CCMP

Intro

Pourquoi le nom de compilateur : vient du fait qu'avant on allait chercher dans les différentes librairies physiques des bouts de code et qu'on les mettait bout à bout pour arriver à ses fins.
↳ On en faisait une compile.

Si cela peut aider on peut considérer un compilateur comme un long pipe dans lequel le code source va subir de nombreuses transitions (to tokens or AST...) et qui va réellement traduire un bout de code depuis un langage source, dans un langage cible. Aujourd'hui il serait plus relevant d'appeler ce "pipe" un traducteur plutôt que compilateur.

Front-end: Partie du Compilateur consacrée à l'analyse du code source et au langage source. Elle est composée essentiellement du scanner (lexer), parser, du binder et du type checker.

Back-end: Partie qui ~~cible~~ ~~du~~ s'occupe du langage cible et de la trad du code dans ce langage (ex: an assembler). (Canon, instruction selection & register allocation.

Middle-end: Partie qui utilise le langage intermédiaire du compilateur. Composée du Translate et canonicalize parts. Une des parties les plus complexes et grandes du compilateur.

I Deux types de compilateurs: En fonction de combien de code on traite en même temps.

Compilateur étroit: On travaille sur 1 ligne, 1 fonction ...
↳ utile dans le temps quand on avait pas la place de stocker un AST complet.

Compilateur large: On envoie tout un fichier à la fois. (do I use every function?)
↳ permet + de retour sur le code
↳ moins de problèmes de synchronisation, tout à la fois allé hop.

L'utilisation des compilateurs étroits étaient dût vraiment à cause du manque de mémoire available de l'époque. Il arrivait même parfois que le compilateur sâche lui-même load and unload ses propres modules pour gagner et préserver de la mémoire.

CCMP

AST : Abstract Syntax Trees.

BASICS

Syntax concrète : syntaxe avec beaucoup de ponctuation ('(' , ';' ...) qui ne servent pas pour les calculs. Ils agissent simplement comme délimiteurs. Sérialiser un arbre dans quelque chose de linéaire.

Calculs et expressions étant fondamentalement arborescent, il n'est pas necessaire de garder l'arbre de syntaxe concrète avec ses éléments superflus surtout là pour rythmer la lecture et aider à la compréhension du code. Un AST est beaucoup plus compacte.

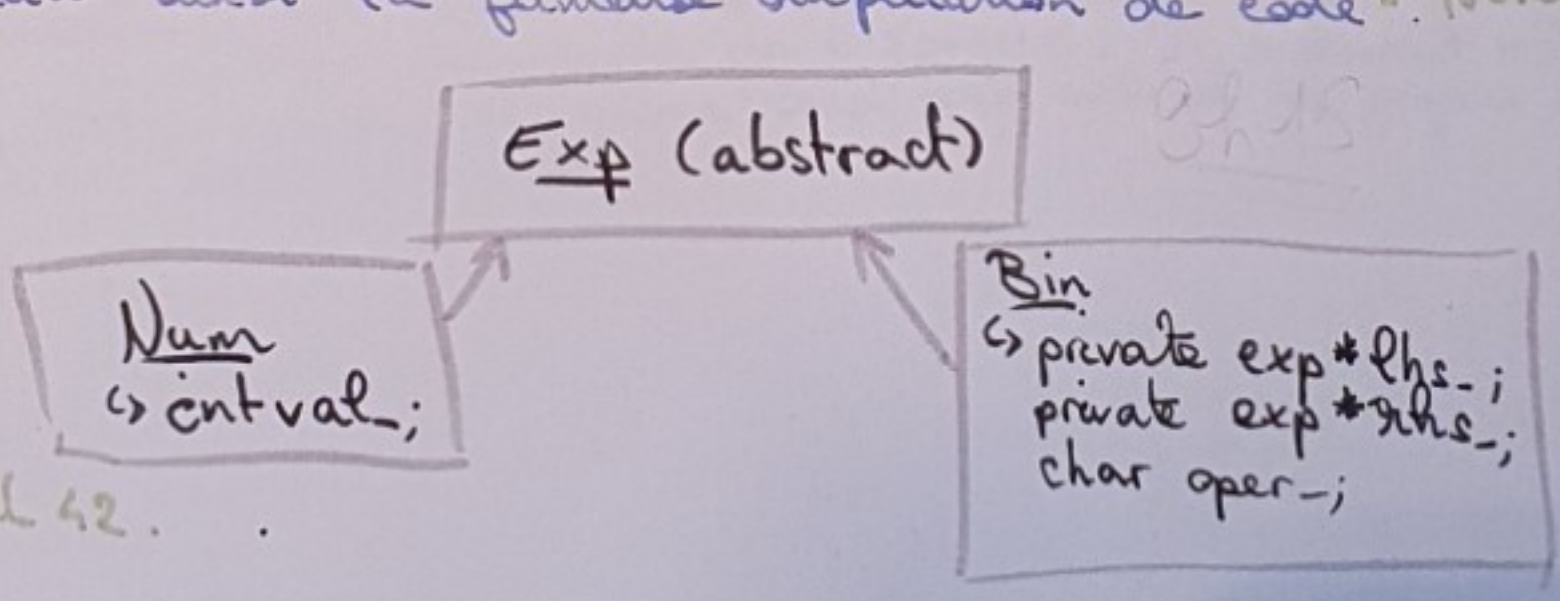
Syntaxe abstraite : syntaxe enlevée de tous les mots clé et ponctuation. (on garde que le nécessaire.

En C++ on peut facilement créer une hiérarchie de noeuds par exemple un object abstrait expr et 5 objets qui en dérive : add, mult, div, sub, Num, qui vont nous servir de noeud specific dans notre AST.

On pourrait aussi avoir aucune autre class et une énum expliquant dans quel type de noeud on est concrètement. On pourrait sinon templater cette unique classe qui permetterait de différencier les different types. Mais cette technique de classe unique reste assez mauvaise d'un point de vue objet et risque de provoquer des problèmes de performances (beaucoup de comparaison pour trouver le bon type) et de clarté.

↳ Ceci dit la solution d'une classe distincte par type de noeud peut entraîner une grosse duplication de code ce qui n'est pas non plus optimal.

↳ Enfin on peut aussi utiliser une hiérarchie un peu plus élaborée possédant toujours la classe abstract "exp" but also une classe (qui hérite de exp) à savoir num (un simple chiffre) et une seconde "Bin" qui va regrouper plus de characteristiques communes aux 4 classes restantes empêchant ainsi la fameuse duplication de code. NOICE



(tentative d'UML ratée) lol

TRAVERSALS

Il est extrêmement utile de pouvoir traverser son AST ne serait-ce que pour le pretty-printer. (ex: analyse des noms (binder), désucrage (ex: for et while -> même instruction à part lors de l'utilisation de continue), type checking, escaping variables / variable d'échappement, inlining, high level optimizations, translation to intermediate representation ...).
BREF le parcours d'arbre est un point clé du compilateur.

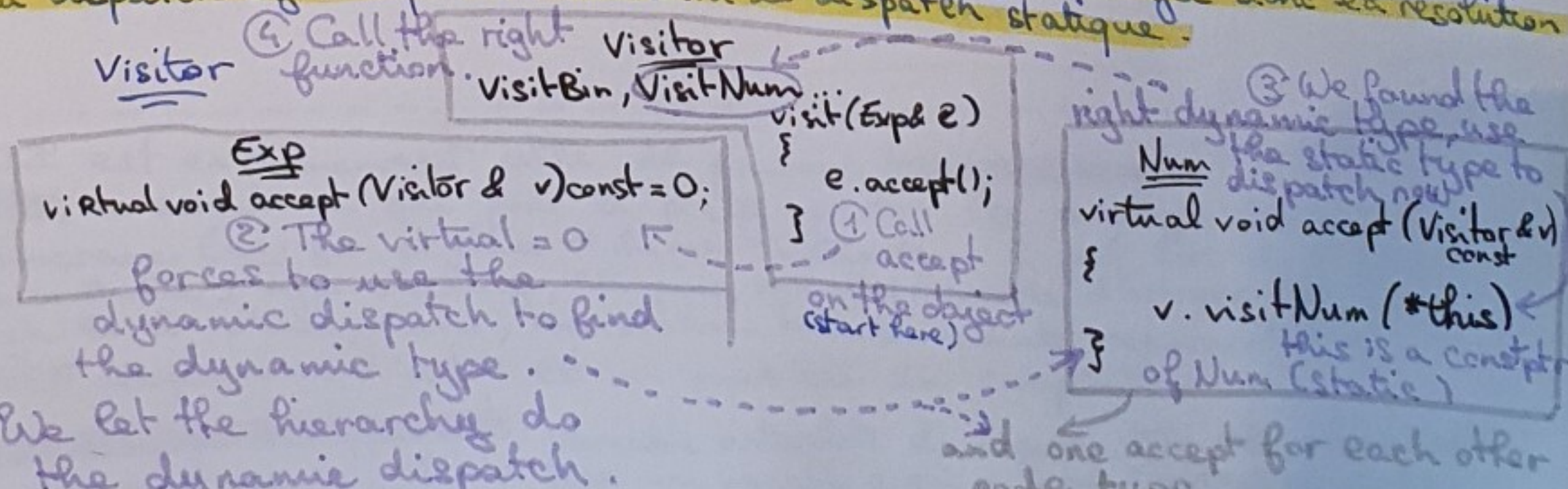
variable en échappement : variable interdite de registre. Ex: Si elle est en Registre 2) elle n'est pas en mémoire DONC une variable a qui on a pris l'address est interdite de registre.

Traversals permettent aussi de get informations et de les stocker. CCMP
Tout type d'informations.

FRONT-END

→ Mais comment faire exactement pour traverser l'AST dans ce cas ? Pour le pretty-print par exemple nous avons une fonction avec un behavior un peu différent pour chaque type de noeud.
↳ On peut peut-être utiliser des amis "friends" qui ont donc accès aux parties privées de chaque type de noeud (et donc peut les print).
Le problème est qu'on peut pas faire de surchage puisqu'on veut pouvoir résoudre de manière dynamique un type alors que la surchage n'utilise que les types statique (la résolution dynamique ne se fait que sur le premier arg d'une fonction (aka l'objet sur lequel on l'appelle). We need du Run time. Dans notre exemple à cause de Bin qui a statiquement deux attributs static (lhs et rhs) qui sont du exp. Donc non pour cette méthode.
↳ On peut utiliser des virtuals pour qu'on obtienne du dispatch au Run time.
On aurait pu utiliser une fonction print qui prend en paramètre un exp (mais c'est une fonction virtual donc elle est implémentée dans toute la hiérarchie) et on appelle ensuite la fonction print sur l'objet lui-même → donc resolution dyn.
It works \o/ mais pas très beau. + On ne veut pas avoir à modifier chacune des classes pour intégrer chaque feature. On ne veut pas que le traitement soit étalé. En + on est pas toujours capable de modifier les fichiers. Ça implique aussi TOUT recompiler à chaque chose added dans le code. Il vaut mieux que les fonctions soient externes à la hiérarchie sur laquelle on veut appliquer le traitement.
=> On veut séparer le dispatch ≠ du traitement
↳ Il faut utiliser un visitor !!! La hiérarchie fait le dispatch pour toi

On a une classe visitor qui veut appeler une fonction sur un type exp. Dans le corps de la fonction du visitor sur l'object exp, en fait en sort que l'on appelle la fonction accept SUR l'objet e de type exp. On a donc une résolution dynamique qui va nous amener sur la fonction accept du type correct. Cette fonction accept va renvoyer vers la fonction du visitor avec son propre type DONC le type dynamique de l'objet exp (qui est static dans sa propre classe). Il y a donc la résolution du dispatch dynamique en utilisant le dispatch statique.



visitor can use the concept of functor in C++ in order to make code more readable and understandable.
(C++ tips: do not mix "virtual" and "templates".)

When you need to traverse with visitors a tree but you only care about some specific nodes (ex: binder who only care about declarations and not about while, if...) you can use default visitors which just do nothing and traverse the AST and derive from that other class of visitors which concentrate only on specific nodes.

Visitor are not only for trees but for anytime we need to make a dynamic dispatch on a hierarchy. (dans des sous-classes d'une classe abstraite). C'est un peu comme une union mais les unions ∄ en C++.

There exists some techniques to optimize visitors by combining some single operations together in order to create a more complex one.

Some applications of visitors :

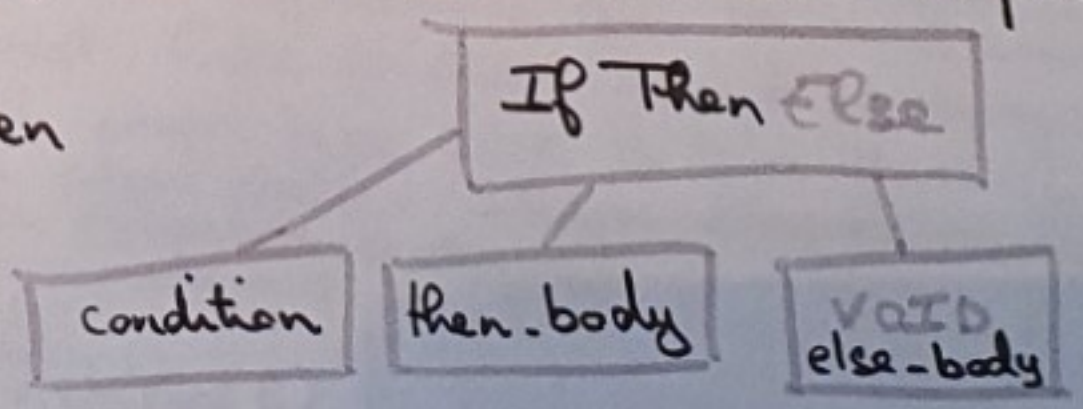
- Only in C++ where other methods are not available.
- désucrage (ex: lambda calculus and curryfication)

pourquoi désucrer? pour simplifier la compréhension du code par le compilateur en effet on permet ainsi à l'utilisateur d'exprimer plus facilement son code tout en ne mourrant pas par la suite lorsque l'on veut faire compiler. (vu que désucrer réduit à une ou quelques façon d'exprimer ce qu'on veut).
Ex: In Haskell the compiler reduce syntactic sugar until reaching the very core of the language, basically only lambda calculus.

Créer des AST est extrêmement compliqué et pourtant on a vraiment envie de pouvoir exploiter ces AST pour plein de raisons. Les developpeurs d'outils tels que pour un IDE qui a besoin de surligner ou faire une indentation correcte se retrouve alors coincé à écrire des algos pour régénérer eux même les AST.

Un exemple de désucrage :

If then → peut toujours être désucrer en If Then Else
en revanche, on sait que le then et le else doivent être du même type donc que mettre dans le else_body de notre AST si on tombe en parsant sur un If then?



null cannot be an option because even though it can match any type, it cannot match void.
If we have a print or nothing in the else_body it then does not respect type checking. The idea is to put void since it does not really makes sense that a If could return something without a else. If it does it is simply a hidden else_clause. So if we have only If then, it MUST be of type void.

→ We can also désucrer for the unary - with a simple 0 - x.
→ On a aussi le α && β que l'on peut désucrer en If α then β else 0 ou plutôt souvent 1 ou 0 : if α then β<>0 else 0.

Autre exemple : Réfléchissons comment désucrer une boucle for.

I said before While loop and for loop are the same things (except if you need to use the continue instruction. (but it does not exist in Tiger).

```
for i := low to hi
    body
```

since i needs to be declared out of the loop scope :

```
let var i := low in
    while i ≤ hi
        (body ; i := i + 1)
```

→ indeed we go until i ≤ hi in TCL.

Pb: if i is for example of type byte & hi is 255. Always true.

↳ maybe add a if somewhere ? Non trivial !

Example : Where

Where is very is, it is just a let. Can be treated during Parsing.

Other example : ?: → Which means a ?: b <=> a ? a : b.

↳ advantage : a evaluated only once.

α ?: β → we could do : if α then α else β
but it would mean that α evaluated twice.

In order to fix the twice-evaluation problem we can use a Let in order to avoid computing 2 times the α AST. So we get :

```
Let var a := α in
    if a then
        a
    else
        β
```

Other example : α < β < γ

α < β < γ → α < β & β < γ

and of Tiger. double eval of β.

We get then.

→ Let var b := β in
α < b & b < γ

→ pb: if α changes things, it needs to be eval before β.

So →

```
Let var a := α in Let var b := β in
a < b & b < γ
```

Finally visitors can be use by :

- desugar (in oriented object programming or not)
- Bounds checking
- Function inlining
- Remove useless function definitions. Known as Pruning and removing dead code
- Escapes Visitor for the escaping variables / variables en échappement
- Translator → translate into intermediary language.

Scanner and Parser

CCMP
FRONT-END

11

symbols : on met une seule occurrence de chaque symbol associé à leur addresse en mémoire ce qui permet de comparer 2 identifiants en comparant leurs adresses sans aucun besoin de faire appel à la mémoire (et donc c'est cool si ~~c'est~~ y des doublons ou whatever ?).
À la fin la table des symboles est vidée.

Le plus simple pour parser serait de ne pas parser du tout.
On pourrait se passer du parser si par exemple on avait la possibilité d'avoir un IDE qui construit lorsque tu tapes, ton AST.
On appelle ça des éditeurs projectifs. Ça enlève pas mal de chose et on devient + rapide since on a déjà fait l'AST.

Pour continuer sur les symboles, il peut y avoir un problème si l'on compare seulement les adresses si l'on veut obtenir non pas une égalité mais une comparaison >. Dans ce cas soit on garde le strcmp pour les comparés seulement pour ça soit on prendre un int associé à l'adresses dans le but de garder l'ordre des identifiers. Sinon si l'on se basait uniquement sur leur addresses et qu'on voulait print la liste dans l'ordre des identifiers, on obtiendrait pas toujours le même résultat (donc non-déterministe).

Reprise sur erreur.

La reprise sur erreur est cruciale pour ne pas s'arrêter à la première erreur rencontrée déjà parce que on veut pas passer 40 ans à fix des erreurs de compil et qu'on veut pas passer 40 ans de plus à attendre que l'on recompile pour chaque erreur TOUTE le programme.

→ Il va falloir que l'on nettoie la pile de tout élément qui nous a amené dans cette erreur.
→ Il faut qu'on jette quelques éléments de l'input jusqu'à trouver un fresh start pour recommencer à parser.
Il faut qu'après ça la pile soit compatible avec le début de l'input...
↳ On peut utiliser par exemple dans Yacc le mot clé "error" et notre règle après qui va permettre de générer un parser avec cette reprise sur erreur ici

This however may introduce new errors.
We can use some values as if the error never existed and maybe create an error class such that whenever this variables that we know is wrong, since introduced by the compiler, if a comparison or an exception trigger an error it will simply ignore this fact. (Prevent the errors to cascade).

parser pure → ne fait pas d'effet de bord sur les variables globales.

parsing driver → Will coordinate actions between lexer and parser
Manipulates also the input flex.

Parser can call it self recursively.

Sometimes you want to have more than only one entry in your parsing (to go back to a certain variable declaration or so)
It can be done by specifying two starts possible in your parser generator, and add some conditions to know which one to choose.

@Binding

front-end

Associé signifiant à son signifié

We have many binding time at different moment of the process of compilation. (for scopes in the code, at the linking of the libraries ...)
For the binding of names and variables we will have first to face the scopes and variables shadowing others. If we declare a variable "a" in a scope, we want that it exists even after we entered and exited an inner scope containing another "a" variable. A simple dictionnary used to map a name to their values is then not sufficient. We could use for example a list of dictionnaries, pushing a new one at each new scope introduction and poping it at the end. To search for a symbol we simply need to search more and more deeply in the list for it until we find it. Pb: We might have to go very deeply to find some symbols introduced early (for example function froms outside libraries).
→ An alternative would be to make a deepCopy of the previous dictionnary at each new scope and push it instead of an empty one. No need to go deeply anymore. Pb: Plz don't do this??? Your poor memory...

But if we think about it, no need to se casser la tête avec le shadowing des variables si celles si ont forcément un nom différent. You cannot really ask that from the developper but you can rename the variables to not have conflicts anymore. Pb: It's time consuming and if you want to use the debugger, the variable names changed... How unfortunate. You can use the 7!! addresses of variables as unique identifiers! \o/ → Now it's okay.

Names, Scopes and Bindings

CCMP
FRONT-END

Before performing the type checking we will do a first traversal of the AST to do the action of binding (liaison des noms) in order to be able to recognize in any case, which variable are we really talking about.

name : identifier, symbole ... désigne quelque chose, une valeur.
refers to some entities : variable, type, function, namespace, constant, control structure ...
(en C: on a le droit a 31 charactères pour un identifier).
↳ peut rendre les choses complexes ex: quand le C++ qui supporte l'overloading traduisait vers le C faisant changer le nom d'une fonction pour chaque "overload" de celle-ci. 31 chars facilement dépassés).
Aujourd'hui C++ n'est plus que simplement un traducteur but still need to be compatible with C et donc with les std libraries et donc le C-linker.

alias: many ways to call 1 thing.
lifetimes of objects: When are they created and destroyed?
Scopes: When are **names** created and destroyed?

Even if you get rid of the shadowing problem by giving an unique identifier, you still need to keep track of the different scopes in order to know the lifetime of a variable, know when it appears, and when it dies. (You also need to know if you are using a **global, static** or **local variable** (OR an **escaping variable** ...)) notamment pour savoir où la trouver.

Les tables de symbols sont en effet pas trivial, il n'y a pas vraiment de bonne data structure bien adaptée, il faudrait une espèce de Hashmap / liste chaînée utilisée pour aussi marquer le moment d'un changement de scope.

→ Cela peut-être aussi utile au moment de détruire les informations gathered during bindingtime. Here again the question is: When to destroy those informations? the end of AST traversal ? end of a scope? destruction de AST?
↳ Dresolution: Call at each new discovered scope recursively the visitor for scope binding, it will have a copy each time, but no more scope problems.

Tout le but du name binding est de mettre des liens entre déclaration de variable et utilisation. On veut étiqueter notre AST. D'où cette question, quand enlever ces étiquettes?

Naming uniquely each variable to its address can be very useful later on for type checking (go to declaration of the AST node which contains the type.
→ Une des choses à faire gaffe : le break: que l'on doit relier à sa boucle. (il faut aussi checker les multiples def et les missing def de variables).

CCMP

Avoir un static scoping permet de enable many advantages such as static binding and hence, static typing (for type checking). For the right languages, it also enable strong typing, which means that no other type error can be introduced at run-time (everything is checked at compile time. (ex: Ada, Tiger ...) Les langages dit "sûre".

Earlier binding → more secure, more clear
→ less flexibility
↳ Very hard to find ax balance.

to handle scope explicitly : avoir une table de symbol qui a l'opération do/undo, qui permet d'être conscient des scopes.

On peut voir l'étape du binding comme un moment où on trace des liens entre définition et utilisation. À chaque utilisation on pose un pointer vers le noeud définition. → identifiant unique = address

Overloading → corresponds to several object statically distinguishable but having the same name. (Synonym) ⇒ ⇑ (Homonym)

Alias → same object having, being referenced by, several names.

L'overloading peut poser pas mal de problème car il n'est plus possible de séparer le type checking de la liaison des noms puisque résoudre le symbole d'une fonction servient à étudier ses types d'entrées (et parfois type de sortie.

↳ overloading est un type de sucre syntaxique, les noms sont unifiés (rendre unique) plus tard, on change leur nom en ajoutant ≠ des préfixes en fonction du prototype de notre fonction. Pb: si en désucrant deux compilos differents ne donne pas le même unique id → Problème de compatibilité.

Autre complication, les variables non-locale en "échappement" (lambda shifting en C par exemple ou on passe l'adresse d'une variable locale à une fonction).

↳ Pb: Comment reconnaître une variable d'échappement ?

On peut se souvenir d'un niveau d'imbrication des variables au moment de leur définition, à chaque nouvelle fonction on augmente d'un niveau, reste plus qu'à verifier lors de l'utilisation d'une variable si celle-ci est du même niveau que la fonction courante. Si elle ne l'est pas on la marque alors en échappement.

⚠ Une variable en échappement ne peut pas être placée en registre car celle-ci est passé par pointeur (incompatible avec registre).

↳ Si on a un doute quant à notre algorithme qui fait le calcul pour décréter qu'une de nos variable est en échappement, il vaut mieux marquer toutes les variables en tant que échappement pour ne pas qu'elles puissent être mises dans des registres.

27/06/20

CCMP
Front-end

Type Checking

After Binding, need to make sure that every operation made is valide, that is, it can be done with the types given.

→ Types are not really useful except to help the programmer, the deeper we get on the compiling process, the less the types matter until eventually reaching assembly language that doesn't contain any. (RK: Assembly can sometimes have types, but they are on the instructions and not the operands).

Coercion → implicit conversion from a type to another. There exists 2 types:
- ↳ widening → to a wider type (ex: int to float)
- ↳ narrowing → to a smaller type (ex: float to int)

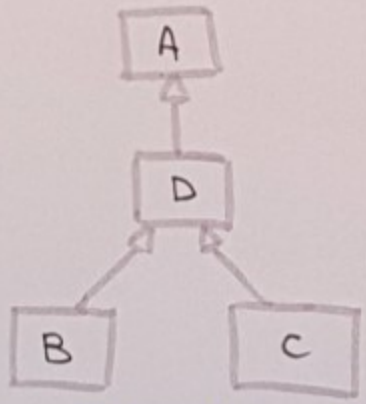
Type equivalence → can change the type without needing coercion.
- ↳ Name type equivalence → two variables defined in the same declaration or the same type name.
- ↳ Structure type equivalence → 2 variables have equivalent types if those types have identical structures.

Les règles d'inférences permettant d'étudier et de déduire d'une certaine expression le type de celle ci.

ex: $\frac{\vdash a : int \quad \vdash b : int \quad \vdash c : bool}{\vdash \text{if } c \text{ then } a \text{ else } b : \cancel{bool}\ int}$

Ici on traite des cas de type primitif. What would happen if for example we want to do the same thing on more complicated types with the notion of heritage?

Let's have this hierarchie:



if a and b have the exact same type (A, B, C or D) then it is simple and the expression is of this same type too.
but if we have:

$\frac{\vdash a : B \quad \vdash b : C \quad \vdash c : bool}{\vdash \text{if } c \text{ then } a \text{ else } b : \rightarrow}$ could be A or D.

Seems wiser to choose the lca (lowest common ancestor) though.

→ It is the notion of lub (least upper bound) defined by:

lub(X, Y) = Z (the lub) (ou X implicitement coercible to Y)
↳ $X \leq Z$ and $Y \leq Z$ (X et Y sont des sous classes de Z).
and if there exist a common ancestor Z' to X and Y then it is to Z too.
$X \leq Z'$ and $Y \leq Z' \Rightarrow Z \leq Z'$

Le type checker peut finir (selon l'implémentation) le travail du binder par exemple pour l'overloading ou la résolution des noms ne suffit pas à associer une fonction à un appel.
Dans ce cas ci on aurait aussi besoin de comparer des fonctions grâce à un $\leq$, ce qui nous permetterait d'avoir la fonction la plus adaptée aux types dynamiques. (overloading pour des args types primitifs ne pose pas ce problème naturellement).

27/06/20

CCMP
Front-end

pour pouvoir comparer plusieurs fonctions entre elles on peut par exemple comparer chacun de leurs arguments 1 à 1 et si l'une a tous ses arguments $\leq$ à la seconde alors cette fonction est $\leq$ à l'autre.
On remarque donc que ça n'est pas un ordre total et donc qu'une fonction peut ne pas être comparable à une autre.

→ Si l'on crée plusieurs niveaux de comparaisons, au niveaux 0 les fonction les plus petite and so on, we just need to go through every level and get the lowest fonction qui répond à nos attentes de types.

↳ Pb: Si plusieurs fonctions valables apparaissent sur le même niveau, alors c'est un appel ambigü et aucune conclusion peut vraiment être faite.