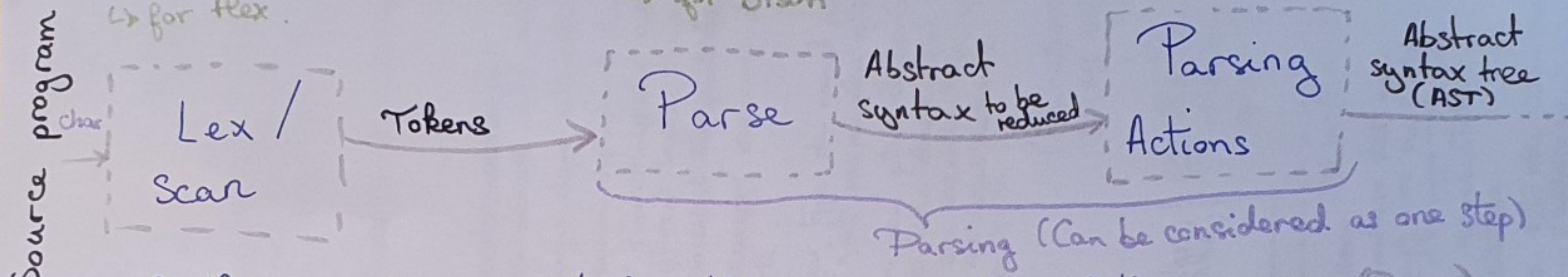
FRONT-END



- Break the source file into individual word, or tokens.
- Basically le cours de THL.
- The goal of the scanner is to provide "tokens" that will be used by the parser.
- Many tools are available like for example Flex which takes a .l or a .ll and generates the right scanner. (usually a .ll is organized in 3 parts separated with %%: definitions, rules and user code).

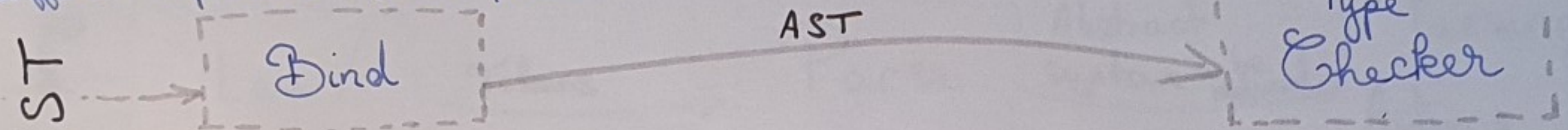
- Analyse the phrase structure of the program. (Parse).
- Build a piece of abstract syntaxd tree corresponding to each ph.
- Le cours de THL peut bien aider ici aussi.
- Bison (takes .y or .yy) takes rules and the list of tokens to parse correctly.
- The goal of the parser is to ensure that the source code provided by the user is valid syntatically (not semantically !) regarding the language.
- It is also the moment to create the AST or some parts.
- In OOP like C++, for AST and in order to pretty-print it, we need to be able to call the "print" function associated to each kind of node without knowing its type beforehand. The dynamic resolu° being available only for the first argument, we need to use some technique cc to be able to the ← call right print function for each object. We use the visitor design pattern. (see the course for more informations).

To make the code nicer we can use functor (by overloading the () operator). (We can also use in C++ xalloc that permits to store information that we want to keep in an ostream in the case you only use << operator wich can take only 2 args no more).

- La reprise sur erreur est très importante car elle permet de ne pas attendre constamment la compilation est de donner toutes les erreurs en même temps. Il faut par exemple en cas d'erreurs trouvée dépop notre pile de parsing et dépop certains look ahead de l'input pour arriver à un state où la pile et l'input redeviennent compatible.

Now we arrive in the part of **semantic analysis**. If the previous part was more about syntax analysis and corresponding to a specific grammar, now we want to determine what each phrase means, relate uses of variables to their definitions, understand scopes and check types of the different expressions of each phrase.

FRONT-END



- The goal of the binder is to link available variables uses to their declarations, wich means match some variable with its identity in order to be sure of which variable we are talking about. It may become complex when variable shadowing and scopes are allowed in the language.
- When we want to bind the names only some types of node in the AST are relevant (we don't care about for loops, if...) we can then use default visitors wich make a "parcours à vide" of the AST and concentrate only on specific nodes.

- The goal of a type checker is to ensure that operations work on compatible types.
(We may run here in the same problems as encountered for the pretty print resolved with the visitor pattern.

Le binder doit s'assurer de:
- annotate / étiqueter l'arbre pour relier utilisation d'un symbol à sa définition.
- utiliser descoped symbol table si le langage supporte le scoping et shadowing... (RK: If all ids unique the problem is reduced to scoping, link name to use and lifetime of a variable and can be solve with regular containers such as #map). Ou des regular containers et la récursion qui reglera seule les problèmes de scope.
- Checks multiple definitions } raise error (trigger eventuellement la reprise sur erreur)
- Checks missing definitions
- Need to bind les "Break" à leur boucle. Il est plus simple que le Parser délègue cette tâche au binder.

→ Pour obtenir un unique id par variable on peut le référencer par son addresse.