CSC-363 Lecture 04B
Semantic Analysis

Colin McKinney

11 February 2026

Semantic Analysis

Semantic analysis is different than tokenizing or parsing. Tokenizing and parsing are in some sense concerned with local structure, whereas semantic analysis is concerned with overall structure and meaning. Things we'd like to be able to analyze:

- Ensure variables are declared and initialized before use
- Handle type-checking: right now we only have ints in ac, but if we added floats, we'd want to make sure that we either disallow mixed BinOps or handled type conversions correctly.
- Related to type-checking: make sure that functions are passed the right datatypes and return the right data type. Not relevant for ac but will be for C.
- We could theoretically also do some optimization here, such as to cull "dead" code (compare example from first lecture). However, usually this is done at the intermediate representation level, which requires doing some partial code generation first.

Syntax vs. Semantics

Consider the two following ac programs:

- `ia, a = 3 +`
- `ia, a = 3 + b`

What's the difference?

Syntax vs. Semantics

Consider the two following ac programs:

- `ia, a = 3 +`
- `ia, a = 3 + b`

What's the difference?

- `ia; a = 3 +` is not syntactically valid. The structure is incomplete, and so we can't derive any meaning from it.
- `ia, a = 3 + b` is syntactically valid: it has the correct structure at a local level. But it is not meaningful. It is not semantically valid.

Parsing asks: is this shaped like a program?
Semantics asks: does this program make sense?

Order Sensitivity

Our tokenizing and parsing operated on a line-by-line basis. Naturally, we did these lines in order using Python's `for X in Y:` loop, but we didn't have to.

Order Sensitivity

Our tokenizing and parsing operated on a line-by-line basis. Naturally, we did these lines in order using Python's `for X in Y:` loop, but we didn't have to.
Semantic Analysis requires us to process the statements in order. Contrast:

Program 1:

```
ia [IntDcl(a)]
a = 3 [Assign(a)][IntLit(3)]
pa [Print(a)]
```

Program 2:

```
ia [IntDcl(a)]
pa [Print(a)]
a = 3 [Assign(a)][IntLit(3)]
```

Order Sensitivity

Our tokenizing and parsing operated on a line-by-line basis. Naturally, we did these lines in order using Python's `for X in Y:` loop, but we didn't have to.
Semantic Analysis requires us to process the statements in order. Contrast:

Program 1:

```
ia [IntDcl(a)]
a = 3 [Assign(a)][IntLit(3)]
pa [Print(a)]
```

Program 2:

```
ia [IntDcl(a)]
pa [Print(a)]
a = 3 [Assign(a)][IntLit(3)]
```

Both programs are syntactically valid, but only Program 1 makes semantic sense. We could have parsed both programs in any order with success, but it is only through a (single) in-order forward pass that we can tell that Program 1 is valid while Program 2 isn't.

Semantic Analysis Algorithm

Core algorithm: take in list of ASTs from parser. Initialize two empty lists for declared and initialized variables. Work through each AST, adding variables to the declared and initialized lists, and also check any use of a variable to make sure it is both declared and initialized before use. Core loop: for each statement in the program, do:

- If statement is a IntDclNode, add the variable name to the declared set. Error case: redeclaring an already declared int.
- If statement is a PrintNode, check if variable name is in declared. If not, error. If so, check if it's initialized. If not, error.
- If statement is AssignNode, mark LHS varname. If varname is not declared, error. Otherwise, we need to recursively process the RHS expressions. If this passes, add varname to the initialized list. Otherwise, the recursive processing will raise an exception.

We raise the first exception we encounter (like with parsing).

Expression Checking

Expression checking is recursive. Function cases:

- If the node is an IntLitNode, simply return. These don't involve variables.
- If the node is a VarRefNode, get the varname and check if it is declared (yes/no) then initialized (yes/no).
- Recursive case: if the node is a BinOpNode, run the expression check on its left and right children.

Importantly, this allows for assignments like `a = a + 1` to be correctly processed in the situation where `a` either is or isn't declared/initialized.