Type system implementation

- We extend our Cuppa3 language to Cuppa4 with the addition of a type system with four types:
  - int
  - float
  - string
  - void
- We also assume that int is a subtype of float and float is a subtype of string, that is, a compiler/interpreter is allowed to insert widening conversions and should flag errors for narrowing conversions,

  int < float < string

Type system implementation

- We want to be able to write programs such as these:

```
int inc(int x) return x+1;
int y = inc(3);
put "the result is" + y;
```

```
float pow(float b, int p) {
   if (p == 0)
      return 1.0;
   else
      return b*pow(b,p-1);
}

float v;
get v;
int p;
get p;
float result = pow(v,p);
put v + " to the power of " + p +" is "+result;
```

Type syste
Syntax

New additions to the language are shown in bold face.

Listing 11.1: An LL(1) grammar for the Cuppa4 language.

```
stmt_list : (stmt)*

stmt : void ID \( formal_args? \) stmt
     | data_type ID decl_suffix
     | ID id_suffix
     | get ID ;?
     | put exp ;?
     | return exp? ;?
     | while \( exp \) stmt
     | if \( exp \) stmt (else stmt)?
     | \{ stmt_list \}

data_type : int
          | float
          | string

decl_suffix : \( formal_args? \) stmt
            | = exp ;?
            | ;?

id_suffix : \( actual_args? \) ;?
          | = exp ;?

exp : exp_low
exp_low : exp_med ((== | =<) exp_med)*
exp_med : exp_high ((\+ | -) exp_high)*
exp_high : primary ((\* | /) primary)*

primary : INTEGER
        : FLOAT
        | STRING
        | ID (\( actual_args? \))?
        | \( exp \)
        | - primary
        | not primary

formal_args : data_type ID (, data_type ID)*
actual_args : exp (, exp)*

ID : <any valid variable name>
INTEGER : <any valid int number>
FLOAT : <any valid floating point number>
STRING : <any valid quoted str constant>
```

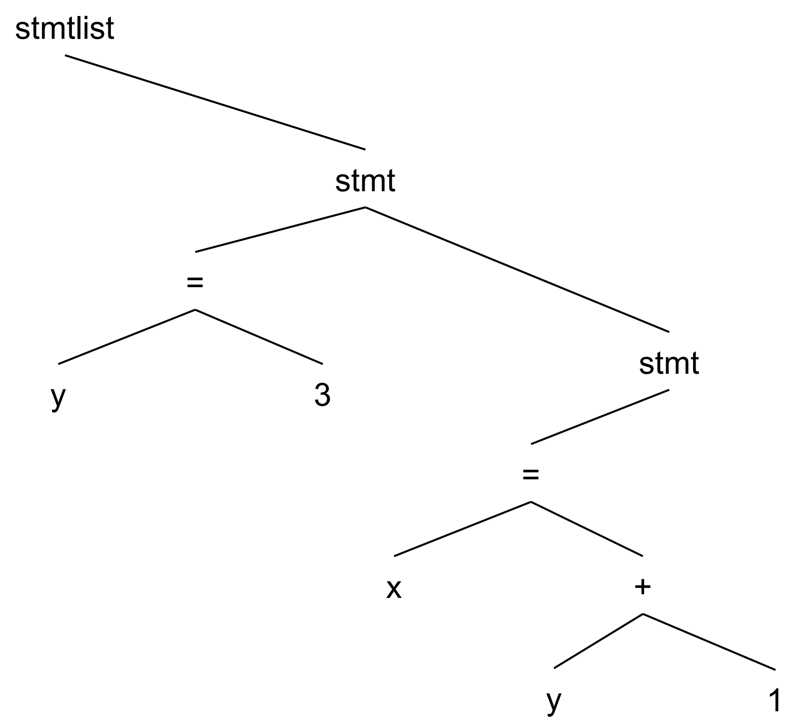
Type system implementation: Semantics

- At the semantic level we *annotate* all ASTs with type information
- We use *type propagation* to check that expressions/statements are properly typed.
  - Type propagation is the systematic tagging of an AST from leafs up with type information.

Type system implementation: Semantics

- Consider the simple example:

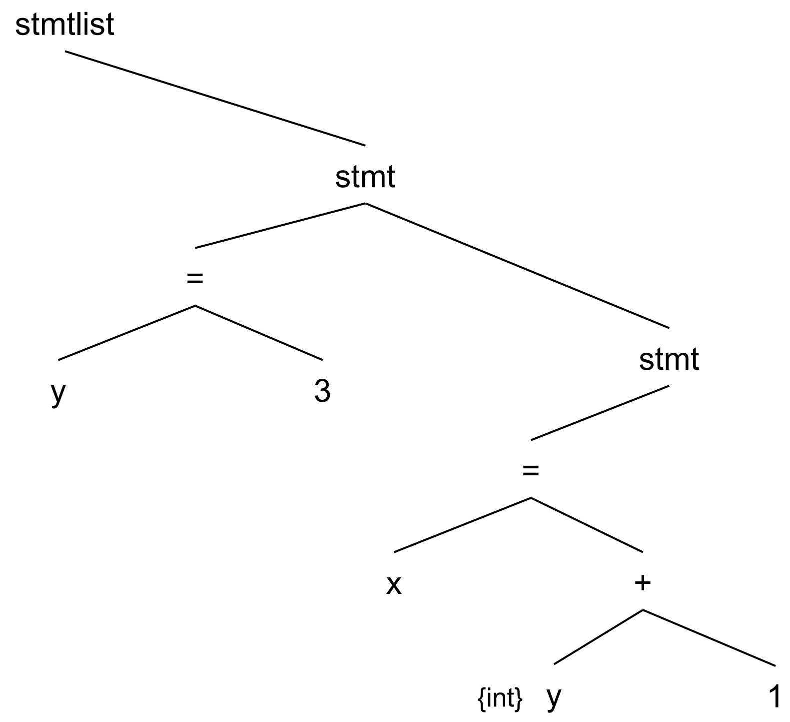
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

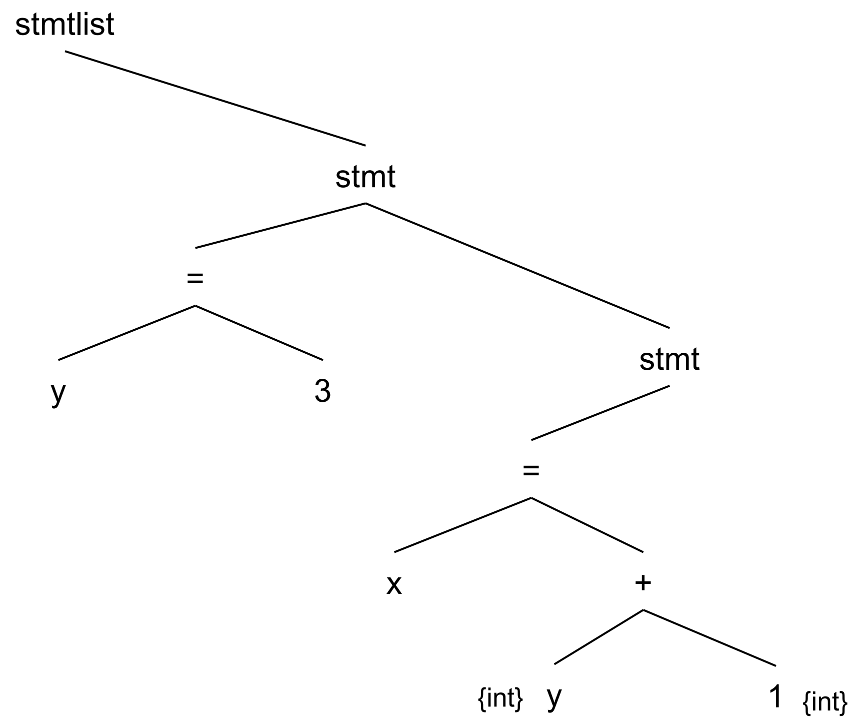
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

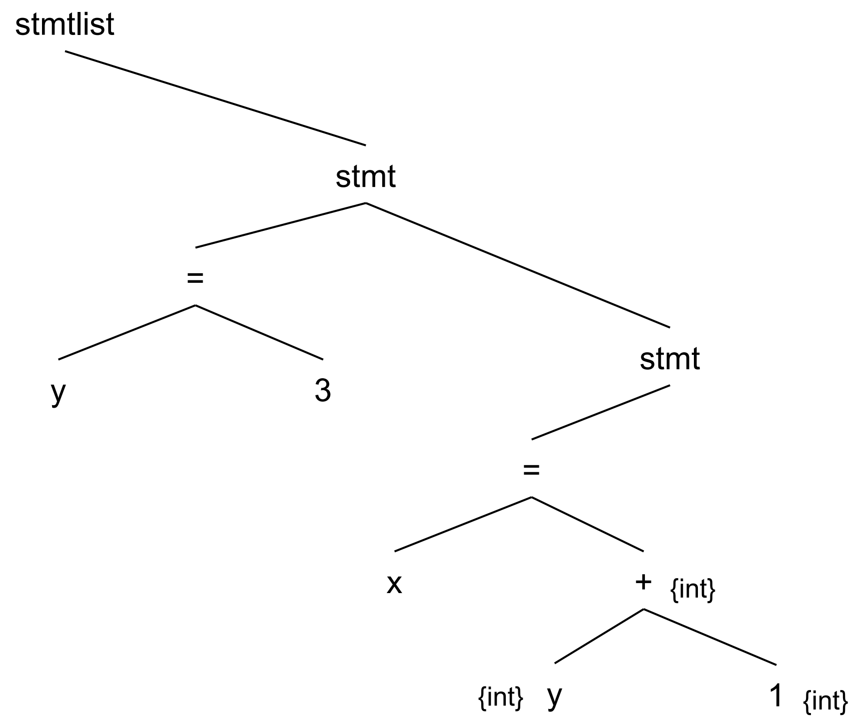
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

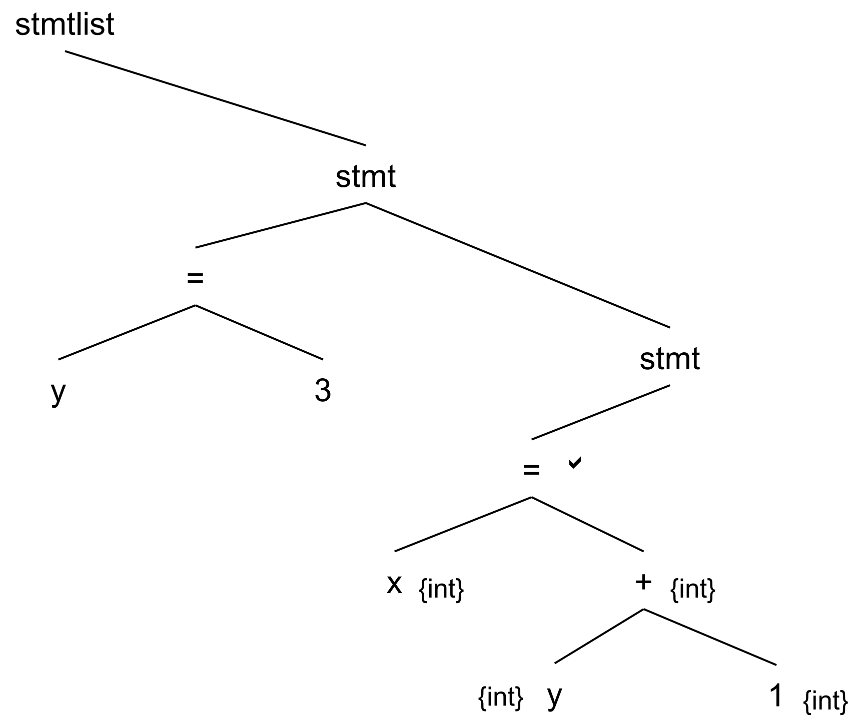
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

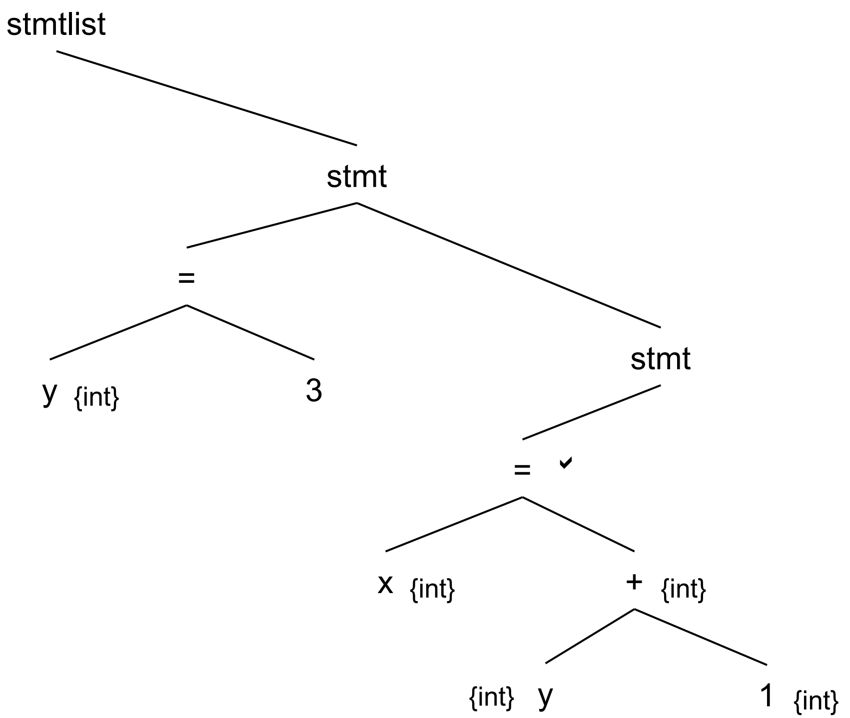
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

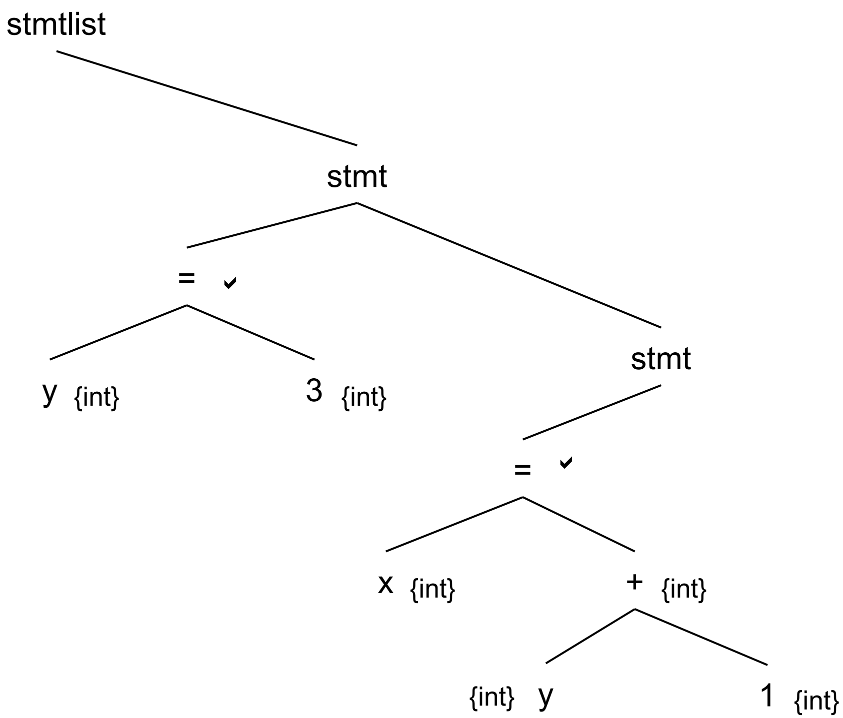
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider the simple example:

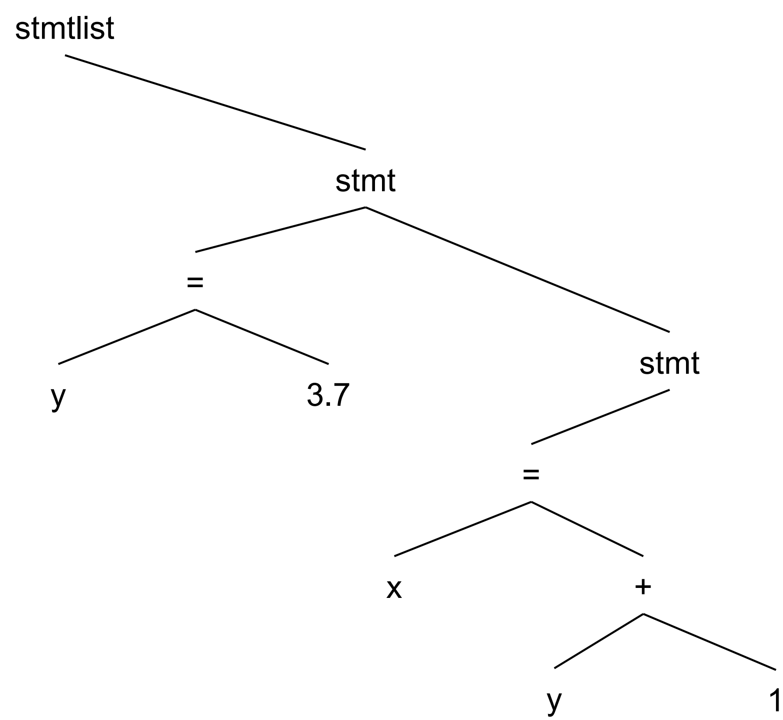
```
int y;
int x;
y = 3;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

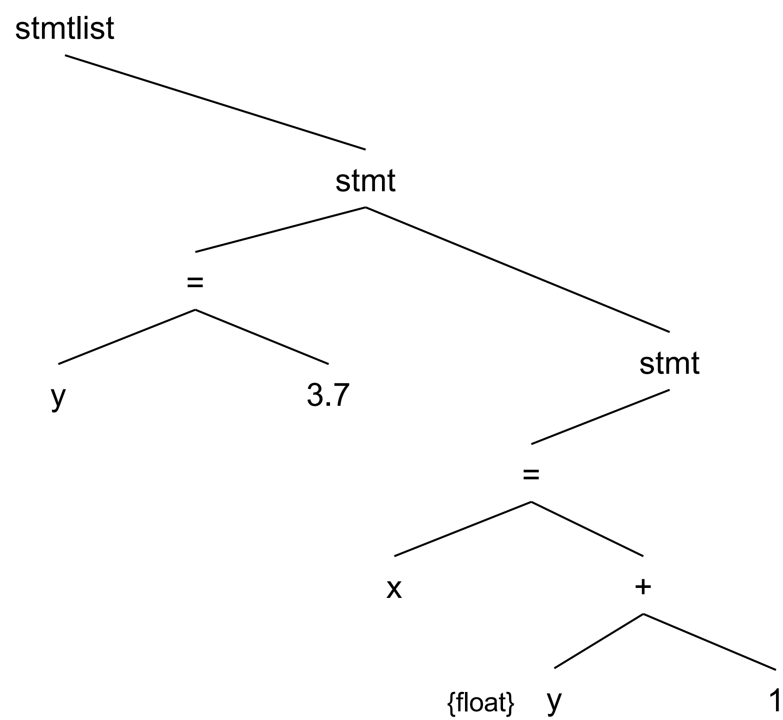
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

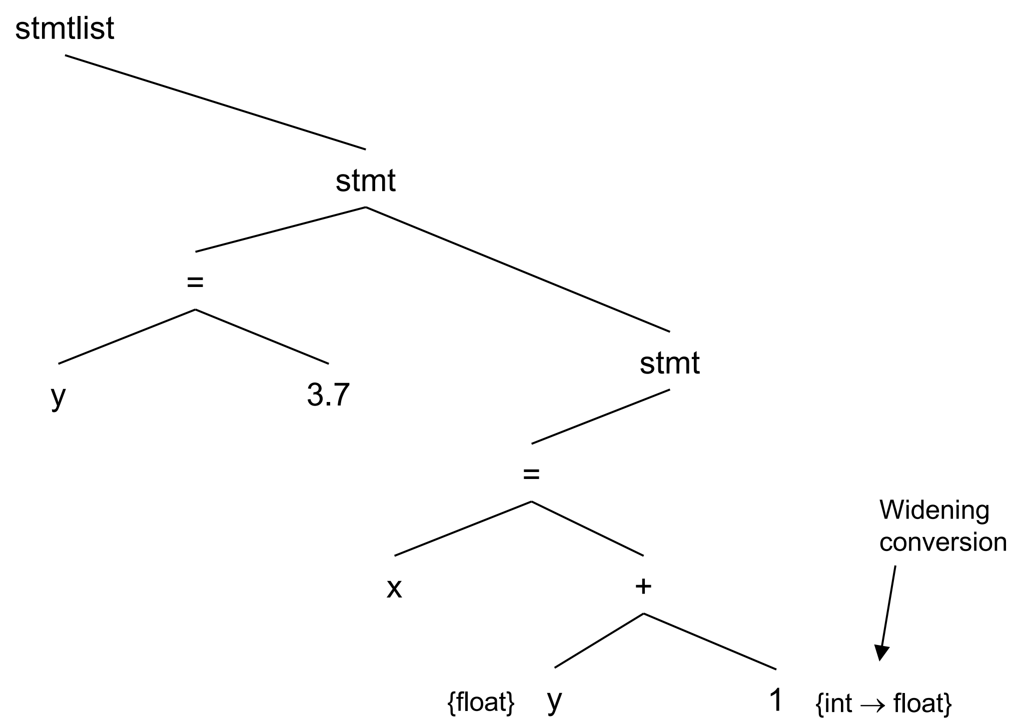
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

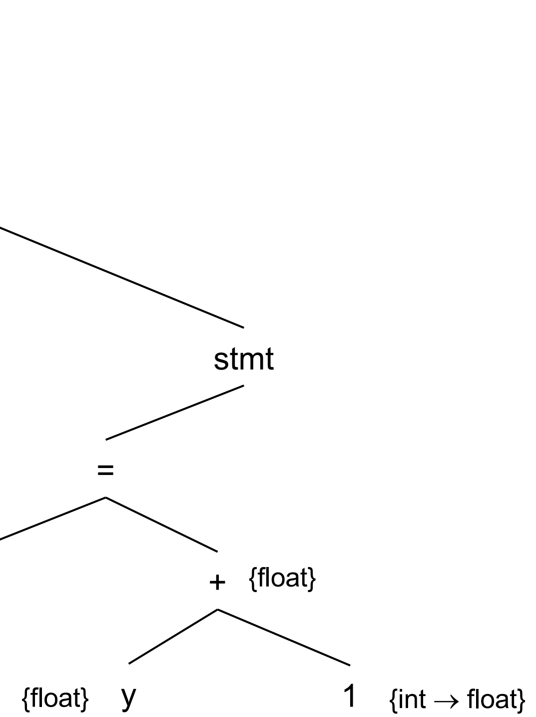
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

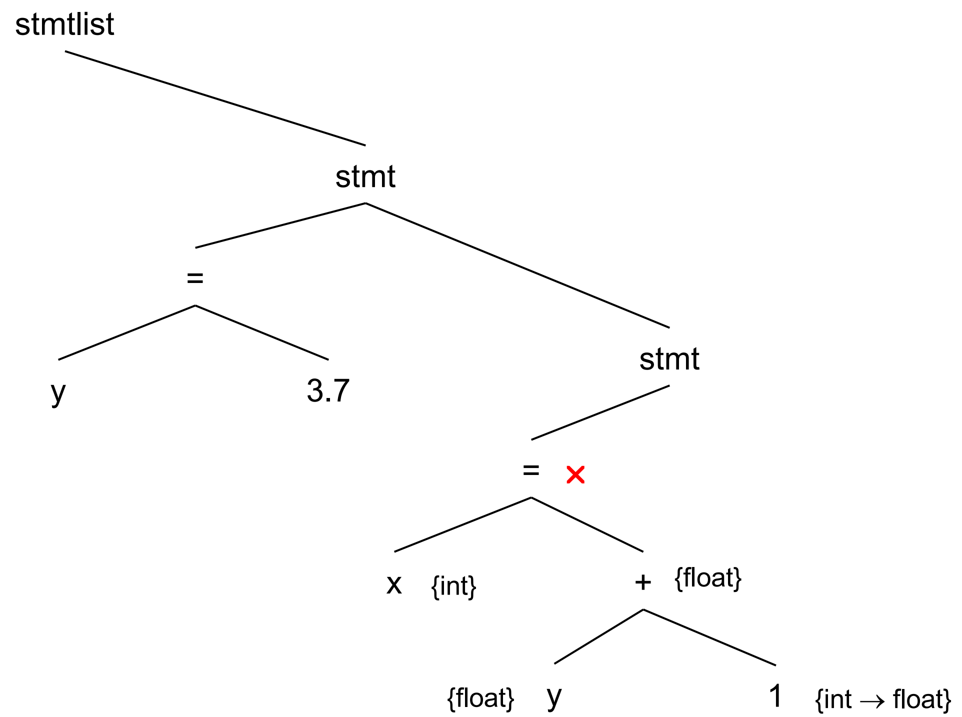
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

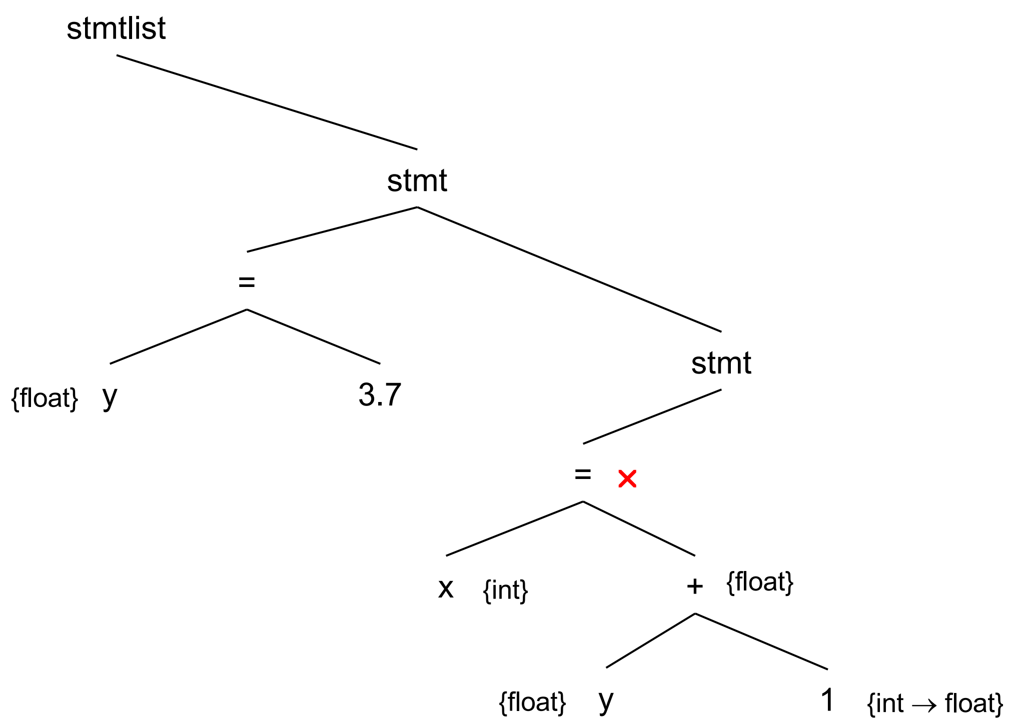
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

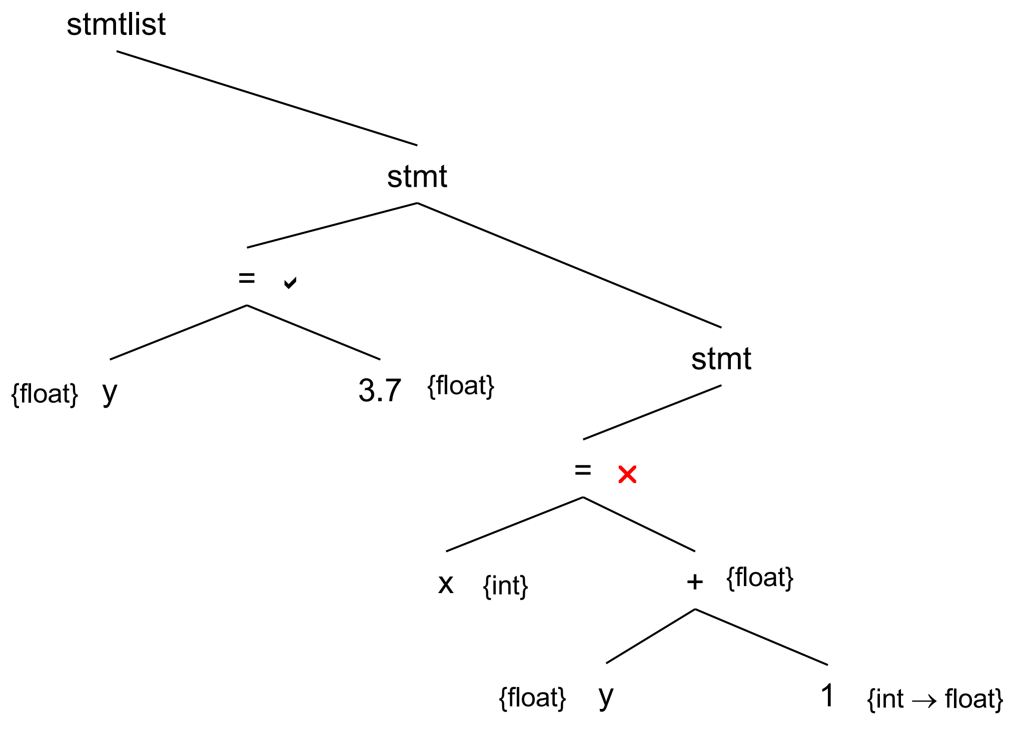
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Consider this example which has a typecheck error:

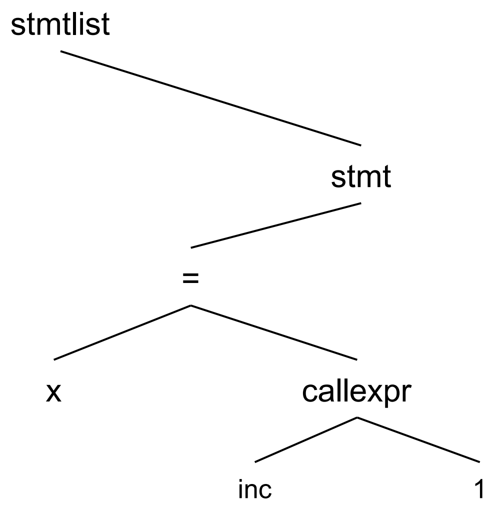
```
float y;
int x;
y = 3.7;
x = y + 1;
```



Type system implementation: Semantics

- Here is an example with a function call:

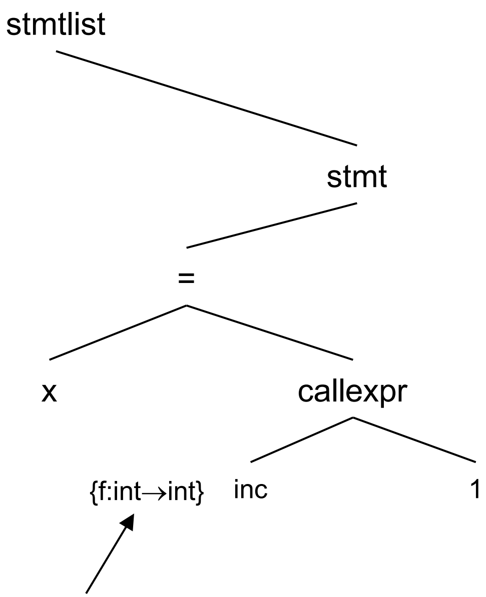
```
int inc(int i) return i+1;
int x;
x = inc(1);
```



Type system implementation: Semantics

- Here is an example with a function call:

```
int inc(int i) return i+1;
int x;
x = inc(1);
```

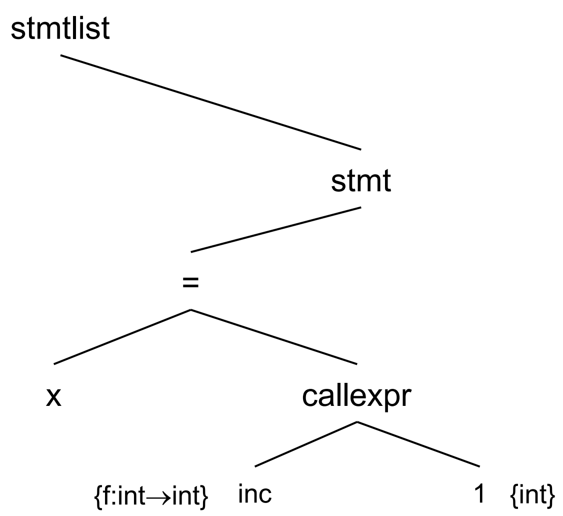


We have to track function symbols, both for their formal parameter types and return types.

Type system implementation: Semantics

- Here is an example with a function call:

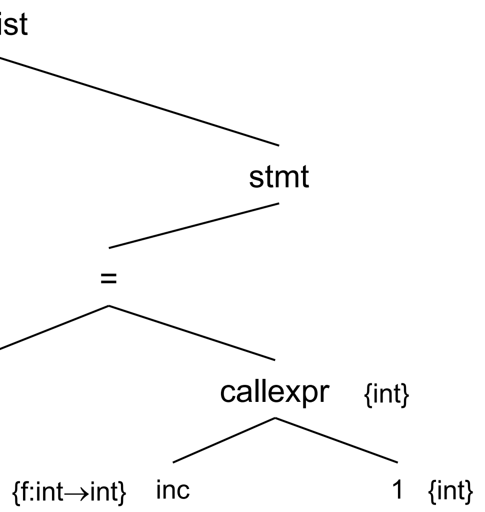
```
int inc(int i) return i+1;
int x;
x = inc(1);
```



Type system implementation: Semantics

- Here is an example with a function call:

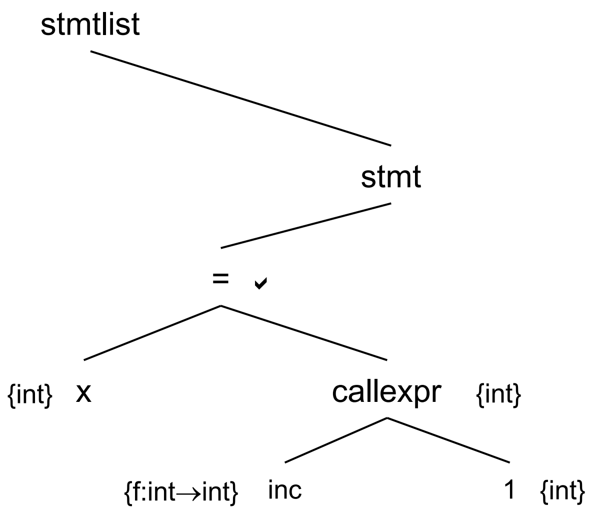
```
int inc(int i) return i+1;
int x;
x = inc(1);
```



Type system implementation: Semantics

- Here is an example with a function call:

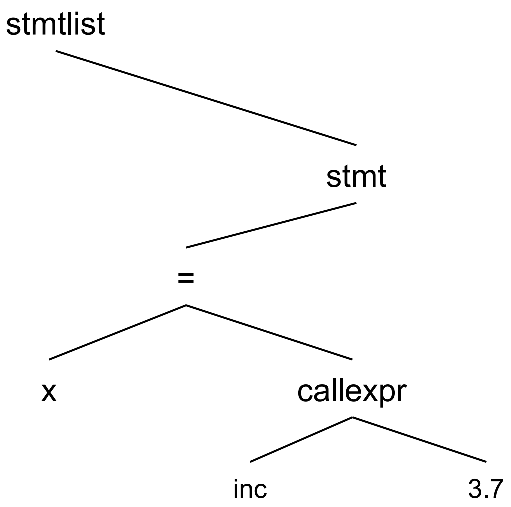
```
int inc(int i) return i+1;
int x;
x = inc(1);
```



Type system implementation: Semantics

- Here is an example with a function call and a type error:

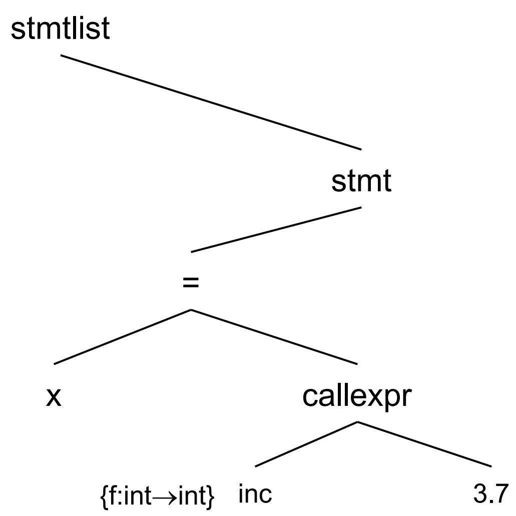
```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```



Type system implementation: Semantics

- Here is an example with a function call and a type error:

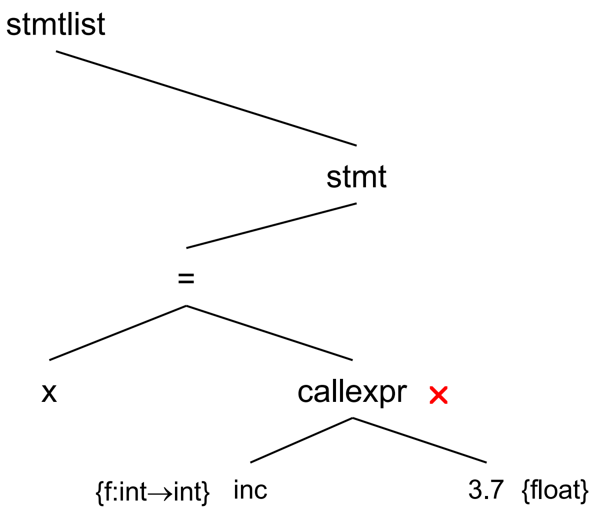
```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```



Type system implementation: Semantics

- Here is an example with a function call and a type error:

```
int inc(int i) return i+1;
int x;
x = inc(3.7);
```



Type System Implementation

- We will implement a static type checker

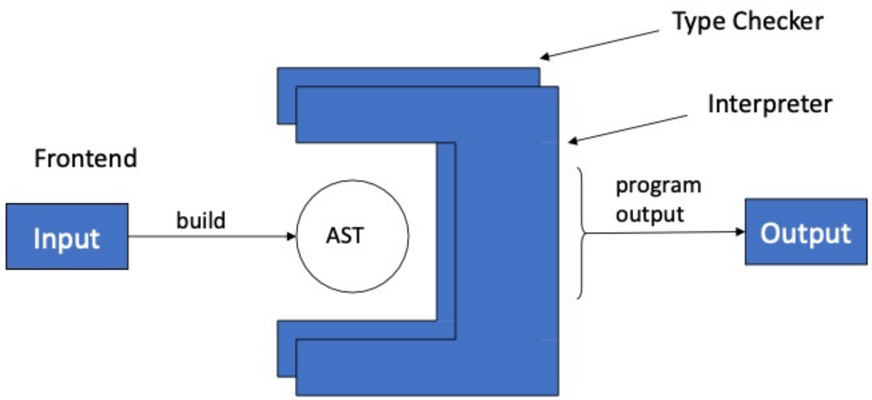


Figure 11.4: The architecture of our Cuppa4 interpreter.

Frontend

- The frontend is the Cuppa3 frontend with explicit type information.
- The changes necessary are simple extensions to the Cuppa3 frontend.

Frontend

```
float add(float a, float b) return a+b;
string c = add(1,2);
put "the result is " + c;
```

(float,float) → float

```
(FUNCTION_TYPE
   |(FLOAT_TYPE)
   |(LIST
   |   |[
   |   |   |(FLOAT_TYPE)
   |   |   |(FLOAT_TYPE)]))
```

```
(STMTLIST
  |[
  |   |(FUNDECL
  |   |   |(ID add)
  |   |   |(FUNCTION_TYPE
  |   |   |   |(FLOAT_TYPE)
  |   |   |   |(LIST
  |   |   |   |   |[
  |   |   |   |   |   |(FLOAT_TYPE)
  |   |   |   |   |   |(FLOAT_TYPE)]))
  |   |   |(LIST
  |   |   |   |[
  |   |   |   |   |(FORMALARG
  |   |   |   |   |   |(FLOAT_TYPE)
  |   |   |   |   |   |(ID a))
  |   |   |   |   |(FORMALARG
  |   |   |   |   |   |(FLOAT_TYPE)
  |   |   |   |   |   |(ID b))])
  |   |   |(RETURN
  |   |   |   |(PLUS
  |   |   |   |   |(ID a)
  |   |   |   |   |(ID b))))
  |   |(VARDECL
  |   |   |(ID c)
  |   |   |(STRING_TYPE)
  |   |   |(CALLEXP
  |   |   |   |(ID add)
  |   |   |   |(LIST
  |   |   |   |   |[
  |   |   |   |   |   |(CONST
  |   |   |   |   |   |   |(INTEGER_TYPE)
  |   |   |   |   |   |   |(VALUE 1))
  |   |   |   |   |   |(CONST
  |   |   |   |   |   |   |(INTEGER_TYPE)
  |   |   |   |   |   |   |(VALUE 2))]))))
  |   |(PUT
  |   |   |(PLUS
  |   |   |   |(CONST
  |   |   |   |   |(STRING_TYPE)
  |   |   |   |   |(VALUE the result is ))
  |   |   |   |(ID c)))])
```

Symbol Table

- Almost identical symbol table!
- We are using the same approach as we did in Cuppa3:
  - Use tags in the symbol table to figure out what kind of types we bound into the symbol table.
- We have to keep track of the return types of functions…we do that at the block scope level.

```
def push_scope(self, ret_type=None):
    # push a new dictionary onto the stack - stack grows to the left
    # Note: every block is associated with a return type
    # even if the return type is None.  If no return
    # type is given in the push instruction then we inherit
    # the return type of the outer block.
    if not ret_type:
        ret_type = self.lookup_ret_type()
    self.scoped_symtab.insert(CURR_SCOPE,({},ret_type))
```

The Type Checker

- As we saw, the type checker is a tree walker
- Turns out that out that it looks very similar to an interpretation walker with one important difference:
  - ☞ It computes TYPES rather than values.
- Types for us are tuples where the first component of the tuple tells us what kind of type we are looking at, e.g.
  - ('INTEGER_TYPE',)
  - ('STRING_TYPE',)
- We are using tuples because complex types such as function types need to store additional information such return type and argument types, e.g.

```
(FUNCTION_TYPE
  |(FLOAT_TYPE)
  |(LIST
  |   |[
  |   |   |(FLOAT_TYPE)
  |   |   |(FLOAT_TYPE)]))
```

The Type Checker

- Central to our implementation is the type promotion table that implements our type hierarchy.
- We use the type promotion table to implement our type propagation and type checking

```
'''
This module implements the Cuppa4 type coercion system through a set
of tables.  These tables implement the type hierarchy

         integer < float < string
         void

Note: we also have a function type but in this language it is not part of
the type system/hierarchy.  However, it does have an impact on why
we use
'''
supported_types = [
    'STRING_TYPE',
    'FLOAT_TYPE',
    'INTEGER_TYPE',
    'VOID_TYPE',
    ]

def supported(t):
    if t[0] not in supported_types:
        raise ValueError("operation does not support type {}".format(t[0]))
    else:
        return True
```

cuppa4_types.py

Note: function types are not supported in our type hierarchy

The Type Checker

cuppa4_types.py

- The type checker uses a number of tables to coerce types

```
# compute the common type for operands of a binary operation
_promote_table = {
  'STRING_TYPE' : {'STRING_TYPE': 'STRING_TYPE', 'FLOAT_TYPE': 'STRING_TYPE', 'INTEGER_TYPE': 'STRING_TYPE',  'VOID_TYPE': 'VOID_TYPE'},
  'FLOAT_TYPE'  : {'STRING_TYPE': 'STRING_TYPE', 'FLOAT_TYPE': 'FLOAT_TYPE',  'INTEGER_TYPE': 'FLOAT_TYPE',   'VOID_TYPE': 'VOID_TYPE'},
  'INTEGER_TYPE': {'STRING_TYPE': 'STRING_TYPE', 'FLOAT_TYPE': 'FLOAT_TYPE',  'INTEGER_TYPE': 'INTEGER_TYPE', 'VOID_TYPE': 'VOID_TYPE'},
  'VOID_TYPE'   : {'STRING_TYPE': 'VOID_TYPE',   'FLOAT_TYPE': 'VOID_TYPE',   'INTEGER_TYPE': 'VOID_TYPE',    'VOID_TYPE': 'VOID_TYPE'},
}

# compute the type coercion function given the target and source types
_coercion_table = {
  'STRING_TYPE' : {'STRING_TYPE': id,    'FLOAT_TYPE': str,   'INTEGER_TYPE': str,   'VOID_TYPE': error},
  'FLOAT_TYPE'  : {'STRING_TYPE': error, 'FLOAT_TYPE': id,    'INTEGER_TYPE': float, 'VOID_TYPE': error},
  'INTEGER_TYPE': {'STRING_TYPE': error, 'FLOAT_TYPE': error, 'INTEGER_TYPE': id,    'VOID_TYPE': error},
  'VOID_TYPE'   : {'STRING_TYPE': error, 'FLOAT_TYPE': error, 'INTEGER_TYPE': error, 'VOID_TYPE': error},
}

# compute whether an assignment is safe based on the target and source type
_safe_assign_table = {
  'STRING_TYPE' : {'STRING_TYPE': True,  'FLOAT_TYPE': True,  'INTEGER_TYPE': True,  'VOID_TYPE': False},
  'FLOAT_TYPE'  : {'STRING_TYPE': False, 'FLOAT_TYPE': True,  'INTEGER_TYPE': True,  'VOID_TYPE': False},
  'INTEGER_TYPE': {'STRING_TYPE': False, 'FLOAT_TYPE': False, 'INTEGER_TYPE': True,  'VOID_TYPE': False},
  'VOID_TYPE'   : {'STRING_TYPE': False, 'FLOAT_TYPE': False, 'INTEGER_TYPE': False, 'VOID_TYPE': False},
}
```

```
def error(_):
    raise ValueError("internal: type coercion error")

def id(x):
    return x;
```

The functions ‘str’ and ‘float’ are Python built-ins.

The Type Checker

- Interface functions to tables

```
def promote(type1, type2):
    supported(type1)
    supported(type2)
    type = (_promote_table.get(type1[0]).get(type2[0]),)
    if type[0] == 'VOID_TYPE':
        raise ValueError("type {} and type {} are not compatible"
                         .format(type1[0],type2[0]))
    return type

def coerce(target, source):
    supported(target)
    supported(source)
    return _coercion_table.get(target[0]).get(source[0])

def safe_assign(target, source):
    supported(target)
    supported(source)
    return _safe_assign_table.get(target[0]).get(source[0])
```

cuppa4_types.py

The Tree Walker

- Architecture wise looks like all our other tree walkers

```
def walk(node):
    # node format: (TYPE, [child1[, child2[, ...]]])
    type = node[0]

    if type in dispatch:
        node_function = dispatch[type]
        return node_function(node)
    else:
        raise ValueError("walk: unknown tree node type: " + type)

# a dictionary to associate tree nodes with node functions
dispatch = {
    'STMTLIST': stmtlist,
    'NIL'     : nil,
    'FUNDECL' : fundecl_stmt,
    'VARDECL' : vardecl_stmt,
    'ASSIGN'  : assign_stmt,
    'GET'     : get_stmt,
    'PUT'     : put_stmt,
    'CALLSTMT': call_stmt,
    'RETURN'  : return_stmt,
    'WHILE'   : while_stmt,
    'IF'      : if_stmt,
    'BLOCK'   : block_stmt,
    'CONST'   : const_exp,
    'ID'      : id_exp,
    'CALLEXP' : call_exp,
    'PAREN'   : paren_exp,
    'PLUS'    : plus_exp,
    'MINUS'   : minus_exp,
    'MUL'     : mul_exp,
    'DIV'     : div_exp,
    'EQ'      : eq_exp,
    'LE'      : le_exp,
    'UMINUS'  : uminus_exp,
    'NOT'     : not_exp
}
```

cuppa4_typecheck.py

The Tree Walker - Statements

```
def assign_stmt(node):

    (ASSIGN, name_exp, exp) = node

    tn = walk(name_exp)
    te = walk(exp)

    if not safe_assign(tn, te):
        raise ValueError("left type {} is not compatible with right type {}"
                         .format(tn[0],te[0]))

    return None
```

No value computation, just type propagation!

```
def if_stmt(node):

    (IF, cond, then_stmt, else_stmt) = node

    ctype = walk(cond)
    if ctype[0] != 'INTEGER_TYPE':
        raise ValueError("if condition has to be of type INTEGER_TYPE not {}"
                         .format(ctype[0]))
    walk(then_stmt)
    walk(else_stmt)

    return None
```

```
def while_stmt(node):

    (WHILE, cond, body) = node

    ctype = walk(cond)
    if ctype[0] != 'INTEGER_TYPE':
        raise ValueError("while condition has to be of type INTEGER_TYPE not {}"
                         .format(ctype[0]))
    walk(body)

    return None
```

The Tree Walker - Declarations

```
def vardecl_stmt(node):

    (VARDECL, (ID, name), type, init_val) = node

    ti = walk(init_val)
    if not safe_assign(type, ti):
        raise ValueError(
            "type {} of initializer is not compatible with declaration type {}"
            .format(ti[0],type[0]))
    symtab.declare(name, type)
    return None
```

```
def return_stmt(node):

    (RETURN, exp) = node

    t = walk(exp)
    ret_type = symtab.lookup_ret_type()
    if t[0] == ret_type[0]:
        # this is for the case void <- void
        return None
    elif not safe_assign(ret_type, t):
        raise ValueError(
            "function return type {} is not compatible with return statement type {}"
            .format(ret_type[0], t[0]))
    else:
        return None
```

```
def fundecl_stmt(node):

    (FUNDECL, (ID, name), type, arglist, body) = node

    symtab.declare(name, type)

    # unpack function type
    (FUNCTION_TYPE, ret_type, arglist_types) = type

    # typecheck body of function
    symtab.push_scope(ret_type=ret_type)
    declare_formal_args(arglist)
    walk(body)
    symtab.pop_scope()

    return None
```

Note: we only store type info in the symtab.

The Tree Walker - Expressions

```
def const_exp(node):

    (CONST, type, value) = node

    return type
```

```
def id_exp(node):

    (ID, name) = node

    val = symtab.lookup_sym(name)

    return val
```

```
def plus_exp(node):

    (PLUS,c1,c2) = node

    t1 = walk(c1)
    t2 = walk(c2)

    return promote(t1,t2)
```

```
def mul_exp(node):

    (MUL,c1,c2) = node

    t1 = walk(c1)
    t2 = walk(c2)
    tr = promote(t1,t2)
    if tr[0] not in ['INTEGER_TYPE','FLOAT_TYPE']:
        raise ValueError("operation on type {} not supported"
                         .format(tr[0]))
    return tr
```

```
def eq_exp(node):

    (EQ,c1,c2) = node

    walk(c1)
    walk(c2)

    return ('INTEGER_TYPE',)
```

No value computation, just type propagation!

The Tree Walker - Calls

```
def call_stmt(node):

    (CALLSTMT, name_exp, actual_args) = node

    check_call(walk(name_exp), actual_args)

    return None
```

```
def call_exp(node):

    (CALLEXP, name_exp, actual_args) = node

    tf = walk(name_exp)

    return check_call(tf, actual_args)
```

```
def check_call(function_type, actual_arguments):

    # unpack
    (FUNCTION_TYPE, ret_type, (LIST, formal_arg_types)) = function_type
    (LIST, actual_args_list) = actual_arguments

    # make sure arguments line up
    if len(formal_arg_types) != len(actual_args_list):
        raise ValueError("expected {} argument(s) got {}"
                         .format(len(formal_arg_types),
                                 len(actual_args_list)))

    # type check association of actuals to formals
    for (tformal, a) in zip(formal_arg_types,actual_args_list):
        tactual = walk(a)
        if not safe_assign(tformal,tactual):
            raise ValueError(
                "actual argument type {} is not compatible with \
                formal argument type {}"
                .format(tactual[0],tformal[0]))

    return ret_type
```

No value computation, just type propagation!

The Interpreter Tree Walk

- The interpreter tree walker walks the type checked AST and computes…wait for it…
  ☞ Values!
  Well, actually we compute type-value tuples.

```
def const_exp(node):

    (CONST, type, (VALUE, value)) = node
    return (type, value)
```

- It uses the type coercion table.
  - Look up appropriate type conversion functions

The Interpreter Tree Walk -- Expressions

```
def plus_exp(node):

    (PLUS,c1,c2) = node

    (t1,v1) = walk(c1)
    (t2,v2) = walk(c2)

    t = promote(t1,t2)

    return (t, coerce(t,t1)(v1) + coerce(t,t2)(v2))
```

```
def eq_exp(node):

    (EQ,c1,c2) = node

    (t1,v1) = walk(c1)
    (t2,v2) = walk(c2)
    t = promote(t1, t2)

    if coerce(t,t1)(v1) == coerce(t,t2)(v2):
        return ('INTEGER_TYPE', 1)
    else:
        return ('INTEGER_TYPE', 0)
```

```
def id_exp(node):

    (ID, name) = node
    (CONST, type, (VALUE, value)) = symtab.lookup_sym(name)

    return (type, value)
```

Very little error checking!
All that is done in the type checker!

```
def call_exp(node):

    (CALLEXP, (ID, name), actual_args) = node

    return_value = handle_call(name, actual_args)

    if not return_value:
        raise ValueError("No return value from function {}".format(name))
    else:
        return return_value
```

The Interpreter Tree Walk -- Statements

```
def assign_stmt(node):

    (ASSIGN, (ID, name), exp) = node

    (t,v) = walk(exp)
    (CONST, ts, (VALUE, vs)) = symtab.lookup_sym(name)
    symtab.update_sym(name, ('CONST', t, ('VALUE', coerce(ts,t)(v))))

    return None
```

```
def call_stmt(node):

    (CALLSTMT, (ID, name), actual_args) = node
    handle_call(name, actual_args)
    return None
```

```
def fundecl_stmt(node):

    (FUNDECL, (ID, name), type, arglist, body) = node

    context = symtab.get_config()
    funval = ('FUNVAL', type, arglist, body, context)
    symtab.declare(name, funval)

    return None
```

```
def while_stmt(node):

    (WHILE, cond, body) = node

    while walk(cond)[1]:
        walk(body)

    return None
```

```
def vardecl_stmt(node):

    (VARDECL, (ID, name), type, init_val) = node

    (ti, vi) = walk(init_val)
    symtab.declare(name, ('CONST', type, ('VALUE', coerce(type,ti)(vi))))

    return None
```

The Interpreter Tree Walk – Handle Call

```
def handle_call(name, actual_arglist):
    '''
    handle calls for both call statements and call expressions.
    '''
    # unpack the funval and type tuples
    (FUNVAL, type, formal_arglist, body, context) = symtab.lookup_sym(name)
    (FUNCTION_TYPE, ret_type, arg_types) = type

    # set up the environment for static scoping and then execute the function
    actual_val_args = eval_actual_args(actual_arglist)
    save_symtab = symtab.get_config()
    symtab.set_config(context)
    symtab.push_scope(ret_type)
    declare_formal_args(formal_arglist, actual_val_args)

    # execute function
    return_value = None
    try:
        walk(body)
    except ReturnValue as val:
        return_value = val.value

    # NOTE: popping the function scope is not necessary because we
    # are restoring the original symtab configuration
    symtab.set_config(save_symtab)

    return return_value
```

Running the Interpreter

```
$ cat pow.txt
float v;
int p;

float pow(float b, int e) {
   if (e == 0)
      return 1.0;
   else
      return b*pow(b,e-1);
}

get v;
get p;
put v+" to the power of "+p+" is "+pow(v,p);

$ python3 cuppa4_interp.py pow.txt
Value for v? 3
Value for p? 2
3.0 to the power of 2 is 9.0
$
```

```
$ cat z.txt
int z(int x) return x;
int y = z + 1;  // semantic error
put y;

$ python3 cuppa4_interp.py z.txt
error: operation does not support type FUNCTION_TYPE
$
```