6.035
Project 2
Semantic Analysis

Jason Ansel

MIT - CSAIL

First Project Wrap-up

- Any questions/comments/concerns about the first project?
- Implementation grade (automated tests; 75%) will be posted by end of week
- Design/doc/write-up grade (subjective; 25%) will be posted in 1-2 weeks

Groups

- You should be forming them
  - See my email
- Later today
  - Project 2 will be posted
  - Groups will be created on athena
    - (for those that emailed me)

---

* Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Group Meetings

- Short meeting with me (the TA) and your group
- Email me to schedule it
- We will go over your proposed IR design
- Catch problems with design early on

Project 2

- 60% Projects
  - 5% P1
  - **7.5% P2 (you are here)**
  - 10% P3
  - 7.5% P4
  - 30% P5
- 30% Quizzes
- 10% Mini-Quizzes (each lecture, 5 so far)

Project Phase 2 Summary

- Create a type system for decaf.
  - Attributed grammar
- Convert concrete syntax of your grammar to high-level IR.
  - Abstract syntax tree plus symbol table(s)
  - much simpler than lecture discussion
- Semantics Analysis (includes type checking):
  - 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 testing infrastructure!
  - JUnit or create your own
- Write type system
- 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
  - Visitor(s) on IR or recursive function on IR
  - Manipulate symbol table(s) during pass(es)
  - Report errors to user

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.

SYMBOL TABLES AND SCOPING

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.

What does a new scope mean?

- Variable definitions of current scope shadow definitions of outer scope.
- Upon entering a scope, must remember state of symbol table.

What do we do in a scope?

- Add binding to symbol table as we visit variable/method definitions.
- Look-up variables in the symbol table as we visit statements and expressions.

What happens when we exit a scope?

- Upon exiting a scope, must restore the symbol table to its state prior to the point when the scope was entered.

ABSTRACT TYPE SYSTEMS

Type System

- Your write-up should include a *Type System* for Decaf on abstract syntax.
- A type system is 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 ANTLR)
- Every non-terminal has an attribute.
- The attribute calculated for the starting production is the attribute calculated for the “parse.”

Attribute Grammar Example

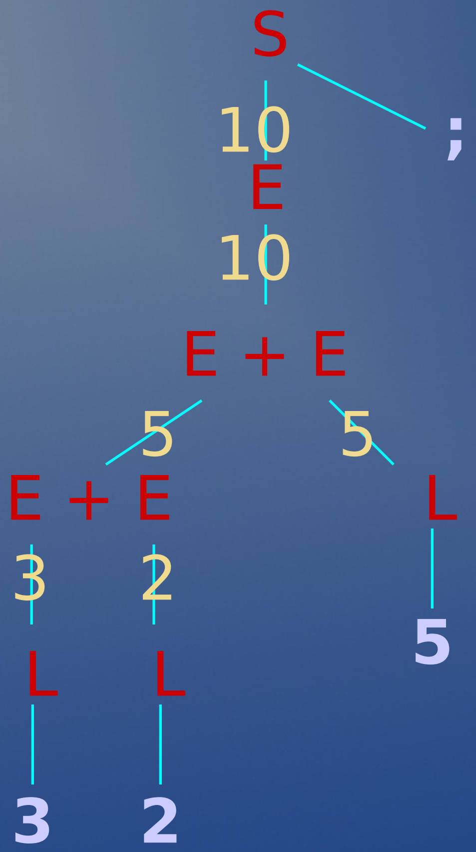
Calculate the *Val* attribute.

Productions	**Attribute Rules**
S -> E ‘;’	S.Val = E.val
E -> E_1 PLUS E_2	E.Val = E_1.Val + E_2.Val
E -> L	E.Val = L.Val
L -> DIGIT	L.Val = 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.

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);
         expr.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 -> { begin_scope(); }
   ‘{’ var_decls stmts ‘}’
   {
      block.type :=  if var_decls.type = error or
    stmts.type = error
        then error
        else void
      end_scope();
   }
```

here `begin_scope()` marks the current state of the symbol tab
d `end_scope()` restores the symbol table to the last mark.

ABSTRACT SYNTAX TREES

Abstract Syntax Tree

- Concrete Syntax (Parse) Tree
  - The parse tree produced by your Antlr grammar
  - Redundant and useless information (punctuation, etc.)
- Abstract Syntax (Parse) Tree
  - Clean up parse tree
  - Conveys structure of the program
  - Represented as data structures in compiler

Choices For Nodes of Parse Tree

- Homogeneous nodes
  - All nodes of the same type
  - General node with child pointer and siblings pointers
  - Distinguish nodes by internal “type” variable
  - Big case statement when walking tree (Antlr can do)

- Heterogeneous nodes
  - Multiple types of nodes with different information and structure
  - Use Visitors to walk tree, each node defines how to visit it

Constructing AST

1. Build your own AST (heterogeneous nodes)
   - From ANTLR's parse of your grammar
   - Constructed with semantic actions.

1. Use ANTLR's AST (homogeneous nodes)
   - Based on grammar
   - Can massage tree structure
   - Can use TreeWalker to walk tree

BUILD YOUR OWN HETEROGENEOUS AST

Abstract Syntax Representation

- Separate class for most non-terminals (kinds) with a sensible class hierarchy:
  - `IR: (line number, column)`
    - `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)`

Antlr Actions

- **Code that is run during the parse.**

```
rule { /* before */ } :
  A { /* during */ } B |
  C D { /* after */ } ;
```

Typical Antlr Actions

```
rule returns [ type varName ]
 { /* initialize vars */ } :
 t:TOK b=rule_b {
 /* set return value,
    can use b to refer to
   rule_b’s return value,
 t to refer to token */
 } ;
```

Antlr Action Example

```
class IRif extends IRStmt {
  IRif( Token t ) { ... }
  void setTest( IRExpr e ) { ... }
  void setStmt( IRStmt S) { ... }
}
```

```
stmt returns [IRStmt n] :
  IF p=expr THEN t=stmt
      { n = new IRif(IF);
          n.setTest(p); n.setStmt(t);} ;
```

Semantics Analysis on Hetero AST

- Use the visitor pattern as a contract for classes that walk the AST.
- Manipulate/access symbol table as you walk.
- Multiple visitors to implement semantic analysis.

USE ANTLR TO BUILD HOMOGENEOUS AST

buildAST=true

```
class DecafParser extends Parser;
  options { buildAST=true; }
```

- With this option, Antlr will create a flat AST for all matched rules.
- But you have control over how it creates the AST and what nodes is creates.
- Antlr TreeWalkers are grammar that specify how to walk the tree.

Antlr Tree Construction Example

```
expr : mexpr ('+' mexpr)* ;
mexpr : INT ('*' INT)* ;
```

Run on "4+5*6" will give all siblings:

4 -> + -> 5 -> * -> 6

Tree Construction Control

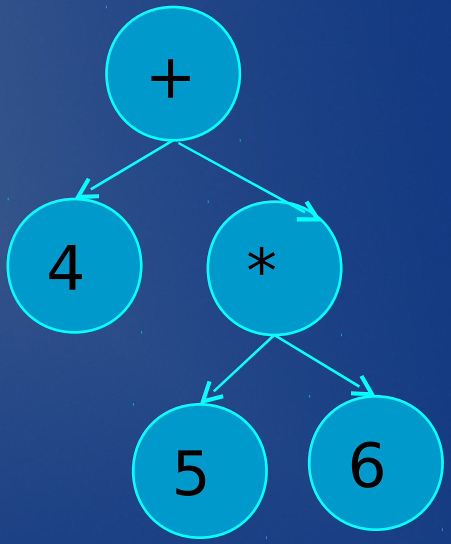
- After a token, ^ makes the node a root of a subtree for the current rule, then we continue to add sibling to the *subtree*.

- After a token, ! prevents an AST node from being built.

Antlr Tree Construction Example

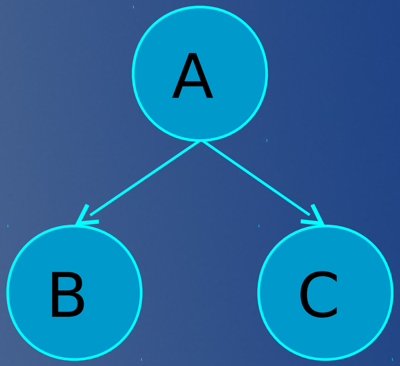
```
expr : mexpr (’+’^ mexpr)* ;
mexpr : atom (’*’^ atom)* ;
atom : INT ;
```

Run on “4+5*6” will give:

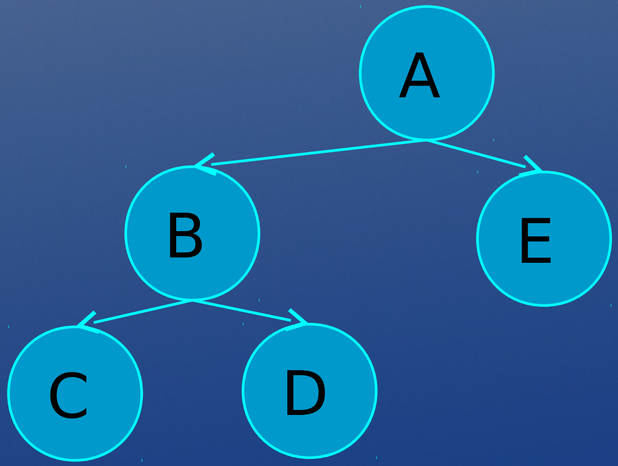


LISP-like Tree Syntax

- #(parent child1 child2 …)
- EX: #(A B C)



- EX: #(A (#B C D) E)



Another Example

```
args:
  "("! ( arg (","! arg)* )? ")"!
   { #args = #([ARGS], args); } ;
```

ARGS

arg **arg** **arg**

What to do?

```
uminus: (MINUS)* expr;
```

Tree Parsers

- Parse a tree as a stream of nodes in two dimensions.
- We can specify the rules for matching a tree
  - The valid structure of a tree
- We can specify actions that happen while walking the tree

Example

```
expr : mexpr (“+”^ mexpr)* ;
mexpr : atom (“*”^ atom)* ;
atom: INT;
```

```
class CalcTreeWalker extends TreeParser;
expr returns [int r]
{
   int a,b;
   r=0;
}
   :   #(“+” a=expr b=expr) {r = a+b;}
   |   #(“*” a=expr b=expr) {r = a*b;}
   |   i:INT
   {r = Integer.parseInt(i.getText());}
   ;
```

Cons of ANTLR AST Construction

- Will take you some time to understand Antlr’s AST construction syntax/semantics.
  - Expect obscure errors
- Might be difficult to write a TreeWalker for your AST
  - TreeWalkers are good for small grammars with few node types.

6.035 Computer Language Engineering
Spring 2010