Hand crafted context sensitive checks

- Many issues that CFG/parsing cannot resolve:
- Variables, functions, constants declarations vs use
- Identifying/resolving scoping issues
- Type checking, implicit type conversions
- Checking number/types of parameters in function calls

Declarations vs use

- If we parse statements in sequence, and require items to be declared before use then we can add to symbol table on declaration and validate on use
- If we allow implicit declarations then can do both the insert and the validation on first use
- For mutually recursive functions it's not possible to declare both before use – allow forward declarations (tell compiler to allow it for now, promises full compatible definition coming later)

Declare anywhere

- What if we can use something before we declare it?
- Compiler could make two passes: first pass fill in symbol table, second pass does error checking
- Could use implicit forward declarations: assume it's ok when sees the use, makes a list of all the uses that need to be checked, then when sees definition it fills in symbol table and goes back to the list of things to check

Scoping issues

- Suppose we have nested (lexical) scopes
- Give each scope a unique identifier
- When item is declared, record its scope in symbol table
- During compilation, keep a stack of current scopes (bottom of stack is global, each time you enter a scope push its id, when you leave the scope pop its id)
- When resolving use of an item, search the stack from top down, looking for “closest” definition

Dynamic scoping

- Dynamic scope: called function can “see” all the items defined in the caller
- Could maintain one stack for each defined item name (e.g. a stack for X's, a stack for Y's, etc.
- Push a new item on top of stack when it is defined, pop it when that item's lexical scope ends
- When code references a name, use the definition on top of stack
- Requires a collection of stacks: one per used identifier

Type checking

- Assuming we have dealt with the declare-before-use vs declare-anywhere issues
- Where a value is used, its actual type must be checked against the expected type
- Where an expression involves an operator and multiple arguments they must all be compared with one another for compatibility
- If types are not identical, must decide if inserting implicit type conversion is appropriate (e.g. integer-->real)

Resolving function calls

- Must address declare-before-use vs declare-anywhere issues (as with variables)
- Must address scoping if nested declarations allowed
- Must check number of parameters passed against number expected (arity), and must check types passed / expected
- For functions with optional parameters, must insert the defaults in call implementation where needed

Handling variadic functions

- Need to decide how to implement functions that accept variable numbers of parameters
- One possibility: in stack frame, push parameters right-to-left (so “first” parameter is on top), then push a count of the number of parameters passed

Code generation...

- Will address many other implementation issues w.r.t. the target language when we get to code generation