Abstract Syntax Trees

COMS W4115

Prof. Stephen A. Edwards
Fall 2003
Columbia University
Department of Computer Science

Parsing and Syntax Trees

Parsing decides if the program is part of the language.
Not that useful: we want more than a yes/no answer.
Like most, ANTLR parsers can include actions: pieces of code that run when a rule is matched.
Top-down parsers: actions executed during parsing rules.
Bottom-up parsers: actions executed when rule is "reduced."

Actions

Simple languages can be interpreted with parser actions.

class CalcParser extends Parser;

expr returns [int r] { int a; r=0; } : r=mexpr ("+" a=mexpr ( r += a; ) )* EOF ;
mexpr returns [int r] { int a; r=0; } : r=atom ("*" a=atom ( r *= a; ) ) * ;
atom returns [int r] { r=0; } : i:INT { r = Integer.parseInt(i.getText()); };

Implementing Actions

In a top-down parser, actions are executed during the matching routines.
Actions can appear anywhere within a rule: before, during, or after a match.
rule { /* before */ } : A { /* during */ } B | C D { /* after */ } ;
Bottom-up parsers restricted to running actions only after a rule has matched.

Implementing Actions

Nice thing about top-down parsing: grammar is essentially imperative.
Action code simply interleaved with rule-matching.
Easy to understand what happens when.

expr returns [int r] { int a; r=0; } : r=mexpr ("+" a=mexpr ( r += a; ) )* EOF ;
public final int expr() { // What ANTLR builds
  int r; int a; r=0;
r=mexpr();
while (((LA(1)==PLUS)) { // ( )*
  match(PLUS); // "+
  a=mexpr(); // a=mexpr
  r += a; // { r += a; }
  match(Token.EOF_TYPE);
return r;
}

What To Build?

Typically, an Abstract Syntax Tree that represents the program.
Represents the syntax of the program almost exactly, but easier for later passes to deal with.
Punctuation, whitespace, other irrelevant details omitted.

Actions

Bottom-up parsers can only build bottom-up data structures.
Children known first, parents later.
→ Constructor for any object can require knowledge of children, but not of parent.
Context of an object only established later.
Top-down parsers can build both kinds of data structures.

Actions

In a top-down parser, actions are executed during the matching routines.
Actions can appear anywhere within a rule: before, during, or after a match.
rule { /* before */ } : A { /* during */ } B | C D { /* after */ } ;
Bottom-up parsers restricted to running actions only after a rule has matched.

Implementing Actions

In a top-down parser, actions are executed during the matching routines.
Actions can appear anywhere within a rule: before, during, or after a match.
rule { /* before */ } : A { /* during */ } B | C D { /* after */ } ;
Bottom-up parsers restricted to running actions only after a rule has matched.

What To Build?

Typically, an Abstract Syntax Tree that represents the program.
Represents the syntax of the program almost exactly, but easier for later passes to deal with.
Punctuation, whitespace, other irrelevant details omitted.
Abstract vs. Concrete Trees

Like scanning and parsing, objective is to discard irrelevant details.

E.g., comma-separated lists are nice syntactically, but later stages probably just want lists.

AST structure almost a direct translation of the grammar.

Example of AST structure

Implementing ASTs

Most general implementation: ASTs are \( n \)-ary trees.

Each node holds a token and pointers to its first child and next sibling:

![Diagram of AST structure]

Example of AST structure

Typical AST Operations

Create a new node; Append a subtree as a child.

Heterogeneous ASTs

Advantage: avoid switch statements when walking tree.
Disadvantage: each analysis requires another method.

```
class BinOp {
    int operator; Expr left, right;
    void typeCheck() { ... };
    void constantProp() { ... };
    void buildThreeAddr() { ... };
};
```

Analyses spread out across class files.
Classes become littered with analysis code, additional annotations.

Comment on Generic ASTs

ANTLR offers a compromise:
It can automatically generate tree-walking code.

→ It generates the big switch statement.
Each analysis can have its own file.
Still have to modify each analysis if the AST changes.
→ Choose the AST structure carefully.

Building ASTs
The Obvious Way to Build ASTs

```java
class ASTNode {
    ASTNode( Token t ) { ... }
    void appendChild( ASTNode c ) { ... }
    void appendSibling( ASTNode c ) { ... }
}
stmt returns [ASTNode n]
| : 'if' p=expr 'then' t=stmt 'else' e=stmt |
| { n = new ASTNode(new Token("IF")); |
| n.appendChild(p); |
| n.appendChild(t); |
| n.appendChild(e); } ;
```

The Obvious Way

Putting code in actions that builds ASTs is traditional and works just fine.
But it's tedious.
Fortunately, ANTLR can automate this process.

Automatic AST Construction

Running

```java
class CalcParser extends Parser;
options { buildAST=true; }
expr : mexpr ('+' mexpr)* EOF ;
mexpr : atom ('*' atom)* ;
atom : INT ;
on
2*3 + 4*5 + 6 gives
```

AST Construction with Annotations

Running

```java
class CalcParser extends Parser;
options { buildAST=true; }
expr : mexpr ('+' mexpr)* EOF !;
mexpr : atom ('*' atom)* ;
atom : INT ;
on
2*3 + 4*5 + 6 gives
```

Choosing AST Structure

Sequences of Things

Comma-separated lists are common
```java
int gcd(int a, int b, int c)
args : "(" ( arg ("," arg)* )? ")" ;
```
A concrete parse tree:
```
```
Drawbacks:
Many unnecessary nodes
Branching suggests recursion
Harder for later routines to get the data they want

Sequences of Things

Better to choose a simpler structure for the tree.
Punctuation irrelevant; build a simple list.
```java
int gcd(int a, int b, int c)
args : "(! ( arg ("," arg)* )? ")!" |
{ #args = #([ARGS], args); } ;
```
What's going on here?

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

Rule generates a sequence of arg nodes.
Node generation suppressed for punctuation (parenth, commas).
Action uses ANTLR's terse syntax for building trees.

```
{ #args = #([ARGS], args); } ;
```

"set the args tree to a new tree whose root is a node of type ARGS and whose child is the old args tree"

---

Removing Unnecessary Punctuation

Punctuation makes the syntax readable, unambiguous.
Information represented by structure of the AST
Things typically omitted from an AST

- Parentheses
  - Grouping and precedence/associativity overrides
- Separators (commas, semicolons)
  - Mark divisions between phrases
- Extra keywords
  - while-do, if-then-else (one is enough)

---

What's going on here?

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

Additional Grouping

The Tiger language from Appel's book allows mutually recursive definitions only in uninterrupted sequences:

```
let
  function f1() = ( f2() ) /" OK"/
  function f2() = ( ... )
in ... end

let
  function f1() = ( f2() ) /" Error"/
  var foo := 42 /" splits group"/
  function f2() = ( ... )
in ... end
```

Grouping

Convenient to group sequences of definitions in the AST. Simplifies later static semantic checks.

```
let
  function f1() = ( ... )
  function f2() = ( ... )
  var foo := 42
in ... end
```

```
defs : ( funcs | vars | types )* ;
  funcs : ( func )+ ;
  vars : ( var )+ ;
  types : ( type )+ ;
```

are ambiguous: Maximum-length sequences or minimum-length sequences?

How Many Types of Tokens?

Since each token is a type plus some text, there is some choice.
Generally, want each "different" construct to have a different token type.
Different types make sense when each needs different analysis.
Arithmetic operators usually not that different.
For the assignment, you need to build a node of type "BINOP" for every binary operator. The text indicates the actual operator.

---

Grouping

Identifying and building sequences of definitions a little tricky in ANTLR.

Obvious rules

```
defs : ( funcs | vars | types )* ;
  funcs : ( func )+ ;
  vars : ( var )+ ;
  types : ( type )+ ;
```

The Greedy Option

Setting greedy true makes "dots" as long as possible

```
string : (dots)* ;
dots : ( options greedy=true; : "." )+ ;
```

Setting greedy false makes each "dots" a single period

```
string : (dots)* ;
dots : ( options greedy=false; : "." )+ ;
```

---

How Many Types of Tokens?

Since each token is a type plus some text, there is some choice.
Generally, want each "different" construct to have a different token type.
Different types make sense when each needs different analysis.
Arithmetic operators usually not that different.
For the assignment, you need to build a node of type "BINOP" for every binary operator. The text indicates the actual operator.
Walking ASTs

Walking ASTs with ANTLR

ANTLR can build "tree parsers" as easily as token parsers.

Much simpler: tree structure is already resolved.

Simple recursive walk on the tree.

Matches are sufficient, not exact.

(Cheaper to implement.)

#( A B ) also matches the larger tree

#( A #(B C) D )

Walking ASTs with ANTLR

class CalcParser extends Parser

expr : mexpr ("+"ˆ mexpr)* ;

mexpr : atom ("*"ˆ atom)* ;

atom : INT | "(" expr ")" ;

class CalcWalker extends TreeParser

eexpr 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 = parseInt(i.getText()); }

;

This walker only has one rule: grammar had three.

Fine: only structure of tree matters.

Comments on walking ASTs

Tree grammars may seem to be ambiguous.

Does not matter: tree structure already known

Unlike proper parsers, tree parsers have only one token of lookahead.

Must be possible to make a decision locally.

Has impact on choice of AST structure.

Optional clauses can cause trouble.

Place them at the end.

stmt

: #("if" expr stmt (stmt)?) // OK

| #("do" (stmt)? expr) // Bad

;

First rule works: can easily decide if there is another child.

Second rule does not: not enough lookahead.

Rewriting Trees with ANTLR

Rewriting Trees with ANTLR

funcdef

: #("func" ID (arg)* stmt)

;

Does not work because the tree walker does not look ahead.

Solution: use a subtree

funcdef

: #("func" #("args" (arg)*) stmt)

;

The placeholder resolves the problem.
Rewriting Trees

Much of compiling is refining and simplifying:
- Discarding unnecessary information
- Reducing high-level things to low-level ones
How to implement this depends on the representation.
Trees are fairly natural: replace one or more children.
ANTLR tree walkers can do semi-automatically.

Rewriting Trees with ANTLR

In the parser, `buildAST=true` adds rules that automatically builds an AST during parsing.
In a tree walker, `buildAST=true` adds code that automatically makes a copy of the input tree.
This is actually useful because you can selectively disable it and generate your own trees.

Rewriting Trees with ANTLR

An example: Replace `x+0` with `x`.
First, make a copying TreeParser:
```java
class FoldZeros extends TreeParser;
options {
  buildAST = true;
}
```
```java
eq ::= (#("+" expr expr)
         | (#("*" expr expr)
         | INT
```

Rewriting Trees with ANTLR

Next, disable automatic rewriting for the `+` operator and add a manual copy.
Adding `!` before a subrule disables AST generation for that subrule.
Tree generation is like that in parsers.
```java
eq ::= ! (#(PLUS left:expr right:expr)
         { #eq = #(PLUS, left, right); })
      | #(STAR expr expr)
      | i:INT
```

Examples of Tree Rewriting

This was incomplete: should do `0+x` case, too.
General constant folding: replace constant arithmetic expressions with their results.
Must do this carefully: watch for overflow, imprecision.
Tricky to do correctly for integers, virtually impossible for floating-point.
Cross-compilation problem: how do you know the floating-point unit on your target machine behaves exactly like the one where you’re compiling?

Examples of Tree Rewriting

Change logical operators `&&` and `||` to if-then statements.
```java
if (a && b && c || d && e) { ... }
if (a) {
  if (b)
    if (c) goto Body;
  else if (d)
    if (e) { Body: ... }
}
```

Examples of Tree Rewriting

Dismantle loops into gotos.
```java
while (a < 3) {
  printf("a is %d", a);
  a++;
}
```
Becomes
```java
goto Continue;
Again:
  printf("a is %d", a);
  a++;
Continue:
  if (a < 3) goto Again;
```