APRES L'ANALYSE SYNTAXIQUE

- Beaucoup des règles que doivent respecter les programmes sont difficiles à exprimer par la syntaxe (p.ex. la déclaration d'une fonction doit inclure au moins une instruction **return**)

- voire impossibles (p.ex. une variable utilisée doit être déclarée, et toutes les règles sur les types)

- Traduction en code machine: très souvent en deux étapes (ou plus): traduction en code intermédiaire avec les caractéristiques d'un code machine mais sans les particularités d'une machine; l'argument $m \times n$

- Ecrire le compilateur en quel langage: en code machine (historique), en un langage de haut niveau (questions des poules et des oeufs), génération automatique à partir d'une description de la machine cible (marche moins bien que pour l'analyse syntaxique)

- Questions intéressantes sur l'affectation d'espace mémoire pour les variables (éventuellement récursives) de façon que l'accès aux variables et les appels des procédures soient efficaces

- Générer du code efficace; plutôt corriger le code généré

- Point de départ: un arbre syntaxique produit par l'analyse; cet arbre peut être réel ou virtuel

ATTRIBUTS (SYNTHETISES)

Une méthode très utile pour spécifier les propriétés non-algébriques des programmes et leur traduction.

- Associer aux noeuds de l'arbre syntaxique des **attributs**, valeurs d'un type quelconque

- Donner des règles locales permettant à calculer l'attribut d'un noeud à partir de ceux de ses fils

- exemple: une expression arithmétique; est-elle de type réel ou entier? un attribut de type Booléen suffit

- La règle à utiliser dépend, en général, de la production de la grammaire utilisée à ce noeud (et donc le cas des feuilles est manié)

- L'écriture des règles de calcul peut se faire dans un langage impératif (C) ou dans un langage spécifique (sans effets de bord?)

  Ecrire sans utiliser des effets de bord, c'est-à-dire dans un style fonctionnel a des avantages, comme ceux de générer un analyseur de la grammaire au lieu de l'écrire à la main:

  - clarté
  - maintenance plus facile
  - simplicité

EXEMPLES

- Une propriété non-algébrique que tout programme doit vérifier: un attribut **valable** qui est vrai si chaque identificateur utilisé dans un programme est aussi déclaré quelque part dans le programme.

- Une propriété qui pourrait être utile pour la génération de code performant: les fonctions sans effets de bord.

ATTRIBUTS HERITES

Un nouveau type d'attribut où la valeur dépend de la valeur du père (et peut-être des frères).

- exemple: quel registre utiliser pour chaque résultat calculé dans une expression?

- En général: chaque noeud a un ensemble d'attributs (déterminé par le terminal ou non-terminal de la grammaire), chaque attribut est classé hérité ou synthétisé

- A chaque production de la grammaire est associée une règle de calcul pour chaque attribut synthétisé du symbole à gauche et chaque attribut hérité de tout symbole à droite.

- Il existe la possibilité que le calcul des attributs boucle!

EXEMPLES

- Code est inaccessible s'il suit une instruction de laquelle il n'y a pas de sortie possible ou s'il est partie d'un bloc de code inaccessible (propriété héritée); Il n'y a pas de sortie possible d'une boucle (**repeat**) qui ne contient pas de **break** (propriété synthétisée)

- (En certains langages) une boucle **for** est interdite d'affecter à la variable de contrôle; quel est la variable de contrôle est une propriété héritée; est-ce qu'une instruction affecte à cette variable est une propriété synthétisée.

TYPES DES EXPRESSIONS

- Le type d'une expression est synthétisée de ceux de ses composants, constantes ou identificateurs

- Le type d'un identificateur dépend d'une table de symboles, héritée

- La table est synthéthisée des informations dans des listes de déclarations

- Les types déclarés dépendent des typedefs antérieurs et donc de la table héritée

- et ainsi de suite?

Donc une question intéressante: dans un ensemble d'attributs interdépendants, comment décider le bon ordre d'évaluation
(s'il y en a!)

YACC/BISON

Yacc/Bison permet d'utiliser des attributs d'une façon très restreinte:
chaque noeud peut avoir un seul attribut et il doit être synthétisé!
dans les règles, l'attribut du symbole à gauche s'appelle $$ et ceux des symboles à droite s'appellent $1, $2 etc.
pour avoir plusieurs attributs synthétisés il suffit de déclarer des types **union**
pour avoir des attributs hérités, il faut écrire du code C qui gère une pile (des attributs de tous les noeuds sur le chemin de la racine);
le programmeur doit décider l'ordre de calcul des attributs;
et ça doit être un ordre qui correspond à un parcours classique de l'arbre, donc, par exemple un sous-arbre ne peut rien hériter d'un oncle droit.

Eli/LIDO

Un système qui permet d'écrire des règles de calcul pour les attributs synthétisés et hérités dans une syntaxe restreinte de type C (l'expression calculée pour la valeur d'un attribut doit être l'appel d'une fonction avec des paramètres simples);
Le système décide l'ordre de calcul des attributs;
...