Compilers Course
Lecture 8: Abstract Syntax Trees

What happens to a program after syntactic analysis?

- Type checking
- Translation to intermediate code

When do we do these things?

Alt. 1: do type checking and translation *while* parsing

+ avoids building an "unnecessary" intermediate data structure.
- complex since three distinct algorithms are mixed.
- side-effects in type checking or translation make certain error-correcting parsing methods impossible.
- language constructs involving forward references or recursive definitions become difficult to handle.

Alt. 2: build a data structure representing the parse tree, traverse it as needed for type checking and translation

+ simpler since distinct algorithms can be kept separate.
+ more flexible since the parse tree can be saved for future use (in an interpreter, for instance).
+ easier to handle forward references and recursive definitions, since the parse tree can be traversed in any order.
+ no side-effects during parsing allows certain error-correcting parsing methods.
- must declare new types for representing the parse tree.
- the parse tree will consume some memory.

Modern compilers follow Alt. 2.

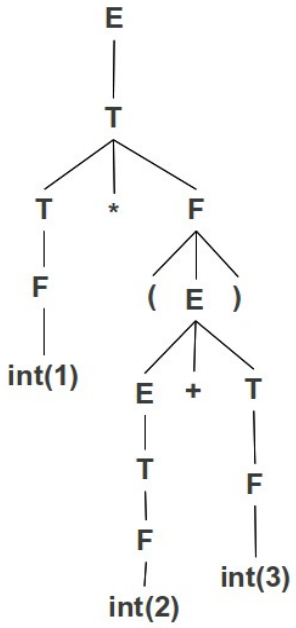
Abstract Syntax Trees == Simplified Parse Trees

Consider this expression grammar:

E → E+T | T
T → T*F | F
F → (E) | int

and the input string "1*(2+3)".

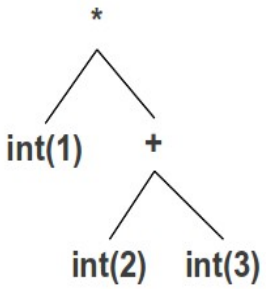
The parse tree for this string will be:



This tree contains many redundancies:

- The unit productions E → T, T → F, and F → int add levels to the tree, but they do not change the meaning of the expression
- The parentheses are only useful during parsing, afterwards they have no real purpose
- Binary expressions have their operators as sub-trees

Let us eliminate these redundancies. This gives:



which is an accurate and more compact representation of the input.

Abstract Syntax Trees (AST):

- *Representations* of the parse trees.
- Focuses on essential structure (semantics), not source syntax.

- Simplified as far as possible.
- Syntactic noise (such as redundant keywords) is eliminated.

Representing ASTs

Let us consider expressions with binary and unary operators. A node in the AST would then be one of the following cases:

- A binary node, with an operator and references to two other nodes
- A unary node, with an operator and a reference to one other node
- An integer constant node, with an integer value
- An identifier node, with a string value

There are several different possible *shapes* of the nodes.

AST Representation 1: Generic Trees

A tree node contains:

- A label (tag, operator)
- An optional attribute of type determined by the label
- An integer value N >= 0
- A sequence of N references to other nodes

Let's use the syntax (label:attribute $node_1$ $node_2$ ... $node_N$) for a node. Then the tree for 1+(2*3) could be represented as

("+" (int:1) ("*" (int:2) (int:3)))

A concrete data type in SML for this could be:

```
datatype label = ADD | MUL | INT | ID

datatype attribute = INTattr of int
                   | STRINGattr of string
                   | NOattr

datatype tree = NODE of {label: label,
                         attribute: attribute,
                         subtrees: tree list}
```

and for our expression the tree would be

```
NODE{label=ADD, attribute=NOattr,
```

```
     subtrees=[NODE{label=INT, attribute=INTattr 1, subtrees=[]},
               NODE{label=MUL, attribute=NOattr,
                    subtrees=[NODE{label=INT, attribute=INTattr 2,
                                   subtrees=[]},
                              NODE{label=INT, attribute=INTattr 3,
                                   subtrees=[]}]}]}
```

The advantage of this representation is that we only need a few simple type declarations, and then we can construct trees of any shape.

The representation in Java will look like:

```
public enum Label { ADD, MUL, INT, ID };

public class Attribute {
     private Integer INTattr = null;
     private String STRINGattr = null;
     private boolean NOattr = false;

     public Attribute(Integer INTattr){
          this.INTattr = INTattr;
     }
     public Attribute(String STRINGattr){
          this.STRINGattr = STRINGattr;
     }
     public Attribute(){
          this.NOattr = true;
     }
}

public class Tree {
     public Label label;
     public Attribute attribute;
     public Tree tree[];
}
```

The disadvantages are:

- Any kind of node can refer to any other kind of node: there is no protection against "junk" trees like a unary expression containing a statement instead of an expression as a sub-tree
- Any node can have any number of sub-trees: there is no protection against nodes with too few or too many sub-trees

AST Representation 2: Typed Trees

The tree contains nodes of different *types*. Each node type contains nodes of different *shapes*. Each shape is a record with a tag (unique in its type), and a shape-specific set of typed fields. Each field contains either an attribute of a specific type or a reference to a node of a specific type.

Example in SML:

```
datatype expr = ADD of expr * expr
              | MUL of expr * expr
              | UMINUS of expr
              | INT of int
              | ID of string
datatype stmt = IF of expr * stmt * stmt
              | WHILE of expr * stmt
              | ...
```

For the "1+(2*3)" example, the expression tree would be

```
ADD(INT(1), MUL(INT(2), INT(3)))
```

The advantage of this representation is that our implementation language prevents us from ever constructing "junk" trees.

The disadvantage is that we need many more type declarations, approximately as many as there are productions in the (simplified) grammar.

Typed Abstract Syntax Trees in Java

Each node type becomes a *class*, with a type-specific tag followed by a *class* of the possible shapes of that type. Each shape becomes a *class* with typed fields. References to other nodes use typed pointers.

```
public enum ExpressionTag { ADD, MUL, UMINUS, INT, ID };

public abstract class Tree{}
public class AddTree extends Tree{
     private Tree left, right;
     private AddTree(){} //No call for new AddTree() from outside
     public AddTree(Tree left, Tree right){
          this.left = left;
          this.right = right;
     }
}
```

```
public class MulTree extends Tree{
     private Tree left, right;
     private MulTree(){} //No call for new MulTree() from outside
     public MulTree(Tree left, Tree right){
          this.left = left;
          this.right = right;
     }
}

public class UniminusTree extends Tree{
     private Tree tree;
     private UniminusTree(){}
     public UniminusTree(Tree tree){
          this.tree = tree;
     }
}

public class IntConstTree extends Tree{
     private Integer value;
     private IntConstTree(){}
     public IntConstTree(Integer value){
          this.value = value;
     }
}

public class IDTree{
     public String name;
     private IDTree(){}
     public IDTree(String name){
          this.name = name;
     }
}

public class Expression{
     private ExpressionTag tag;
     private Tree tree;

     /* instances of Expression should only be constructed by
      calling the following utility methods */
     private Expression(){}
```

```
    /**
     * For each shape there should be a function for constructing
     * nodes of that shape, given parameters for its attributes and
     * references to other nodes.
     **/


    public static Expression makeAddTree(Tree left, Tree right){
         Expression expr = new Expression();
         expr.tag = ExpressionTag.ADD;
         expr.tree = new AddTree(left, right);
         return expr;
    }

    public static Expression makeMulTree(Tree left, Tree right){
         Expression expr = new Expression();
         expr.tag = ExpressionTag.MUL;
         expr.tree = new MulTree(left, right);
         return expr;
    }

    public static Expresssion makeINTTree(Integer value){
         Expression expr = new Expression();
         expr.tag = ExpressionTag.INT;
         expr.tree = new IntConstTree(value);
         return expr;
    }

    //...
    //MORE METHODS FOR OTHER KINDS OF TREES
}
```

Simplifiying same-shape operators

Consider binary operators. They all have the same shape (two sub-expressions) so there will be a large number of essentially identical shapes.

Solution: use a single "binary operator" node shape, and make the operator itself (+, *, etc) an attribute of the shape.

```
Example in SML:
datatype binop = ADD | MUL | ...
datatype expr = BINARY of binop * expr * expr
...
```

and similarly in Java:

```
public enum BinaryOperation { ADD, MUL, ... };
public enum ExpressionTag { BINARY, ... }
public abstract class Tree{}
public class BinaryTree{
     private BinaryOperation op;
     private Tree left, right;
}

public class Expression {
     private ExpressionTag exprTag;
     private Tree tree;
     public static Expression makeBinaryExpression
                         (BinaryOperation op, Tree left, Tree right){
          Expression expr = new Expression();
          expr.exprTag = ExpressionTag.BINARY;
          expr.exprTag = new BinaryTree(op, left, right);
          return expr;
     }
}
```

Constructing ASTs in LR parsers

Each position in the state stack corresponds to a terminal, or a variable that is the result of reducing some production.

Both terminals and variables can have attributes.

We add a "semantic stack" SEM that parallels the state stack S, such that SEM[SP] contains the AST node for the symbol that caused the transition to the state in S[SP].

At actions shift q':
    push t's attribute on SEM (SEM[SP] := t's attribute)

At actions reduce $A \rightarrow \beta$:
    create a new AST node before popping S;
    after pushing the new state, push AST node on SEM

Example:
T → T x F (MKBINARY(MUL,T,F))

```
T_AST := SEM[SP-2]
F_AST := SEM[SP-0]
NEW_AST := MKBINARY(MUL, T_AST, F_AST)
SP := (SP - 3) + 1
SEM[SP] := NEW_AST
```

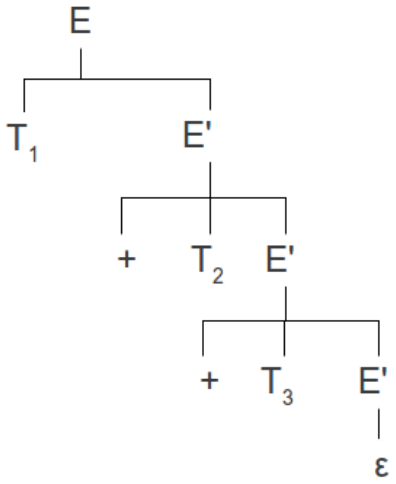
Constructing ASTs in RD parsers

Change each parsing procedure to be a function that constructs and returns the AST, as a synthesized attribute.

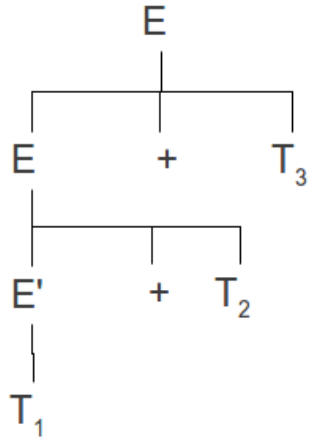
```
S():AST =
  case token() of
    IF => (match(IF);
           e = E();
           match(THEN);
           s1 = S();
           match(ELSE);
           s2 = S();
           return mkIF(e, s1, s2))
    ID => (name = attr();
           match(ID);
           match(ASSIGN);
           e = E();
           return mkASSIGN(name, e))
    BEGIN => (match(BEGIN);
              s1 = S();
              match(SEMI);
              s2 = S();
              match(END);
              return mkBEGIN(s1, s2))

E():AST =
  case token() of
    ID => (name = attr(); match(ID); return mkID(name))
    NUM => (val = attr(); match(NUM); return mkNUM(val))
```

Building AST after left-recursion elimination

E → E + T
  | T



E → T E'
E' → +T E' | ε



An operator's left operand occurs above and to the left of it in the parse tree. We need to propagate it right and down as an inherited attribute.

```
E():AST =
  case token() of
    FIRST(T) => (e1 = T(); e2 = E'(e1); return e2)
    ...
```

```
E'(left:AST):AST =
  case token() of
    PLUS => (match(PLUS); right = T(); e1 = mkADD(left,right);
             e2 = E'(e1); return e2)
    default => (return left)
```