COMS W4115
Programming Languages and Translators
Programming Assignment 1: Scanner, Parser, and AST

Prof. Stephen A. Edwards
Columbia University

Assigned January 30th, 2002
Due February 13th, 2002

> Write the front-end of your Tiger compiler using ANTLR and Java. Have it generate the AST, but do not perform static semantic analysis (you will do this in the next assignment). Write the scanner, parser, and AST generator in one file called Tiger.g. Hook it up to the supplied XML generator to test it.

For this assignment, you will be using the ANTLR compiler tool to build part of the front-end of the Tiger compiler. ANTLR takes a grammar-like specification that may include fragments of Java code and generates Java source for scanners, parsers, and tree walkers. The documentation for ANTLR is available in the `~cs4115/antlr/doc` subdirectory, off the www.columbia.edu/~cs4115 website, and on the main ANTLR website, www.antlr.org.

Helpful files for this assignment are in `~cs4115/prog1` on the cunix cluster.

1 Getting Started

The software you will need is installed on Columbia's CUNIX cluster (use "`ssh -l` *uni* `cunix.columbia.edu`" to log in), although feel free to download and install it yourself elsewhere.

First, set your CLASSPATH environment variable so you can run ANTLR and the programs it generates:

```
# Under bash
$ CLASSPATH=~cs4115/antlr:.
$ export CLASSPATH

# Under csh
% setenv CLASSPATH ~cs4115/antlr:.
```

It should now be possible to run ANTLR:

```
$ java antlr.Tool
ANTLR Parser Generator   Version 2.7.1
usage: java antlr.Tool [args] file.g
```

2 ANTLR

ANTLR generates top-down recursive-descent scanners and parsers. Such parsers accept a slightly more restricted class of languages than the bottom-up parsers generated by the well-known lex and yacc tools, but the generated code is easier to understand because it more closely matches the input and error recovery can be easier.

You use ANTLR by writing one or more grammar files (file suffix `.g`) that contain definitions for lexical analyzers, parsers, and/or tree parsers. From such a file, ANTLR generates one or more `.java` files with code for classes that implement the parsers, etc.

To give you an idea how ANTLR works, here is a small but useful example: a calculator that evaluates expressions like "`3*4+2;`".

The following `calc.g` file is an ANTLR grammar that describes a lexer, parser, and tree walker for the calculator.

```
class CalcLexer extends Lexer;

WS : ( ' ' | '\t' | '\n' { newline(); }
     | '\r' ) { $setType(Token.SKIP); }
   ;
LPAREN : '(' ;
RPAREN : ')' ;
STAR : '*' ;
PLUS : '+' ;
SEMI : ';' ;

protected
DIGIT : '0'..'9' ;
INT : (DIGIT)+ ;

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

stmt : expr SEMI! ;
expr : mexpr (PLUS^ mexpr)* ;
mexpr : atom (STAR^ atom)* ;
atom : INT
     | LPAREN! expr RPAREN!
     ;

class CalcTreeWalker extends TreeParser;

expr returns [float r] { float a,b; r=0; }
 : #(PLUS a=expr b=expr ) {r = a+b;}
 | #(STAR a=expr b=expr ) {r = a*b;}
 | i:INT
   { r = (float)Integer.parseInt(i.getText()); }
 ;
```

From this, ANTLR generates a scanner, parser, and tree walker than can be invoked using the following (`Calc.java`).

```
import antlr.CommonAST;

class Calc {
  public static void main(String[] args) {
    try {
      CalcLexer l = new CalcLexer(System.in);
      CalcParser p = new CalcParser(l);
      p.stmt();
      CommonAST a = (CommonAST) p.getAST();
      CalcTreeWalker walker =
        new CalcTreeWalker();
      float r = walker.expr(a);
      System.out.println(r);
    } catch (Exception e) {
      System.err.println("exception: " + e);
    }
  }
}
```

These two files (in the directory `~cs4115/calc`) can be compiled and run as followed:

```
$ java antlr.Tool calc.g
ANTLR Parser Generator   Version 2.7.1
$ javac Calc.java
$ java Calc
10*(2+3*2);
80.0
```

Let's look at this example in some detail.

The definition for the lexer comes first. Each starts with a line of the form "class *Classname* extends Lexer;" and contains rules that define tokens. A rule starts with the name of the token (which must start with an uppercase latter) followed by a colon, an expression defining what characters constitute the token, and finally a semicolon to terminate the rule.

Rules, e.g., for LPAREN can be quite simple. An LPAREN is simply the single open-paren character. The rule for WS ("whitespace") is more complicated. The rule is a choice among different possibilities separated by vertical bars (`|`). The code enclosed in braces is an action: stylized Java code that is executed when the rule is matched. The action after the newline character (`'\n'`) simply calls the newline() method of the lexer, which updates the line count used to report error messages. The action after the parenthesis-contained choices sets the type of the returned token, in this case, SKIP, which indicates the token is to be discarded and not passed to the parser.

The rule for INT says INT is one or more DIGITs (this is what is meant by the `()+` notation). The rule for DIGIT is protected, meaning it can only be used within another rule; the parser will never see a token of type DIGIT. The notation `'0'..'9'` in the DIGIT rule means a digit is a character between and including 0 and 9.

In operation the lexer looks at the next few characters in the input stream and attempts to match them to one of the non-protected rules. One it has, the lexer returns the matched token and will start looking beyond the end of the last token the next time it is invoked.

The parser, which appears second in this example, starts with a class definition and an options block. The "buildAST=true" option setting causes the parser to build an abstract syntax tree (AST) when it is running. In this example, once built, this tree is passed to the CalcTreeWalker class, which traverses the tree to compute the result.

Like the lexer, parser rules start with the name of the keyword being defined (which must begin with a lowercase letter in the parser) followed by a colon and the rule.

AST construction in the parser is controlled by single-character suffixes after the tokens in the rules. The two interesting ones are `!`, which prevents the parser from generating a node for the token (used, for example, to supress puctuation such as semicolons and parenthesis), and `^`, which means the token should become a root of a subtree. In general, you will want to mark every operator token with a `^`; here, these are PLUS and STAR.

The rule for `expr` says an `expr` is an `mexpr` followed by zero or more sequences of the form "`+ mexpr`". This is exactly how the parser will parse such an expression: it will look first for an mexper, then, if it can, it will try try to match a "+" followed by another `mexpr`, and it will continue to do this as long as it can. Writing it in this style means the "+" operator is treated as being left-associative, that is, a sequence such as "1+2+3" is treated as meaning "(1+2)+3" not "1+(2+3)".

The precedence of `+` relative to `*` is defined here by breaking the rules into `expr`, `mexpr`, and `atom`. Intuitively, since `*` is meant to bind more tightly, meaning "1+2*3" should be parsed as "1+(2*3)". This is accomplished by writing the rules as they are: this makes the parser try to parse the higher-precendence `*` first (an `mexpr`) before trying to parse the lower-precedence `+`. (Implementing arithmetic expressions is definitely more awkward with ANTLR than with other parser generators that allow precedence and associativity to be defined explicitly.)

The third part of this example is a tree walker. While not necessary for this first assignment, we will use its abilities later. A tree walker traverses an AST built by the parser according to given rules. This particular tree walker visits the AST nodes in a depth-first order, calculating the expressions it represents along the way.

The rules in a tree walker look much like those for lexers and parsers. A few variants are present: "returns [float r]" means that matching an `expr` produces a float value (this is exactly the return value of the recursive `expr` match method generated by ANTLR for this example), and an initial action declares two local variables and initializes the result.

In a tree walker definition, the notation `#( )` means attempt to match a subtree. The first symbol in the parentheses is the root of the tree and the others are the children. Thus, `#(PLUS expr expr)` attempts to match a subtree rooted at a PLUS node with two children that match `expr`s. The `a=expr` notation means assign the result of matching `expr` (a float) to the local Java variable `a`.

```
class TigerASTGram extends TreeParser;
options {
  exportVocab = Tiger;
}

lvalue
  : ID
  | #( FIELD lvalue ID )       // lvalue.field
  | #( SUBSCRIPT lvalue expr)  // lvalue[expr]
  ;

expr
  : "nil"
  | lvalue
  | STRING
  | NUMBER
  | #( NEG expr )             // - expr
  | #( BINOP expr expr )      // e.g., expr + expr
  | #( ASSIGN lvalue expr )   // lvalue := expr
  | #( CALL ID (expr)* )      // foo(expr, expr)
  | #( SEQ (expr)* )          // expr ; expr
  | #( RECORD ID              // type { a=b, c=d }
      (#(FIELD ID expr))* )
  | #( NEWARRAY ID expr expr ) // type [ex] of ex
  | #( "if" expr expr (expr)? )
  | #( "while" expr expr )
  | #( "for" ID expr expr expr )
  | "break"
  | #( "let" #(DECLS (#(DECLS (decl)+ ))* ) expr )
  ;

decl
  : #( "type" ID type )
  | #( "var" ID (ID | "nil") expr )
  | #( "function" ID fields (ID | "nil") expr )
  ;

type
  : ID
  | fields                    // { a:b, c:d }
  | #( "array" ID )           // array of type
  ;

fields : #( FIELDS ( #(FIELD ID ID) )* ) ;
```

Figure 1: The file `TigerASTGram.g`: a tree parser defining the AST for Tiger.

3 AST

Figure 1 shows `TigerASTGram.g`: ANTLR rules for a tree parser defining the AST you need to generate for Tiger. The important thing about this syntax is, for example, `#(BINOP expr expr)` represents a subtree rooted at a BINOP node with two children that are both exprs.

All binary operators (e.g., +, &) are represented with the single token type BINOP. The text of this token should contain the actual operator (e.g., &, >=). The text of the STRING and NUMBER tokens similarly should contain their actual values.

The var and function subtrees each have the same number of children regardless of whether a type is defined. A variable declaration with an undefined type or a procedure (function without a return type) each have the `"nil"` token instead of a type. In Java code, this is represented as `LITERAL_nil`.

A list of declarations is represented as a list of lists because Tiger treats adjacent function and type declarations as potentially mutually recursive. Therefore, each sequence of function or type declarations should end up in its own DECLS list so that all the functions or types defined in a single DECLS subtree are mutually visible.

4 Hints for writing the Tiger Grammar

- String constants may contain escape sequences such as `\n`. It's best to translate these into the characters they actually represent during lexical analysis. ANTLR's lexical analzers can change the text of a token using `$setText()` in an action. See the ANTLR documentation for more information.

- Use the technique of multiple rules for expressions to implement Tiger's different precedence levels.

- ANTLR parsers and lexers can be told to use different amounts of lookahead: how many characters/tokens to look forward before making a decision about what rule to match. For example, the lexer rule `COMPARISON : '<' | "<=" ;` is ambiguous with a single character of lookahead (both rules appear to match when the next character is <). I found `k=2;` was necessary in the options section for both the Tiger lexer and parser.

- Sometimes, more elaborate lookahead is necessary to avoid ambiguities. You will probably find ANTLR considers Tiger's "type [expr] of expr" construct ambiguous since it begins the same way as one of the lvalue rules. The solution to this is to have the ANTLR-generated parser try a more complicated experiment before matching a rule, something done with the => operator.

  In my parser, one of the alternatives for `expr` looks like

  ```
  (ID "[" expr "]" "of") =>
   ID "[" expr "]" "of" expr
  ```

  This tells the parser to try to parse a prefix of an ID, a bracket, and expression, a bracket, and the token "of" before attempting to match the rest of the rule. Such far-sighted lookahead removes the ambiguity.

- Rules that choose between trying to match something and continuing without matching (e.g., the `()?`, `()*`, and `()+` constructs) can be made either "greedy" or "non-greedy." A greedy rule tries to match its own tokens before giving up and trying to match its successors; a non-greedy rule tests these in the opposite order.

  "Who owns the else?" is a common source of ambiguity in many grammars, Tiger's included. To illustrate the problem, consider

```
if A then if B then C else D
```

Does the "else D" belong to "if A" or "if B"? Most languages choose the nearest "if", i.e., "if B" in this case. ANTLR's greediness control allows this to be specified explicitly. The rule

```
"if" expr "then" stmt ("else" stmt)?
```

is ambiguous because the parser does not know whether to first try to match the "else" or trying to match what may follow. The solution is to make the rule for the "else" greedy:

```
"if" expr "then" stmt
  (options {greedy=true;} : "else" stmt)?
```

Greediness is also useful for scanning comments and grouping sequences of declarations.

- Comments are usually the trickiest thing in a scanner. Use `{greedy=false;}` in the rule for comments (see the discussion in the ANTLR documentation) and make sure you make your comments nest, i.e., `/*/**/ this is ignored */`.
- Make sure to set `buildAST = true;` in the options block for your parser to enable the AST-building machinery.
- Make your lexer tokens all uppercase (e.g., "`PERIOD`") and parser rules all lowercase (e.g., "`expr`"). Do this for readability (ANTLR only constrains the case of the first letter).
- Use `$setType(Token.SKIP)` to ignore whitespace and comments.
- There's a special EOF token that signals end-of-file. It appears to be necessary to have the outermost parser rule be

  ```
  file : expr EOF! ;
  ```

- If you use the `~` operator in the lexer (I did not have to) to invert a set of characters, you will probably want to use the `charVocabulary` option. See the options section of the ANTLR documentation for details.
- ANTLR's rule are written in extended Backus-Naur form (EBNF). Some of the most useful constructs are `( )?` (zero or one instances), `( )*` (zero or more instances), and `( )+` (one or more instances). I used all three in my grammar.
- When building the AST, the two useful token suffixes are `!`, which prevents an AST node from being built for the token, and `^`, which makes the node a root of a subtree.
- ANTLR does most of the work of building an AST for you, but there are some cases where you need to help. The most common is when you want a node representing a list of zero or more instances of something. This is done by adding an action that augments the subtree returned by a rule. Consider building an AST for a comma-separated list of zero or more elements. The rule

  ```
  list : element (","! element)*
         { #list = #([LIST], list); }
       ;
  ```

  often does what you want. The bare rule generates a simple sequence of elements (the ! following the comma means no node is generated for the comma), but often you want this sequence to be a rooted tree. The action fixes this: it replaces the just-built sequence with a subtree rooted at a node called "LIST."

- Another trick I found useful when parsing arithmetic expressions gives you the ability to change the type of a token. I used this to change arbitrary operators to a token of type BINOP.

  ```
  expr : expr1
    ("+"^ expr1 { #expr.setType(BINOP); } )*
  ```

- The syntax of most languages is a collection of numbers and identifiers (names) structured with punctuation (e.g., $+$) and keywords (e.g., "if"). Usually, the keywords are syntactically similar to identifers, which leads to a quandary: when is an identifier actually a keyword?

  There are two standard approaches. The first separates keywords from identifiers while the token is being assembled. E.g., if the first character is "i" and the next character is "f", return the "if" token and otherwise return an identifier. The second approach, which ANTLR employs, parses everything as an identifer and later checks to see if it was actually a keyword.

  After an ANTLR-generated scanner matches any token, it checks to see whether the text in that token appears in its literal (keyword) table and returns the keyword instead of the token if it does. Each double-quoted string in the parser grammar is placed in the scanner's literal table. Thus it is possible to write rules such as

  ```
  expr : "while" expr "do" stmt ;
  ```

  Here, "while" and "do" end up in the scanner's literal table. The rule will work like you expect provided the strings "while" and "do" are parsed as tokens (e.g., as identifiers).

  Using this can cause subtle problems: in fact, you need to turn off literal comparison for the double-quoted string rules. See the ANTLR documentation for how to do this.

- The grammar in the Tiger language reference manual is ambiguous, however it can be made unambiguous through careful rewriting, paying attention to such things as precedence and associativity. Doing this is one of the main points of the assignment. Your final grammar should be unambiguous, i.e., not produce any ANTLR warnings.
- I've supplied `TigerASTGram.g` (Figure 1) in `~cs4115/prog1`. Use it to verify the content and structure of the AST you generate.

- Use the following template for your `Tiger.g` file:

  ```
  class TigerParser extends Parser;
  option {
    importVocab = Tiger;
    buildAST = true;
    /* other options */
  }
  /* rules */

  class TigerLexer extends Lexer;
  /* rules */
  ```

  The `Tig2xml` class expects these names for the parser and scanner classes.

  The `importVocab` option makes ANTLR use the token types generated by ANTLR from the `TigerASTGram.g` file. Make sure to run ANTLR on this file before running it on your `Tiger.g`.

- I've provided `~cs4115/prog1/Tig2xml.java` to assist with testing your scanner and parser. It calls your scanner and parser to read a `.tig` source file, writes the output in XML (Internet Explorer can display this in a convenient tree form), and then runs the tree walker in `TigerASTGram.g` on the generated AST. It will complain (although not too helpfully, unfortunately) if the AST does not comply with the specification. Make sure your scanner and parser works correctly with this test fixture.

- Keep it short: My `Tiger.g` file for the scanner and parser was only 213 lines, although it has no error handling.

5 Deliverables

Submit the following:

- Your `Tiger.g` file.

- A `README` file describing your scanner, parser, and test-cases. In particular, I want to hear how you dealt with

  - Testing your front-end
  - Nested Comments
  - Syntax Errors
  - Parsing sequences of like declarations (e.g., functions, vars, etc.) into their own DECLS subtrees.
  - Distinguishing between `arraytype [10] of 0` and `array[3]`.
  - The dangling else problem.
  - Operator precedence and BINOP subtrees
  - Strings containing keywords, e.g., `"if"`

- A Makefile with at least two rules

  - "make" by itself should compile your source files so that "java Tig2xml" runs in this directory.
  - "make test" should run tests on your front end.

- A subdirectory called "tests" containing test programs.

- A file called `MEMBERS` that contains a space-separated list of the uni IDs of each of the members in your group, e.g.,

  ```
  se2007 mkf1998 dw1969
  ```

- The files from the `~cs4115/prog1` directory.

To submit your assignment, create a subdirectory with just these files (e.g., don't include any `.class` files or anything else) and run `~cs4115/bin/submit_code`, which will submit everything in the current directory.

To test your submission, make sure "make" and "make test" both work in the directory starting from just those files. This is how we will grade your assignment; make sure your program compiles without any warnings.

Only one member of the group should submit: all the members listed in the MEMBERS file will be credited.

After a successful submission, you will receive a confirmation email from the class account with a list of files received. In case you did not receive the confirmation, try again and contact the TA staff. You can submit multiple times, but the last submission is what counts. Each submission will be time stamped. The assignment is due an hour before before the beginning of class on the due date.