

6.035

Fall 2006

Project 2 Semantic Analysis

Michael Gordon

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Project 1 Impressions

- Overall a great job!
 - But only 5% of your grade
- Remember to correct any errors/inadequacies that were found.
 - Use the (revealed) hidden tests
- Create a testing infrastructure!
 - JUnit or create your own

Project Phase 2 Summary

- Convert concrete syntax of your grammar to high-level IR.
 - Abstract syntax tree plus symbol table(s)
 - much simpler than lecture discussion
- Semantics Analysis:
 - Traverse AST to perform semantic checks
 - Build and query symbol table during traversal
- Pretty print AST and symbol table during traversal when in debug mode.
 - You decide format

Possible Project Flow

- Create a high-level representation of the program
 - Convert the concrete syntax to abstract syntax
 - Employ parser actions to construct high-level IR during parse
- Run semantic checking on high IR
 - Attributed visitor(s) on IR or recursive function on IR
 - Manipulate symbol table(s) during pass(es)
 - Report errors to user

Abstract Syntax and CUP

- Each terminal / non-terminal may have an associated type:

- non terminal IRExpr expr ;

- Semantic Actions:

- executed whenever CUP reduces the preceding terminal/non-terminal

- Name terminal/non-terminal to use in action

- assign to RESULT assign to LHS

- must return a value of appropriate type

- `expr ::= expr:e1 PLUS expr:e2`

- `{: RESULT = new BinExpr(e1, e1, OP.PLUS); :}`

Abstract Parse Trees

- Separate Parsing from Semantics.
 - CUP performs bottom-up, left-to-right traversal (post-order)
 - Not convenient for semantic checking
- We don't want to use the concrete parse tree that is produced by the parse
 - Based on grammar
 - Could have superfluous tokens
- Build an Abstract Syntax Tree
 - Conceptually, we have an abstract grammar
 - not bothered by ambiguity, parse already happened!

Abstract Syntax Representation

- Recommendation: Use a non-objected-oriented style.
- Separate class for most non-terminals (kinds) with a sensible class hierarchy:
 - IR: (line number)
 - Decl(...)
 - VarDecl(...)
 - » FieldDecl(...)
 - » LocalDecl(...)
 - MethodDecl
 - VarDecls(List<vardecl>)
 - Statement(...)
 - For (Expr initExpr, Expr endExpr, Block block)
 - If (Expr expr, Block trueBlock, Block falseBlock)
 - Block (VarDecls varDecls, Statements stmts)
 - Expr(...)
 - BinaryExpr: (Expr expr1, Expr expr2, int operator)
 - MethodCallExpr: (Method method, ?? args)

Abstract Syntax Representation

- Separate classes that visit the IR to perform operations (Interpretations)
 - Implements a visitor interface on the IR (see Design Patterns)
 - Examples: Semantic Check, Pretty Print

Building the AST, Example

```
non terminal VarDecls      var_decls;
non terminal Block        block;

block ::= '{' var_decls:vds stmts:ss '}'
      { : RESULT = new Block(line, vds, ss); : }

var_decls ::= var_decls:vds var_decl:v
           { :
             List children = vds.getChildren();
             children.add(v);
             RESULT = new VarDecls(line, children);
           : } |
           var_decl:v
           { :
             List children = new LinkedList();
             children.add(v);
             RESULT = new VarDecls(line, children);
           : }
```

Semantic Analysis

- Perform semantic checking to ensure that the program is (statically) legal.
 - If error(s), provide the user with meaningful and helpful error message(s).
- Operates on the high-level IR
 - Traverses AST
 - Manipulates symbol table(s)

Symbol Tables

- A symbol table maps identifiers to types and locations.
- For this phase we will build/use the symbol table while performing semantic checking.
- Terminology: symbol table part of *environment* that contains *bindings*.
 - Your environment could include multiple symbol tables for multiple name spaces (see Tiger Book for example)
- Implementation decisions entirely at your discretion.
 - Write-up should include complete description of your implementation.

Symbol Tables

- Functionality:
 - Newer bindings have precedence over older bindings.
 - Need a mechanism to undo a set of bindings:
 - Used when popping out of a scope
- Many possible choices:
 - How many symbol tables?
 - Hashing?
 - Functional vs. Imperative
 - Destructive updates (imperative)
 - Immutable, persistent (functional)

Bindings

- The symbol table is filled with bindings.
- Ex:
 - Id -> Type (for value variables)
 - Id -> Signature (for methods)
 - Id -> Type (for type variables)
- What do you need for decaf?

Scoping

- Scope Rules: Associate name with declaration.
- A new scope is created upon entering a block.
 - Variable definitions of current scope shadow definitions of outer scope.
 - Upon entering a scope, must remember state of symbol table.
 - Add binding to symbol table as we visit variable/method definitions.
 - Look-up variables in the symbol table as we visit statements and expressions.
 - Upon exiting a scope, must restore the symbol table to its state prior to the point when the scope was entered.

Semantic Checks

- Flow of control checks
 - Ex: cannot exit from meth without returning a value of correct type (if meth returns a value)
- Uniqueness check
 - Ex: identifier cannot be defined twice in same scope
- Type checks
 - Ex: each expression has correct type for use
- Your write-up should include a list of all the checks you implemented.

Type System

- Your write-up should include a *Type System* for decaf on abstract syntax.
 - see Dragon Book chapter 6
- A type system is a mathematical object used to define the typing rules of a programming language.
 - A collection of rules for assigning types to various parts of the program.
 - The type system will be implemented in your compiler.

Type System

- A type system is *sound* if it allows us to statically determine if a program has a type error.
- A language is *strongly typed* if we can create a sound type system for it.

Attribute Grammars

- Grammar with productions and associated actions (just like CUP)
- Every non-terminal has an attribute.
- The attribute for a LHS non-terminal is calculated from the attributes of the non-terminals in the RHS
- The attribute calculated for the starting production is the attribute calculated for the "parse."

Attribute Grammar Example

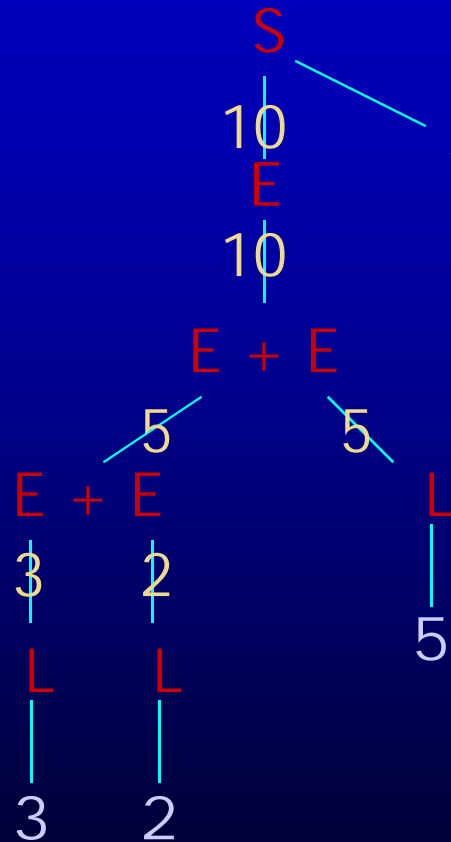
Calculate the *Val* attribute.

Productions	Attribute Rules
$S \rightarrow E \text{ ;}$	$S.Val = E.val$
$E \rightarrow E_1 \text{ PLUS } E_2$	$E.Val = E_1.Val + E_2.Val$
$E \rightarrow L$	$E.Val = L.Val$
$L \rightarrow \text{DIGIT}$	$L.Val = \text{digit}$

Attribute Annotated Parse Tree

3 + 2 + 5;

S.Val = 10



Attribute Grammar as a Type System

- Every non-terminal has an attribute, *type*.
- If the attribute computed for the program is not *error*, then the program type checks.
- Ignore other errors that aren't type related:
 - Flow of control checks
 - Uniqueness checks

Type System Example

```
expr ::= e1 PLUS e2
      { : expr.type := if e1.type = int and e2.type = int
        then int
        else error : }
```

```
int_lit ::= INT_LITERAL
        { : int_lit.type := int : }
```

Type System Example Con't

```
program ::= ... var_decls methods ...  
        { : program.type := if vardecls.type != error and  
                               methods.type != error  
                               then void  
                               else error : }
```

...

```
var_decl ::= type ids  
         { : foreach id in ids {put(id, type);}  
           var_decl.type := void : }
```

...

```
stmt ::= if e then block  
       { : stmt.type := if e.type = boolean  
         then block.type  
         else error : }
```

Type System Example Con't

```
expr ::= id ( expr1, expr2, ... , exprN)
      { : sig = lookup(id);
        stmt.type := if sig.type != method and
                      sig.numArgs = N and
                      expr1.type = sig.arg1.type and
                      expr2.type = sig.arg2.type ...
        then sig.returnType
        else error :}
```

Type System Examples Con't

```
stmt ::= RETURN expr `;`  
      { : sig = getEnclosingSig();  
        expr.type :=      if sig.returnType != void and  
                        sig.returnType = expr.type  
                        then void  
                        else error  
      : }
```

Where `getEnclosingSig()` returns the type signature of the enclosing method.

Type System Example Con't

```
block ::= '{' var_decls stmts '}'
      { : begin_scope();
        block.type := if var_decls.type = error or
                        stmts.type = error
                      then error
                      else void
        end_scope();
      : }
```

Where `begin_scope()` marks the current state of the symbol table and `end_scope()` restores the symbol table to the last mark.

Meetings

- Schedule group meeting with me for end of week (Thursday or Friday).
- Each member needs to attend!
- Be prepared to discuss (informally):
 - Questions on decaf semantics
 - Design of your IR
 - Abstract syntax
 - Symbol table
 - Your type system
 - Work division among group