Building the Abstract Syntax Trees

Lecture 23
Section 5.3

Robb T. Koether

Hampden-Sydney College

Wed, Mar 18, 2015

Outline

The Abstract Syntax Tree

- Each statement will be represented as a distinct AST.
- Once the AST is constructed, we will traverse the tree, emitting x86 assembly code as appropriate.
- Because our parsing is bottom-up, our AST construction will also be bottom-up.

Outline

TreeNode Constructors

TreeNode Constructors

```
TreeNode(int n);
TreeNode(IdEntry id);
TreeNode(int op, TreeNode lf, TreeNode rt);
```

- `TreeNode(int n)` – Construct a `NUM` tree node containing the value `n`.
- `TreeNode(IdEntry id)` – Construct an `ID` tree node containing the `IdEntry`.
- `TreeNode(int op, TreeNode lf, TreeNode rt)` – Construct a tree node with the operation `op` at its root and with left and right subtrees `lf` and `rt`.

Outline

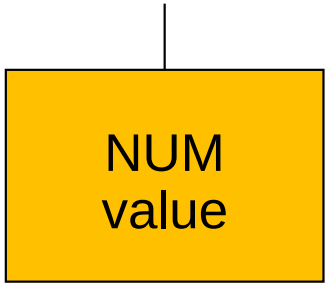
NUM Nodes

- The production for numbers is

$$num \longrightarrow \texttt{NUM}$$

NUM Nodes

- A number node is of the form



NUM Nodes

NUM Nodes

```
new TreeNode(NUM.lexval);
```

- Construct a NUM node.

Outline

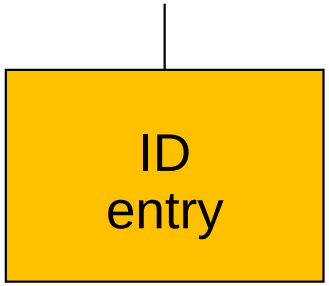
`ID` Nodes

- Two productions for identifiers are

$$dcl \longrightarrow type \texttt{ID}$$
$$lval \longrightarrow \texttt{ID}$$

ID Nodes

- An identifier node is of the form



ID Nodes

NUM Nodes

```
new TreeNode(ID.entry);
```

- Construct an ID node.

Outline

Declarations

- The production for declarations is

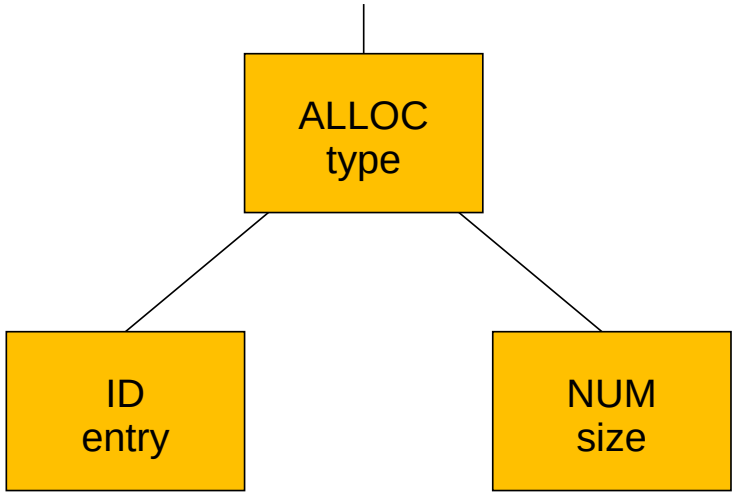
$$dcl \rightarrow type \texttt{ID ;}$$

Declarations

- The action taken by a declaration is to allocate memory for the variable.
- That will be represented by an allocation node (`ALLOC`) in the syntax tree.
- The left subtree will be the `ID` node.
- The right subtree will be a `NUM` node whose value is the number of bytes to allocate.

ALLOC Trees

- An allocation tree is of the form



ALLOC Trees

ALLOC Trees

```
TreeNode lf = new TreeNode(ID.entry);
TreeNode rt = new TreeNode(NUM.size);
new TreeNode(ALLOC, lf, rt);
```

- Construct an allocation tree.

Outline

Declarations

- The production that requires dereferencing an identifier is

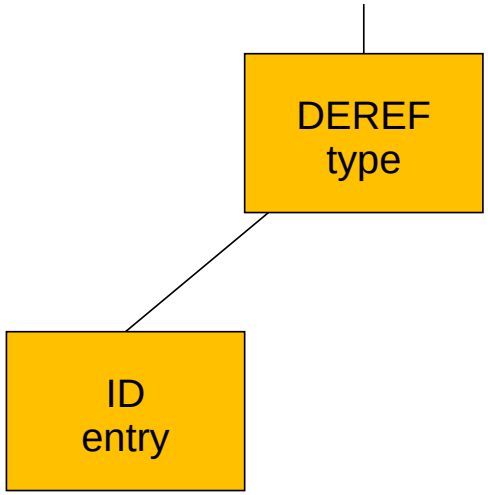
$$expr \longrightarrow lval$$

Dereferencing

- When a variable (*lval*) is used in an expression, it must be dereferenced to convert it to an *r*-value.
- That will be represented by an dereference node (`DEREF`) in the syntax tree.
- The left subtree will be the `ID` node.
- There is no right subtree.

DEREF Trees

- An dereference tree is of the form



DEREF Trees

DEREF Trees

```
new TreeNode(DEREF, id.tree, null);
```

- Construct a dereference tree.

Outline

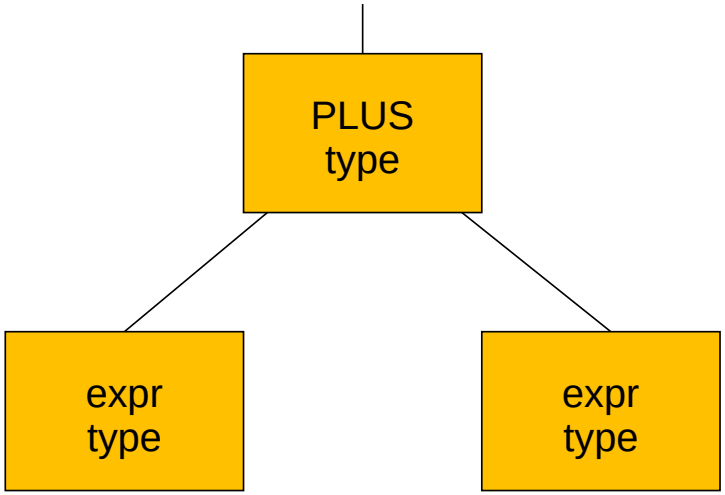
Expressions

- There are many productions for expressions.
- One such production is

$$expr \longrightarrow expr + expr$$

PLUS Trees

- The tree for addition is



PLUS Trees

```
new TreeNode(PLUS, expr1.tree, expr2.tree);
```

- Construct a tree for addition.

Outline

Assignments

- The production for assignments (not statements) is

$$expr \longrightarrow lval = expr$$

Assignments

Assignments

```
a = b = c = 0;
if (str1[i++] = str2[i++]);
```

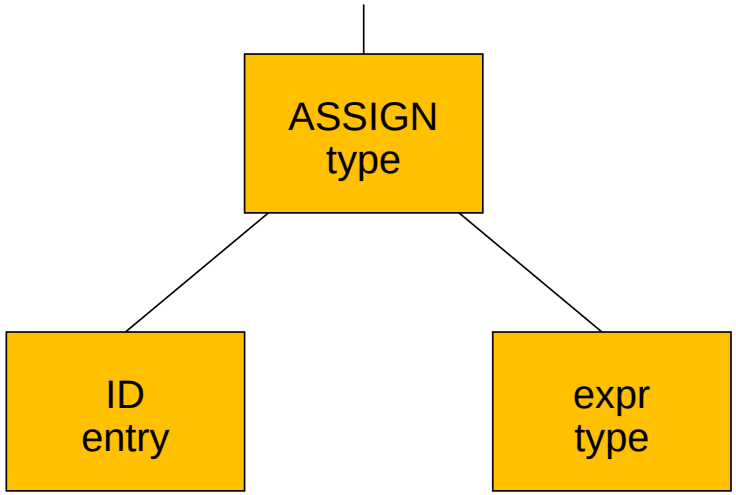
- These are *expressions*, not *statements*, because they can be used wherever that an expression is expected.
- Even though that is thought by many not to be good practice.
- The value of the expression `a = b` is the value that was assigned to `a`.

Assignments

- An assignment will be represented by an assignment tree (ASSIGN).
- The left subtree will be the *lval*.
- The right subtree will be the expression.

ASSIGN Trees

- An ASSIGN tree is of the form



ASSIGN Trees

ASSIGN Trees

```
new TreeNode(ASSIGN, id.tree, expr.tree);
```

- Construct an assignment tree.

Outline

Other Productions

Other Productions

$$\textit{expr} \rightarrow \texttt{(}\ \textit{expr}\ \texttt{)}$$
$$\textit{expr} \rightarrow \texttt{-}\ \textit{expr}$$
$$\textit{stmt} \rightarrow \texttt{READ}\ \textit{lval}$$
$$\textit{stmt} \rightarrow \texttt{PRINT}\ \textit{lval}$$

- How would we build the syntax trees for the above productions?

Outline

Assignment

Assignment

- p. 323: 1.