IN2009 - Language Processors
Week 8
Typechecker & Interpreter

Igor Siveroni
City University London

Session plan

- JJTree
  - Traversal of ASTs
  - Example: Calculator of arithmetical expressions
- The SIMPLE Programming Language
  - Syntax
  - JJTree specification
- Semantic Analysis
  - Identification
  - Typechecking
- Execution of SIMPLE programs
  - Interpreter implementation

JJTree

- A preprocessor for JavaCC that :
  - Inserts abstract syntax tree (AST) building actions at various places in the JavaCC source.
  - It generates:
    - A JavaCC file (.jj) from a .jjt file.
    - Java classes that implement the AST, including the Node interface and SimpleNode superclass.
  - The generated AST classes can be extended with:
    - New fields e.g. to store the numeric value of a node.
    - New methods e.g. Tree traversals that implement print operations (dump), semantics analysis (typecheck) or execution (interpret)

JJTree specification

- Let’s consider the syntax for arithmetical expressions used in Lab 6 (week 7):

  E → T (“+” T | “-” T)*
  T → F (“*” F | “/” F)*
  F → <NUM> | “(“ E “)”

The JJTree specification for E:

```
void E() :  // default – generates an ASTE node
 {  }
 {    T()  (        "+“ T()
            |    "-" T()
          )*
 }
```

Generates AST nodes with one or more subnodes, depending on the number of T()‘s matched.
There is no way of knowing if it came from an addition or subtraction!!

JJTree specification

JJTree annotations override the default. We can decide the names of the AST nodes, and where they are generated. For example:

```
void E()  #void  :   // #void produces no default node.
    {  }
    {    T()  (        "+“ T() #PlusExp(2)      // produces an ASTPlusExpNode
                 |    "-" T()  #MinusExp(2)   // produces an ASTMinusNode
               )*
    }
```

Generates ASTPlusExp and MinusExp nodes, depending on the operator matched by the parser. Each Node has 2 subnodes (children), corresponding to the last T()’s matched.

JJTree nodes and state

- all AST nodes implement interface Node
  - The generated AST nodes extend class SimpleNode, which implements Node.
  - useful methods provided include:
    - public void jjGetNumChildren() which returns the number of children
    - public void jjtGetChild(int i) which returns the i'th child
- the `state' is in a parser field called **jjtree**
  - the root is at **Node rootNode()**
  - so you can display the tree with

    ((SimpleNode)parser.jjtree.rootNode()).dump("");
  - method dump() is defined in class SimpleNode

Extending JJTree class definitions

- JJTree AST class definitions can be modified to include new fields and methods.
- For example, let's consider an initial specification of F:

```
void F() #void :   //  No ASTF node is created
     { Token t; }
     {        <NUM> #NumExp   // produces ASTNumExp node (no children)
          | "(" E() ")“
     }
```

Produces an ASTNumExpNode if the token <NUM> is matched.

Problem: We have lost the value associated to the number!!

... continued

- In order to store the value associated with <NUM> we need to:
  - Add a field to ASTNumExp.java to store the numeric value of <NUM>:

```
class ASTNumExp extends SimpleNode  {
...
      public int val;   // new field declaration
}
```

  - Update the JJTree spec to initialise the new field:

```
void F() #void :
{ Token t; }
{        (t=<NUM>  { jjtThis.val = Integer.parseInt(t.image);})  #NumExp
      | "(" E() ")“
}
```

... continued

- We can also add a new method that evaluates the arithmetical expression represented by the AST.
- Let's call this new method **int evaluate()**
- We need to:
  - Declare the new method in interface Node.java

    ```
    public int evaluate();
    ```

  - Give a default implementation in SimpleNode.java:

    ```
    public int evaluate()
    {   return 0;  // default   }
    ```

... continued

- And provide implementations of **evaluate** to each ASTxxxx.java:

```
public  class ASTNumExp extends SimpleNode   {
...
   public   int  evaluate() {
      return val;
    }
}

public class ASTPlusExp extends SimpleNode   {
......
    public int evaluate() {
        return jjtGetChild(0).evaluate() + jjtGetChild(1).evaluate();
    }
}
```

The SIMPLE Programming Language

SIMPLE → CompilationUnit

CompilationUnit → (VarDeclaration “**;**”)* Statement*

VarDeclaration → (“**boolean**” | “**int**”) <ID>

Statement →

SkipStatement | AssignStatement |

IfStatement | WhileStatement |

ReadStatement | WriteStatement

SkipStatement → “**skip**” “**;**”

AssignStatement → <ID> “**=**“ Expression “**;**”

IfStatement → “**if**” “**(**“ Expression “**)**” StatementBlock

[“**else**” StatementBlock]

WhileStatement → “**while**” “**(**“ Expression “**)**” StatementBlock

ReadStatement → “**read**” <ID> “**;**”

WriteStatement → “**write**” <ID> “;”

StatementBlock → “**{**“ Statement+ “**}”**

The SIMPLE Programming Language

Expression → OrExp

OrExp → AndExp (“**or**” AndExp)*
AndExp → EqExp (“**and**” EqExp)*
EqExp → RelExp ((“**==**“ | “**!=**“) RelExp)*
RelExp → AddExp ((“**<**“ | “**>**” | “**<=**“| “**>=**“) AddExp)*
AddExp → MultExp ((“**+**” | “**-**”) MultExp)*
MultExp → PrimExp ((“*****” | “**/**”) PrimExp)*
PrimExp → Literal | <ID> | “**(**“ Expression “**)**”
Literal → <INT>
BooleanLiteral → > “**true**” | “**false**”

SIMPLE JJTree: Example

```
void VarDeclaration() :  /* By default will create ASTVarDeclaration nodes. */
    { Token t; }
    {  (                    /* `type' declared in ASTVarDeclaration.java. */
         <KEYBOOL> { jjtThis.type = TPLTypes.boolType; }
      |
        <KEYINT> { jjtThis.type = TPLTypes.intType; }
      )    t = <IDENTIFIER>     {  jjtThis.name = t.image; }
       /* `name' declared in ASTVarDec.java  -    'image' is the text of the token. */
    }
```

- The JJTree specification above assumes that ASTVarDeclaration.java has been extended with two fields:
  - Field ‘type’ type TPLTypes (boolType or IntType). Required for typechecking.
  - Field ‘name’ of type String.

SIMPLE JJTree: Example

```
void MultiplicativeExpression() #void :        /* #void=produce no default node. */
{}
{
 PrimaryExpression()
 (
   "*" PrimaryExpression() #Mul(2)             /* produce an ASTMul node. */
  |
   "/" PrimaryExpression() #Div(2)             /* produce an ASTDiv node. */
 )*
}
void Literal() #void :
{   Token t;}
{ (  t=<INTEGER_LITERAL>     {   jjtThis.val = Integer.parseInt(t.image);   }   )

                #IntConst  /* Create an ASTIntConst leaf (note no children). */
 |  BooleanLiteral()
}
```

With field `val' declared in ASTIntConst.java.

SIMPLE Example

- For example, the string

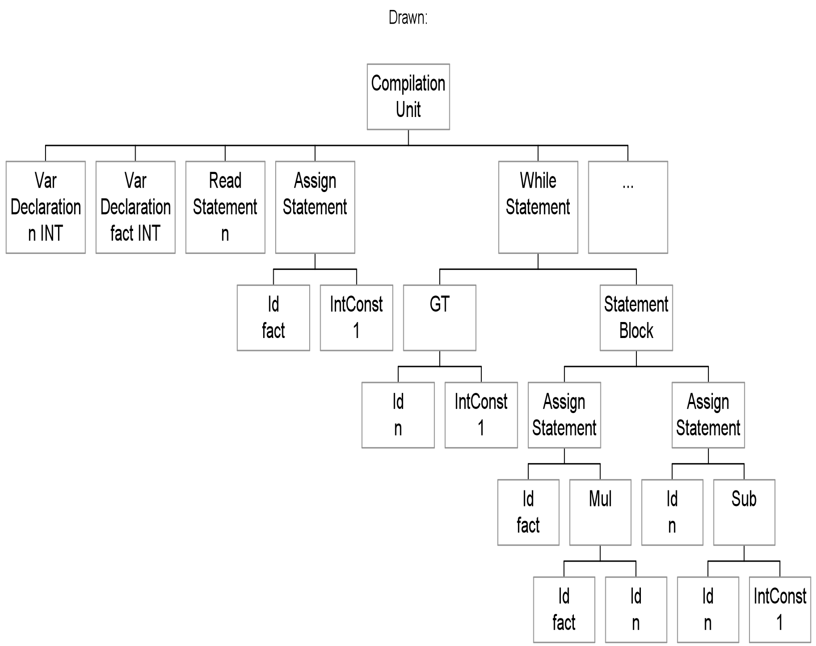
```
“ int n;  int fact;
read n;
fact = 1;
while (n > 1) {
    fact = fact * n;   n = n – 1;
}
write n;”
```

Is a valid SIMPLE program i.e. It satisfies its syntax.

- Given the JJTree specification shown in the previous slides (an extract of the spec given in your coursework), the result of a call to **dump**, and the representation of the AST generated by JavaCC for the program above is ….

ASTs from JJTree

```
CompilationUnit
 VarDeclaration n INT
 VarDeclaration fact INT
 ReadStatement n
 AssignStatement
  Id fact
  IntConst 1
 WhileStatement
  GT
   Id n
   IntConst 1
  StatementBlock
   AssignStatement
    Id fact
    Mul
     Id fact
     Id n
   AssignStatement
    Id n
    Subtract
     Id n
     IntConst 1
 WriteStatement fact
```



Contextual or Semantic Analysis

- Parsing does not find all the legal programs
  - some rules are *context dependent*
  - depend on non-syntactic aspects of the program
  - eg the expression (0=1)+2 is not *well-formed* although it is legal in our TPL syntax, because it does not typecheck
- We will perform two analyses:
  - **Identification**:
    Applying the scope and binding rules of the language to find applied occurences of identifiers (eg variable names in expressions) and relating to the appropriate declarations.
    Output: Symbol table.
  - **Type checking**:
    infer the type of each expression and check that it matches the expected type
  - The symbol table is used by other semantics analyses e.g. type checking, and processes e.g. code generation.

Identification

- collect all identifiers in a symbol table with attributes like name, type, scope level, etc
- uses variable declaration ASTs to:
  - Identify types and variable names, and them to the symbol table, perhaps with initial value.
  - Identify scope level (depth of nesting), if necessary
- traverses ASTs
  - checking that applied occurrences of identifiers have related declarations (are in table) at appropriate scope level.
- for SIMPLE:
  - static Hashtable of IdSymbols, declared in SimpleNode
  - traverse AST by writing identification() method for each node type, called initially on rootNode()

Identification in SIMPLE
Method **void identification()**

- In Node.java:

```
public void identification ();   // now, part of the interface
```

- In SimpleNode.java

```
protected static java.util.Hashtable symtab = new java.util.Hashtable();

 /* This will be overridden by those AST nodes doing identification. */
  public void identification () {    /* Do nothing. */  }
```

- In ASTCompilationUnit.java

```
public void identification ()
{     int i, k = jjtGetNumChildren();
      for (i = 0; i < k; i++)
              jjtGetChild(i).identification();
}             // visits each declaration and statement
```

Identification in SIMPLE

- The hash table stores
  - The name of the variable, of type String (the key).
  - A pair made of the variable's type and initial value, stored in an object of class **IdSymbol**.
- In ASTVarDeclaration.java

```
public void identification () {
 if (type == TPLTypes.boolType)
      symtab.put (name, new IdSymbol (type, false));
else
      symtab.put (name, new IdSymbol (type, 0));

}
```

populates the symbol table.

Identification in SIMPLE

- The rest of the AST nodes call identification on their children i.e. Tree traversal.
- With the exception of ASTId.java:

```
 public void identification () {
if (!(symtab.containsKey(name))) {         /* Insert if not declared. */
        System.out.println("TPL Identification: "+name+" not declared.");
        symtab.put (name, new IdSymbol (TPLTypes.intType, 0));
}  }
```

which checks if a variable has been declared by looking up the symbol table. If not, a warning message is displayed and the symbol table is updated with the defaults (type int and value 0).

Type checking

- traverse AST from bottom up inferring type of each expression
  - check that an inferred type is of expected type
    - eg boolean in if statement condition
    - eg an actual parameter matches a formal parameter
  - check that two inferred types conform (not necessarily equal)
    - eg equality operator is valid for both booleans and ints
    - eg may be possible to assign an int to a float
    - Eg subtyping, inheritance
- for SIMPLE
  - traverse AST by writing typecheck() method for each node type, initially called on rootNode()
  - this method stores the inferred type in each node
  - checks the inferred type for children's nodes against expected types, issuing appropriate error messages

Typechecking of SIMPLE programs

- Each node will contain a type and a method that implements type checking. In SimpleNode.java:

```
 public void typecheck ()  { /* Error if this is executed */
     System.out.println("TPL Typechecker: Simple Node version panic");
 }
public  int  NodeType;  // the type assigned to the node
public  int  GetNodeType () {   return NodeType;  }
```

where NodeType can be any of the four values declared in TPLTypes.java:

```
 public static final int  intType = 0, boolType = 1,
                          decType =2,     stmType = 3;
```

Typechecking of SIMPLE programs

- In ASTIntConst.java.java

```
public void typecheck () {
        NodeType = TPLTypes.intType;
}
```

  the type of an ASTIntConst is intType!!

- In ASTTrue.java.java

```
        NodeType = TPLTypes.boolType;  // yes, a boolean
```

- In ASTId.java:

```
        NodeType = ((IdSymbol)symtab.get(name)).type;
```

  the typechecker looks up the type of the variable in the symbol table.

Typechecking of SIMPLE programs

- In ASTAssignStatement:

```
public void typecheck () {
  jjtGetChild(0).typecheck();    // typecheck identifier
  jjtGetChild(1).typecheck();     // typecheck expression

  if (jjtGetChild(0).GetNodeType() == TPLTypes.intType) {
     if (jjtGetChild(1).GetNodeType() != TPLTypes.intType)
        System.out.println("TPL Typechecker: assign of non-int exp
                                              to int   var.");    }
  else if (jjtGetChild(0).GetNodeType() == TPLTypes.boolType) {
     if (jjtGetChild(1).GetNodeType() != TPLTypes.boolType)
        System.out.println("TPL Typechecker: assign of non-bool exp
        to bool var.");
 }
NodeType = TPLTypes.stmType;
}
```

Rule: Typecheck children nodes and make sure that the identifier and expression (right hand side) have the same type

Typechecking of SIMPLE programs

- In ASTWhileStatement.java

```
public void typecheck () {
jjtGetChild(0).typecheck();    // typecheck conditional
jjtGetChild(1).typecheck();    // typecheck body

if (jjtGetChild(0).GetNodeType() != TPLTypes.boolType)
        System.out.println("TPL Typechecker: while statement
                                    condition non-bool");

NodeType = TPLTypes.stmType;
}
```

Rule: Typecheck all children and make sure that the type of the conditional expression is boolType.

Task: Figure out the typechecking rules of SIMPLE by looking at the rest of the code (given as part of the lab/coursework)

Implementation of SIMPLE

- We want to write a program that executes SIMPLE programs. This can be done by writing:
  - An Interpreter: Parses the SIMPLE program and executes each of its nodes by traversing the AST.
  - A compiler: Parses a SIMPLE program and generates code that can be executed separately e.g. Machine code.
- We will implement a SIMPLE interpreter: a program that takes as input a file/string that encodes a SIMPLE program, and executes it.
- Recall the SIMPLE code extract shown previously. Our interpreter should be able to read that program, typecheck it, ask for a number and write its factorial.
- The interpreter should do this for any SIMPLE program!

Implementation - Interpretation

- traverse the AST, interpreting each node according to pre-defined evaluation rules (SIMPLE semantics)
- need a state vector to track the values of variables
- declarations initialise state's variables
- expressions must be evaluated bottom-up
- statements have structure
  - eg the while statement node must be continually reinterpreted until the Expression interprets to false
- for SIMPLE
  - the state vector is the symbol table (has place for values)
  - we use a stack for expression evaluation

Interpretation rules: Summary

CU	$\rightarrow$ D ; S	[interpret D then interpret S with D's names]
D	$\rightarrow D_1 ; D_2$	[interpret D_1 then interpret D_2]
	[interpreting D_n means creating a state (name,type,value) entry]	
S	$\rightarrow$ I = E	[eval E, store in state for id I]
	| $S_1 ; S_2$	[interpret S_1 then interpret S_2]
	| if E { S_1 }	[eval E interpret S_1 if true, else do nothing]
	| if E { S_1 } else {S_2}	[eval E interpret S_1 if true, else interpret S_2]
	| while E {S}	[eval E interpret S if true, and repeat, else do nothing]
	| skip	[do nothing]
E	$\rightarrow$ N	[value is the n constant]
	| B	[value is the b constant]
	| ID	[value is value of ID from the state (stable)]
	| $E_1 + E_2$	[eval E_1 eval E_2 add]
	| $E_1 = E_2$	[eval E_1 eval E_2 equal]
	| $E_1 <= E_2$	[eval E_1 eval E_2 less than or equal]
	| E_1 and E_2	[eval E_1 if false value is false else eval E_2 value is E_2]

Interpreting SIMPLE ASTs

- ASTVarDeclaration.java:

```
public class ASTVarDeclaration extends SimpleNode { ...
 public void interpret () {
   // do nothing – symbol table is initialised by identification method
}
```

- ASTAssignStatement:

```
public class ASTAAssignStatement extends SimpleNode { …
  public void interpret() {  /* Overwrite the symbol value appropriately. */
     IdSymbol i;
      jtGetChild(1).interpret();   // The right-hand side Expression

     i = (IdSymbol)symtab.get(((ASTId)jjtGetChild(0)).name);
     if (i.type == TPLTypes.intType)
        i.intvalue = ((Integer)stack.pop()).intValue();
     else
      i.boolvalue = ((Boolean)stack.pop()).booleanValue();  }
}
```

The method replaces the old value stored in the symbol table with the new computed value (left on top of the stack by interpreter)

Interpreting SIMPLE ASTs

- If statement node example:

```
public class ASTIfStatement extends SimpleNode { ...
 public void interpret () {
   jjtGetChild(0).interpret();

   if (((Boolean)stack.pop()).booleanValue())
     jjtGetChild(1).interpret();
   else if (jjtGetNumChildren() == 3)
     jjtGetChild(2).interpret();
}
```

- Add node example:

```
public class ASTAdd extends SimpleNode { …
 public void interpret () {
   int left, right;
   jjtGetChild(0).interpret();
     jjtGetChild(1).interpret();

   right = ((Integer)stack.pop()).intValue();
   left = ((Integer)stack.pop()).intValue();
   stack.push(new Integer(left+right));
  }
```

Putting all together

- The Java cide below executes the two analyses and interpreter after parsing a valid SIMPLE program:

```
parser.CompilationUnit();
    ((SimpleNode)parser.jjtree.rootNode()).dump("”);
    parser.jjtree.rootNode().identification();
    parser.jjtree.rootNode().typecheck();
    parser.jjtree.rootNode().interpret();
```