6 Syntactic analysis

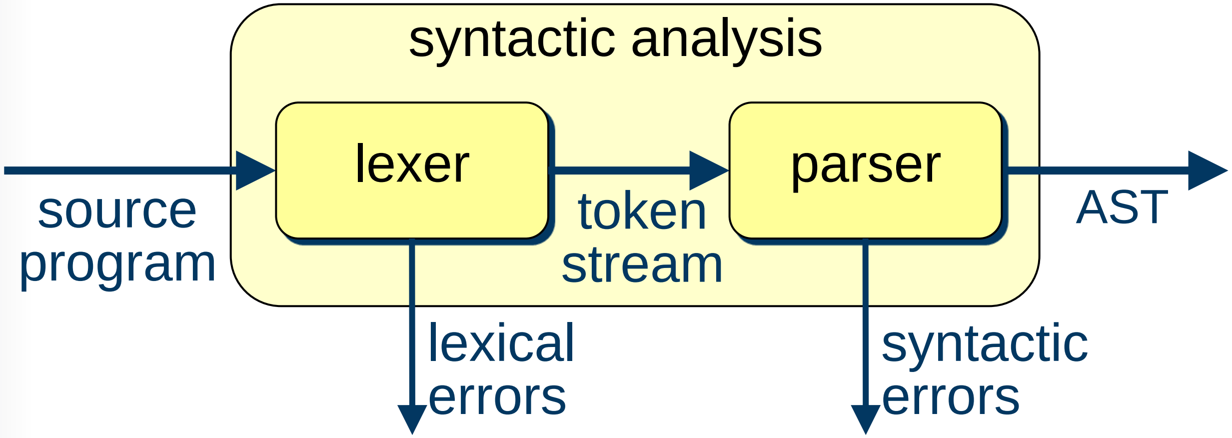
- Aspects of syntactic analysis
- Tokens
- Lexer
- Parser
- Applications of syntactic analysis
- Compiler generation tool ANTLR
- Case study: Calc
- Case study: Fun syntactic analyser

Aspects of syntactic analysis *(1)*

- **Syntactic analysis** checks that the source program is well-formed and determines its phrase structure.
- Syntactic analysis can be decomposed into:
    - a **lexer**
      (which breaks the source program down into tokens)
    - a **parser**
      (which determines the phrase structure of the source program).

Aspects of syntactic analysis *(2)*

- *Recall:* The syntactic analyser inputs a source program and outputs an AST.
- Inside the syntactic analyser, the lexer channels a stream of tokens to the parser:



Tokens

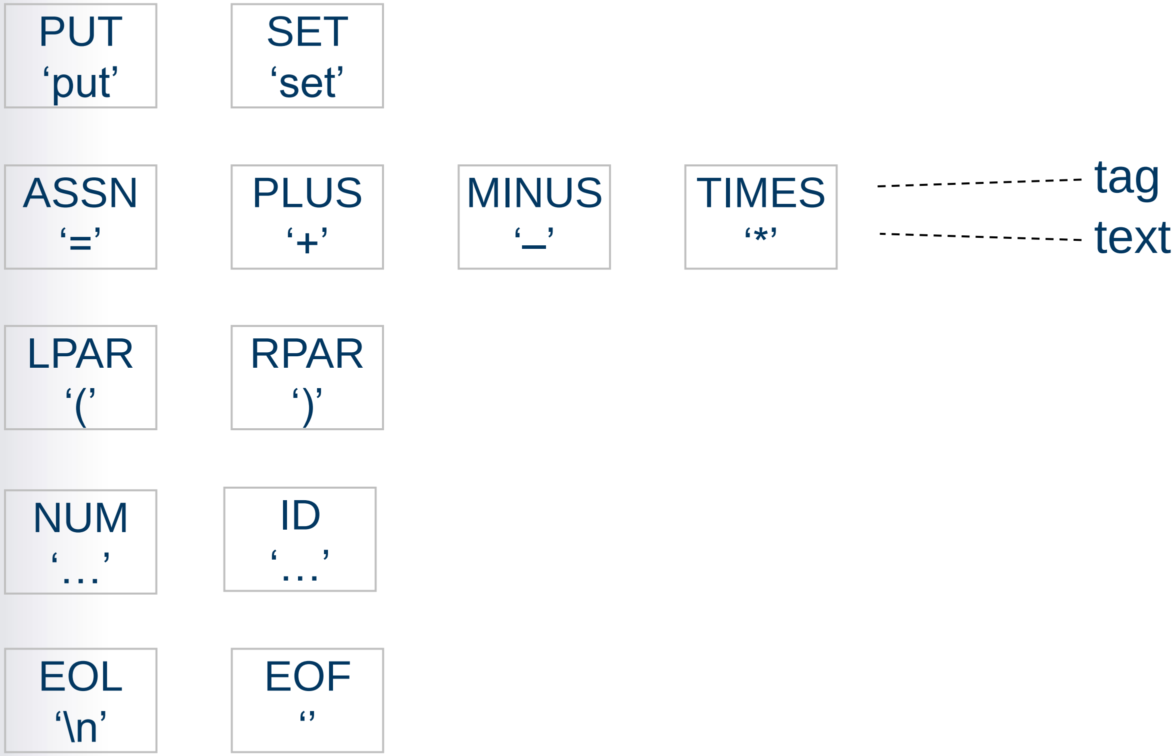
- **Tokens** are textual symbols that influence the source program’s phrase structure, e.g.:
  - literals
  - identifiers
  - operators
  - keywords
  - punctuation (parentheses, commas, colons, etc.)
- Each token has a **tag** and a **text**. E.g.:
  - the addition operator might have tag PLUS and text ‘+’
  - a numeral might have tag NUM, and text such as ‘1’ or ‘37’
  - an identifier might have tag ID, and text such as ‘x’ or ‘a1’.

Separators

- **Separators** are pieces of text that *do not* influence the phrase structure, e.g.:
  - spaces
  - comments.
- An end-of-line is:
  - a separator in most PLs
  - a token in Python (since it delimits a command).

Case study: Calc tokens

- Complete list of Calc tokens:



PUT ‘put’ SET ‘set’

ASSN ‘=’ PLUS ‘+’ MINUS ‘–’ TIMES ‘*’

LPAR ‘(’ RPAR ‘)’

NUM ‘…’ ID ‘…’

EOL ‘\n’ EOF ‘’

tag

text

Example: Calc tokens

- Calc source program and token stream:

```
set x = 7
put x*(x+1)
```

SET ‘set’
ID ‘x’
ASSN ‘=’
NUM ‘7’
EOL ‘\n’
PUT ‘put’
ID ‘x’
TIMES ‘*’
LPAR ‘(’
ID ‘x’
PLUS ‘+’
NUM ‘1’
RPAR ‘)’
EOL ‘\n’
EOF ‘’

Lexer *(1)*

- The **lexer** converts source code to a token stream.
- At each step, the lexer inspects the next character of the source code and acts accordingly *(see next slide)*.
- When no source code remains, the lexer outputs an EOF token.

Lexer *(2)*

- E.g., if the next character of the source code is:
  - a *space:*
    discard it.
  - the *start of a comment:*
    scan the rest of the comment, and discard it.
  - a *punctuation mark:*
    output the corresponding token.
  - a *digit:*
    scan the remaining digits, and output the corresponding token (a numeral).
  - a *letter:*
    scan the remaining letters, and output the corresponding token (which could be an identifier or a keyword).

Parser

- The **parser** converts a token stream to an AST.
- There are many possible parsing algorithms.
- **Recursive-descent parsing** is particularly simple and attractive.
- Given a suitable grammar for the source language, we can quickly and systematically write a recursive-descent parser for that language.

Recursive-descent parsing *(1)*

- A recursive-descent parser consists of:
  - a family of parsing methods `N()`, one for each nonterminal symbol *N* of the source language's grammar
  - an auxiliary method `match()`.
- These methods "consume" the token stream from left to right.

Recursive-descent parsing *(2)*

- Method `match(t)` checks whether the next token has tag `t`.
  - If yes, it consumes that token.
  - If no, it reports a syntactic error.
- For each nonterminal symbol *N*, method `N()` checks whether the next few tokens constitute a phrase of class *N*.
  - If yes, it consumes those tokens (and returns an AST representing the parsed phrase).
  - If no, it reports a syntactic error.

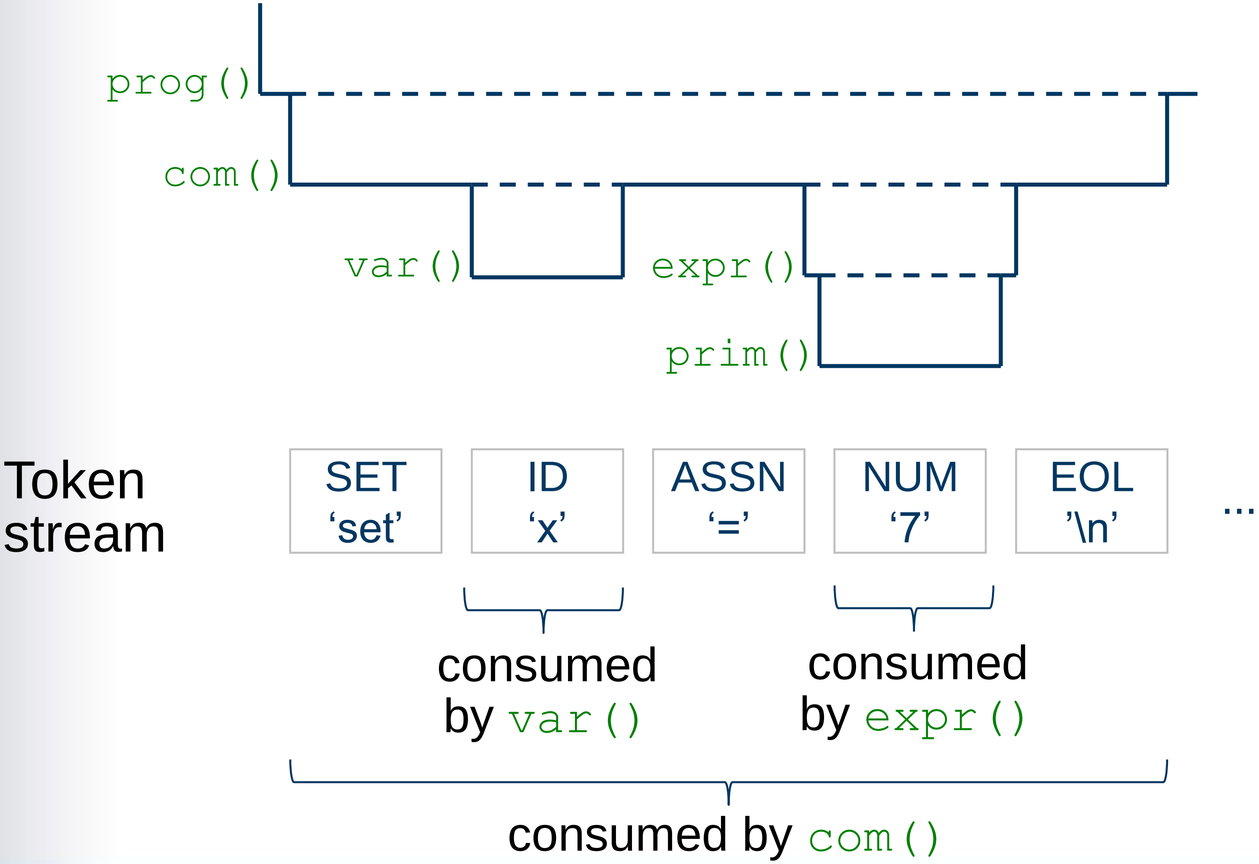
Example: Calc parser *(1)*

- Parsing methods:

`prog()`	parses a program
`com()`	parses a command
`expr()`	parses an expression
`prim()`	parses a primary expression
`var()`	parses a variable

Example: Calc parser *(2)*

- Illustration of how the parsing methods work:



`prog()`

`com()`

`var()`

`expr()`

`prim()`

Token stream

SET ‘set’	ID ‘x’	ASSN ‘=’	NUM ‘7’	EOL ’\n’	…

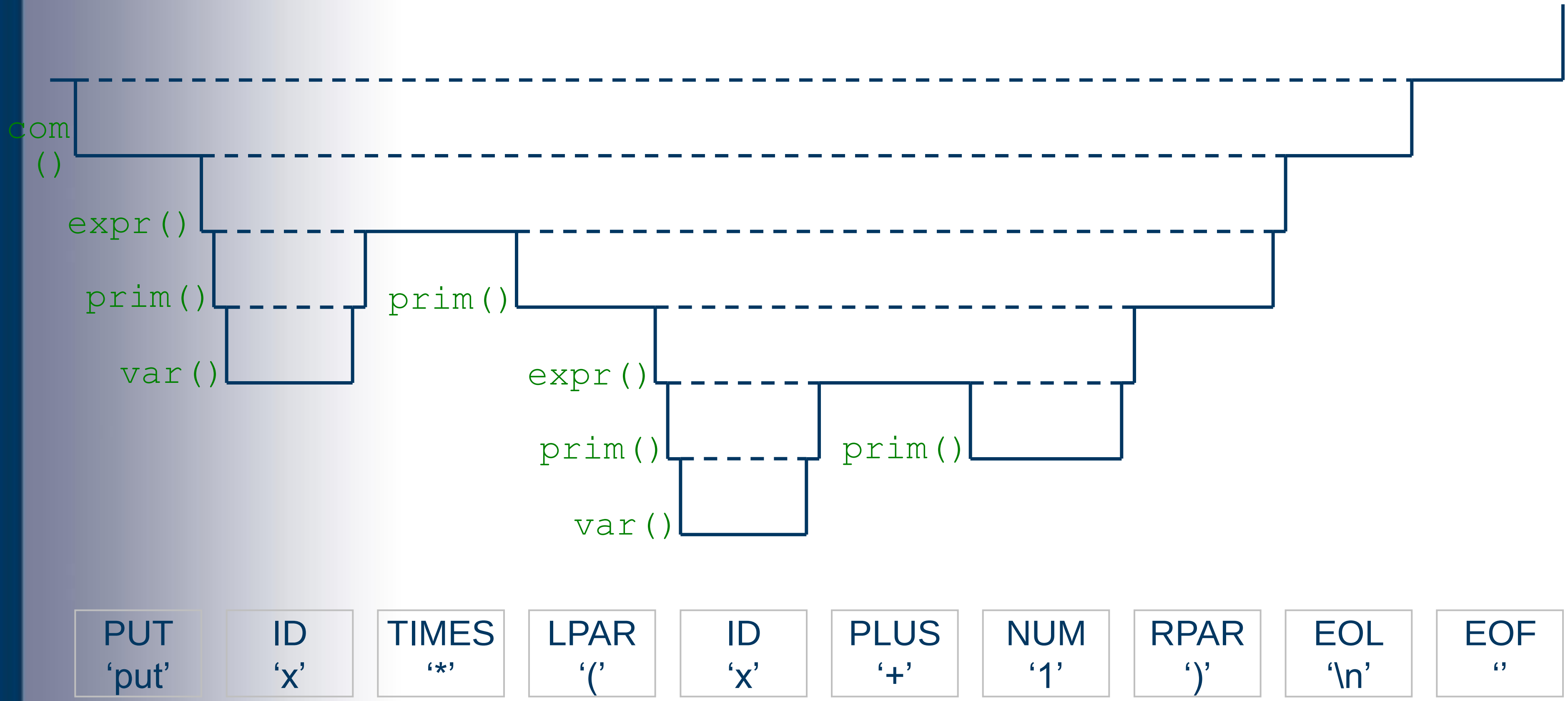
consumed by `var()`

consumed by `expr()`

consumed by `com()`

Example: Calc parser *(3)*

- Illustration *(continued)*:



PUT ‘put’	ID ‘x’	TIMES ‘*’	LPAR ‘(’	ID ‘x’	PLUS ‘+’	NUM ‘1’	RPAR ‘)’	EOL ‘\n’	EOF ‘’

University of Glasgow

Example: Calc parser *(4)*

- Recall the EBNF production rule for commands:

```
com  =  'put' expr eol
     |  'set' var '=' expr eol
```

- Parsing method for commands *(outline)*:

```
void com () {
   if (…) {
      match(PUT);
      expr();
      match(EOL);
   }
```

if the next token is 'put'

Example: Calc parser *(5)*

- Parsing method for commands *(continued)*:

```
    else if (…) {
       match(SET);
       var();
       match(ASSN);
       expr();
       match(EOL);
    }
    else
       …
 }
```

if the next token is ‘set’

report a syntactic error

Example: Calc parser *(6)*

- Recall the EBNF production rule for programs:

  prog = *com* * *eof*

- Parsing method for programs *(outline)*:

```
void prog () {
   while (…) {
      com();
   }
   match(EOF);
}
```

while the next token is ‘set’ or ‘put’

General rules for recursive-descent parsing *(1)*

- Consider the EBNF production rule:

  N = *RE*

- The corresponding parsing method is:

```
void N () {
    match the pattern RE
}
```

General rules for recursive-descent parsing *(2)*

- To match the pattern *t*
  (where *t* is a terminal symbol):

  match(*t*);

- To match the pattern *N*
  (where *N* is a nonterminal symbol):

  N();

- To match the pattern RE_1 RE_2:

  match the pattern RE_1
  match the pattern RE_2

General rules for recursive-descent parsing *(3)*

- To match the pattern $RE_1 \mid RE_2$:

```
if (the next token can start $RE_1$)
    match the pattern $RE_1$
else if (the next token can start $RE_2$)
    match the pattern $RE_2$
else
    report a syntactic error
```

- *Note:* This works only if no token can start both RE_1 and RE_2.
  - In particular, this does not work if a production rule is left-recursive, e.g., $N = X \mid N\ Y$.

General rules for recursive-descent parsing *(4)*

- To match the pattern *RE* *:

```
while (the next token can start RE)
    match the pattern RE
```

- *Note:* This works only if no token can both start and follow *RE*.

Applications of syntactic analysis

- Syntactic analysis has a variety of applications:
    - in compilers
    - in XML applications (parsing XML documents and converting them to tree form)
    - in web browsers (parsing and rendering HTML documents)
    - in natural language applications (parsing and translating NL documents).

Compiler generation tools

- A **compiler generation tool** automates the process of building compiler components.
- The input to a compiler generation tool is a *specification* of what the compiler component is to do. E.g.:
  - The input to a parser generator is a grammar .
- Examples of compiler generation tools:
  - lex and yacc *(see Advanced Programming)*
  - JavaCC
  - SableCC
  - ANTLR.

The compiler generation tool ANTLR

- ANTLR (ANother Tool for Language Recognition) is the tool we shall use here. See www.antlr.org.
- ANTLR can automatically generate a lexer and recursive-descent parser, given a grammar as input:
  - The developer starts by expressing the source language’s grammar in ANTLR notation (which resembles EBNF).
  - Then the developer enhances the grammar with actions and/or tree-building operations.
- ANTLR can also generate contextual analysers *(see §7)* and code generators *(see §8)*.

Case study: Calc grammar in ANTLR *(1)*

- Calc grammar expressed in ANTLR notation:

```
grammar Calc;

prog
        :  com* EOF
        ;

com
        :  PUT expr EOL
        |  SET var ASSN expr EOL
        ;

var
        :  ID
        ;
```

Case study: Calc grammar in ANTLR *(2)*

- Calc grammar *(continued)*:

```
expr
        :  prim
           ( PLUS prim
           | MINUS prim
           | TIMES prim
           )*
        ;

prim
        :  NUM
        |  var
        |  LPAR expr RPAR
        ;
```

Case study: Calc grammar in ANTLR *(3)*

- Calc grammar *(continued – lexicon)*:

```
PUT    :  'put'  ;
SET    :  'set'  ;

ASSN   :  '='  ;
PLUS   :  '+'  ;
MINUS  :  '-'  ;
TIMES  :  '*'  ;
LPAR   :  '('  ;
RPAR   :  ')'  ;

ID     :  'a'..'z'  ;
NUM    :  '0'..'9'+  ;

EOL    :  '\r'? '\n'  ;
SPACE  :  (' ' | '\t')+      {skip();}  ;
```

Tokens and separators have upper-case names.

This says that a `SPACE` is a separator.

Case study: Calc driver *(1)*

- Put the above grammar in a file named `Calc.g`.
- Feed this as input to ANTLR:

```
…$ java org.antlr.Tool Calc.g
```

- ANTLR automatically generates the following classes:
  - Class `CalcLexer` contains methods that convert an input stream (source code) to a token stream.
  - Class `CalcParser` contains parsing methods `prog()`, `com()`, …, that consume the token stream.

Case study: Calc driver *(2)*

- Write a driver program that calls `CalcParser`'s method `prog()`:

```
public class CalcRun {

   public static void main (String[] args) {
      InputStream source =
          new InputStream(args[0]);
      CalcLexer lexer = new CalcLexer(
          new ANTLRInputStream(source));
      CommonTokenStream tokens =
          new CommonTokenStream(lexer);
      CalcParser parser =
          new CalcParser(tokens);
      parser.prog();
   }

}
```

creates an input stream

creates a lexer

runs the lexer, creating a token stream

creates a parser

runs the parser

Case study: Calc grammar in ANTLR *(6)*

- When compiled and run, `CalcRun` performs syntactic analysis on the source program, reporting any syntactic errors.
- However, `CalcRun` does nothing else!

Enhancing a grammar in ANTLR

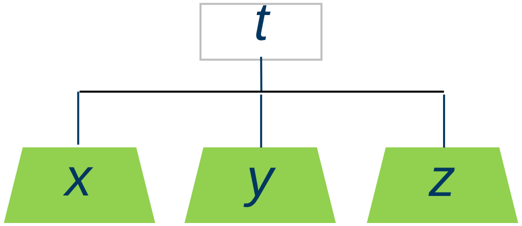
- Normally we want to make the parser do something useful.
- To do this, we enhance the ANTLR grammar with either actions or tree-building operations.
- An ANTLR **action** is a segment of Java code:

```
{ code }
```

- An ANTLR **tree-building operation** has the form:

```
-> ^(t x y z)
```

where `t` is a token and `x`, `y`, `z` are subtrees.



Case study: Calc grammar in ANTLR with actions *(1)*

- Suppose that we want `CalcRun` to perform actual calculations:
  - The command “`put` *expr*” should evaluate the expression *expr* and then print the result.
  - The command “`set` *var* `=` *expr*” should evaluate the expression *expr* and then store the result in the variable *var*.

Case study: Calc grammar in ANTLR with actions *(2)*

- We can augment the Calc grammar with actions to do this:
  - Create storage for variables ‘a’, …, ‘z’.
  - Declare that `expr` will return a value of type `int`. Add actions to compute its value. And similarly for `prim`.
  - Add an action to the `put` command to print the value returned by `expr`.
  - Add an action to the `set` command to store the value returned by `expr` at the variable’s address in the store.

Case study: Calc grammar in ANTLR with actions *(3)*

- Augmented Calc grammar:

```
grammar Calc;

@members {
     private int[] store
        = new int[26];
}

prog
     :  com* EOF
     ;
```

storage for variables ‘a’, …, ‘z’

Case study: Calc grammar in ANTLR with actions *(4)*

- Augmented Calc grammar *(continued)*:

```
com
        :  PUT v=expr EOL   { println(v); }

        |  SET ID ASSN
             v=expr EOL     { int a =
                               $ID.text.charAt(0)
                                - 'a';
                              store[a] = v; }

        ;
```

`$ID.text` is the text of `ID` (a string of letters)
`$ID.text.charAt(0)` is the 1st letter.
`$ID.text.charAt(0)-'a'` is in the range 0..25.

Case study: Calc grammar in ANTLR with actions *(5)*

- Augmented Calc grammar *(continued)*:

```
expr                           returns [int val]

        :  v1=prim             { $val = v1; }
           ( PLUS v2=prim      { $val += v2; }
           | MINUS v2=prim     { $val -= v2; }
           | TIMES v2=prim     { $val *= v2; }
           )*
        ;
```

Case study: Calc grammar in ANTLR with actions *(5)*

- Augmented Calc grammar *(continued)*:

```
prim                          returns [int val]
     :  NUM                   { $val = parseInt(
                                  $NUM.text); }
     |  ID                    { int a =
                                 $ID.text.charAt(0)
                                  - 'a';
                                $val = store[a]; }
     |  LPAR v=expr RPAR { $val = v; }
     ;
```

Case study: Calc grammar in ANTLR with actions *(6)*

- Run ANTLR as before:

```
…$ java org.antlr.Tool Calc.g
```

- ANTLR inserts the `@members{…}` code into the `CalcParser` class.
- ANTLR inserts the above actions into the `com()`, `expr()`, and `prim()` methods of `CalcParser`.

Case study: Calc grammar in ANTLR with actions *(7)*

- When compiled and run, `CalcRun` again performs syntactic analysis on the source program, but now it also performs the actions:

```
…$ javac CalcLexer.java CalcParser.java \
      CalcRun.java

…$ java CalcRun test.calc

16
56
72
```

```
set c = 8
set e = 7
put c*2
put e*8
set m = (c*2) + (e*8)
put m
```

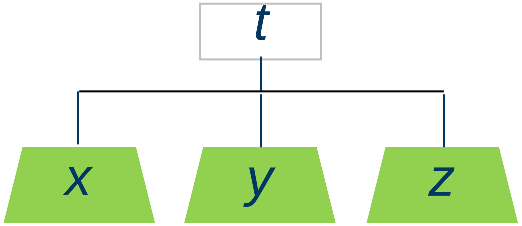
ANTLR notation

- At the top of the `expr` production rule, “`expr returns [int val]`” declares that parsing an `expr` will return an integer result named `value`.
- Within actions in the `expr` production rule, “`$val = …`” sets the result.
- In any production rule, “`v=expr`” sets a local variable `v` to the result of parsing the `expr`.

AST building with ANTLR

- What if the parser is required to build an AST?
- Start with an EBNF grammar of the source language, together with a summary of the ASTs to be generated.
- Express the grammar in ANTLR’s notation. Then add tree-building operations to specify the translation from source language to ASTs.
- *Recall:* An ANTLR tree-building operation has the form:

```
-> ^(t x y z)
```



Case study: Fun grammar in ANTLR *(1)*

- Fun grammar *(outline)*:

```
grammar Fun;

prog
     :  var_decl* proc_decl+ EOF
     ;

var_decl
     :  type ID ASSN expr
     ;

type
     :  BOOL
     |  INT
     ;
```

Case study: Fun grammar in ANTLR *(2)*

- Fun grammar *(continued)*:

```
com
        :  ID ASSN expr
        |  IF expr COLON seq_com DOT
        |  …
        ;

seq_com
        :  com*
        ;
```

Case study: Fun grammar in ANTLR *(3)*

- Fun grammar *(continued)*:

```
expr : sec_expr …

sec_expr
    : pri_expr
        ( (PLUS | MINUS | TIMES | DIV)
          pri_expr
        )*
    ;

pri_expr
    : NUM
    | ID
    | LPAR expr RPAR
    | …
    ;
```

Case study: Fun grammar in ANTLR with AST building *(1)*

- Augmented Fun grammar *(outline)*:

```
grammar Fun;

options {
        output = AST;
        …;
}

tokens {
        PROG;
        SEQ;
        …;
}
```

states that this grammar will generate an AST

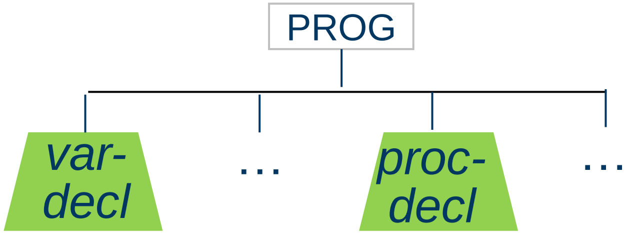
lists special tokens to be used in the AST (in addition to lexical tokens)

Case study: Fun grammar in ANTLR with AST building *(2)*

- Augmented Fun grammar *(outline)*:

```
prog
        :  var_decl* proc_decl+ EOF
                              -> ^(PROG
                                  var_decl*
                                  proc_decl+)
        ;
```

builds an AST like this:

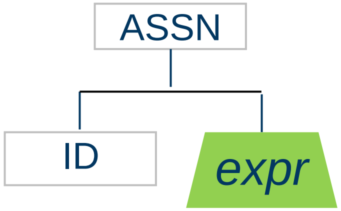


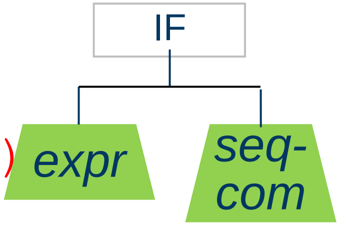
Case study: Fun grammar in ANTLR with AST building *(3)*

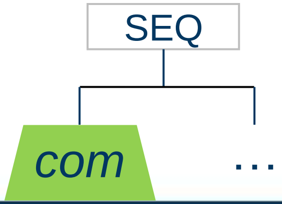
- Augmented Fun grammar *(continued)*:

```
com
   :  ID ASSN expr        -> ^(ASSN
                                ID
                                expr)
   |  IF expr COLON
        seq_com DOT       -> ^(IF
                                expr
                                seq_com)
   |  …
   ;

seq_com
   :  com*                -> ^(SEQ
                                com*)
   ;
```







Case study: Fun grammar in ANTLR with AST building *(4)*

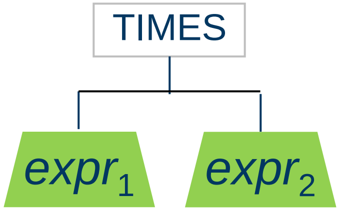
- Augmented Fun grammar *(continued)*:

```
expr : sec-expr …

sec_expr
     : prim_expr
         ( (PLUS^ | MINUS^ | TIMES^ | DIV^)
             prim_expr
         )*
     ;

prim_expr
     : NUM                  -> NUM
     | ID                   -> ID
     | LPAR expr RPAR       -> expr
     | …
     ;
```

builds an AST like this:



Case study: Fun syntactic analyser *(1)*

- Put the above grammar in a file named `Fun.g`.
- Run ANTLR to generate a lexer and a parser:

```
…$ java org.antlr.Tool Fun.g
```

- ANTLR creates the following classes:
  - Class `FunLexer` contains methods that convert an input stream (source code) to a token stream.
  - Class `FunParser` contains parsing methods `prog()`, `var_decl()`, `com()`, …, that consume the token stream.
- The `prog()` method now returns an AST.

Case study: Fun syntactic analyser *(2)*

- Program to run the Fun syntactic analyser:

```
public class FunRun {

   public static void main (String[] args) {
      InputStream source =
         new FileInputStream(args[0]);
      FunLexer lexer = new FunLexer(
         new ANTLRInputStream(source));
      CommonTokenStream tokens =
         new CommonTokenStream(lexer);
      FunParser parser =
         new FunParser(tokens);
      CommonTree ast = (CommonTree)
         parser.prog().getTree();
   }

}
```

runs the parser

gets the resulting AST