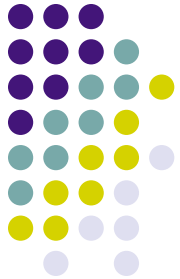
Tree Pattern Matching

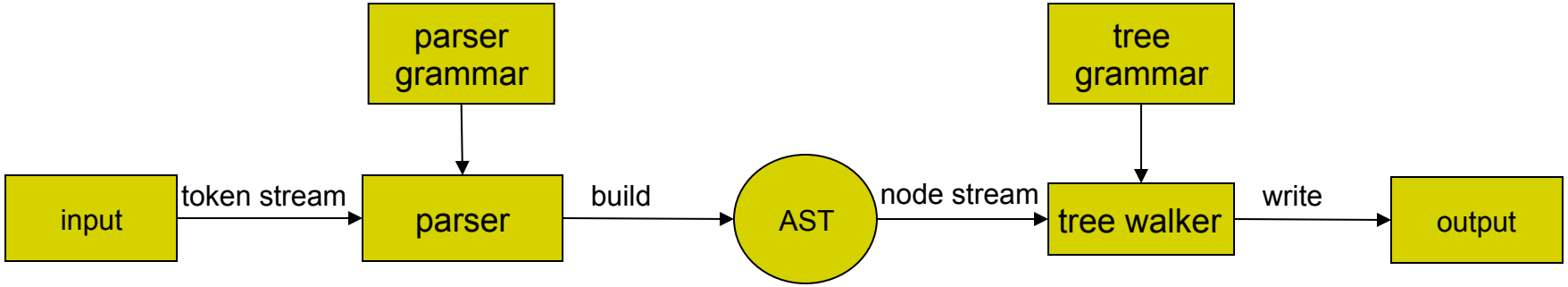


- As we have seen, tree walking and tree transformation are at the core of most language applications
- It is therefore no surprise that ANTLR provides us with a tool to create tree walkers and tree transformers without having to write explicit visitors ☞ Tree Grammars
- All this is based on ANTLR's internal AST representation

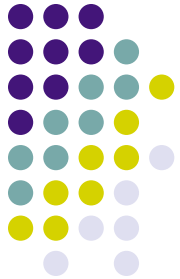
Read chapter 7 & 8 in the reference book

Tree Pattern Matching

- Tree grammars work analogously to parser generator grammars except that instead of working on a token stream the generated code works on a node stream that is created from an AST.
- Language applications written in this style have the following architecture



Building ANTLR ASTs



- Before getting into pattern matching we need to understand how ANTLR generates its internal AST
- ANTLR uses an abstract notation for its AST, e.g.

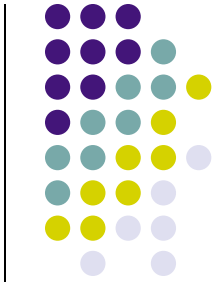
  ^(‘+’ $e1 $e2)

  represents an expression tree.
- another example,

  ^(‘if’ $e $s1 $s2)

  represents an if-then-else statement.

Building ANTLR ASTs



- Here is another look at the pretty printer that also folds constants
- Here the reader is implemented completely using ANTLR's AST

The Parser Grammar

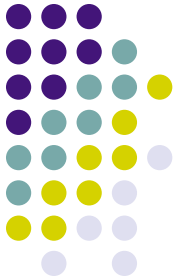
```
grammar simple1;

//*************************************************************
// variable definition usage analyzer and pretty printer
// for the simple1 scripting language
// this version uses ANTLR's AST mechanism

options{
output=AST;
ASTLabelType=CommonTree;
}

// we define some additional tokens not defined
// in the lexer that we need for tree building
tokens{STMTLIST;BLOCKSTMT;}
```

The Parser Grammar

```
prog    :   (stmt)+                                   -> ^(STMTLIST stmt+)
        ;

stmt    :   VAR '=' exp ';'?                          -> ^('=' VAR exp)
        |   'get' VAR ';'?                            -> ^('get' VAR)
        |   'put' exp ';'?                            -> ^('put' exp)
        |   'while' '(' exp ')' s=stmt                -> ^('while' exp $s)
        |   'if' '(' exp ')' s1=stmt ( 'else' s2=stmt )?  -> ^('if' exp $s1 $s2?)
        |   '{' (stmt)+ '}'                           -> ^(BLOCKSTMT stmt+)
        ;

exp     :   relexp
        ;

relexp  :   addexp (('==' ^ addexp) | ('<=' ^ addexp))* ;
addexp  :   mulexp (('+' ^ mulexp) | ('-' ^ mulexp))*  ;
mulexp  :   atom (('*' ^ atom) | ('/' ^ atom))* ;

atom    :   '(' exp ')'          -> exp
            // default tree VAR -> ^(VAR)
        |   VAR                  {PPrint.usageMap.put($VAR.text,new Boolean(true)); }
            // default tree NUM -> ^(NUM)
        |   NUM
        |   '-'v=NUM             -> ^(NUM["-"+$v.text])
        ;
```

Note: compare this to the explicit AST we built in lecture notes for Program Analysis and AST (Part 2).

The Parser Grammar



- The notation

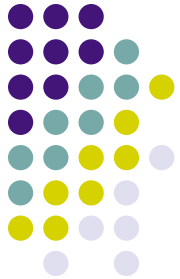
  addexp ('==' ^ addexp)*

  means
  - either return tree addexp
  - or return a tree ^(‘==‘ $a1 $a2)
  - or return a tree ^(‘==‘ ^(‘==‘ $a1 $a2) $a3)
  - etc.
- The notation

  ^(NUM[”3”])

  means build a node of type NUM with the value 3.

Tree Pattern Matching



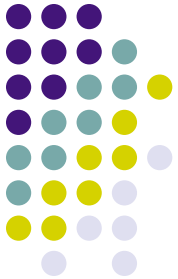
- A *tree pattern* is usually a snippet of (abstract syntax) tree that has to match some object tree exactly or may contain *wild cards*.
- In ANTLR's built-in AST the following is a pattern that matches the addition of two numbers:

  ^('+' NUM NUM)

- The next pattern uses a wild card for the right operand

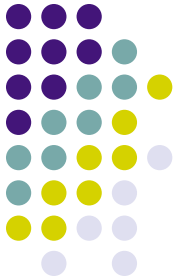
  ^('+' NUM .)

The Pretty Printer



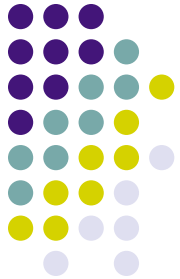
- We implement the Pretty Printer as a tree walker using a tree grammar.
- In a tree walker we have to supply an *exhaustive* set of tree patterns, that is we have to provide a pattern for each kind of AST node.

The Pretty Printer



```
tree grammar gen;

//*************************************************************
// this is the code generation back end for our pretty printer

options{
tokenVocab=simple1;               // use token definitions from simple1
ASTLabelType=CommonTree;   // use CommonTree AST
}

@members{
// indentation function to make the output look nice
public void indent(int ix) {
                    for (int i = 0; i < ix; i++)
                                        System.out.print("     ");
}
}
```

The Pretty Printer



- Tree grammars have the following structure:

<pattern name> : <pattern 1> | <pattern 2> | …| <pattern N>

```
prog    :    ^(STMTLIST stmt+) ;

stmt    :    ^('=' VAR  exp)
        |    ^('get' VAR)
        |    ^('put' exp)
        |    ^('while'  exp stmt)
        |    ^('if' exp stmt stmt?)
        |    ^(BLOCKSTMT stmt+)
        ;
```

```
prog        :       ^(STMTLIST stmt[0]+) ;

stmt [int ix]
            :       ^('=' VAR
                                { indent(ix);
                                  System.out.print($VAR.text+" = ");}
                      exp)
                                { if (PPrint.usageMap.get($VAR.text) == null)
                                        System.out.print("  // -- not used -- ");
                                  System.out.println(); }
            |       ^('get' VAR)
                                { indent(ix);
                                  System.out.print("get "+$VAR.text);
                                  if (PPrint.usageMap.get($VAR.text) == null)
                                        System.out.print("  // -- not used --");
                                  System.out.println(); }
            |       ^('put'
                                { indent(ix); System.out.print("put "); }
                      exp)
                                { System.out.println(); }
            |       ^('while'
                                { indent(ix); System.out.print("while ( "); }
                      exp
                                { System.out.println(") "); }
                      stmt[ix+1])
                                { System.out.println(); }
            |       ^('if'
                                { indent(ix); System.out.print("if ( "); }
                      exp
                                { System.out.println(") "); }
                      stmt[ix+1]
                      ({ indent(ix); System.out.println("else"); } stmt[ix+1])?)
                                { System.out.println(); }
            |       ^(BLOCKSTMT
                                { ix=ix>0?ix-1:0; indent(ix); System.out.println("{"); }
                      stmt[ix+1]+)
                                { indent(ix); System.out.println("}"); }
            ;
```

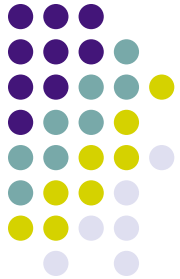
-the pretty printer tree grammar with the actions filled in.

-actions can appear anywhere in the pattern matching code (see the else statement).

-we have a new kind of argument to pattern names; we can send values to patterns, e.g.,

stmt [int ix]

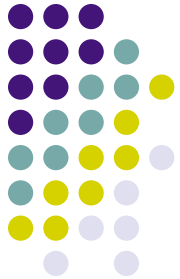
The Pretty Printer



- Expressions are particularly interesting
  - We have matching in a tree now
  - Therefore, no precedence or associativity headaches!

```
exp     :       ^('==' exp {System.out.print("== ");} exp)
        |       ^('<=' exp {System.out.print("<= ");} exp)
        |       ^('+'  exp {System.out.print("+ ");} exp)
        |       ^('-'  exp {System.out.print("- ");} exp)
        |       ^('*'  exp {System.out.print("* ");} exp)
        |       ^('/'  exp {System.out.print("/ ");} exp)
        |       VAR {System.out.print($VAR.text+" ");}
        |       NUM {System.out.print($NUM.text+" ");}
        ;
```

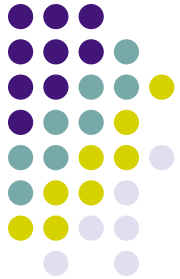
The Pretty Printer



```
// open up the input file
ANTLRFileStream input = new ANTLRFileStream(args[0]);
// create the lexer with the input stream
simple1Lexer lexer = new simple1Lexer(input);
// create a token stream for the parser
CommonTokenStream tokens = new CommonTokenStream(lexer);
// create a parser object with the token stream
simple1Parser parser = new simple1Parser(tokens);

// call the toplevel recursive descent parsing function to construct
// our IR
CommonTree ast = (CommonTree)parser.prog().getTree();

// Dump out the AST representation
System.out.println("Parse Tree: "+ast.toStringTree());

// tree node stream for pretty printer
CommonTreeNodeStream nodes = new CommonTreeNodeStream(ast);
// create new tree parser
gen generator = new gen(nodes);
// start pattern matching at toplevel name
generator.prog();
```

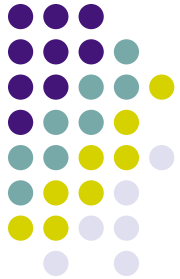
Top-level Driver

Tree Transformation



- The difference between tree walkers and tree transformers is that
  - Tree walkers need to see/know about every possible node in the tree
  - Tree transformers are only interested in the part of the tree that they can rewrite
- Tree transformers are implemented as *filters* in ANTLR
  - Filters watch the node stream and only take action when they see a node that they want to transform/rewrite

Constant Folding Revisited



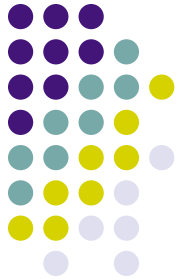
- Here we rewrite the constant folder from the Pretty Printer we looked at in Tree Walking (Part 2)
- For the most part the constant folder will work just like the pretty printer except that we do not walk the tree but watch for the nodes that interest us

Constant Folding Revisited

```
tree grammar fold;

//************************************************************************
// constant folder for the simple1 scripting language

options{
tokenVocab=simple1;                 // use token definitions from simple1
output=AST;                         // generate AST
ASTLabelType=CommonTree;            // use CommonTree AST
filter=true;                        // make it a tree rewriter
}

bottomup:      exp;  // make sure the children are folded first - postfix  node stream

exp            :               ^('==' i=NUM j=NUM)      -> ^(NUM[String.valueOf(($i.int==$j.int)?1:0)])
               |               ^('<=' i=NUM j=NUM)      -> ^(NUM[String.valueOf(($i.int<=$j.int)?1:0)])
               |               ^('+'  i=NUM j=NUM)      -> ^(NUM[String.valueOf($i.int+$j.int)])
               |               ^('-'  i=NUM j=NUM)      -> ^(NUM[String.valueOf($i.int-$j.int)])
               |               ^('*'  i=NUM j=NUM)      -> ^(NUM[String.valueOf($i.int*$j.int)])
               |               ^('/'  i=NUM j=NUM)      -> ^(NUM[String.valueOf($i.int/$j.int)])
               ;
```

That's it!

Constant Folding Revisited



- The last thing we have to do is to set up the node stream for the tree rewriter
- Since this is a filter there is no start symbol, we will call the system start symbol called ‘downup’

```
// tree node stream for tree parser
CommonTreeNodeStream nodes = new CommonTreeNodeStream(inast);
// create constant folder object
fold folder = new fold(nodes);
CommonTree outast = (CommonTree)folder.downup(inast, true);
System.out.println("Transformed Tree: "+outast.toStringTree());
```

Code available online

- SIMPLE1PATTERNS.zip