Abstract Syntax Trees
COMS W4115

Prof. Stephen A. Edwards
Fall 2007
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

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;
    }
    return r;
}

Actions

Usually, actions build a data structure that represents the
program.
Separates parsing from translation.
Makes modification easier by minimizing interactions.
Allows parts of the program to be analyzed in different
orders.

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.

Implementing ASTs

Most general implementation: ASTs are n-ary trees. Each node holds a token and pointers to its first child and next sibling:

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.

Comment on Generic ASTs

Is this general-purpose structure too general? Not very object-oriented: whole program represented with one type. Alternative: Heterogeneous ASTs: one class per object.

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.

Comment on Generic ASTs

Building ASTs
The Obvious Way to Build ASTs

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.

Building an AST Automatically with ANTLR

class TigerParser extends Parser;
options {
    buildAST=true;
}

By default, each matched token becomes an AST node.
Each matched token or rule is made a sibling of the AST for the rule.
After a token, "$" makes the node a root of a subtree.
After a token, "!" prevents an AST node from being built.

Automatic AST Construction

Running

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

gives
2*3+4*5+6

AST Construction with Annotations

Running

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

gives
2*3+4*5+6

Choosing AST Structure

Designing an AST Structure

Sequences of things
Removing unnecessary punctuation
Additional grouping
How many token types?

Sequences of Things

Comma-separated lists are common
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.
int gcd(int a, int b, int c)
args : "( ! ( arg (","! arg) )? " )\" !
    { #args = #((ARGS), args); } ;

ARGs
arg arg arg
int a int b int c
What's going on here?

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

Rule generates a sequence of arg nodes. Node generation suppressed for punctuation (parens, commas).

Action uses ANTLR's terse syntax for building trees.

```lang
{ #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"

---

Additional Grouping

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

```lang
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.

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

---

The Greedy Option

Setting greedy true makes "dots" as long as possible

```lang
string : (dots)* ;
dots : ("."+ ;
```

Setting greedy false makes each "dots" a single period

```lang
string : (dots)* ;
dots : ("."+ ;
```

---

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)

---

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 )

These examples are for ANTLR 2.0

Walking ASTs with ANTLR

```java
class CalcParser extends Parser
    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 = parseInt(i.getText()); }
;
```

This walker only has one rule: grammar had three.
Fine: only structure of tree matters.

Comments on walking ASTs

Optional clauses can cause trouble.
Place them at the end.

```java
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.

Comments on walking ASTs

Lists of undefined length can also cause trouble

```java
funcdef
    : #("func" ID (arg)* stmt)
    ;
```

Does not work because the tree walker does not look ahead.
Solution: use a subtree

```java
funcdef
    : #("func" #("args" (arg)*) stmt)
    ;
```

The placeholder resolves the problem.