ANTLR #2

User-defined ASTs & Error Handling

University of Twente
The Netherlands

Vertalerbouw **HC 9**

VB HC9 http://fmt.cs.utwente.nl/courses/vertalerbouw/

Michael Weber
kamer: INF 5037
telefoon: 3716
email: michaelw@cs.utwente.nl

Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools

Overview of Lecture 9

- Homework Assignment
- Final Project (Compiler)
- Advanced ANTLR 3
  - user-defined ASTs
  - error handling
  - predicates
  - string templates

Mededelingen

- Practicum - eerste deel (wk 1-6)
  - stricte deadline voor aftekenen: dinsdag 2 juni 2009 aan begin van practicum
  - eindopdracht alleen mogelijke als op dinsdag 2 juni alle opdrachten van de eerste vijf weken afgetekend zijn
- Practicum - tweede deel (wk 7-9): eindopdracht
  - niet langer verplicht
  - assistentie:
    - dinsdag 5-8: alle studentassistenten
    - woensdag 3+4: wel verroosterd, geen assistentie
  - in de weken 10, 11 en 12 geen assistentie
  - deadline verslag: maandag 13 juli 2009 (een week na tentamens)

resultaten zijn pas medio/eind augustus 2009 beschikbaar

Eindopdracht

Zie de handleiding voor een beschrijving van de opdracht.

- Eindopdracht
  - eigen programmeertaal ontwerpen
  - vertaler schrijven voor deze programmeertaal
  - verslaglegging
- Beoordeling
  - basiscijfer hangt af van
    - taalfeatures van de programmeertaal
    - keuze van doelarchitectuur: TAM, JVM of .NET
  - bonus/malus op grond van
    - kwaliteit van de programmatuur
    - kwaliteit van verslag
    - kwaliteit van tests

Voor de eindopdracht dient gebruik te worden gemaakt van ANTLR 3.2. Het is niet toegestaan om ANTLR versie 2.x te gebruiken.

Globale planning

Eindopdracht is 'begroot' op zo'n 50 uur (per student).

wk 7 (23)	definitie van taal: syntax, context en semantiek testprogramma's schrijven scanner specificatie in ANTLR parser specificatie in ANTLR
wk 8 (24)	testen: syntax van voorbeeldprogramma's symbol table klasse ontwikkelen context checker: treeparser in ANTLR testen: contexteisen van voorbeeldprogramma's
wk 9 (25)	code generatie: treeparser in ANTLR testen: gegenereerde code van voorbeeldprogramma's
wk 10 (26)	verslaglegging
wk 11 (27) wk 12 (28)	eventuele uitloop — deadline verslag: maandag 13 juli 2009

Verslageisen

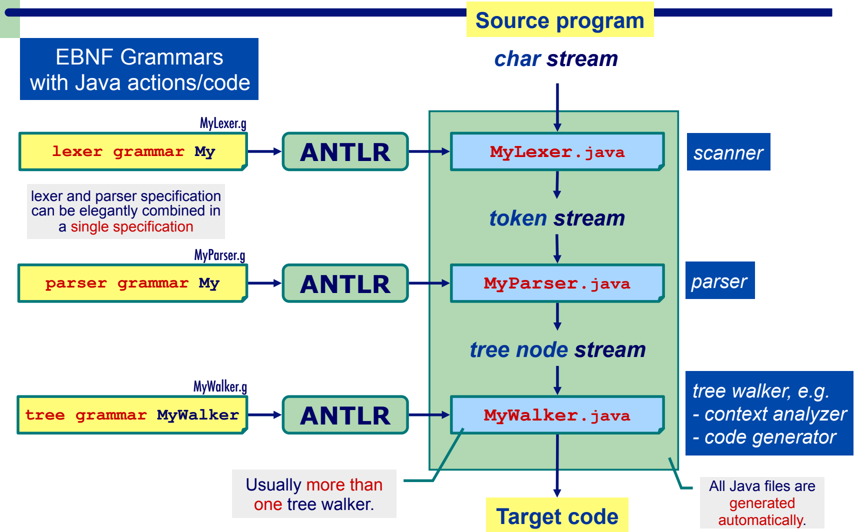
- Verslag
  - Inleiding
  - Beknopte beschrijving van taal
  - Problemen en oplossingen
  - (Formele) specificatie van de taal
    - syntax
    - contextbeperkingen
    - semantiek: vertaalregels
  - Beschrijving van extra programmatuur (bv. symbol table)
  - Testplan
  - Conclusies
- Appendices
  - ANTLR listings: scanner, parsers en alle treewalkers
  - Invoer- en uitvoer van uitgebreid testprogramma

Aandachtspunten

- Taalkeuze en features
  - Zoveel mogelijk houden aan beschrijving van features.
  - Afwijken van de eisen kost altijd punten.
    - voorbeeld: statements en declaraties *niet* door elkaar
- Testen
  - Testen van vertaler is heel belangrijk.
    - correcte programma's
    - incorrecte programma's
  - Veel en uitgebreide testprogramma's bijleveren op CD-R maar ook beschrijven in verslag (en appendix).
- Verslag
  - Zorg dat alle genoemde onderdelen aanwezig zijn.

ANTLR – overview

HC4



ANTLR - AST building revisited

HC4

- In an ANTLR **Parser** specification it is easy to construct an AST node.

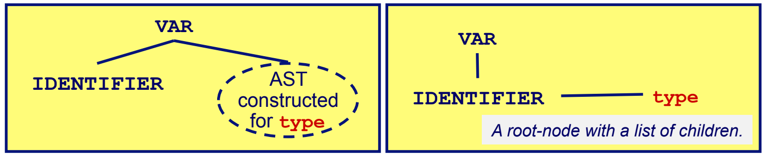
becomes top-node | will be discarded

e.g.
```
declaration : VAR^ IDENTIFIER COLON! type ;
```

or:
```
declaration : VAR IDENTIFIER COLON type
              -> ^(VAR IDENTIFIER type);
```

Notations cannot be mixed!

constructs the following AST | or seen alternatively:



- An ANTLR **tree parser** can now parse this AST pattern:

e.g.
```
declaration : ^(VAR IDENTIFIER type) ;
```

ANTLR uses a prefix pattern language for AST nodes.

User-defined AST Class (1)

- So far (i.e., in the Calc-compiler), we used ANTLR to construct an AST using default AST nodes.
  - For little languages, ANTLR's default AST class **tree.CommonTree** suffices.
  - However, if one needs to store additional information (types, identifier information, memory addresses, etc.) a user-defined AST class has to be defined.
  - With ANTLR it is easy to define your own AST class:

    MyTree extends CommonTree
    user-defined AST node

    MyTreeAdaptor extends CommonTreeAdaptor
    adaptor to create MyTree nodes

develop two classes

User-defined AST Class (2)

- How to use your own AST class in ANTLR:

+ override some specific methods of **CommonTree**.

1. Define a class **MyTree** as subclass of **CommonTree**.
```
public class MyTree extends CommonTree { ...
```
2. Define a class **MyTreeAdaptor** as subclass of **CommonTreeAdaptor** (= a factory class for MyTree nodes).
```
class MyTreeAdaptor extends CommonTreeAdaptor { ...
```
3. Specify in the ANTLR specification of the **tree parser** that **MyTree** class is used for AST nodes.
```
options { ... ASTLabelType = MyTree; }
```
4. Tell the **Parser** that **MyTree** nodes should be constructed:
```
MyParser parser = new MyParser(tokens);
parser.setTreeAdaptor(new MyTreeAdaptor());
```


List – simple List language

- List – explanation by example.
  - The List language defines computations as operations on a list of elements. The elements of such a list can be
    - numbers
    - lists
  - An example of a sentence of the List-language is:
    ```
    +[3, 5, *[2, 5], +[3, 7, +[2, 5], 11], 27, 51]
    ```
  - We define our own AST node to store:
    - for a each (sub)list, the (computed) value of this list
    - furthermore, we only want to retain the toplevel list

All source files (**.g** and **.java**) will be put on the Vertalerbouw-website.

List – ListNode

ListNode is a subclass of ANTLR's default AST class: CommonTree.

```
public class ListNode extends CommonTree {
    protected int value = 0;

    public ListNode()              { super();      }
    public ListNode(Token t)       { super(t);     }

    /** Get the List value of this node. */
    public int getValue()          { return value; }

    /** Set the List value of this node. */
    public void setValue(int value) { this.value = value; }

    public String toString() {
        String s = super.toString();
        try { Integer.parseInt(this.getText()); }
        catch (NumberFormatException ex)
            { s = s + " {=" + getValue() + "}"; }
        return s;
    }
}
```

For the string representation, add the value to non-numeric nodes.

Usual set- and get-methods for the extra instance variable of ListNode.

Warning: do not override CommonTree's getType or getText.

List – ListNodeAdaptor

```
class ListNodeAdaptor extends CommonTreeAdaptor {
    public Object create(Token t) {
        return new ListNode(t);
    }
}
```

The method create is used to build to ListNode objects.

List – lexer and parser

As usual, we only let the parser construct the AST.

```
grammar List;

options {
    k=1;
    language=Java;
    output=AST;
}

tokens {
    ...
}

top         :   list EOF! ;
list        :   operator^ elems ;
elems       :   LBRACKET! elem (COMMA! elem)* RBRACKET! ;
elem        :   NUMBER
            |   list
            ;
operator    :   PLUS
            |   TIMES
            ;
...
```

build an AST

A (List)AST node is created: the operator-TOKEN is the root-node, and the elements of elems are the children.

= "+"

= "*"

Straightforward lexer rules for NUMBER, whitespace and comments have been omitted.

List – ListWalker (1)

Computes the values of List-nodes (i.e. PLUS- or TIMES-nodes) and stores this value in the corresponding ListNode node.

```
list     : operator^ elems ;
elems    : elem+
elem     : NUMBER | list ;
operator : PLUS | TIMES ;
```

```
tree grammar ListWalker;

options { ... ASTLabelType=ListNode; }
@members { /* ... see next slide ... */ }

list        :       { int sum=0; ListNode l=null; }
                ^(p=PLUS
                    { l=(ListNode)input.LT(1); }
                    list
                    { sum += l.getValue(); }
                 )+
                )   { $p.setValue(sum); }

            |   ^(t=TIMES list+)
                    { $t.setValue(product(t)); }

            |   n=NUMBER
                    { $n.setValue(Integer.parseInt($n.text)); }
            ;
```

The alternative for PLUS computes the sum while walking its children (preferred way).

We need to refer to the actual ListNodes of the elements of the sublist.

The alternative for TIMES computes the product after all children have been parsed (see the method product on the next slide).

List – ListWalker (2)

Now add the method **product** as a private method to the class **ListWalker**.

```
list
  :  ...
  | ^(t=TIMES list+)
    { $t.setValue(product(t)); }
  ;
```

```
tree grammar ListWalker;
...
@members {
    ...

    private int product(ListNode root) {
        int prod = 1;
        for (int i=0; i<root.getChildCount(); i++)
            prod *= ((ListNode) root.getChild(i)).getValue();
        return prod;
    }
}
```

Walk the children of a node **root** and computes the product.

Implementations of Tree

Some methods from **CommonTree** and its superclass **BaseTree**.

```
public class BaseTree implements Tree
{
  public int      getChildCount()
  public Tree     getChild(int i)
  public List     getChildren()

  public void     addChild(Tree t)
  public void     addChildren(List kids)
  public void     setChild(int i,Tree t)

  public int      getChildIndex()
  public void     setChildIndex(int ix)
  public Tree     getParent()
  public void     setParent(Tree t)

  public String   toString();
  public String   toStringTree();

  ...
}
```

```
public class CommonTree extends BaseTree
{

  public Token    getToken()
  public Tree     dupNode()
  public boolean  isNil()

  public int      getType()
  public String   getText()
  public int      getLine()

  ...
}
```

- The **BaseTree** is a generic tree implementation with no payload. You must subclass **BaseTree** to actually have any user data.
- A **CommonTree** node is wrapper for a **Token** object.

List – ListTopLevel

This tree walker only retains the List-structure of the top-level AST node. In this top-level List all elements are numbers: inner-Lists are replaced by new **NUMBER** nodes which represent the computed values of the original Lists.

```
tree grammar ListTopLevel;

options {
   tokenVocab=List;
   ASTLabelType=ListNode;
   output=AST;
   rewrite=true;
}

root   :  ^(PLUS  (elem)+)
       |  ^(TIMES (elem)+)
       ;

list   :  ( ^(p=PLUS  (elem)+) -> ^(NUMBER[p.getToken(),""+p.getValue()])
          | ^(t=TIMES (elem)+) -> ^(NUMBER[t.getToken(),""+t.getValue()])
          )
       ;

elem   :   NUMBER
       |   list
       ;
```

output a (new) AST

in-line replacement of nodes

in-line replacement of nodes

List – main

Putting it all together.

```
public static void main(String[] args) {
  ...
  try {
     ListLexer lexer =
         new ListLexer(new ANTLRInputStream(System.in));
     CommonTokenStream tokens = new CommonTokenStream(lexer);
     ListParser parser = new ListParser(tokens);
     parser.setTreeAdaptor(new ListNodeAdaptor());

     ListParser.top_return result = parser.top();
     ListNode tree = (ListNode) result.getTree();

     TreeNodeStream nodes = new CommonTreeNodeStream(tree);
     ListWalker walker = new ListWalker(nodes);
     walker.top();
     ...
     System.out.println(">> Total: " + tree.getValue());

  } catch (RecognitionException e) { ... }
}
```

Make sure that **ListNode** objects are created.

Print the value of the root node.

Heterogeneous AST trees (1)

- Homogeneous trees
  - In standard ANTLR (and our List-examples) all AST nodes had the same type: **`CommonTree`** (or **`ListNode`**).
  - Sometimes it is more appropriate to have several different AST node types. See for instance the W&B approach.
- Brute force (but homogeneous) approach:
  - Homogeneous AST node with a single instance variable **`Map properties`**.
    - In this variable **`properties`** you can store whatever *(key, value)* pair you want.
    - Drawback: not type-safe, difficult to maintain.
- Heterogeneous trees
  - Fortunately, since version 3.1, ANTLR (again) supports the use of heterogeneous AST trees.

http://www.antlr.org/wiki/display/ANTLR3/Tree+construction

Heterogeneous AST trees (2)

```
program         :   declarations statements EOF
                       -> ^(PROGRAM declarations statements)
                ;
declarations    :   (declaration SEMICOLON!)*
                ;
statements      :   (statement SEMICOLON!)+
                ;
declaration     :   VAR^ IDENTIFIER<IdNode> COLON! type
                ;
statement       :   assignment
                |   print
                ;
assignment      :   lvalue BECOMES^ expr

                ;
print           :   PRINT^ LPAREN! expr RPAREN!
                ;
lvalue          :   IDENTIFIER<IdNode>
                ;
expr            :   operand (( PLUS<BinExprNode>^
                             | MINUS<BinExprNode>^) operand)*
                ;
operand         :   IDENTIFIER<IdNode>
                |   NUMBER
                |   LPAREN! expr RPAREN!
                ;
type            :   INTEGER<TypeNode>
                ;
```

With the `<...>` suffix annotation, one can specify the node type of a node.

In this example for Calc, there are three extra node types:
`IdNode`
`BinExprNode`
`TypeNode`
All these classes have to be defined as subclasses of (a subclass of) **`CommonTree`**. Just like we did for **`ListNode`**.

Resist the urge to define and use many (>10) heterogeneous AST nodes.

With ANTLR (usually) at most a handful is needed. Due to the complete OO approach, W&B had to use a complete heterogenous approach.

Error Handling (1)

- When constructing compilers with ANTLR, errors (in the source text) are modelled by Java exceptions.
  - **`RecognitionException`** is the base class of all ANTLR Exceptions.
- In the Calc example of week 3 and 4, in CalcChecker.g, we threw a **`CalcException`** (as subclass of **`Recognition-Exception`**) when a semantic error was detected.
  - We had the following **`@rulecatch`** clause (to disable ANTLR's default exception handlers):

```
@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}
```

With this **`@rulecatch`** clause, we specified that an **`RecognitionError`** is not handled, but re-thrown to the **`main`** method. This essentially means that the Calc compiler stops at the first error.

Error Handling (2)

- The **`Parser`** and **`TreeParser`** classes already have their own exception handlers which catch all **`RecognitionException`**'s and report them.

  *To signal an error (i.e., context constraint violation) one can throw a **`RecognitionException`** to let the Parser (or TreeParser) report the error and continue parsing. For example:*

ListWalker

```
list  :  ...
      |  n=NUMBER
           { if ($n.text.equals("211035"))
                 throw new RecognitionException(
                         "211035 on line " + $n.getLine() +
                         " is not a valid number");
             else
                 $n.setValue(Integer.parseInt($n.text));
           }
      ;
```

The number "211035" is tagged as a **`RecognitionException`**. The **`ListWalker`** class will catch the **`Exception`** and report the error. Then it will proceed in walking the tree. Note that we use the line number that is associated with the Token of the NUMBER node.

Error Handling (3)

- User-defined error reporting routines for **Parser** or **TreeParser** classes:
  - override the **displayRecognitionError** method

```
tree grammar ListWalker;
...
@members {
  protected int nrErr = 0;
  public    int nrErrors() { return nrErr; }

  public void displayRecognitionError(
              String[] tokenNames, RecognitionException e) {
      nrErr = nrErr+1;
      if (e instanceof ListException)
          emitErrorMessage("[List] error: " + e.getMessage());
      else
          super.displayRecognitionError(tokenNames, e);
    }
}
```

Counting the total number of errors.

ListException is an user defined exception (in the style of **CalcException**).

Error Handling (4)

- It is also possible to catch Exceptions in any rule of your grammar:

```
rule : foo BAR SEMI!
     ;
     catch [RecognitionException re] {
         reportError(re);
         consumeUntil(input, SEMI);
         input.consume();
     }
```

Error recovery: consume all tokens until and including the SEMI token.

Syntactic Predicates (1)

- Consider the following rule:

```
rule : X Y
     | X Z
     ;
```

LL(1) problem.

can be solved using left-factorization:

```
rule : X (Y | Z) ;
```

- ANTLR support syntactic predicates which let you 'look into the tokenstream'.

```
rule : (X Y) => X Y
     | X Z
     ;
```

Only when X Y appears in the tokenstream take this alternative.

This can be regarded as 'locally setting k to 2'.

- Syntax for syntactic predicates:

```
( prediction block ) => production
```


Syntactic Predicates (2)

- Syntactic predicates allow us to use arbitrary lookahead.
  - This can be quite useful in the few cases where finite LL(k) for k>1 is insufficient.
    - e.g., in expression: function call or variable

```
expr : (ID LPAREN) => ID LPAREN params RPAREN
     | ID BECOMES ...
     | ...
     ;
```

```
foo(int x);
x=...
```

- Syntactic predicates are a form of selective backtracking.
  - Actions are turned off while evaluating a syntactic predicate so that actions do not have to be undone.

Semantic Predicates (1)

- A semantic predicate specifies a condition that must be met (at run-time) before parsing may proceed.
  - Syntax: `{ semantic-predicate-expression } ?`
- Validating predicates are predicates which throw exceptions (i.e., `FailedPredicateException`) if their conditions are not met while parsing a production.

```
decl : ^(VAR id=IDENTIFIER type)
       { if (isDeclared($id.text))
             throw new CalcException(...);
         else
             declare($id.text); };
```

CalcChecker

```
decl : ^(VAR id=IDENTIFIER type)
       { !isDeclared($id.text) }?
       { declare($id.text); }
    ;
```

with validating predicate

Semantic Predicates (2)

- Disambiguating predicates that are hoisted into the prediction expression for the associated production.

```
stat   :   // declaration "type varName;"
           { isTypeName(input.LT(1).getText()) }?
           ID ID SEMICOLON                  // declaration
       |   ID BECOMES expr SEMICOLON        // assignment
       ;
```

```
int x;
x=3;
```

Disambiguating predicates must be the first item of an alternative.

first lookahead-`Token`

```
public interface TokenStream {
  ...
  /** Return the i-th token of lookahead */
  public Token LT(int i);
}
```


StringTemplate (1)

http://www.stringtemplate.org/

- In the laboratory session of week 4 we built an ANTLR tree parser that could generate code for the TAM machine.
  - A straightforward implementation of such a code generator simply has numerous `emit`() statements as actions in the grammar specification.

```
assignment
    : ^(BECOMES id=IDENTIFIER expr)
          { int addr = dict.get($id.text);
            emit("STORE(1)" + addr + "[SB]");
          }
    ;
```

```
expr
    : operand
    | ^(PLUS expr expr)
          { emit("CALL  add"); }
    | ^(TIMES expr expr)
          { emit("CALL  mult"); }
    ...
    ;
```

```
operand
    : id=IDENTIFIER
          { int addr = dict.get($id.text);
            emit("LOAD(1)" + addr + "[SB]");
          }
    | n=NUMBER
          { emit("LOADL" + $n.text); }
    ;
```

ANTLR's StringTemplates can be used to collect all these emit strings in a separate file (i.e., a template).

StringTemplate (2)

A string template allows another level of indirection to isolate the target instructions.

CalcCodeGeneratorStringTemplate.g

```
tree grammar Generator;
options { ...
   output=template;
}
```

build a template

```
statement
   : ^(BECOMES id=IDENTIFIER expr)
     -> assign(addr={dict.get($id.text)},
               expr={$expr.st})
   | ^(PRINT expr)
     -> print(expr={$expr.st})
```

```
operand
   : id=IDENTIFIER
     -> loadvar(addr={dict.get($id.text)})
   | n=NUMBER
     -> loadnum(val={$n.text})
   ;
```

tam.stg

```
assign(addr,expr) ::= <<
<expr>
STORE(1) <addr>[SB]
>>
```

```
print(expr) ::= <<
<expr>
CALL  putint
CALL  puteol
>>
```

```
loadvar(addr) ::= <<
LOAD(1) <addr>[SB]
>>
```

```
loadnum(val) ::= <<
LOADL <val>
>>
```

Every non-terminal has a variable `st` which corresponds with the string template that it returns.

Note the resemblance between string templates and W&B's code templates.

StringTemplate (3)

original code generator

```
expr
  : operand
  | ^(PLUS expr expr)
      { emit("CALL  add"); }
  | ^(TIMES expr expr)
      { emit("CALL  mult"); }
  ;
```

Note that each node (expr) of the AST is responsible for generating its code.

CalcCodeGeneratorStringTemplate.g

```
expr
  : operand
    { $st = $operand.st; }

  | ^(PLUS  x=expr y=expr)
    -> binexpr(e1={x.st}, e2={y.st},
               instr="add")

  | ^(TIMES x=expr y=expr)
    -> binexpr(e1={x.st}, e2={y.st},
               instr="mult")
  ;
```

You have to explicitly state that the child's string template has to be copied.

tam.stg

```
binexpr(e1,e2,instr) ::= <<
<e1>
<e2>
CALL <instr>
>>
```

All machine specific code generation definitions appear together within the template definition (i.e., **tam.stg**).

It is 'easier' to port the code generator to a different target machine: "just" write a new template definition.

The source code of a code generator for Calc using String Templates will be made avaliable on the website.

Parser debugging

- ANTLR 2.x provided several useful command line options (e.g. `-traceParser`) and grammar options (e.g, `analyzerDebug`) to debug ANTLR parsers.
  - Unfortunately, ANTLR 3 does not longer support these options from the command line.
  - However, ANTLRWorks has more or less the same functionality and more (in a nice GUI).
  - Also have a look at gUnit, a unit testing framework for ANTLR grammars.
- Look at the generated Java code!
  - readable, recurisive descent parser
- Write a lot of test programs!

Final Notes

- Browse the ANTLR website
  http://www.antlr.org/
  to find a wealth of useful information (documentation, examples, mailing list discussions, etc.) on the tool.

MORE ANTLR MATERIAL

- After this slide there is some more material on using ANTLR.
  This extra material might be incorporated into VB 200*.

Additional Notes

- Show some error messages of ANTLR
  - especially LL(k) conflicts: non-determinism
- Show a 2-dimensional pictures of ASTs:
  - show the parse tree made by ANTLR
  - show the AST tree which is produced build by the parser
  - show how the Treeparser can walk this AST tree
- Explain that the grammar of an AST can be much more general than the parser grammar: all LL(1) difficulties of the parser are gone; we only have simple AST nodes.

Additional Notes

- Explain how the TreeParser deals with rules like:
  program : (statement)+
- If the TreeParser encounters a node which is not a statement, the TreeParser just stops (without an error message).
  - Can we turn explicit checking on?

Additional Notes

- Within Triangle it is exactly known how many children each AST node has. ANTLR, however, allows for an unlimited number of nodes (due + and *).
  - Show how this is represented in ANTLR.
  - Advantages of ANTLR's approach
    - more flexible
    - rules can be more general
  - Disadvantages of ANTLR's approach
    - passing information between sub trees is more difficult.
  - Note however that it is always possible to implement the Triangle-way in ANTLR.

OLD ANTLR 1

- THE NEXT SLIDES ARE SLIDES THAT WERE ONCE PART OF THE FIRST LECTURE ON ANTLR.
  - The slides contain rough ideas for adding material to the first lecture on ANTLR. Some of them have already made it to the second ANTLR lecture.
  - Most of them are clearly too advanced for an introductory course.
  - Perhaps we could get some ideas from them to improve this second lecture on ANTLR.
  - TCR, 1 May 2005.

CURRENTLY MISSING IN THE LECTURE!

OLD ANTLR 1

- showing more graphical pictures of the ASTs
  - especially: how the operator precedence ASTs are built
- semantic/syntactic predicates
- error handling and –recovery, throwing exceptions
- building your own AST nodes
- subclassing grammars
- demo: showing the AST nodes as ASTFrame
- see COMS4115 2002 - Programming Assignment 1, for some general tips on using ANTLR

Semantic Predicates

OLD ANTLR 1

- validating: middle of production, throws Exception
- disambiguating: first element in a production
  - ```
    stat:   // declaration "type varName;
            { isTypeName(LT(1)) }? ID ID ";"
    ```
    - ```
      if (LA(1) == ID && isTypeName(LT(1)) ) {
        match production one
      }
      ```
- guarded predicate
  - `(lookahead-context-for-predicate) => {predicate}?`
  - ```
    a   :    (ID) => { isType(LT(1))}? (ID|INT)
        |    ID
        ;
    ```
    - The predicate is only applicable when an ID is found on the input stream. It should not be evaluated when an INT is found.

Passing information

OLD ANTLR 1

- return values
  - foo returns [T x]
    - foo returns a value of type T, the actual value returned is the final value of x when foo returns
    - The part [T x] is a local declaration within the generated method for foo, so we can initialize T, e.g. [int n = 0]
  - x=foo in rule
    - foo is parsed and the returned value is put into x.
- parameters
  - foo [formal parameters]

Notes (1)

OLD ANTLR 1

- Useful macros
  (23 mei 2006 - dit klopt volgens mij niet, zie antlr.Parser)
  - LA(1)
    - current lookahead token
  - LT(1)
    - text representation of current lookahead token
    - useful for semantic predicates
- Exceptions: when a non-ANTLR Exception is thrown by P
  - ```
    class P extends Lexer | Parser | TreeParser ;
    ...
    start throws MyException
    ```

OLD ANTLR 1

Notes (2)

- Lexer
  - rule for strings:
    - `STRING : '"'! ('"' '"'! | ~('"'))* '"'! ;`
      - the !'s after a character means that the character will not be in the representation of this token
- On tree walkers
  - #(PLUS expr expr) means:
    - match a tree whose root is a PLUS token with two children that match the expr rule.
- operator precedence has to be hardcoded in the structure of the grammar

OLD ANTLR 1

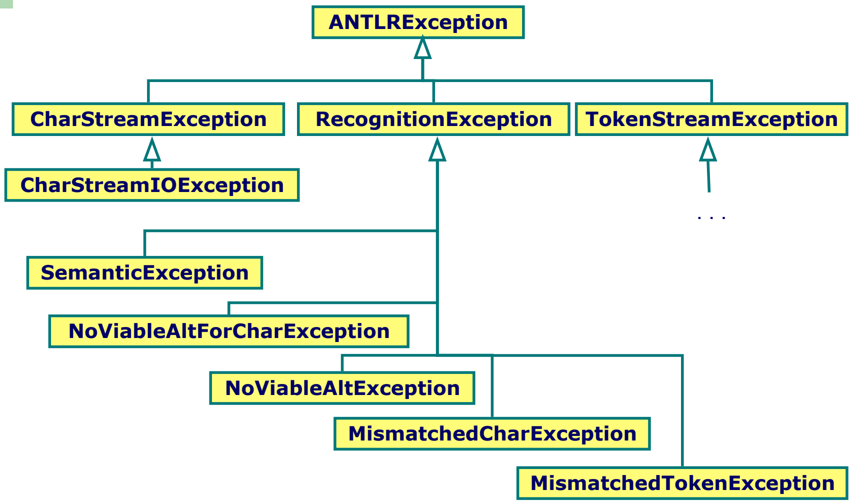
Notes (3)

- Keep amount of lookahead (i.e. k) low; especially in the parser. If more lookahead is necessary, use ANTLR's syntactic predicates .
  - expr : ...
    (ID "[" expr "]" "of") =>
    ID "[" expr "]" "of" expr

  which tells the parser to try to parse a ID, a bracket, and expression, a bracket and a token "of" before attempting to match the rest of the rule.

OLD ANTLR 2.7.x stuff

Error Handling (2)



see also: OLD ANTLR 2.7.x stuff
http://www.antlr.org/article/parse.trees/index.tml

Debugging (2)

Suppose we let ANTLR generate a Parser as follows

```
$ java antlr.Tool -traceParser tinyc.g
```

When giving the input

```
int i;
```

ANTLR will show what it is doing while parsing:

```
> program; LA(1)==int
  > declaration; LA(1)==int
   > variable; [guessing]LA(1)==int
    > type; [guessing]LA(1)==int
    < type; [guessing]LA(1)==i
    > declarator; [guessing]LA(1)==i
    < declarator; [guessing]LA(1)==;
   < variable; [guessing]LA(1)==null
   > variable; LA(1)==int
    > type; LA(1)==int
    < type; LA(1)==i
    > declarator; LA(1)==i
    < declarator; LA(1)==;
   < variable; LA(1)==null
  < declaration; LA(1)==null
 < program; LA(1)==null
```

part of tinyc.g

```
program
   : (declaration)* EOF
   ;
declaration
   : (variable) => variable
   | function
   ;
variable
   : type declarator SEMI
   ;
declarator
   : id:ID
   | STAR id2:ID
   ;
...
```

Final Notes

OLD ANTLR 2.7.x stuff

- The ANTLR grammar structure is itself specified as a ANTLR 2.7.2 grammar (of course): `antlr.g`.

  This (mother-of-all) ANTLR specification(s) nicely illustrates the language-features of ANTLR.