Comp 204: Computer Systems and Their Implementation

Lecture 21: Semantics and Code Generation

Semantics

- Specify meaning of language constructs
  - usually defined informally
- A statement may be syntactically legal but semantically meaningless
  - “colourless green ideas sleep furiously”
- Semantic errors may be
  - static (detected at compile time)
    e.g. a := ‘x’ + true;
  - dynamic (detected at run time)
    e.g. array subscript out of bounds

Question

- If the array x contains 20 ints, as defined by the following declaration:

```
int x[] = new int[20];
```

- What kind of message would be generated by the following line of code?

```
    a := 22;
val := x[a];
```

a) A Syntax Error.
b) A Static Semantic Error.
c) A Dynamic Semantic Error.
d) A Warning, rather than an error.
e) None of the above.

Answer: c
A dynamic semantic error – the value of a would cause an array out of bounds error

Semantics

- Also needed to generate appropriate code e.g. a = b
  - in Java and C, this means assign b to a
  - in Pascal and Ada, this means compare equality of a and b
  - hence, generate different code in each case

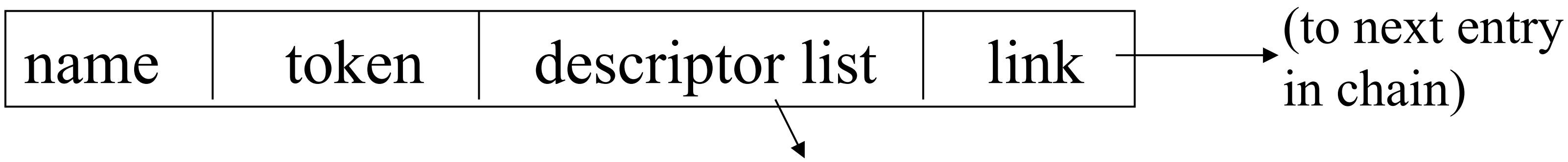
Semantic Routines

1) Semantic analysis
   - Completes analysis phase of compilation
   - Object descriptors are associated with identifiers in symbol table
   - Static semantic error checking performed

2) Semantic synthesis
   - Code generation

Object Descriptors

Symbol table entry

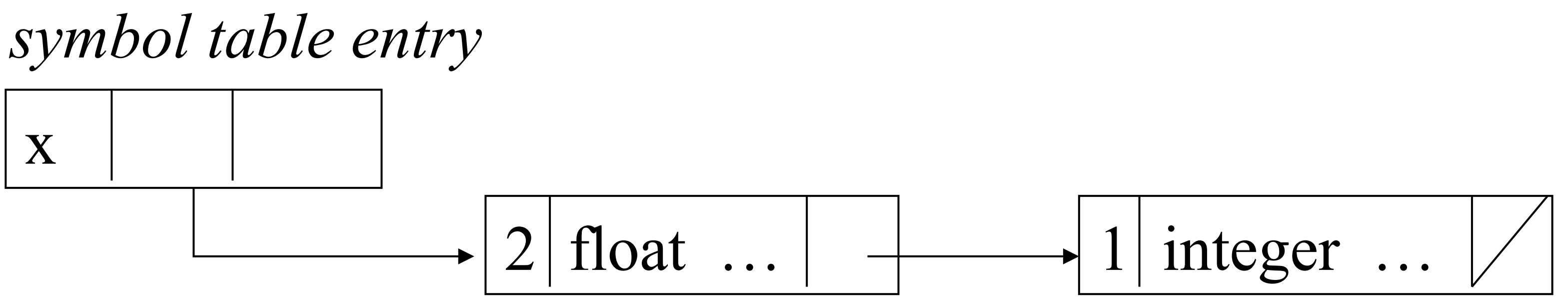


- 'Token' tells us what 'name' is
  - e.g. while-token, if-token, identifier, etc.
- A descriptor contains things like type, address, array bounds, etc.
- Need a list of descriptors because of identifier re-use

Identifier Re-use

- Can have code such as:

```
int x;              // level 1

main() {
   float x;         // level 2
}
```



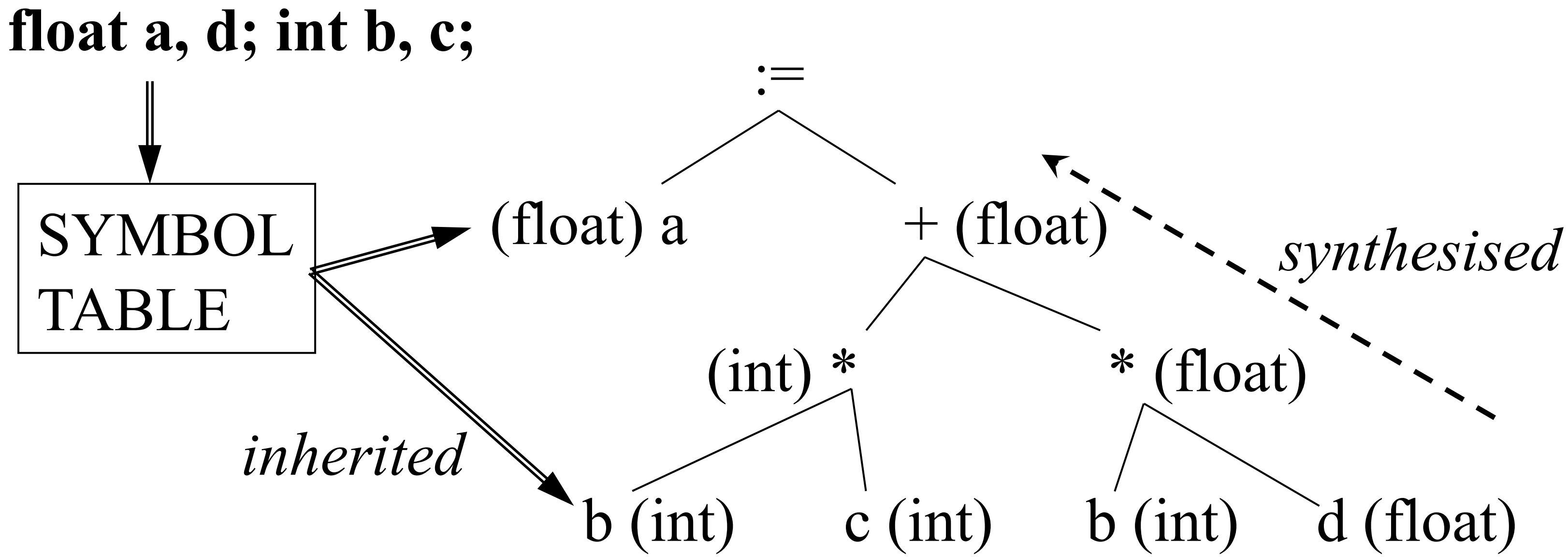
Descriptor Lists

- For efficiency, the most local descriptors are kept at the front of the list
- At the end of a block, all descriptors declared in that block must be deleted
- To aid in this, all descriptors within same block may be linked together

Attribute Propagation

- Before code can be generated, semantic attributes may need to be propagated through tree
- Top-down (inherited attributes)
  - declarations processed to build symbol table
  - identifiers looked up in table to attach attribute info to nodes
- Bottom-up (synthesised attributes)
  - determine types of expressions based on operators and types of identifiers
- Propagation can be done at same time as static semantic error checking, and often forms next pass
  - May also be combined with code generation

Example: a := b*c +b*d



- Type attribute recorded in extra field of each node
- After propagation, tree is said to be decorated

Static Semantic Error Checking

- With info from attribute propagation, static checking often trivial, e.g.
  - "type mismatch"
    (compare type attributes)
  - "identifier not declared"
    (null descriptor field in symbol table)
  - "identifier already declared"
    (descriptor with current level number already present)

Question

- A BNF grammar includes the following statement:

```
<statement> ::= <iden> := ( <expr> );
```

- What kind of message would be produced by the following line of code?

```
a := (2 + b;
```

a) A Syntax Error.
b) A Static Semantic Error.
c) A Dynamic Semantic Error.
d) A Warning, rather than an error.
e) None of the above.

Answer: a
A syntax error – all the tokens are valid, but the close parenthesis is missing, resulting in an error in the grammar

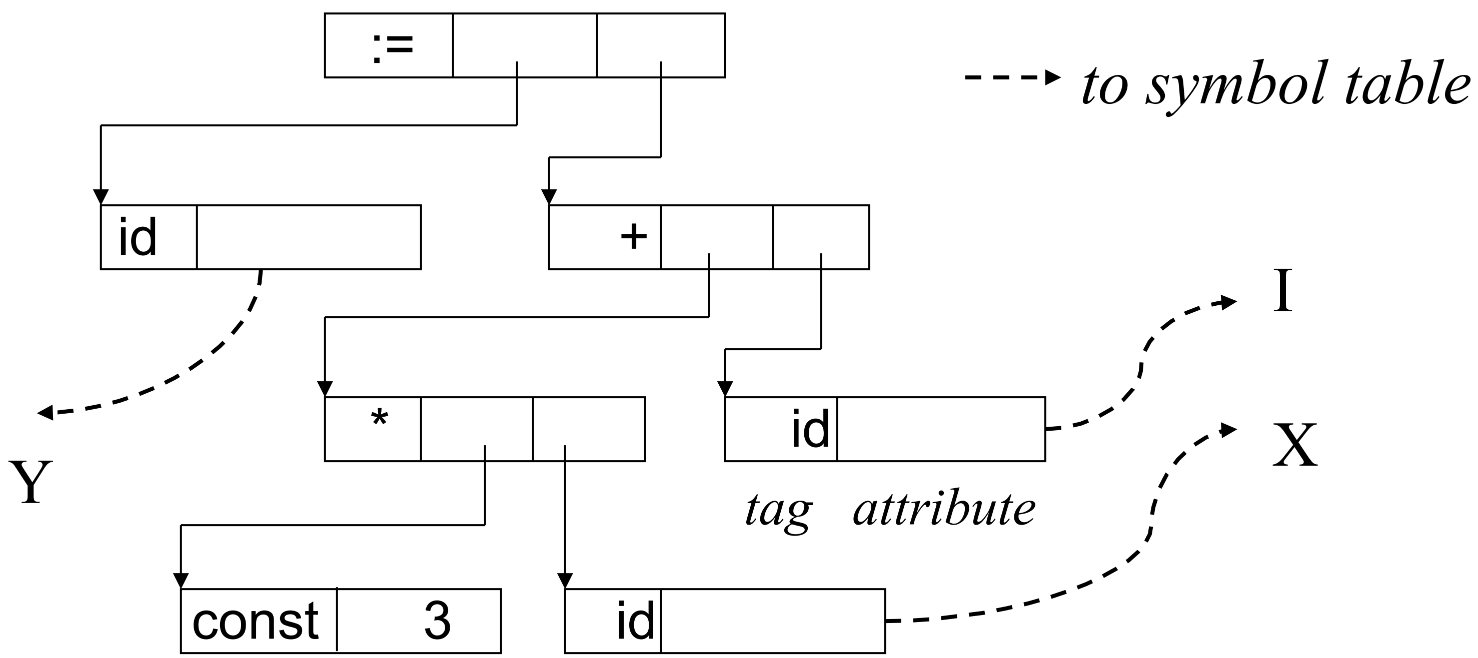
Code Generation

- Often performed by tree-walking the AST

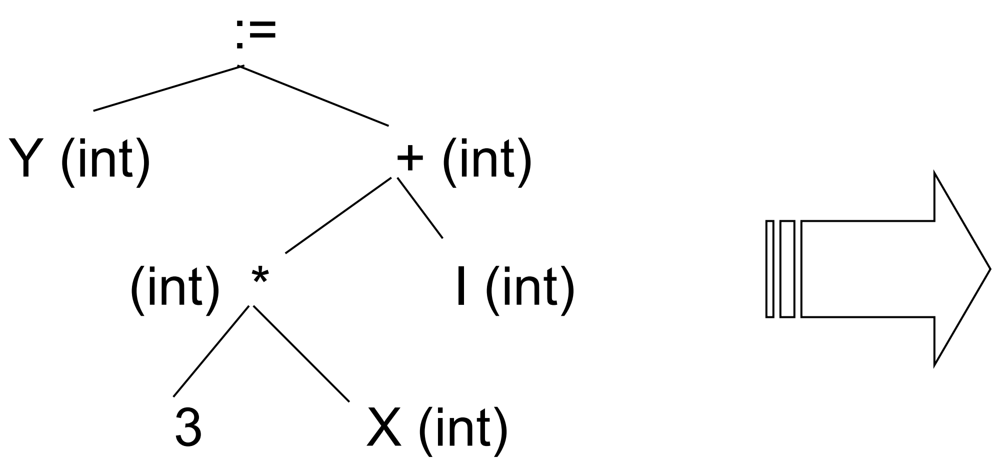
```
GenAssign(node) {
   // Gen code for RHS, leaving result in R1
   GenExpr(node.rhs, R1);
   //Calculate addr for LHS
   GenAddr(node.lhs, Addr);
   Gen(STORE, R1, Addr)
}

GenExpr(node, reg) {
   if (node.type == op) {
      GenExpr(node.lhs, reg);
      GenExpr(node.rhs, reg+1);
      Gen(node.opcode, reg, reg+1);
      ...
   }
}
```

Abstract Syntax Tree (AST) Again

- More compact form of derivation tree
  - contains just enough info. to drive later phases e.g. Y := 3*X + I



Tree Walking




```
LOAD R1, #3
LOAD R2, X
MULT R1, R2
LOAD R2, I
ADD R1, R2
STORE R1, Y
```

- Advantage of AST is that order of traversal can be chosen
  - code generated in one-pass compiler corresponds to strictly fixed traversal of tree
    (hence, code not as good)

Intermediate Code (IC)

- Instead of generating target machine code, semantic routines may generate IC.
  - can form input to separate code generator (CG)
  - advantage is that all target machine dependencies can be limited to CG

- Postfix
  - e.g. **`a := b*c + b*d`**
    `a b c * b d * + :=`
  - Concise and simple, but not very good for generating code unless stack-based architecture used

Postfix

- In normal algebraic notation the arithmetic operator appears between the two operands to which it is being applied

- This is called infix notation
  - example: a / b + c

- It may require parentheses to specify the desired order of operations
  - example: a / (b + c)

- In postfix (or Reverse Polish) notation the operator is placed directly after the two operands to which it applies

- Therefore, in postfix notation the need for parenthesis is eliminated

Operator Precedence

- To do the conversion from infix to postfix, we need to prioritise operators as follows:

^	highest priority
*, /	
+, -	
<, >, =, ...	
& (and)	
| (or)	lowest priority

Exercise

- Convert the following infix expressions into postfix:

    a+b/c

    a*c+(b-d)

    a*c+b-d

Postfix

- Example 1:
  - The infix expression: a ^ b + c
  - Becomes in postfix: a b ^ c +

- Example 2:
  - The infix expression: a ^ (b + c)
  - Becomes in postfix: a b c + ^

- Example 3:
  - The infix expression: b * c + 5 ^ (3 + 6 / a)
  - Becomes in postfix: b c * 5 3 6 a / + ^ +

Question

- Which of the following postfix expressions is equivalent to the following expression?

  a*b – c/d

  a) a b c d * - /
  b) a b * - c d /
  c) a b c d / - *
  d) a b * c d / -
  e) a b c * - d /

Answer: d
a b * c d / -