Languages and Compilers
Recursive-Descent Parsing Basics

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University

Agenda.

- Parsing.
- Recursive-Descent Parsing.
- Recursive-Descent Recognisers.

Parsing.

- Syntax Analysis.
- The Parser.
- The Parse Tree.
- Parsing Approaches.

Syntax Analysis.

- *Parsing* or *recognising*, performed by a *syntax analyser*, *parser* or *recogniser*.
- Determines if a program is syntactically correct according to the EBNF.
  - i.e. finds the derivation sequence from the starter symbol, by
  - parsing the input tokens into their corresponding syntactic components.
- Also detects, reports and recovers from syntactic errors.

The Parser …

… processes the stream of tokens output from the scanner.

… creates a *syntax* or *parse tree* representing the derivation sequence for the input program.

… *may* be the control component of a **single-pass** compiler.

- … initiates scanning, semantic analysis and artifact generation as each syntactic construct is processed.

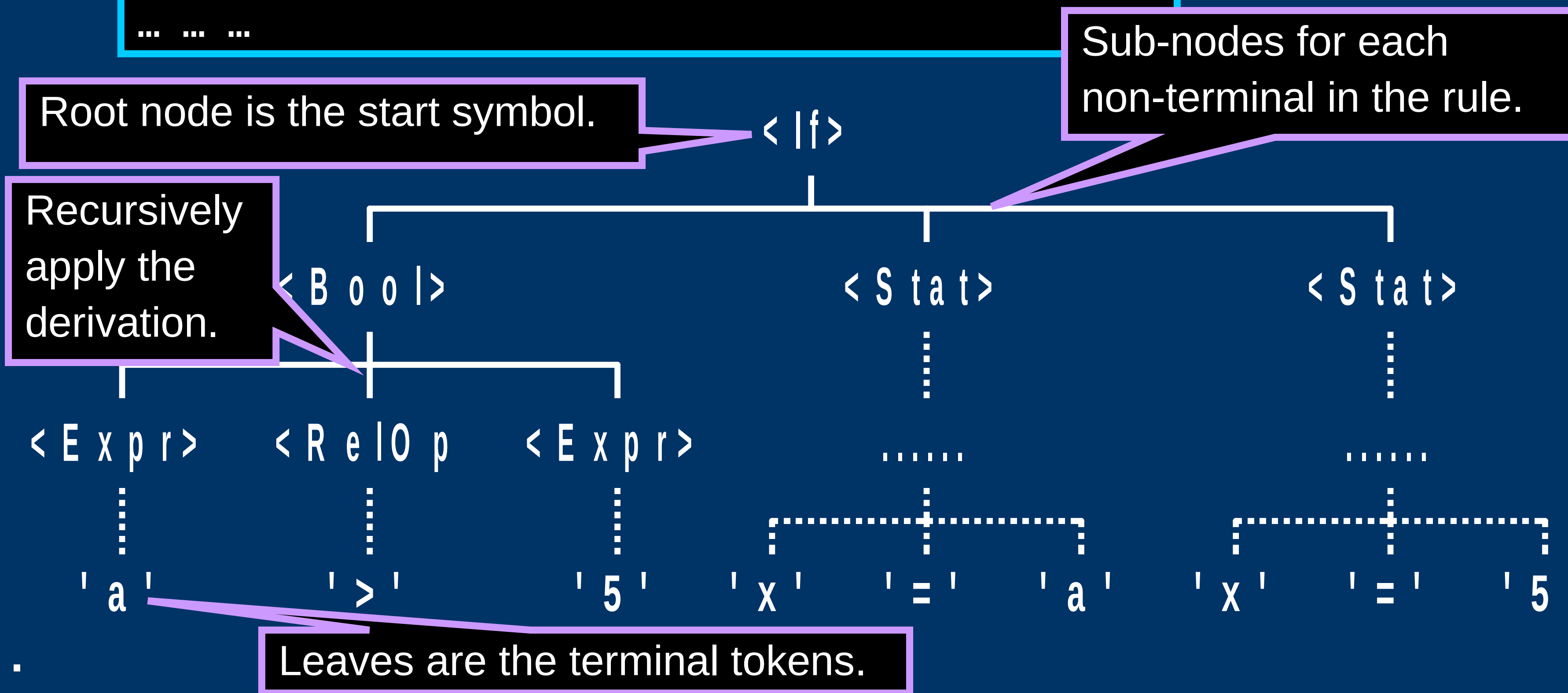
An Example Parse Tree.

EBNF

```
<If> ::= if <Bool> then <Stat> else <Stat> ;
<Bool> ::= <Expr> <RelOp> <Expr> ;
<RelOp> ::= "<" | "<=" | = | ">" | ">=" ;
… … …
```

Input

```
if a>5
then x=a
else x=5
```



The Parse Tree (AST)

- Represents steps of the derivation sequence
- Identifies
  - the alternate productions used,
  - the structure of the productions, and
  - the terminal tokens.
- Is built up incrementally as the parse proceeds through the input program.
- Is used subsequently in generating artifacts.
- May be represented by some object collections.
- … but may not be required as a data structure for some parsing techniques (eg RD).

Parsing Approaches.

- Parsers can be
  - top-down or bottom-up
  - single or multi-pass
- The approach taken depends on the attributes of the EBNF specification defining the language.
- The approach to constructing *recursive-descent* parsers we will use in this course applies only to LL(1) specifications.
- Languages that cannot be defined using an LL(1) specification must use other approaches.
  - These are outside the scope of this course.

Recursive-Descent Parsing.

- Recursive-Descent.
- LL(1) Specifications & Parsing.
- RD Parser Structure.
- Using the Parser.
- The RD Primitives.

Recursive-Descent (RD).

- A one-pass parsing implementation technology.
- Applies only to languages defined by LL(1) specifications.
- The parser is derived directly from the BNF rules, productions and clauses.
- The parser is top-down, deterministic with no back-tracking or look-ahead.
- No internal representation of the syntax tree is required – but you can construct one if necessary.

LL(1) Specifications.

- LL(1) specifications
  - process the input tokens from left to right,
  - derive the leftmost non-terminal first,
  - require only 1 token look-ahead.
- The next step in a derivation (i.e. which alternate production to apply) is defined
  - by the current input token, and
  - the terminals in the director sets of the alternate productions.

LL(1) Specifications & Parsing.

- Parsing a program written in a language defined by an LL(1) specification...
  - reads the input tokens one at a time,
  - requires you only to keep track of the current input token,
  - needs to know the director sets of each production,
  - can process a non-terminal derivation independantly of other derivations.

Recursive-Descent Recognisers.

- Recogniser Methods.
- Recognising Productions.
- Alternative Productions.
- Handling The Null Production.

Recogniser Methods.

- The parser can be written directly from the BNF following the steps given in the following slides.

```
<Block> ::= …
<Statement-List> ::= …
<Statement> ::= …
… … …
```

1) There should be one recogniser method for every non-terminal rule in the main (non-microsyntax) BNF.

```
private void recStarter() {…}
private void recStatementList() {…}
private void recStatement() {…}
… … …
```

- The *void xxx()* signature may be changed when semantic analysis or artifact gen are added.

The RD Primitive Methods.

- The RD primitive methods are common to all recursive-descent parsers.

- **`void mustBe (String s)`**
  - asserts that the current token on the input stream must be *s*.
  - If it is then the next token is read in;
  - If it is not then a syntax error has been detected.

- **`bool have (String s)`**
  - Returns true if the current input token is *s*, returns false otherwise.

Recognising Productions.

- A production of terminal and/or non-terminal tokens is recognised by a sequence of *mustBe(...)* and recogniser method calls.

```
<Let-Statement> ::= let Identifier := <Expression> ;
```

2) Terminal tokens are recognised by calling *mustBe()* for that token.
Non-terminals are recognised by calling the corresponding recogniser method.

```
private void recLetStatement() {
  mustBe ("let");
  mustBe (Token.IdentifierToken);
  mustBe (":=");
  recExpression();
}
```

- Use literal strings for simple terminal tokens.
- Use *constants* of the token class for microsyntax terminal types.

Alternative Productions.

- Alternative productions in a rule are selected with an **`if (have(…))`** statement.

```
<Block> ::= <Statement> |
            begin <Statement-List> end ;
```

3) use *have(…)* to test if the next token is in the director set of each production.

```
private void recBlock () {
  if (have ("begin")) {
    mustBe ("begin");
    recStatementList ();
    mustBe ("end");
  }
  else
  {  recStatement(); }
}
```

- In this example the test is opposite from the order of the productions to avoid too many *have(…)* tests.
- If there is a syntax error then this will be left to be detected in *recStatement()*.

A Different Version.

- In this version we test explicitly for the director sets of each production and indicate an error.

```
private void recBlock () {
  if (have("let") || have("for") ||
      have("get") || have("put"))
    recStatement();
  else if (have ("begin")) {
    mustBe ("begin");
    recStatementList ();
    mustBe ("end");
  }
  else
    syntaxError ("<Block>");
}
```

- This version is **more robust** than the previous version.
- If an invalid token starts this part of the parse then it will be detected here rather than pushing it down to one of the other recogniser methods.

Handling The Null Production.

```
<Statement-List> ::= <Statement> <Statement-List> | <> ;
```

4) For a null production we do nothing; that is we use the empty {}.

```
private void recStatementList () {
  if (have("let") || have("for") ||
      have("get") || have("put")) {
    recStatement();
    recStatementList();
  }
  else {} // do nothing.
}
```

- If we wish to explicitly test for the null prodn. director set then this will contain the **follower tokens** of the rule.

```
else if (have("end") ||
             have(EndOfFileToken.Tag))
  {}  // do nothing
else
  syntaxError("<Statement-List>");
```

- So the valid follower tokens are "end" and end-of-file.

Where Is The Parse Tree?

- An explicit parse tree is not required to keep track of the derivation.
  - The recogniser method call sequence is an implicit parse tree.
- If an explicit parse tree is needed then create it in the recogniser methods as they are called.

Extract From Block-1 BNF.

```
<Block> ::= <Statement> | begin <Statement-List> end ;
<Statement-List> ::= <Statement> <Statement-List> | <> ;
<Statement> ::= <Let-Statement>  | … ;
<Let-Statement> ::= let Identifier := <Expression> ;
<Expression> ::= <Term> <Rest-Expr> ;
<Rest-Expr> ::= + <Term><Rest-Expr> | - <Term><Rest-Expr> | <> ;
<Term> ::= <Factor> <Rest-Term> ;
<Rest-Term> ::= * <Factor><Rest-Term> | / <Factor><Rest-Term> | <> ;
<Factor> ::= Identifier | IntValue | "(" <Expression> ")" ;

microsyntax
Identifier <|[a-zA-Z][\w_\.]*
IntValue <|\d+
```

- Some <Statement> productions have been left out of this extract.
- See the full BNF at *Resources\Block-1\Block-1.BNF.txt*
- This is an LL(1) specification.
- It has been written using only standard BNF.

A Derivation.

Input

```
let a := b * (2+c)
```

<Block>

→ <Statement>

→ <Let-Statement>

→ let Identifier := <Expression>

→ let a := <Term> <Rest-Expr>

→ let a := <Factor> <Rest-Term> <Rest-Expr>

→ let a := Identifier <Rest-Term> <Rest-Expr>

→ let a := b * <Factor> <Rest-Term> <Rest-Expr>

→ let a := b * (<Expression>) <Rest-Term> <Rest-Expr>

→ let a := b * (<Term> <Rest-Expr>) <Rest-Term> <Rest-Expr>

→ let a := b * (<Factor> <Rest-Term> <Rest-Expr>) <Rest-Term> <Rest-Expr>

→ let a := b * (IntValue <Rest-Term> <Rest-Expr>) <Rest-Term> <Rest-Expr>

→ let a := b * (2 <> <Rest-Expr>) <Rest-Term> <Rest-Expr>

→ let a := b * (2 + <Term> <Rest-Expr>) <Rest-Term> <Rest-Expr>

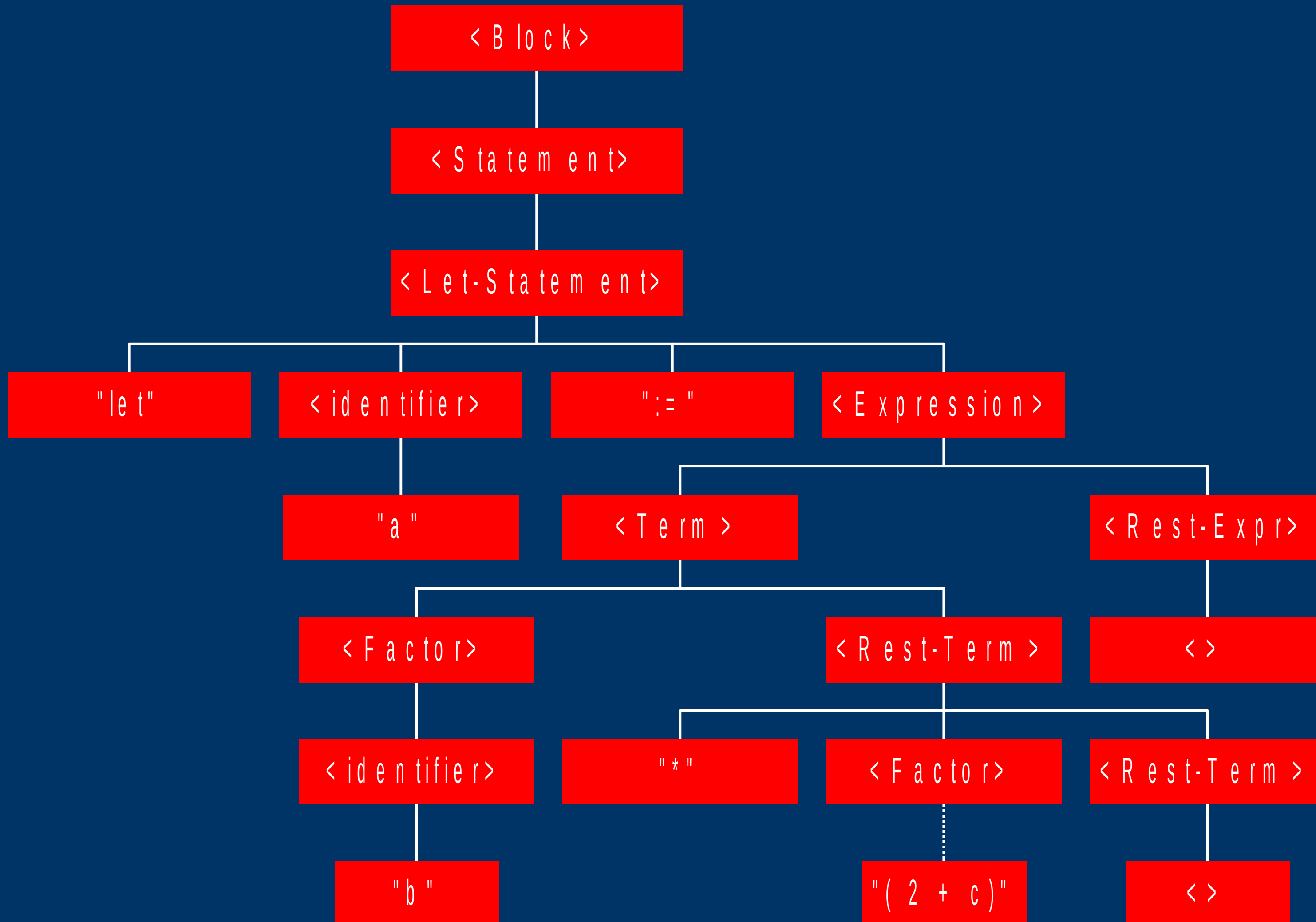
→ let a := b * (2 + <Factor> <Rest-Term><Rest-Expr>) <Rest-Term><Rest-Expr>

→ let a := b * (2 + Identifier <Rest-Term><Rest-Expr>) <Rest-Term><Rest-Expr>

→ let a := b * (2 + c <>) <> <> = let a := a * (2 + c)

The Parse Tree.

```
let a := b * (2+c)
```



Recogniser Calls.

```
let a := b * (2+c)
```

```
recBlock();
```

```
if haves ("begin")) {
  mustBe ("begin");
  recStatementList ();
  mustBe ("end");
}
else
  recStatement();
```

```
if (have("let"))
    recLetStatement();
...
```

```
mustBe ("let");
mustBe (Token.IdentifierToken);
mustBe (":=");
recExpression();
```

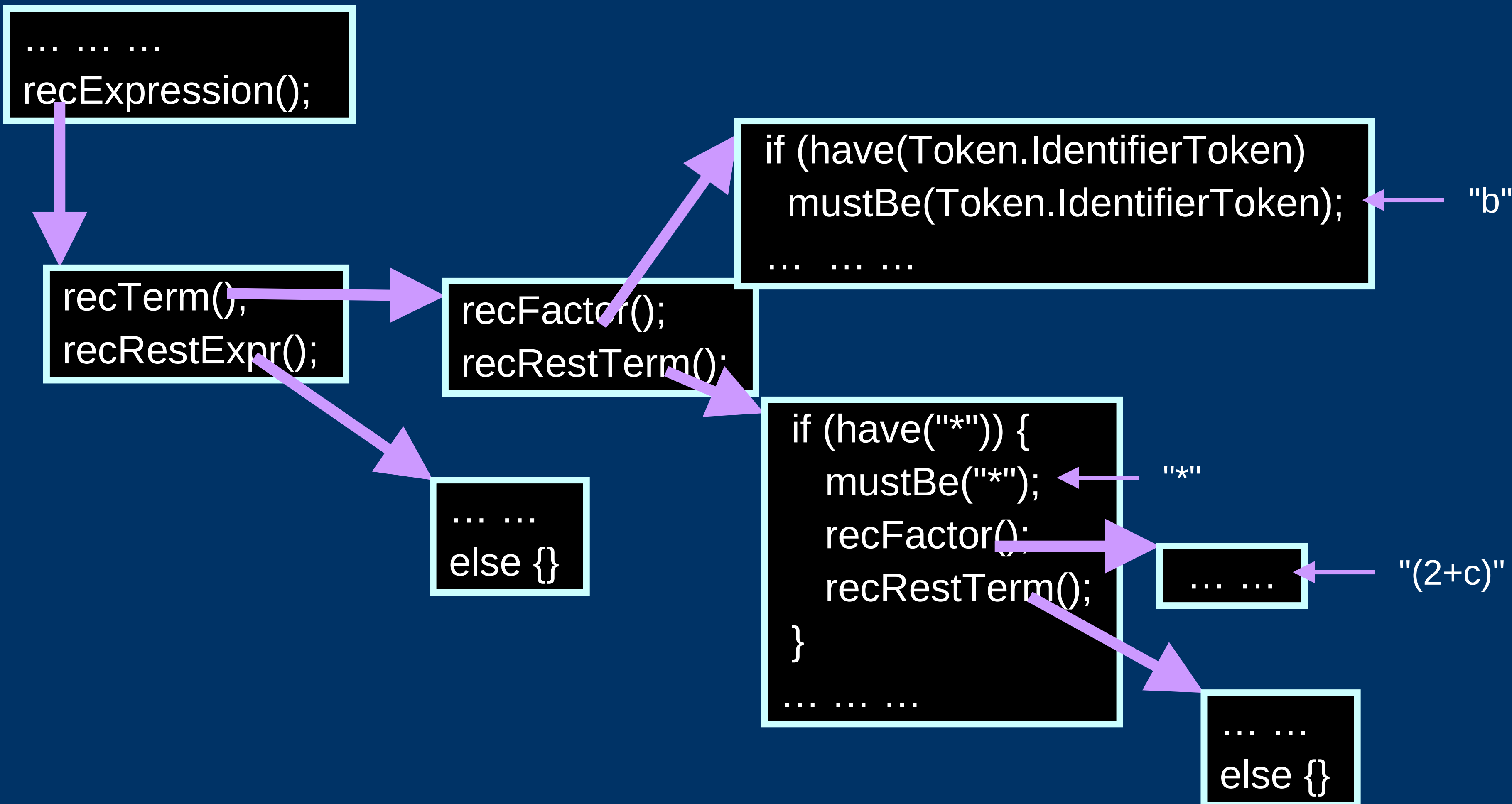
"let"

"a"

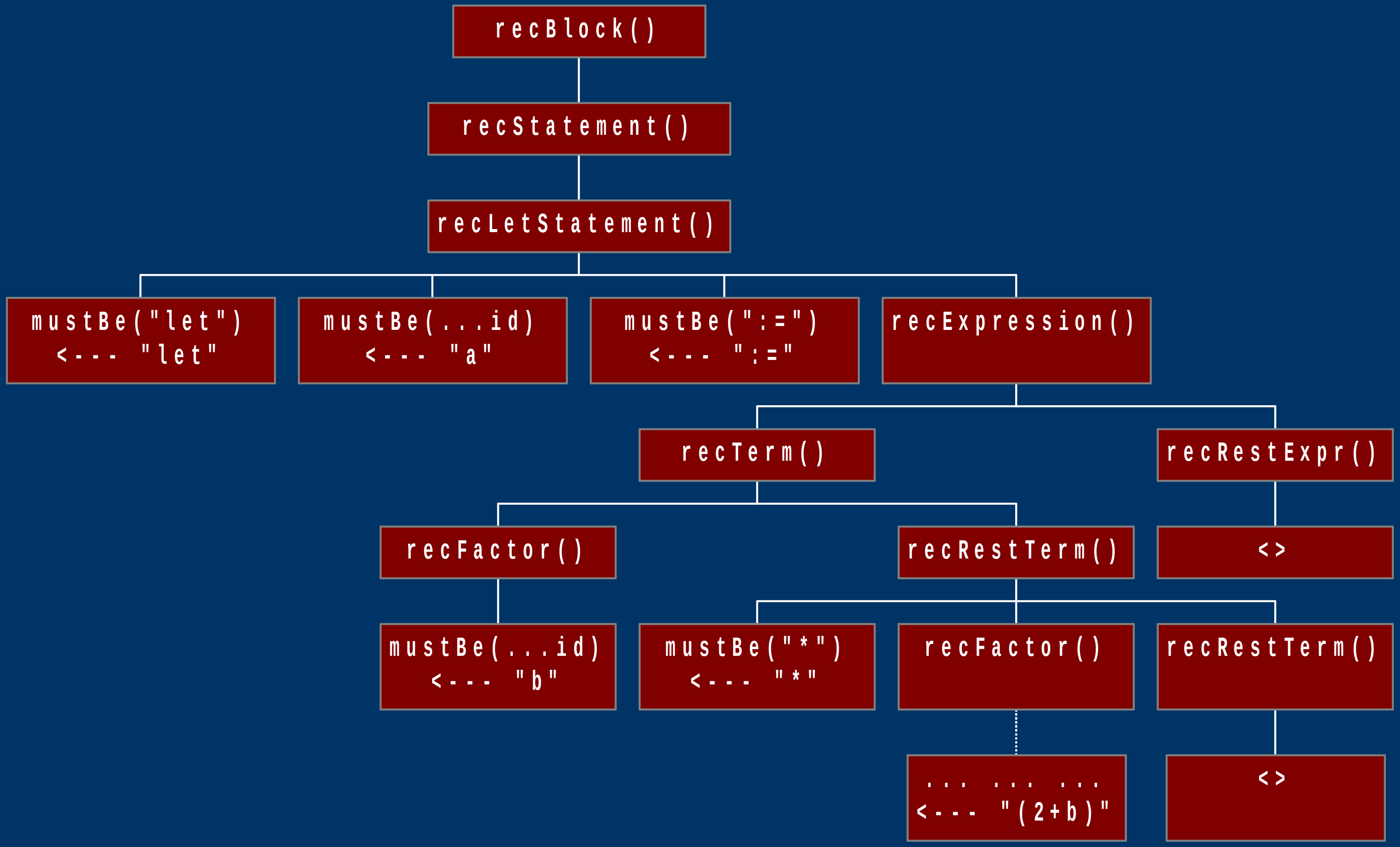
":="

Recogniser Calls.

```
let a := b * (2+c)
```



Another View Of The Calls



Does It Have To Be LL(1)?

- No – I'm oversimplifying
- I've been saying that you **must** have an LL(1) language to do recursive-descent parsing...
- … but in practice there are lots of compilers for non-LL(1) languages (e.g. C) written in recursive-descent style
- You need more complex primitives than `mustBe`/`have` for this, though!

From GCC, `gcc/c/c-parser.c` ...

```
/* Parse a parenthesized condition from an if, do or
   while statement.
   condition:
     ( expression )
*/
static tree
c_parser_paren_condition (c_parser *parser)
{
  tree cond;
  if (!c_parser_require (parser, CPP_OPEN_PAREN,
                         "expected %<(%>"))
    return error_mark_node;
  cond = c_parser_condition (parser);
  c_parser_skip_until_found (parser, CPP_CLOSE_PAREN,
                             "expected %<)%>");
  return cond;
}
```

Summary.

And Now You Know About …

… why LL(1) was a useful property to have!

… recursive-descent parsing.

… implementing a recursive-descent parser.

… writing recogniser methods directly from BNF.

… the relationship between the derivation sequence, the order of method calls in an RD parser, and the abstract syntax tree.