Augmenting CFGs

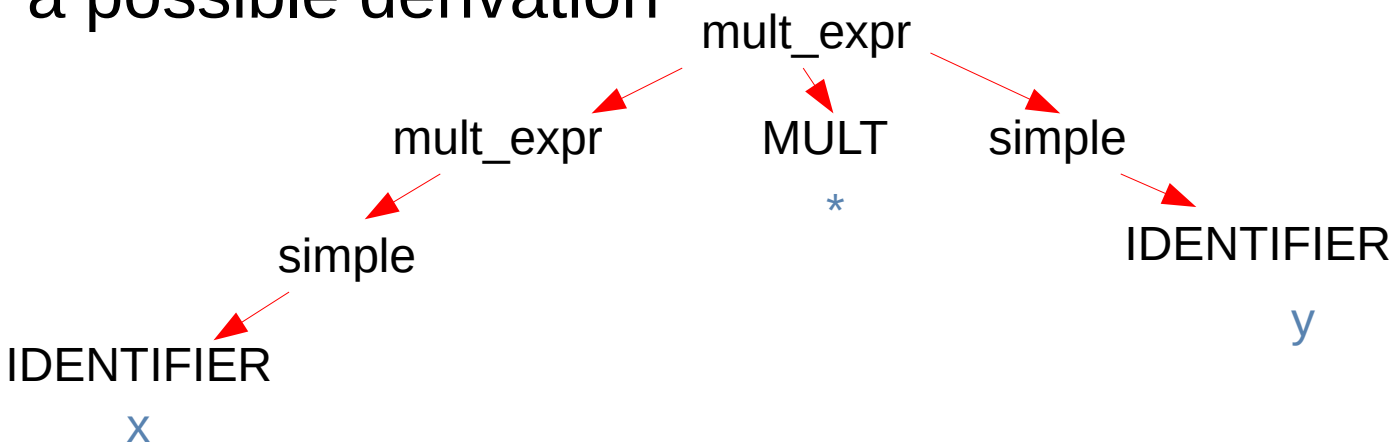
- Context free grammars can check the structure of your token sequence, but there are many things they simply can't check, such as
  - Checking variables, functions are declared before use
  - Checking expression operand/operation types are compatible
  - Identifying implicit type conversions where necessary
  - Identifying how the values of variables, parameters, etc should be computed and updated
- To check these things, we need to add/associate extra functionality on top of our CFG
- Fortunately, lex and yacc (and similar tools such as flex, bison, antlr, etc) provide us with ways to do this

Context and grammars

- Context free grammars apply their rules without (wait for it) knowledge of their context!
- Thus a rule knows nothing about what happens before or after, and is unable to do things like typecheck, or check for declaration-before-use
- Our “augmentations” involve communicating extra information with each applied derivation step, identifying the context it is operating in
- this can be by associating extra data with the derivation step itself, or through global tables (so data can be looked up as needed)

Example: simple expression

- Consider an expression like x*y: we need to know the data type for each, whether they have been declared, which specific variable they refer to in overlapping scopes (e.g. if there is a local x and a global x)
- Consider a possible derivation



Passing info up/down the tree

- Our parse trees can be generated/analyzed recursively, from our top “start” non-terminal: the leaves or bottom layer of the recursion is our set of tokens
- As the tree is constructed (the top-down recursive calls) we can theoretically pass environmental context, such as the current scope, the currently declared variables, etc
- As the tree is evaluated (the bottom-up returns from the recursive calls) we can fill in information such as the specific types of operands in an expression

A lookup table approach

- Many compilers store information about symbols in a table as they process the source code, giving each symbol a unique name, storing its type, scope, value (if known), etc
- They can also store information about scopes, giving each scope a unique identifier and tracking which scopes are nested in which others
- Between the two tables, whenever a symbol is used in the source code it is possible to check it has been declared (and in which scope) and perform type-checking

A lispy approach

- An alternative to global symbol tables is to pass each function a list of its environment values (which scope it is in, what variables/functions are visible, what their values are, etc)
- Recall that the structure of lambda-block-closures and lambda functions in gcl included three environment lists, containing just such information
- The environment data passed can become get quite large, but it also enables lisp functions to parse their own environment information

lex and yacc

- We'll do some work with lex and yacc, each of which provides us with the ability to
  - associate a struct with tokens/nonterminals, containing relevant data
  - create C data types and global variables usable in those derivation steps
  - run C code when a derivation step is applied: e.g. to apply type checking, or to act as a compiler or interpretter and translate the source code into something else