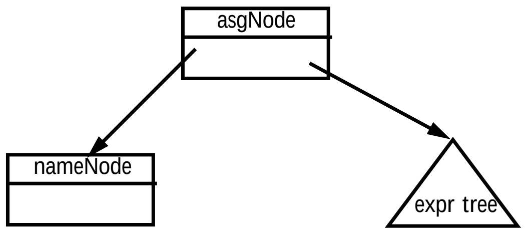
Reading Assignment

Get and read Chapter 9 of Crafting a Compiler featuring Java.

(Available from DoIt Tech Store)

Type Checking Assignments

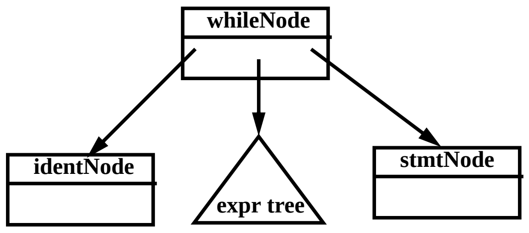


Type checking steps:

1. Type check the nameNode.
2. Type check the expression tree.
3. Check that the nameNode's kind is assignable (Variable, Array, ScalarParm, or ArrayParm).
4. If the nameNode's kind is scalar then check the expression tree's kind is also scalar and that both have the same type. Then return.

5. If the nameNode's and the expression tree's kinds are both arrays and both have the same type, check that the arrays have the same length. (Lengths of array parms are checked at run-time). Then return.
6. If the nameNode's kind is array and its type is character and the expression tree's kind is string, check that both have the same length. (Lengths of array parms are checked at run-time). Then return.
7. Otherwise, the expression may not be assigned to the nameNode.

Type Checking While Loops

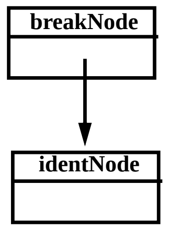


Type checking steps:

1. Type check the condition (an expr tree).
2. Check that the condition's type is Boolean and kind is scalar.
3. If the label is null (no identNode is present) then type check the stmtNode (the loop body) and return.

4.If there is a label (an identNode) then:
  (a) Check that the label is not already present in the symbol table.
  (b) If it isn't, enter label in the symbol table with kind=VisibleLabel and type= void.
  (c) Type check the stmtNode (the loop body).
  (d) Change the label's kind (in the symbol table) to HiddenLabel.

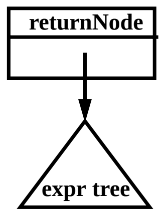
Type Checking Breaks and Continues



Type checking steps:

1. Check that the identNode is declared in the symbol table.
2. Check that identNode's kind is VisibleLabel. If identNode's kind is HiddenLabel issue a special error message.

Type Checking Returns

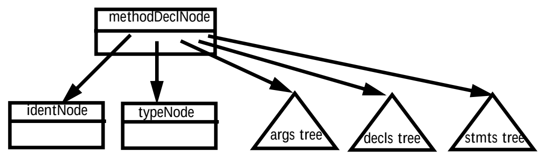


It is convenient to arrange that a static filed named *currentMethod* will always point to the methodDeclNode of the method we are currently checking.

Type checking steps:

1. If returnVal (an expr) is null, check that currentMethod.returnType is Void.
2. If returnVal (an expr) is not null then check that returnVal's kind is scalar and returnVal's type is currentMethod.returnType.

Type Checking Method Declarations

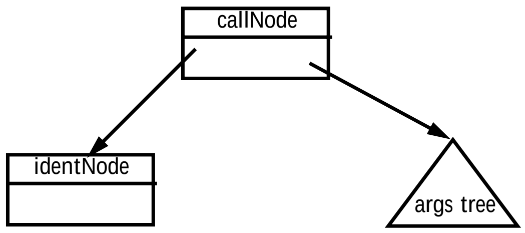


Type checking steps:

1. Create a new symbol table entry m, with type = typeNode.type and kind = Method.
2. Check that identNode.idname is not already in the symbol table; if it isn't, enter m using identNode.idname.
3. Create a new scope in the symbol table.
4. Set currentMethod = this methodDeclNode.

5. Type check the args subtree.
6. Build a list of the symbol table nodes corresponding to the args subtree; store it in m.
7. Type check the decls subtree.
8. Type check the stmts subtree.
9. Close the current scope at the top of the symbol table.

Type Checking Method Calls



We consider calls of procedures in a statement. Calls of functions in an expression are very similar.

Type checking steps:

1. Check that identNode.idname is declared in the symbol table. Its type should be Void and kind should be Method.
2. Type check the args subtree.

3. Build a list of the expression nodes found in the args subtree.
4. Get the list of parameter symbols declared for the method (stored in the method's symbol table entry).
5. Check that the arguments list and the parameter symbols list both have the same length.
6. Compare each argument node with its corresponding parameter symbol:
(a) Both should have the same type.
(b) A Variable, Value, or ScalarParm kind in an argument node matches a ScalarParm parameter. An Array or ArrayParm kind in an argument node matches an ArrayParm parameter.

Virtual Memory & Run-Time Memory Organization

The compiler decides how data and instructions are placed in memory.

It uses an *address space* provided by the hardware and operating system.

This address space is usually *virtual*— the hardware and operating system map instruction-level addresses to "actual" memory addresses.

Virtual memory allows:

- Multiple processes to run in private, protected address spaces.
- Paging can be used to extend address ranges beyond actual memory limits.

Run-Time Data Structures

Static Structures

For static structures, a fixed address is used throughout execution.

This is the oldest and simplest memory organization.

In current compilers, it is used for:

- Program code (often read-only & sharable).
- Data literals (often read-only & sharable).
- Global variables.
- Static variables.

Stack Allocation

Modern programming languages allow recursion, which requires *dynamic allocation*.

Each recursive call allocates a *new copy* of a routine's local variables.

The number of local data allocations required during program execution is not known at compile-time.

To implement recursion, all the data space required for a method is treated as a distinct data area that is called a *frame* or *activation record*.

Local data, within a frame, is accessible only while a subprogram is active.

In mainstream languages like C, C++ and Java, subprograms must return in a stack-like manner—the most recently called subprogram will be the first to return.

A frame is pushed onto a *run-time stack* when a method is called (activated).

When it returns, the frame is popped from the stack, freeing the routine's local data.

As an example, consider the following C subprogram:

```
p(int a) {
   double b;
   double c[10];
   b = c[a] * 2.51;
}
```

Procedure **p** requires space for the parameter **a** as well as the local variables **b** and **c**.

It also needs space for control information, such as the return address.

The compiler records the space requirements of a method.

The *offset* of each data item relative to the beginning of the frame is stored in the symbol table.

The total amount of space needed, and thus the size of the frame, is also recorded.

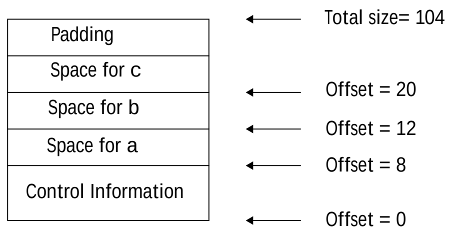
Assume **p**'s control information requires 8 bytes (this size is usually the same for all methods).

Assume parameter **a** requires 4 bytes, local variable **b** requires 8 bytes, and local array **c** requires 80 bytes.

Many machines require that word and doubleword data be *aligned*, so it is common to pad a frame so that its size is a multiple of 4 or 8 bytes.

This guarantees that at all times the top of the stack is properly aligned.

`Here is p`'s frame:



Within **p**, each local data object is addressed by its offset relative to the start of the frame.

This offset is a fixed constant, determined at compile-time.

We normally store the start of the frame in a register, so each piece of data can be addressed as a **(Register, Offset)** pair, which is a standard addressing mode in almost all computer architectures.

For example, if register **R** points to the beginning of **p**'s frame, variable **b** can be addressed as **(R,12),** with **12** actually being added to the contents of **R** at run-time, as memory addresses are evaluated.

Normally, the literal `2.51` of procedure **p** is not stored in **p**'s frame because the values of local data that are stored in a frame disappear with it at the end of a call.

It is easier and more efficient to allocate literals in a *static area*, often called a *literal pool* or *constant pool*. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.