CS 4120
Introduction to Compilers

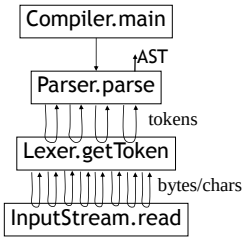
Ross Tate
Cornell University

Lecture 8: AST construction
and semantic analysis

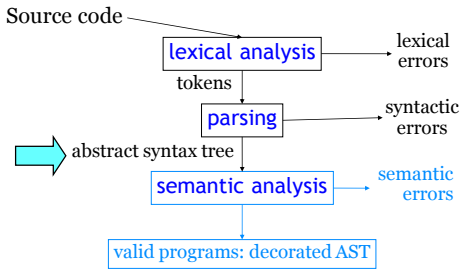
Compiler 'main program'

```
class Compiler {
        void compile() throws CompileError {
            Lexer l = new Lexer(input);
            Parser p = new Parser(l);
            AST tree = p.parse();
            typeCheck(tree);
            IR = genIntermediateCode(tree);
            IR.emitCode(output);
        }
}
```


Thread of Control

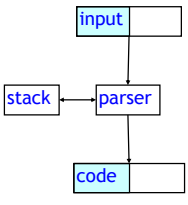


AST Construction



Do we need an AST?

- Old-style compilers: semantic actions generate code during parsing!
- Especially for stack machine:




```
expr : expr PLUS expr
    { emitCode(add); };
```

Problems:

- hard to maintain
- limits language features (e.g., recursion)
- bad code!

AST

- **A**bstract **S**yntax **T**ree is a tree representation of the program. Used for
  - semantic analysis (e.g. type checking)
  - some optimization (e.g. constant folding)
  - intermediate code generation (sometimes intermediate code = AST with somewhat different set of nodes)
- Compiler phases = recursive tree traversals
  - building new tree or modifying tree in place for next compiler phase
- Object-oriented and functional languages both convenient for defining AST nodes

Building the AST bottom-up

- Semantic actions are attached to grammar statements
- E.g. Java statement attached to each production

```
expr [returns Expr e]
    : e1=expr PLUS e2=expr
      { $e = new Add($e1.e, $e2.e); };
```

grammar production

semantic action

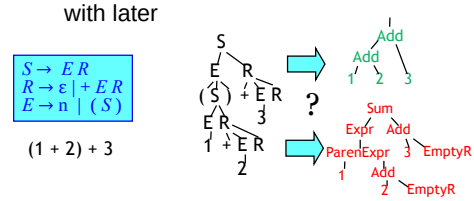
- *Semantic action* executed when parser reduces a production
- Variable e is a *value* of non-terminal symbol being reduced
- AST is built bottom-up along with parsing

How not to design an AST

- Introduce a tree node for every node in parse tree
  - not very abstract
  - creates a lot of useless nodes to be dealt with later

$S \rightarrow E\ R$
$R \rightarrow \varepsilon \mid +\ E\ R$
$E \rightarrow n \mid (\ S\ )$

(1 + 2) + 3



How not to design the AST, part II

- Simple approach: have one class AST_node
- Problem: must have fields for every different kind of node with attributes
- E.g. need information for if, while, +, *, ID, NUM

```
class AST_node {
        int node_type;
        AST_node[ ] children;
        String name; int value; …etc…
}
```

- Not extensible, Java type checking no help

Using class hierarchy

- Can use subclassing to solve problem
  - write *abstract* class for each "interesting" non-terminal in grammar
  - write non-abstract subclass for (almost) every production

$$E \rightarrow E + E \mid E * E \mid -E \mid (\ E\ )$$

```
abstract class Expr { … } // E
class Add extends Expr { Expr left, right; … }
class Mult extends Expr { Expr left, right; … }
// or: class BinExpr extends Expr { Oper o; Expr l, r; }
class Negative extends Expr { Expr e; …}
```


Creating the AST

```
expr [returns Expr e]
: MINUS e1=expr
    { $e = new Negative($e1.e); :}
| e1=expr TIMES e2=expr
    { $e = new Add($e1.e, $e2.e); }
| e1=expr PLUS e2=expr
    { $e = new Mult($e1.e, $e2.e); :}
| LPAREN e1=expr RPAREN
    { $e = $e1.e; };
```

"Component e of expr has type Expr"

ANTLR Bottom-Up

```
expr                          : …
   |    (DASH | BANG)    expr

   |  expr     (STAR | SLASH | PERCENT)    expr

   |  expr (PLUS | DASH)    expr

   | …;
```

ANTLR Bottom-Up

```
expr returns [XiExpression xi] : …
    | op=(DASH | BANG) e=expr
      { $xi = $op.type == DASH ? new XiNegative($e.xi)
            : new XiNegate($e.xi); }
    | l=expr op=(STAR | SLASH | PERCENT) r=expr
      { $xi = $op.type == STAR ? new XiMultiply($l.xi, $r.xi)
            : $op.type == SLASH ? new XiDivide($l.xi, $r.xi)
            : new XiMod($l.xi, $r.xi); }
    | l=expr (PLUS | DASH) r=expr
      { $xi = $op.type == PLUS ? new XiAdd($l.xi, $r.xi)
            : new XiSubtract($l.xi, $r.xi); }
    | …;
```


Top-Down AST Construction

- parse_*X* method for each non-terminal *X*
- Return type is abstract class for *X*

```
Stmt parseStmt() {
    switch (next_token) {
      case IF: consume(IF); consume(LPAREN);
        Expr e = parseExpr;
        consume(RPAREN);
        Stmt s2, s1 = parseStmt();
        if (next_token == ELSE) { consume(ELSE);
                                  s2 = parseStmt(); }
        else s2 = new EmptyStmt();
        return new IfStmt(e, s1, s2); }
      case ID: …
```


ANTLR Top-Down

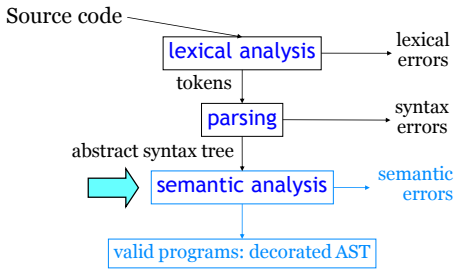
```
expr : …
      | ID LPAREN      exprs RPAREN

      | …;
exprs
      :
      (   expr
            (COMMA     expr                    )*
      )?;
```


ANTLR Top-Down

```
expr : …
      | ID LPAREN es=exprs RPAREN
      { $xi = new XiFunctionCall($ID.text, $es.xi); }
      | …;
exprs returns [List<XiExpression> xi]
      : { $xi = new ArrayList<XiExpression>(); }
      (e=expr { $xi.add($e.xi); }
            (COMMA e=expr { $xi.add($e.xi); })*
      )?;
```


Semantic Analysis



Goals of Semantic Analysis

- Find all possible remaining errors that would make program invalid
  - undefined variables, types
  - type errors that can be caught *statically*
  - uninitialized variables, unreachable code
- Figure out useful information for later compiler phases
  - types of all expressions
  - data layout: memory sizes

Recursive semantic checking

- Program is tree, so...
  - recursively traverse tree, checking each component
  - traversal routine returns information about node checked

```
class Add extends Expr {
        Expr e1, e2;
        Type typeCheck() throws SemanticError {
                Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
                if (t1 == Int && t2 == Int) return Int;
                else throw new TypeCheckError("type error +");
        }
```


Type-checking identifiers

```
class Id extends Expr {
        String name;
        Type typeCheck() {
                return ?
        }
}
```

Need an **environment** that keeps track of types of all identifiers in scope: **symbol table**

Symbol Table – Type Context

- Can write formally as set of *id* : *type* pairs: { x: int, y: array[string] }

```
int i, n = ...;
for (i = 0; i < n; i++) {
        boolean b = ...
}
```

{ i: int, n: int }

{ i: int, n: int, b: boolean }

Specification

- Symbol table maps identifiers to types

```
class SymTab {
  Type lookup(String id) …
  void add(String id, Type binding) …
}
```


Using the symbol table

- Symbol table is argument to all checking routines

```
class Id extends Expr {
    String name;
    Type typeCheck(SymTab s) {
      try {
        return s.lookup(name);
      } catch (NotFound exc) {
        throw new UndefinedIdentifier(this);
      }
    }
}
```


Propagation of symbol table

```
class Add extends Expr {
        Expr e1, e2;
        Type typeCheck(SymTab s) {
                Type t1 = e1.typeCheck(s),
                     t2 = e2.typeCheck(s);
                if (t1 == Int && t2 == Int) return Int;
                else throw new TypeCheckError("+");
        }
```

- Same variables in scope – same symbol table used
- When do we add new entries to symbol table?

Adding entries

- Java: statement may declare new variables. { a = b; int x = 2; a = a + x }
- Suppose $\{stmt_1;\ stmt_2;\ stmt_3 \ldots\}$ represented by AST nodes:

```
abstract class Stmt { … }
class Block { List<Stmt> stmts; … }
```

- And declarations are a kind of statement:

```
class Decl extends Stmt {
    String id; TypeExpr typeExpr; …
```


A stab at adding entries

```
class Block {
    Vector stmts;
    void typeCheck(SymTab s) {
        for (st : stmts) {
            st.typeCheck(s);
            if (st instanceof Decl) {
                Decl d = (Decl) st;
                s.add(d.id, d.typeExpr.interpret());
            }
        }
    }
}
```


Restoring Symbol Table

```
int x = 5;
{ int y = 1; }
x = y;  // should be illegal in Java!
```

scope of y in Java

Handling declarations

```
class Block {
    Vector stmts;
    Type typeCheck(SymTab s) {
        SymTab s1 = s.clone();
        for (int i = 0; i < stmts.length(); i++) {
            stmts[i].typeCheck(s1);
            if (stmts[i] instanceof Decl) {
                Decl d = (Decl) stmts[i];
                s1.add(d.id, d.typeExpr.interpret());
            }
        }
    }
}
```

Declarations added in block (to s1) don't affect code after the block

Storing Symbol Tables

- Many symbol tables constructed during checking
  - May keep track of more than just variables: type definitions, break & continue labels, …
  - Top-level symbol table contains global variables, type & module declarations,
  - Nested scopes result in extended symbol tables containing additional definitions for those scopes.
- Can reconstruct symbol tables, but useful to save in corresponding AST nodes to avoid recomputation

How to implement Symbol Table?

- Stateful? Three operations:

```
Object lookup(String name);
void add(String name, Object type);
SymTab clone();
```

- Stateless? Two operations:

```
Object lookup(String name);
SymTab add(String, Object);
```

Stateful: Linked list of tables

```
class SymTab {
    SymTab parent;
    HashMap table;
    Object lookup(String id) {
      if (table.get(id) != null) return table.get(id);
      else return parent.lookup(id); // can cache..
    }
    void add(String id, Object t)
      { table.add(id,t); }
    SymTab(Symtab p)
      { parent = p; } // =clone
}
```


Functional: Binary trees

- Discussed in Appel Ch. 5
- Implements the two-operation interface

  Object lookup(String name);

  SymTab add(String, Object);
  - non-destructive add so no cloning is needed
  - O(log n) performance: clones only the path from added node to the root.

Decorating the tree

- How to remember expression type?
- One approach: record in the node

```
abstract class Expr {
        private Type type = null;
        public final Type typeCheck()
            { if (type == null) type = calcType();
              return type; }
        protected Type calcType();
}
class Add extends Expr { Type calcType() {
        Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
        if (t1 == Int && t2 == Int) return Int;
        else throw new TypeCheckError("+");
}}
```

- Maybe useful to record symbol table used (if not destructively modified later…)