans un langage de haut niveau en code machine ou en bytecode. Il existe 2 composants principaux:
- **Front-end**: analyse lexicale, syntaxique et sémantique.
- **Back-end**: optimisation et génération de code.

Processus de compilation

1. Analyse lexicale: code source en une suite de tokens.
2. Analyse syntaxique: tokens en un arbre syntaxique abstrait (AST).
3. Analyse sémantique: validité sémantique du programme (types, portées, etc.).
4. Génération de la représentation intermédiaire (IR).
5. Optimisation de la représentation intermédiaire.
6. Génération du code machine de la représentation intermédiaire optimisée.

Stratégies de compilation

- **Compilation Ahead-of-Time (AOT)**: compilation avant l'exécution.
- **Just-in-Time (JIT)**: compilation pendant l'exécution.
- **Transpilation**: traduction entre langages de haut niveau.

```
// ES6 JavaScript Code
const hello = (name) => {
  return `Hello, ${name}! `;
};

const person = {
  name: "John",
  age: 30,
  greet() {
    console.log(hello(this.name));
  },
};

person.greet();
```

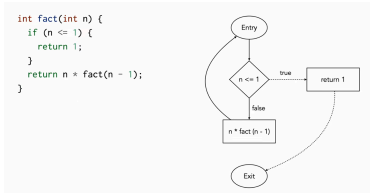
```
// Transpiled ES5 JavaScript Code
var hello = function (name) {
  return "Hello, " + name + "! ";
};

var person = {
  name: "John",
  age: 30,
  greet: function () {
    console.log(hello(this.name));
  },
};

person.greet();
```

Représentation intermédiaire

1. **Three-adress code**: chaque instruction est représentée par une opération et jusqu'à trois opérandes.
2. **Static single assignment (SSA)**: chaque variable est assignée une seule fois, ce qui facilite l'analyse et l'optimisation.
3. **Control flow graph (CFG)**: représente le flux de contrôle du programme sous forme de graphe.



Machines abstraites

- **Turing machines**: modèle théorique de calcul.
- **Stack-based machines**: utilisent une pile pour les opérations (ex: calculatrice).
- **Virtual machines**: environnement d'exécution abstrait (ex: JVM).

Cycle d'exécution

1. **Fetch**: récupération de l'instruction à exécuter.
2. **Decode**: décodage de l'instruction pour déterminer l'opération à effectuer.
3. **Execute**: exécution de l'instruction.
4. **Memory access**: accès à la mémoire si nécessaire (lecture/écriture).
5. **Write back**: écriture du résultat dans le registre ou la mémoire.

Génération du code

1. **Instruction selection**: choisir les instructions machine appropriées pour chaque opération.
2. **Register allocation**: assigner des variables aux registres de la machine.
3. **Instruction scheduling**: organiser les instructions pour optimiser l'utilisation des ressources de la machine.
4. **Code emission**: produire le code machine final.
5. **Linking and loading** (optionnel): combiner plusieurs modules de code et charger le programme en mémoire pour l'exécution.

Architecture cible

Pour générer du code, le compilateur doit connaître l'architecture cible, y compris:
1. **Performance Optimization**: optimisation du code pour tirer parti des caractéristiques spécifiques de l'architecture cible (par exemple, jeux d'instructions, pipeline, etc.).
2. **Memory Model**: gestion de la mémoire en fonction de l'architecture cible (par

exemple, alignement des données, hiérarchie de cache, etc.).
3. **Instruction Set**: utilisation des instructions spécifiques à l'architecture cible.
4. **Endianness**: gestion de l'ordre des octets en fonction de l'architecture cible (big-endian vs little-endian).
5. **Platform Constraints**: prise en compte des contraintes spécifiques à la plateforme cible (par exemple, taille des registres, modes d'adressage, etc.).

Jeu d'instructions

1. Instructions arithmétiques (ADD, SUB, MUL, DIV)
2. Instructions logiques (AND, OR, NOT, XOR, CMP, TEST)
3. Data transfer instructions (LOAD, STORE, MOVE)
4. Control flow instructions (JUMP, CALL, RETURN, BRANCH)

Sélection d'instructions

1. Prupose: traduis correctement les opérations de la représentation intermédiaire.
2. Criteria for Selection: choisir les instructions qui minimisent le nombre d'instructions générées.
3. Techniques: utiliser du pattern matching pour identifier les séquences d'instructions optimales ou un approche table-driven pour mapper les opérations aux instructions.

Environnement runtime

Les différentes responsabilités d'un environnement d'exécution sont:
1. Resource management
2. Memory handling
3. I/O operations
4. OS Communication
5. Function calls

Gestion de la mémoire

Un environnement d'exécution doit gérer la mémoire utilisée par le programme. Cela inclut:
1. **Memory allocation**: allocation de mémoire pour les variables et les structures de données.
2. **Memory deallocation**: libération de la mémoire lorsque celle-ci n'est plus nécessaire.
3. **Memory protection**: protection de la mémoire pour éviter les accès non autorisés.
4. **Garbage collection**: gestion automatique de la mémoire pour libérer les objets inutilisés.

Organisation de la mémoire

L'organisation de la mémoire dans un environnement d'exécution comprend plusieurs segments:

1. **Text segment**: contient le code exécutable du programme.

2. **Data segment**: contient les variables globales et statiques.

3. **Heap**: zone de mémoire utilisée pour l'allocation dynamique.

4. **Stack**: zone de mémoire utilisée pour les appels de fonctions et les variables locales.

Stack et base pointers

```
+--------------------------+
| main's local variables   |
| main's parameters        |
| Return Address to OS     |
+--------------------------+
| "Main program starts\n"  |
+--------------------------+
| foo's local variables    |
| foo's parameters (n=2)   |
| Return Address to main   |
| Pointer to main's frame  |
+--------------------------+
| "Entering foo(2)\n"      |
+--------------------------+
| foo's local variables    |
| foo's parameters (n=1)   |
| Return Address to foo(2) |
| Pointer to foo(2)'s frame|
+--------------------------+
|                          |  <--- SP
+--------------------------+
```

Stack frames

Un stack frame est une structure de données utilisée pour stocker les informations d'une fonction lors de son appel. Durant un appel de fonction:
1. La stack frame est créée et empilée sur la pile.
2. Les variables locales et les paramètres sont accédés via le base pointer.
3. À la fin de la fonction, la stack frame est dépilée et la mémoire est libérée.

Fonction Prolog:
1. Alloue de l'espace sur la pile pour les variables locales.
2. Sauvegarde les registres utilisés par la fonction.
3. Met à jour le base pointer.

Fonction Epilog:
1. Nettoie l'espace alloué sur la pile.
2. Restaure les registres sauvegardés.
3. Retourne au point d'appel.

Optimisation du code

Plusieurs techniques d'optimisation du code peuvent être appliquées pour améliorer les performances:
1. **Constant folding**: évaluation des expressions constantes à la compilation.
2. **Common subexpression elimination**: élimination des calculs redondants.
3. **Dead code elimination**: suppression du code qui n'est jamais exécuté.
4. **Constant propagation**: remplacement des variables constantes par leurs valeurs.
5. **Function inlining**: remplacement des appels de fonctions par le corps de la fonction.

Niveaux d'optimisation

Les compilateurs offrent souvent différents niveaux d'optimisation qui permettent de contrôler l'agressivité des optimisations appliquées au code généré. Les niveaux d'optimisation courants sont:
1. **-O0**: pas d'optimisation, le code est généré tel quel.
2. **-O1**: optimisations de base qui n'affectent pas significativement le temps de compilation.
3. **-O2**: optimisations plus agressives qui améliorent la performance sans augmenter excessivement le temps de compilation.
4. **-O3**: optimisations très agressives qui peuvent augmenter le temps de compilation mais produisent le code le plus performant possible.

Gestion de la mémoire

Les langages utilisent diverses techniques pour gérer la mémoire : contrôle direct (`C` avec `malloc`/`free`) ou automatisation via garbage collection (`Java`, `Python`).

Cycle de vie de la mémoire

1. **Allocation**: la mémoire est allouée pour stocker des données.
2. **Utilisation**: opérations de lecture et d'écriture sur la mémoire allouée.
3. **Libération**: la mémoire n'est plus nécessaire et doit être libérée.

Gestion explicite vs implicite

Explicite : le programmeur gère l'allocation/libération. Offre un contrôle précis mais peut entraîner des erreurs:
- **Dangling pointer**: pointeur référençant une zone mémoire déjà libérée.
- **Memory leak**: mémoire allouée jamais libérée.
- **Double free**: tentative de libérer une zone déjà libérée.

Implicite : le système de runtime gère automatiquement via garbage collection. Réduit les erreurs mais peut introduire des pauses d'exécution.

Définition: Si un bloc mémoire est atteignable, alors il est vivant ; sinon, il peut être récupéré par le garbage collector.

Garbage Collection

Technique de gestion automatique identifiant et libérant la mémoire inutilisée. Prévient les fuites et améliore la sécurité.

Trois techniques principales :
1. **Reference counting**: compteur de références par objet, libéré quand le compteur atteint zéro.
2. **Marking and sweeping**: marque les objets accessibles, balaie pour libérer les non-marqués.
3. **Copying GC**: mémoire divisée en deux zones, objets vivants copiés d'une zone à l'autre.

Structures de données GC

Free list : liste chaînée des blocs de mémoire disponibles. Chaque bloc contient un pointeur vers le prochain bloc libre.

Block header : structure au début de chaque bloc contenant taille, état (alloué/libre), et pointeurs vers blocs adjacents.

BiBOP (Big Bag of Pages) : mémoire divisée en pages dédiées à un type d'objets, optimisant allocation et GC.

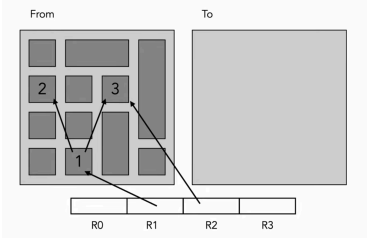
Fragmentation :
- **Externe**: blocs libres dispersés dans la mémoire.
- **Interne**: blocs alloués plus grands que nécessaire.

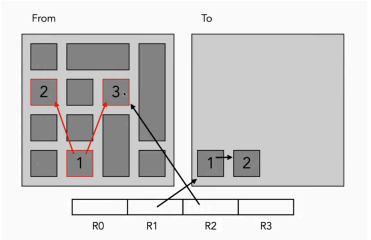
Reference counting

Compteur indiquant le nombre de références vers chaque objet, smart pointers, gère pas les cycles.

Copying GC

Mémoire divisée en deux zones. Lors de la collecte:
1. Parcourir les racines pour trouver objets atteignables.
2. Copier les objets atteignables dans la zone de destination.
3. Mettre à jour les références vers nouvelles adresses.





Avantages : compresse la mémoire, élimine la fragmentation.

Inconvénient : nécessite le double de mémoire (une moitié toujours inutilisée).

Forwarding pointers : maintiennent les références aux objets déplacés. Un pointeur de redirection est créé à l'ancienne adresse pointant vers la nouvelle.

Cheney's algorithm : approche élégante avec un passage en largeur sur les objets atteignables, nécessitant qu'un seul pointeur comme état.

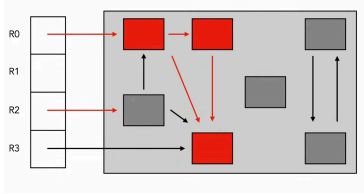
Mark and Sweep GC

Fonctionne en deux phases:
1. **Mark**: parcourt objets atteignables depuis les racines et les marque comme vivants.
2. **Sweep**: parcourt toute la mémoire et libère les objets non marqués.

Phase optionnelle **compactage**: réduit fragmentation en déplaçant objets vivants vers zone contiguë.

Avantage : gère les cycles de références.

Inconvénient : pauses d'exécution pendant la collecte.



Marquage d'objets : ajout d'un bit supplémentaire ou réutilisation du bit de poids faible si adresses alignées sur adresses paires.

Allocation policy :
- **First fit**: allouer premier bloc libre suffisamment grand.
- **Best fit**: allouer plus petit bloc libre qui convient.

Bloc grand - divisé | Bloc petit - fusionné