# Building Your Compiler: Type Checking

# Building Your Compiler: Type Checking

Static type checking is a compile time operation that works to verify that certain types of program errors will be detected and reported. Some examples of type checking are:

- Type compatibility: ensuring that types are compatible (i) in expressions, (ii) of the argument list and parameter list of a function call, and (iii) between the LHS and RHS (Left/Right Hand Side) of an assignment
- Illegal duplicate declarations of symbols
- Undeclared symbols
- Control flow errors (attempts to branch to an undefined label)

These tests are embedded within the parser and implementing these tests generally requires that type information be passed up and down the method calls of a parser so that the testing of type compatibility can be performed.

Constructing the type checking requires that symbol declaration information is recorded in the symbol table.

# nt_retType

In general, each Non-Terminal production will have a production specific input argument set.

However the return type for Non-Terminal productions should ideally be a common reusable type. For example:

```
class nt_retType
  bool: returnCode
end class

class nt_retType : nt_retType_expr
  tokenType: *tt
end class
```

So we have a generic base type for Non-Terminal Productions and a derived class for Non-Terminals for expression processing. Other derived classes are also possible.

# Variable Declarations

```
variable_decl :
  ’variable’ ID ’:’
  type_mark()
  ( ’[’ num() ’]’ )?
```

```
nt_retType variable_decl ()
  scan_assume<’variable’>
  if tok.tt != ’ID’
    reportError(’Missing name in var decl’)
  // verify that this is not a duplicate symbol declaration
  if not ckPrevDecl(tok)
    reportError(’Duplicate declaration for {}’.format(tok))
    return new nt_retType(’false’, NULL)
  varName = tok
  tok = scan()
  scan_assume<’:’>
  varType = type_mark()
  // verify that we have a valid type
  if not varType->isType()
    reportError(’Illegal type {} in var decl’.format(tok))
    varType = NULL
  // if an array declaration, process the array bounds
  if tok.tt = ’[’
    tok = scan()
    numElements = num()
    if not (atoi(numElements->tokStr) > 0)
      reportError(’Array size must be a positive integer’)
      return new nt_retType(’false’, NULL)
    scan_assume<’]’>
    varType->makeArryType(varType,numElements)
  return new nt_retType(’true’, makeVariable(varName, varType))
```

# number()

```
nt_retType_expr number ()

  numTok = tok
  // apply any num tests; this might be simply: numTok.tt == NUM
  if num_TypeCheck(numTok)
    tok = scan()
    return new nt_retType_expr('true', numTok)
  else
    return new nt_retType_expr('false', NULL)
```

# identifier()

```
nt_retType_expr identifier ()

  // verify properly declared variable
  if not verifyDefinedIdentifier(tok)
    reportError(’Undefined identifier {}’.format(tok)
    return new nt_retType_expr(’false’, NULL)
  idTok = tok
  tok = scan()
  if tok.tt == ’[’
    tok = scan()
    arrayIndx = expr()
    scan_assume<’]’>
    idTok = makeArrayRef(idTok, arrayIndx)
  // this changes in code generation
  return new nt_retType_expr(’true’, idTok)
```

# Revisiting `factor`

```
nt_retType_expr factor ()

  if tok.tt == ’(’
    tok = scan()
    exprVal = expr()
    res = scan_assume<’)’>
    if res == ’false’ then
      exprVal->returnCode = ’false’
    return exprVal

  if tok.tt == ID
    return identifier()
  if tok.tt == NUM
    return number()

  // factor() failed to expand properly
  reportError(’Malformed expression’)
  return new nt_retType_expr(’false’, NULL)
```

# Revisiting `expr`

```
nt_retType_expr expr ()
  termVal = term()
  exprPVal = exprPrime(termVal)
  return exprPVal

nt_retType_expr exprPrime (nt_returnType_expr *leftExpr)
  retExpr = leftExpr
  if tok.tt == ADD_OP
    opVal = tok
    tok = scan()
    termVal = term()
    // verify type compatibility and build return object for ADD_OP
    retExpr = verifyCompatible(ADD_OP, leftExpr, termExpr)
  return retExpr
```

- So the general exercise is to verify type compatibility to the operands on either side of the unary/binary operators.
- I am showing left-to-right type checking by passing type info up/down the parse. However, it can also be performed bottom up. In our case we will be performing code generation in the same pass, so its best if we type check left-to-right.

# Revisiting cond_expr

```
nt_retType_expr cond_expr()
  exprVal = expr()
  if exprVal->isBoolType()  // true if the type compatible to ’bool’
    return exprVal
  else
    reportError(’Boolean expression required; expression type is {}’.format(exprVal))
    return new nt_retType_expr(’false’, NULL)
```

# Revisiting assignment_stmt

```
nt_retType assignment_stmt ()
  lhs = identifier()
  scan_assume<’:=’>
  rhs = expr()
  return verifyCompatible(ASSIGN_OP, lhs, rhs)
```