Sémantique
Associe les variables et les références aux fonctions avec leur définition, check les erreurs de type.
Checklist : Lecture de variable avant initialisation, label dupliqué dans switch, réassignement de constante, complétude du pattern matching, visibilité d'une méthode invoquée.
Erreures sémantiques problème de typage, utilisation incorrecte de variables, utilisation incorrecte de fonc, erreur de logique.
Avertissements sémantiques var non initialisées, conversions implicites, var/fonc inutilisées, param inutilisés, code inatteignable…
Symbol table contient des infos sur les symboles du code (scope, type, bindings), remplie duant le parsing et l'analyse sémantique.
Typing rules
Composé d'une liste de prémisses aux dessus de la ligne et d'une conclusion en dessous, si la liste de prémisses est vide on a un axiome.
On lit une règle de type de bas en haut, jusqu'à ce qu'on rencontre un axiome.
Deviation tree = pile de règles de type, sert à prouver qu'un expression est bien typée.
Avec le Γ on représente l'environnement de typage qu'on peut augmenter avec des bindings. EX :

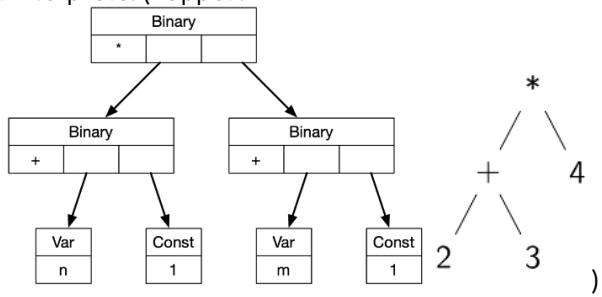
$$\textsc{BinOp}\ \frac{\textsc{Ident}\ \dfrac{\Gamma(x) = \textbf{int}}{\Gamma \vdash x : \textbf{int}} \qquad \textsc{IntLit}\ \dfrac{}{\Gamma \vdash 1 : \textbf{int}}}{\Gamma \vdash x + 1 : \textbf{int}}$$

$$\textsc{LetIn}\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma[x \mapsto T_1] \vdash e_2 : T_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : T_2}$$

```
data Type = ... | TBool
data Expr = ... | If Expr Expr Expr
type Env = [(String, Type)]
typecheck :: Expr -> Env -> Type
typecheck (If cond thenn elze) env =
        case (typecheck cond env, typecheck thenn env,
typecheck elze env) of
                (TBool, t1, t2) | t1 == t2 -> t1
                _ -> error "typing error"
```

Rappel haskell contraintes types Num, Ord (>=, <…), Eq, Show, Read, Integral (div, modulo), Fractional, Enum (succ, pred…).
Interpreteur
ex Python, Javascript…

AST-based interpreters implémente l'analyse lexicale et syntaxiale pour faire un abstract syntax tree et ensuite l'interprète. (Rappel :



Read-eval-print loop REPL ex ghci, lis les inputs, évalue le code, affiche le résultat et loop.
Avantage interpréteurs facile à apprendre (pas de compilation), rapide à l'utilisation, contrôle de l'execution car dynamique et indépendant de la platforme (tout OS).
Désavantages interpréteurs lent, insecurité des données, moins efficace (mémoire), plus d'erreurs (qu'à l'exécution).
Evaluation d'expressions
Pour les languages OOP implémentent le patterne visiteur ou interpreteur pour évaluer les expressions

```
abstract class Expr {
        abstract int eval();
}
class Const extends Expr {
        /* ... */
        @Override
        int eval() {
                return value;
}}
class Binary extends Expr {
        /* ... */
        @Override
        int eval() {
                int lhs = left.eval();
                int rhs = right.eval();
                switch (op) {
                        case '+': return lhs + rhs;
                        case '-': return lhs - rhs;
                        case '*': return lhs * rhs;
                        default : throw new
RuntimeException("unsupported operation");
}}}
```

Pour les languages fonctionnels récursif

```
data Expr = Const Int
        | Binary Expr Char Expr
eval :: Expr -> Int
eval (Const value) = value
eval (Binary left op right) =
        case op of
                '+' -> lhs + rhs
                '-' -> lhs - rhs
                '*' -> lhs * rhs
                _ -> error "unsupported operation"
        where (lhs, rhs) = (eval left, eval right)
```

Runtime errors on les détectent avant l'exécution de tout code (EX : division par 0).
Statements
Assignements, appels fonc, conditions, boucles, IO.
Différemment des expressions les statements fonct des action mais ne retourne pas de valeurs.
L'évaluation se fait de la même manière que les expressions (OOP et récursifs).
Environnement = structure de données qui garde les bindings (Map string value) qui associent les variables aux valeurs, on va donc prendre en compte cet environnement pour les expressions.
Un état (state) fait référence aux valeurs de toutes les variables et structure de données dans la mém.

```
data State = State {
globals :: Env,
locals :: Env
}
```

Appel de fonction

```
data Expr = ... | Call String [Expr]
```

Déclaration de fonction

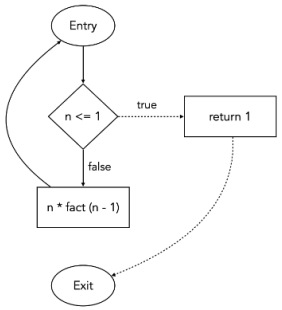
```
data Decl = ... | Fun String [String] Expr
```

Closures

```
type Env = ...
data Expr = ...
data Value = ... | Closure [String] Expr Env
```

Compilateurs
1. Analyse lexicale, 2. Analyse syntaxe, 3. Analyse sémantique, 4. Génération intermédiaire de code, 5. Optimisation, 6. Génération de code.
Représentation en graphe de control flow

```c
int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
}
```



Machine abstraite

Stack

Consider the expression:

```
3 + (4 * 5)
```

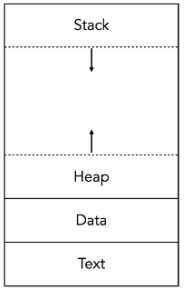
The stack machine performs the following operations to evaluate it:

1. Push 3 onto the stack
2. Push 4 onto the stack
3. Push 5 onto the stack
4. Multiply the top two values (4 and 5), push the result 20
5. Pop two values (3 and 20), add them, push the result 23

Memory management

Memory Segments:

1. Text Segment: Program's executable code and constants, usually read-only.
2. Data Segment: Initialized and uninitialized data, including global and static variables.
3. Heap: Region for dynamic memory allocation, lasting until explicitly deallocated or collected.
4. Stack: Stores local variables, function parameters, and control information, using Last-In-First-Out (LIFO) structure.



Appel aux fonctions → push les paramètre sur la stack et transfère le controle à la fonction.

Optimisation

Constant folding évalue les expressions constantes et remplace les expressions par leur valeur calculée (60 * 60 → 360).

Common subexpression elimination

```
int result = a (b - c) + d * (b - c)
-------------------------------------------
int common = b - c
int result = a * common + d * common
```

Dead code elimination élimine ce qui n'est pas utilisé.

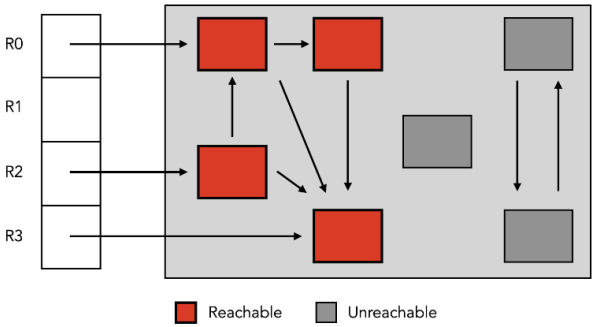
Constant propagation remplace les variables par des constantes (ou litéraux), simplifie les expressions et réduit les références aux variables.

```
x = 5
y = x + 3
----------
y = 5 + 3
```

Function inlining remplace les appels aux fonctions par leur corps.

Memory management

Garbage collection Reachable objects

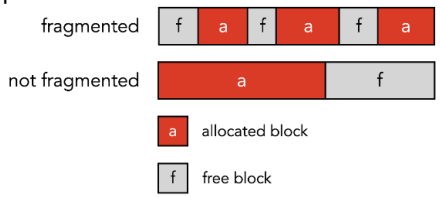


Free list = permet au memory manager de savoir quel partie du heap est libre.

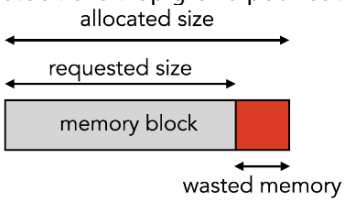
BiBOP (big bag of pages) groupe les objets de taille identiques dans des aires contigues de mémoire appelées pages, les pages ont la même taille $s = b^2$ et commence à un multiple de s, la taille des objets de la page est stockée au début de celle-ci et peut être récupérée en masquant les b moins important bits de l'addresse d'un objet.

Fragmentation

external fragmentation = fragmentation de mémoire libre en petit blocs.



internal fragmentation = gachis de mémoire en utilisant un bloc libre trop grand pour satisfaire la requête.

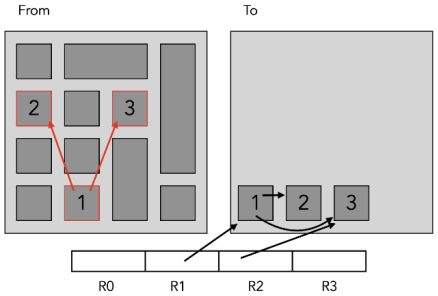


Reference counting

Compte le nombre de références et quand = 0 alors on libère la mémoire, mais attention coût exec et structure cyclique (structure inatteignable mais références présentes car se réfère elle-même).

Copying GC

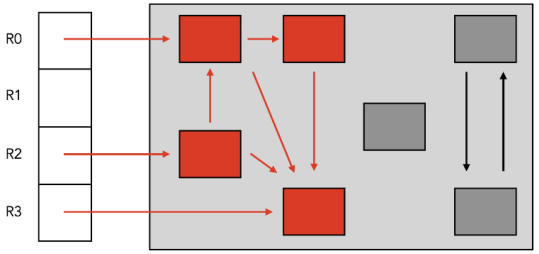
Coupe le heap en 2 égaux et quand le from space est plein (on alloue linéairement dedans) on copie les objets atteignables dans le to space (avec maj ref/ptr) puis on échange les rôles.



On garde un pointeur sur la nouvelle copie pour les réutilisations.

Mark and sweep GC

2 phases, marquage on marque les object atteignable, sweeping les objets alloués sont examiné et on libère les non-marqués (se déclenche quand mémoire pleine et on pause le programme pendant).



Marque les objets avec un bit dans le header (ex : si mm taille alors on utilise le least significant bit de la taille du bloc).

On utilise la free list pour la mémoire dans ce cas car les bloc libres ne sont pas contigus (linked list avec chaque link dans les blocs libres).

Allocation policy best fit ou first fit

Split and coalescing collapse les blocs libres en un quand plusieurs côté et les coupe quand pas tout utilisé.

On peut lazy sweep à la demande de mémoire.

Quand on sweep on parcourt tout le heap et on reconstruit la free list en coalescing en mm temps.