# COMPILER DESIGN

## Generating an Abstract Syntax Tree using Syntax-Directed Translation

# Abstract Syntax Tree: Definition

- An abstract syntax tree (AST) is a tree representation of the *abstract syntactic structure* of source code.
- Each node of the tree denotes a construct occurring in the source code.
- The syntax is "abstract" in not representing every detail appearing in the real syntax.
- For instance, grouping parentheses have been removed, and a syntactic construct like an `if-then-else` may be denoted by means of a single node with three branches.

- This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees.
- Once built, additional information is added to the AST by means of subsequent processing steps such as semantic analysis and code generation.

# Abstract Syntax Tree: Goal

- **Goal**: to *aggregate* information gathered during the parse in order to get a broader understanding of the meaning of *whole syntactic constructs*
- At the leaves of the tree is fine – grained information.
- As the information is migrated up in the tree, information is aggregated.

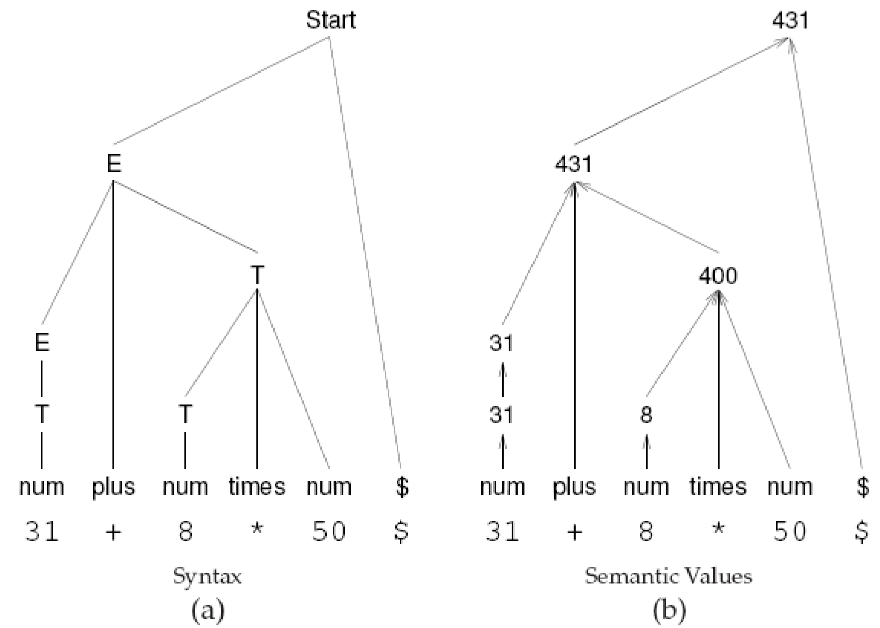



Figure 7.1: (a) Parse tree for the displayed expression;
(b) Synthesized attributes transmit values up the parse tree toward the root.

# Abstract Syntax Tree: data structure requirements

- The AST structure is constructed bottom-up:
  - A list of siblings is generated.
  - The list later is adopted by a parent.
- The AST structure should allow for adding of siblings at either end of the list.
- Some AST nodes require a fixed number of children
  - Arithmetic operators
  - if-then-else statement
- Some AST nodes require zero or more number of children
  - Parameter lists
  - Statements in a statement block
- In order to be generally applicable, AST nodes should allow for any number of children.

# Abstract Syntax Tree: data structure design

- According to depth-first-search tree traversal.

- Each node needs connection to:
  - **Parent**: to migrate information upwards in the tree
    - Link to parent
  - **Siblings**: to iterate through (1) a list of operands or (2) members of a group, e,g, members of a class, or statements.
    - Link to right sibling (thus creating a single linked list of siblings)
    - Link to leftmost sibling (in case one needs to traverse the list as a sibling is being processed).
  - **Children**: to generate/traverse the tree
    - Link to leftmost child (who represents the head of a linked list of children).

# Abstract Syntax Tree: data structure design

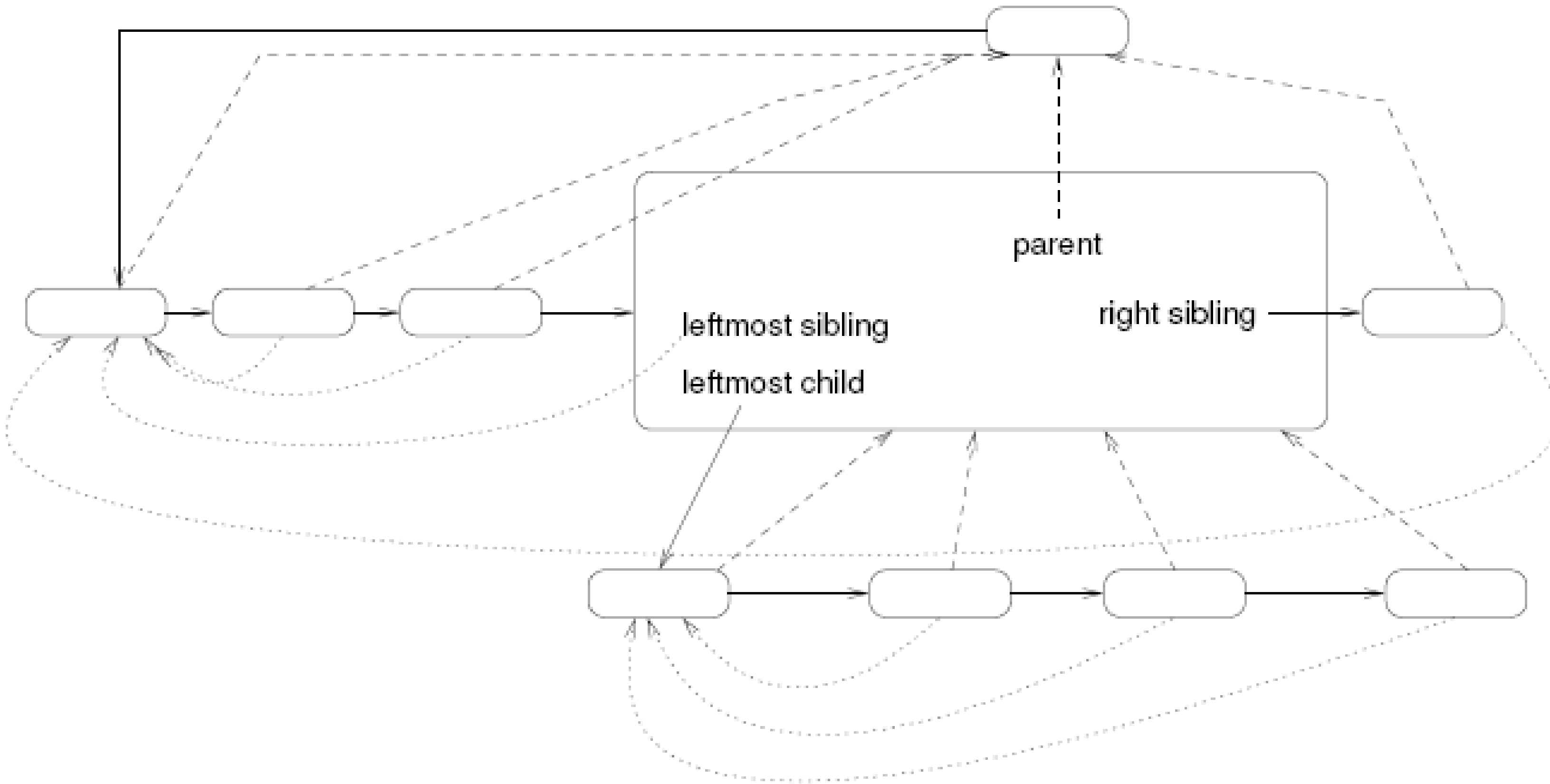



Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

# Abstract Syntax Tree: data structure implementation

- **`makeNode(t)`**

  factory method that creates/returns a node whose members are adapted to the type of the parameter t. For example:

  - **`makeNode(intNum i)`**: instantiates a node that represents a numeric literal value. Offers a get method to get the value it represents.
  - **`makeNode(id n)`**: instantiates a node that represents an identifier. Offers get/set methods to get/set the symbol table entry it represents, which stores information such as its type/protection/scope.
  - **`makeNode(op o)`**: instantiates a node that represents composite structures such as operators, statements, or blocks. There should be one for each such possible different nodes for each different kind of composite structures in the language. Each offers get/set methods appropriate to what they represent.
  - **`makeNode()`**: instantiates a null node in order to represent, e.g. the end of siblings list.

# Abstract Syntax Tree: data structure implementation

- **x.makeSiblings(y)**

  inserts a new sibling node **y** in the list of siblings of node **x**.

```
function makeSiblings( y ) returns Node
    /* Find the rightmost node in this list */
    xsibs ← this
    while xsibs.rightSib ≠ null do xsibs ← xsibs.rightSib
    /* Join the lists */
    ysibs ← y.leftmostSib
    xsibs.rightSib ← ysibs
    /* Set pointers for the new siblings */
    ysibs.leftmostSib ← xsibs.leftmostSib
    ysibs.parent ← xsibs.parent
    while ysibs.rightSib ≠ null do
        ysibs ← ysibs.rightSib
        ysibs.leftmostSib ← xsibs.leftmostSib
        ysibs.parent ← xsibs.parent
    return (ysibs)
end
```

- **x.adoptChildren(y)**

  adopts node **y** and all its siblings under the parent **x**.

```
function adoptChildren( y ) returns Node
    if this.leftmostChild ≠ null
    then this.leftmostChild.makeSiblings( y )
    else
        ysibs ← y.leftmostSib
        this.leftmostChild ← ysibs
        while ysibs ≠ null do
            ysibs.parent ← this
            ysibs ← ysibs.rightSib
end
```

# Abstract Syntax Tree: data structure implementation

- **makeFamily(op, $kid_1$, $kid_2$, …, $kid_n$)**: generates a family with n children under a parent **op**. For example:

```
function MAKEFAMILY(op, kid1, kid2) returns Node
    return (makeNode(op).ADOPTCHILDREN(kid1.MAKESIBLINGS(kid2)))
end
```

- One such function to create each kind of sub-tree, or one single variadic function.

- Some (many) programming languages do not allow variadic functions.

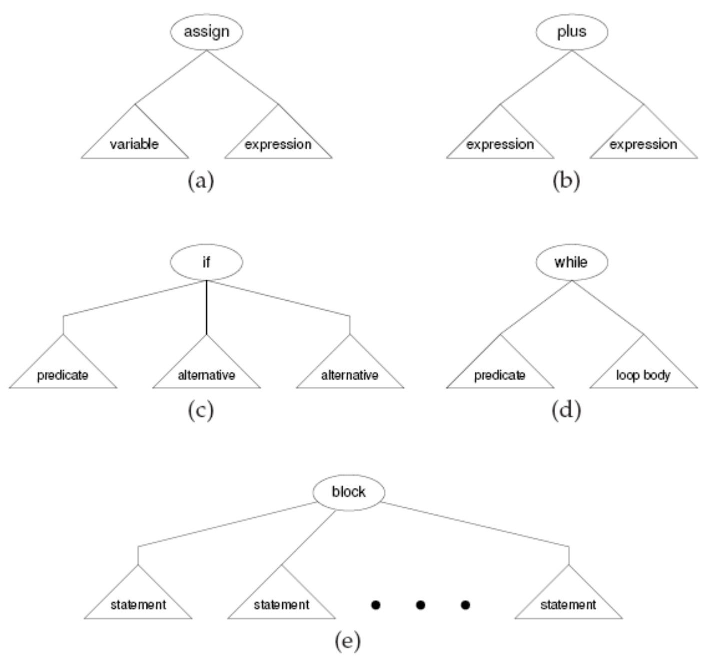



Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

# Insert semantic actions in the grammar/parser

- Example simple grammar:

```
 1  Start  → Stmt $
 2  Stmt   → id assign E
 3         | if lparen E rparen Stmt else Stmt fi
 4         | if lparen E rparen Stmt fi
 5         | while lparen E rparen do Stmt od
 6         | begin Stmts end
 7  Stmts  → Stmts semi Stmt
 8         | Stmt
 9  E      → E plus T
10         | T
11  T      → id
12         | num
```

# Insert semantic actions in the grammar/parser

- Example grammar with semantic actions added.
- AST leaf nodes are created when the parse reaches leaves in the parse tree (23, 24) (**`makeNode`**).
- Siblings lists are constructed as lists are processed inside a structure (19) (**`makeSiblings`**).
- Subtrees are created when an entire structure has been parsed (14, 15, 16, 17, 18, 21) (**`makeFamily`**).
- Some semantic actions are only migrating information across the tree (20).

| | | | |
|---|---|---|---|
| 1 | Start | → $\text{Stmt}_{ast}$ \$ | |
| | | **return** (*ast*) | (13) |
| 2 | $\text{Stmt}_{result}$ | → $\text{id}_{var}$ assign $\text{E}_{expr}$ | |
| | | *result* ← MAKEFAMILY(assign, *var*, *expr*) | (14) |
| 3 | | \| if lparen $\text{E}_p$ rparen $\text{Stmt}_s$ fi | |
| | | *result* ← MAKEFAMILY(if, *p*, *s*, MAKENODE( )) | (15) |
| 4 | | \| if lparen $\text{E}_p$ rparen $\text{Stmt}_{s1}$ else $\text{Stmt}_{s2}$ fi | |
| | | *result* ← MAKEFAMILY(if, *p*, *s1*, *s2*) | (16) |
| 5 | | \| while lparen $\text{E}_p$ rparen do $\text{Stmt}_s$ od | |
| | | *result* ← MAKEFAMILY(while, *p*, *s*) | (17) |
| 6 | | \| begin $\text{Stmts}_{list}$ end | |
| | | *result* ← MAKEFAMILY(block, *list*) | (18) |
| 7 | $\text{Stmts}_{result}$ | → $\text{Stmts}_{sofar}$ semi $\text{Stmt}_{next}$ | |
| | | *result* ← *sofar*.MAKESIBLINGS(*next*) | (19) |
| 8 | | \| $\text{Stmt}_{first}$ | |
| | | *result* ← *first* | (20) |
| 9 | $\text{E}_{result}$ | → $\text{E}_{e1}$ plus $\text{T}_{e2}$ | |
| | | *result* ← MAKEFAMILY(plus, *e1*, *e2*) | (21) |
| 10 | | \| $\text{T}_e$ | |
| | | *result* ← *e* | (22) |
| 11 | $\text{T}_{result}$ | → $\text{id}_{var}$ | |
| | | *result* ← MAKENODE(*var*) | (23) |
| 12 | | \| $\text{num}_{val}$ | |
| | | *result* ← MAKENODE(*val*) | (24) |

# Example: parse tree

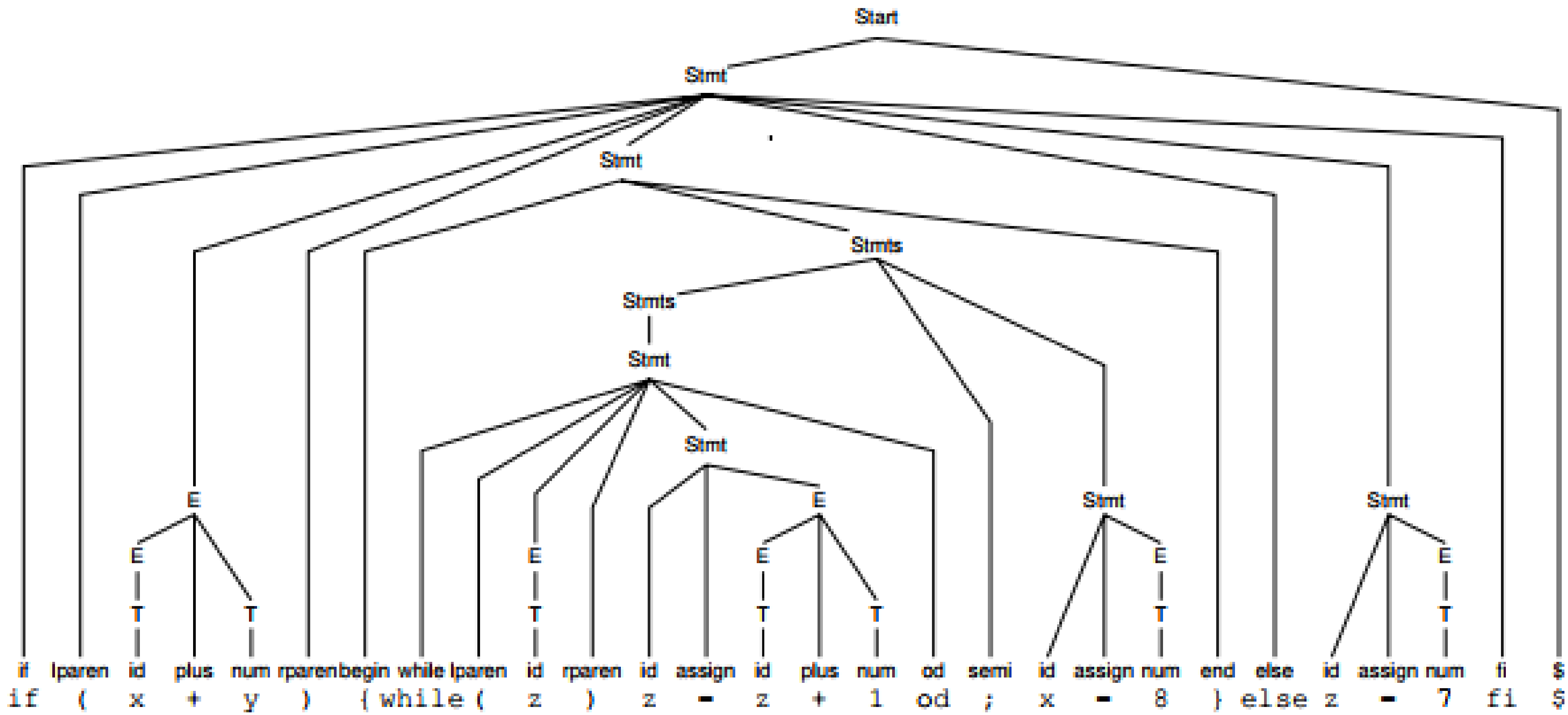



Figure 7.18: Concrete syntax tree.

# Example: corresponding AST

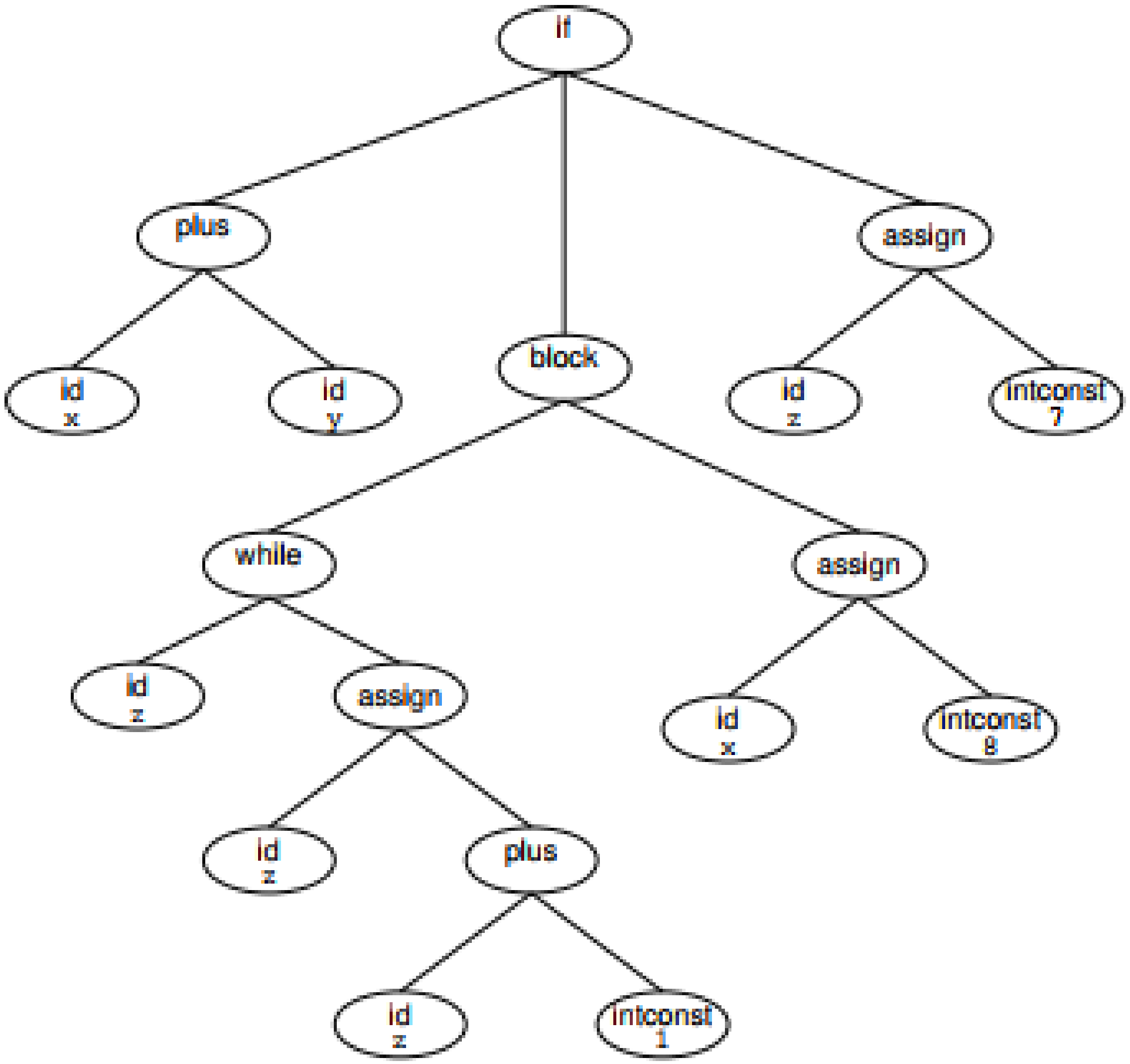



Figure 7.19: AST for the parse tree in Figure 7.18.

# AST generation: implementation in recursive-descent predictive parser

```
Parse(){
  AST Es                               //blank AST created
                                       //before the call
  lookahead = NextToken()
  if (E(Es);Match('$'))                //passed as a reference
                                       //to parsing functions
                                       //that will create the tree
    return(true);
  else
    return(false);
}
```

- **AST** variables represents tree nodes that are created, migrated and grafted/adopted in order to construct an abstract syntax tree.

# AST generation: implementation in recursive-descent predictive parser

```
E(AST &Es){
  AST Ts,E's
  if (lookahead is in [0,1,(])
    if (T(Ts);E'(Ts,E's);)        // E' inherits Ts from T
      write(E->TE')
      Es = E's                     // Synthetised attribute sent up
      return(true)                 // by way of the Es reference
    else                           // parameter of E()
      return(false)
  else
    return(false)
}
```

- Each parsing function potentially (i.e. not all of them) defines its own AST nodes used locally the processing of its own subtree.
- **Ts,E's** are ASTs produced/used by the **T()** and **E'()** functions and returned by them to the **E()** function.

# AST generation: implementation in recursive-descent predictive parser

```
E’(AST &Ti, type &E's){
  AST Ts,E'2s
  if (lookahead is in [+])
    if (Match('+');T(Ts);E'(Ts,E'2s))    // (3) E' inherits Ts from T
      write(E'->TE')
      E's = makeFamily(+,Ti,E'2s)        // (1) AST subtree creation
      return(true)                       // sent up in the parse tree
    else                                 // by way of the E’s parameter
      return(false)
  else if (lookahead is in [$,)]
    write(E'->epsilon)
    E's = Ti                             // (2) Synth. attr. is inherited
    return(true)                         // from T (sibling, not child)
  else                                   // and sent up
    return(false)
}
```

- Some semantic actions will do some semantic checking and/or semantic aggregation, such as a tree node adopting a child node, or inferring the type of an expression from two child operands (1).
- Some semantic actions are simply migrating an AST subtree upwards in the parse tree (2), or sideways to a sibling tree (3).

# AST generation: implementation in recursive-descent predictive parser

```
T(AST &Ts){
  AST Fs, T's
  if (lookahead is in [0,1,(])
    if (F(Fs);T'(Fs,T's);)        // T' inherits Fs from F
      write(T->FT')
      Ts = T's                    // Synthetized attribute sent up
      return(true)
    else
      return(false)
  else
    return(false)
}
```

# AST generation: implementation in recursive-descent predictive parser

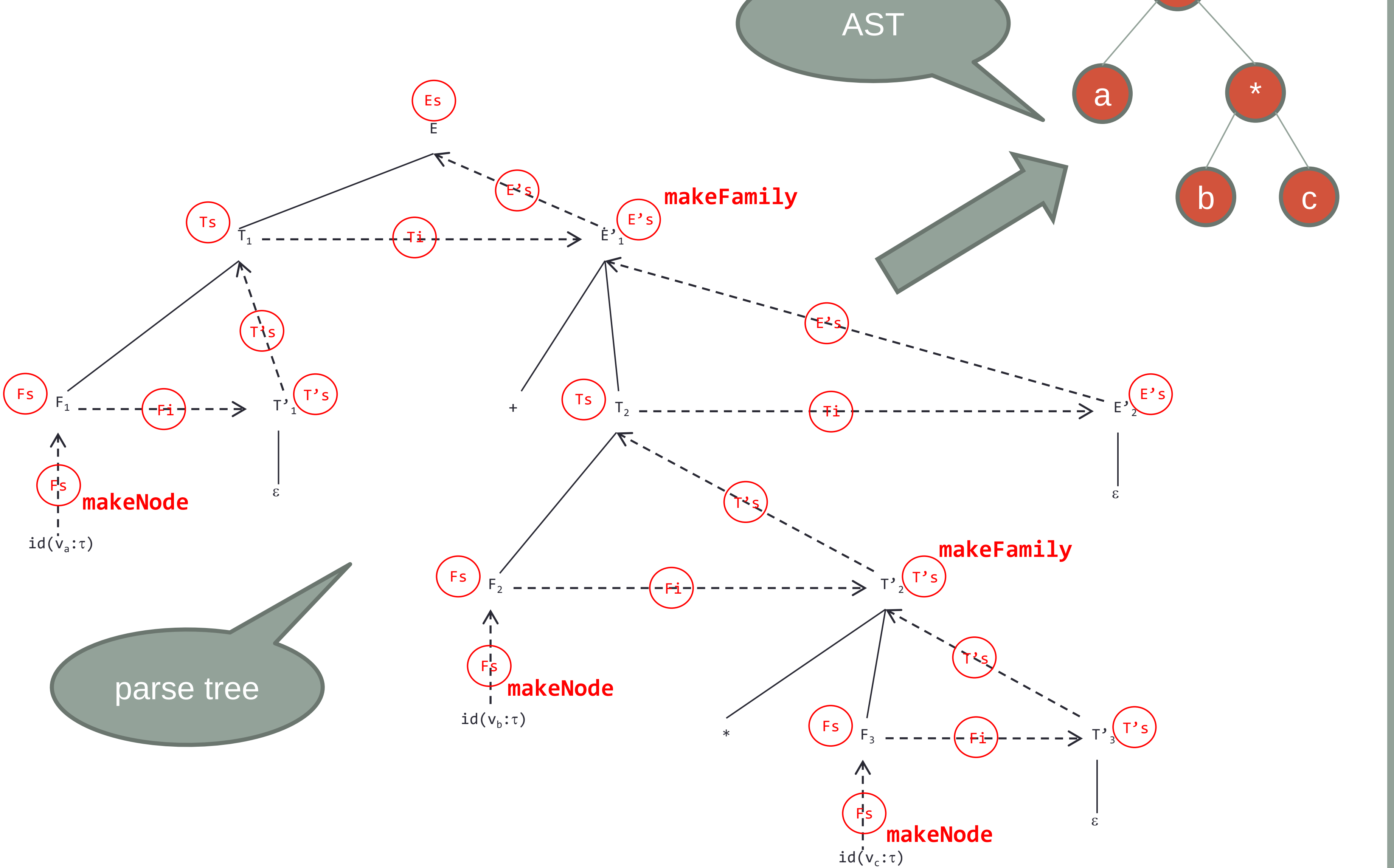
```
T'(AST &Fi, type &T's){
  AST Fs, T'2s
  if (lookahead is in [*])
    if (Match('*');F(Fs);T'(Fs,T'2s))    // T' inherits Fs from F
      write(T'->*FT')
      T's = makeFamily(*,Fi,T'2s)        // AST subtree creation
      return(true)                       // using left operand migrated
    else                                 // from left sibling parse tree
      return(false)                      // received as Fi parameter
  else if (lookahead is in [+,$,)]
    write(T'->epsilon)
    T's = Fi                             // Synthetized attribute is
                                         // inhertied from F sibling
                                         // and sent up the tree
    return(true)
  else
    return(false)
}
```

# AST generation: implementation in recursive-descent predictive parser

```
F(AST &Fs){
  AST Es
  if (lookahead is in [id])
    if (Match('id'))
      write(F->id)
      Fs = makeNode(id)                       // create a leaf node
      return(true)                            // and send it up the parse tree
    else
      return(false)
  else if (lookahead is in [(])
    if (Match('(');E(Es);Match(')'))
      write(F->(E))
      Fs = Es                                 // Synthetized attribute from E
      return(true)                            // i.e. AST of whole expression
    else return(false)                        // sent up in the parse tree
  else return(false)                          // as AST subtree representing
}                                             // the '(E)' successfully parsed
```

# Attribute migration: example

a+b*c

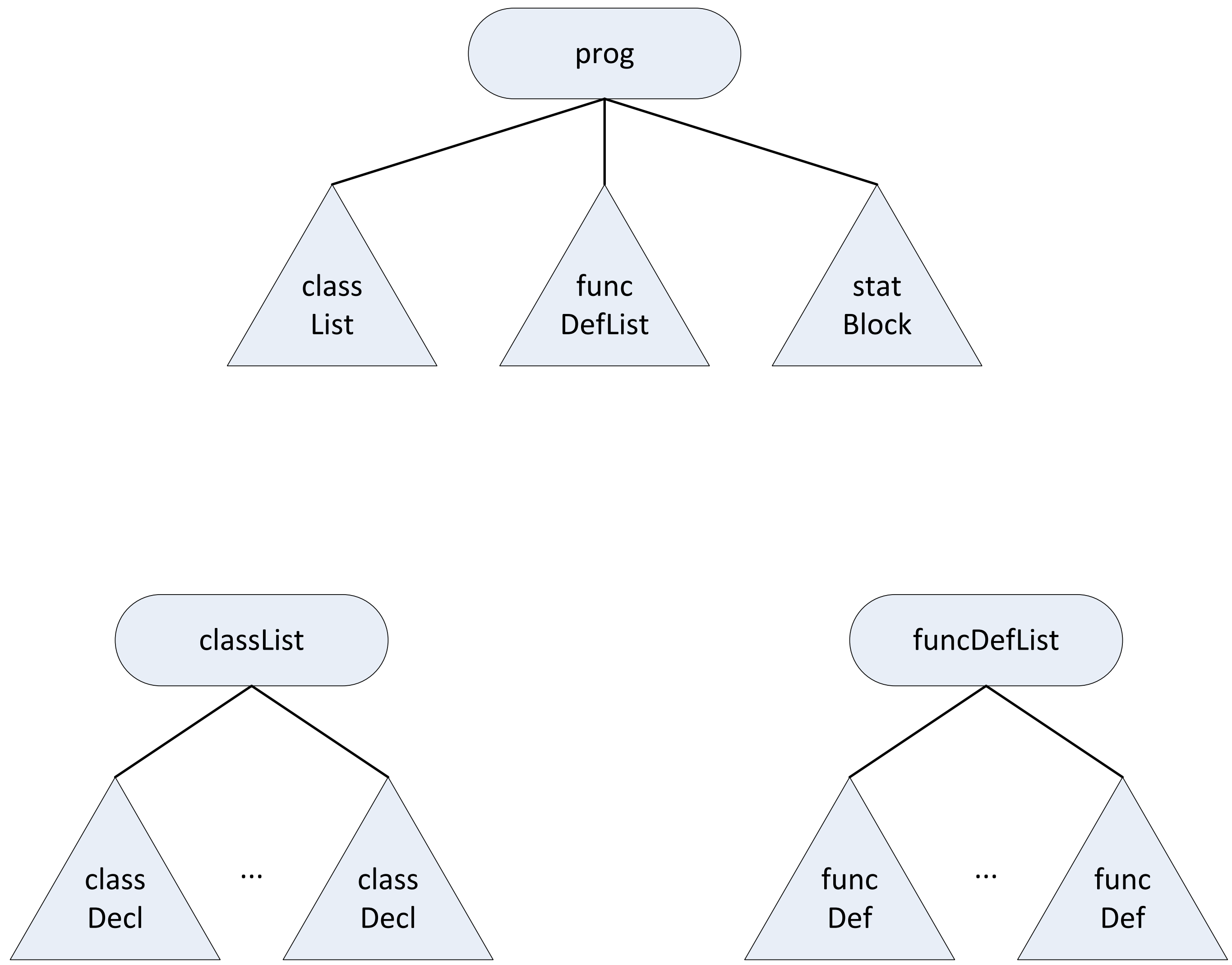


# Abstract Syntax Tree structural elements

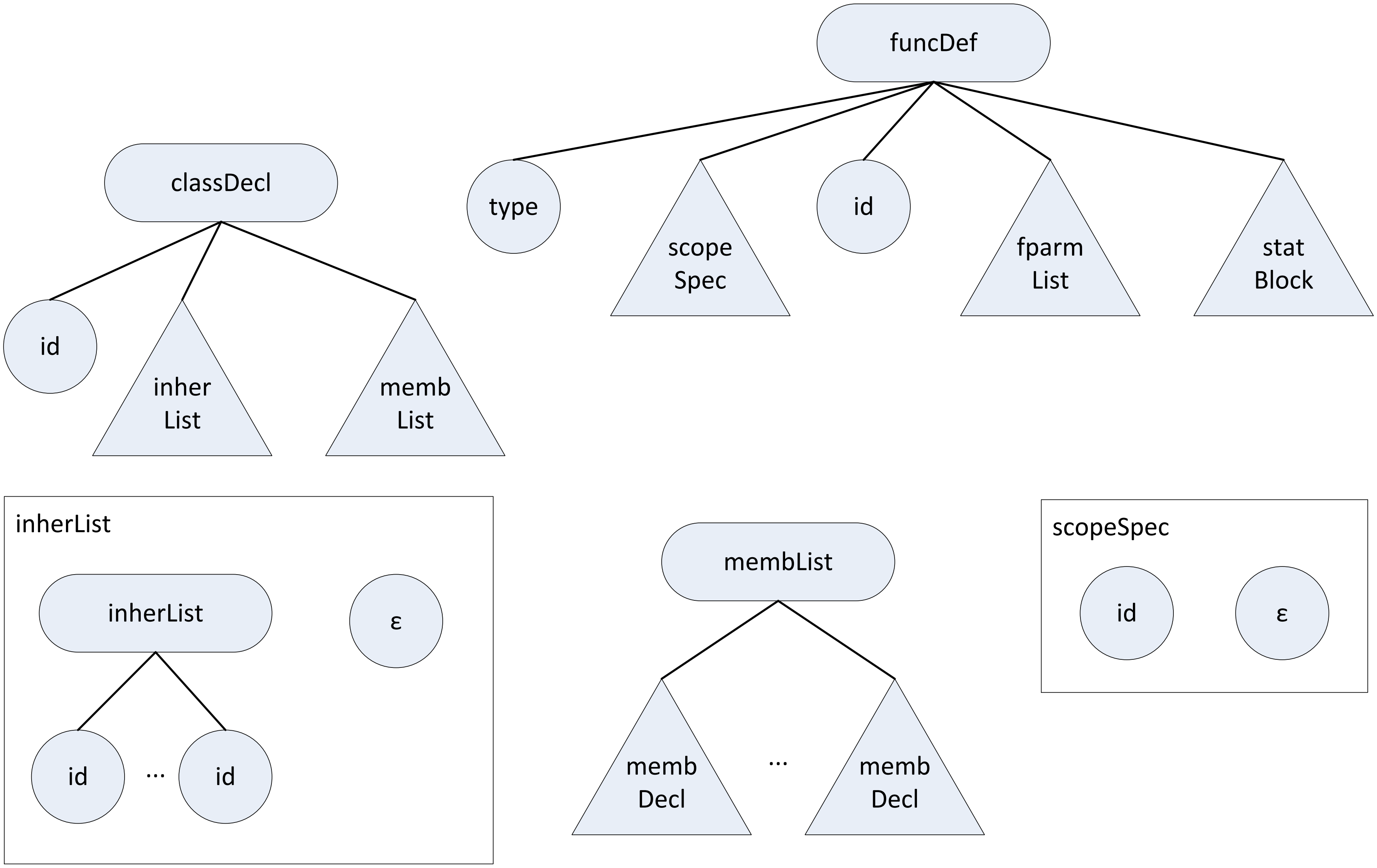
# Grammar

```
prog         -> {classDecl} {funcDef} 'program' funcBody ';'
classDecl    -> 'class' 'id' [':' 'id' {',' 'id'}] '{' {varDecl} {funcDecl} '}' ';'
funcDecl     -> type 'id' '(' fParams ')' ';'
funcHead     -> type ['id' 'sr'] 'id' '(' fParams ')'
funcDef      -> funcHead funcBody ';'
funcBody     -> '{' {varDecl} {statement} '}'
varDecl      -> type 'id' {arraySize} ';'
statement    -> assignStat ';'
              | 'if'     '(' expr ')' 'then' statBlock 'else' statBlock ';'
              | 'for'    '(' type 'id' assignOp expr ';' relExpr ';' assignStat ')' statBlock ';'
              | 'get'    '(' variable ')' ';'
              | 'put'    '(' expr ')' ';'
              | 'return' '(' expr ')' ';'
assignStat   -> variable assignOp expr
statBlock    -> '{' {statement} '}' | statement | EPSILON
expr         -> arithExpr | relExpr
relExpr      -> arithExpr relOp arithExpr
arithExpr    -> arithExpr addOp term | term
sign         -> '+' | '-'
term         -> term multOp factor | factor
factor       -> variable
              | functionCall
              | 'intNum' | 'floatNum'
              | '(' arithExpr ')'
              | 'not' factor
              | sign factor
variable     -> {idnest} 'id' {indice}
functionCall -> {idnest} 'id' '(' aParams ')'
idnest       -> 'id' {indice} '.'
              | 'id' '(' aParams ')' '.'
indice       -> '[' arithExpr ']'
arraySize    -> '[' 'intNum' ']'
type         -> 'int' | 'float' | 'id'
fParams      -> type 'id' {arraySize} {fParamsTail} | EPSILON
aParams      -> expr {aParamsTail} | EPSILON
fParamsTail  -> ',' type 'id' {arraySize}
aParamsTail  -> ',' expr
assignOp     -> '='
relOp        -> 'eq' | 'neq' | 'lt' | 'gt' | 'leq' | 'geq'
addOp        -> '+' | '-' | 'or'
multOp       -> '*' | '/' | 'and'
```

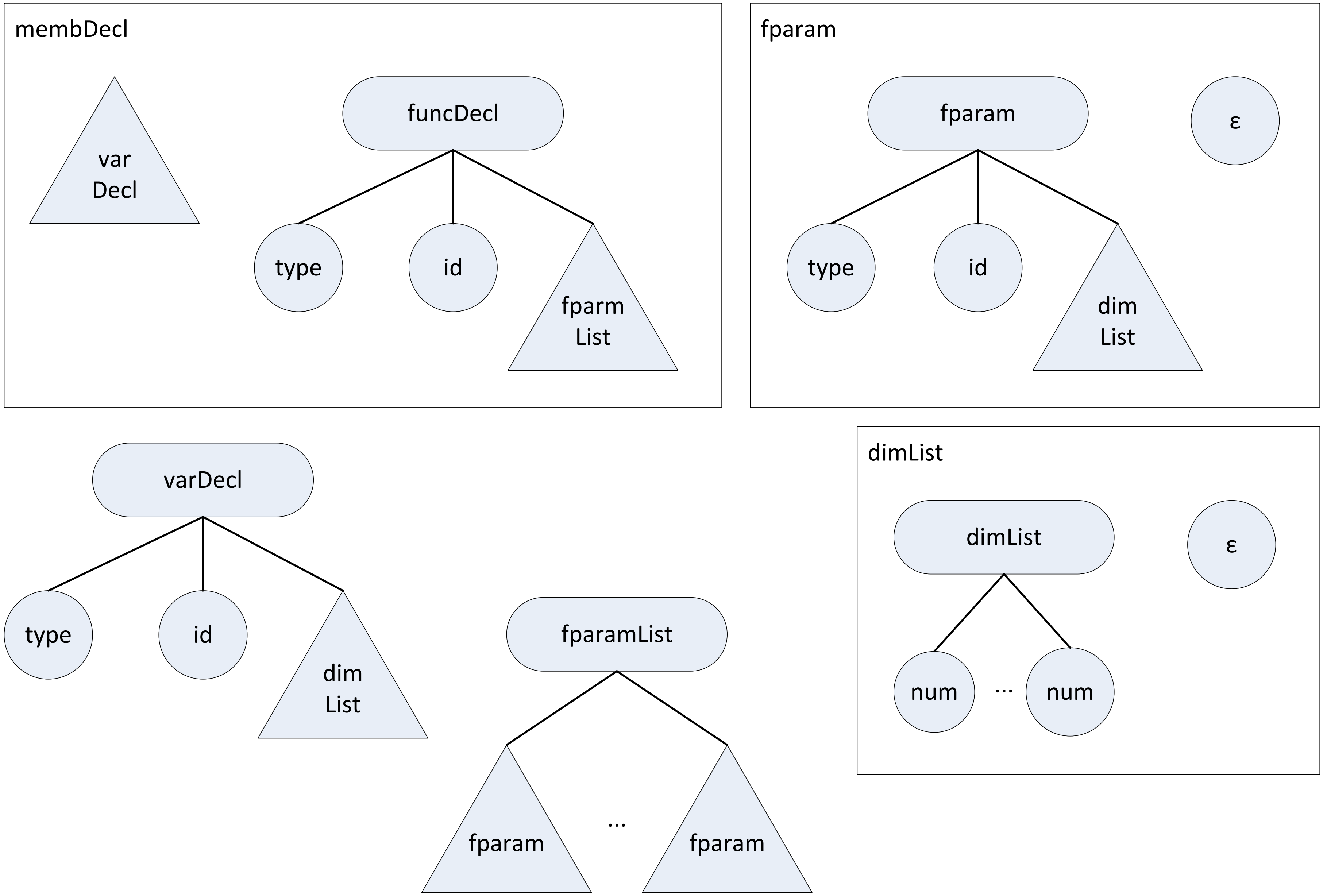
# Abstract Syntax Tree structural elements



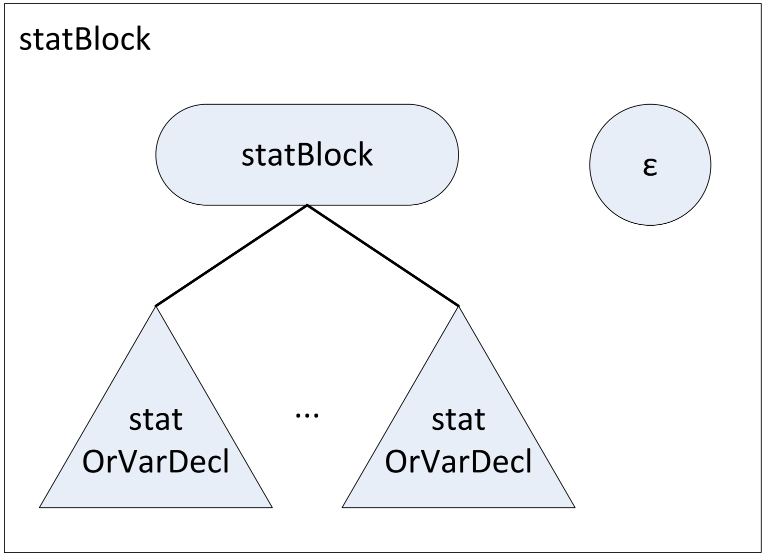
# Abstract Syntax Tree structural elements

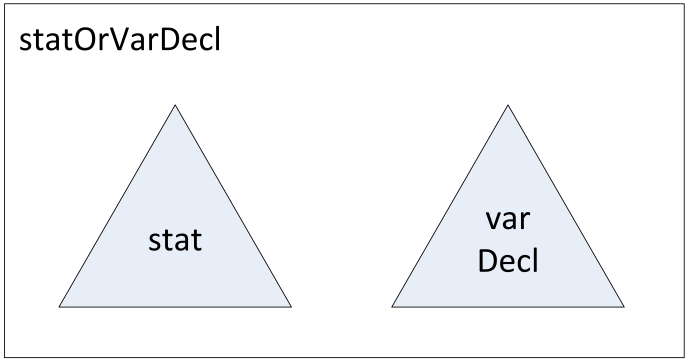


# Abstract Syntax Tree structural elements

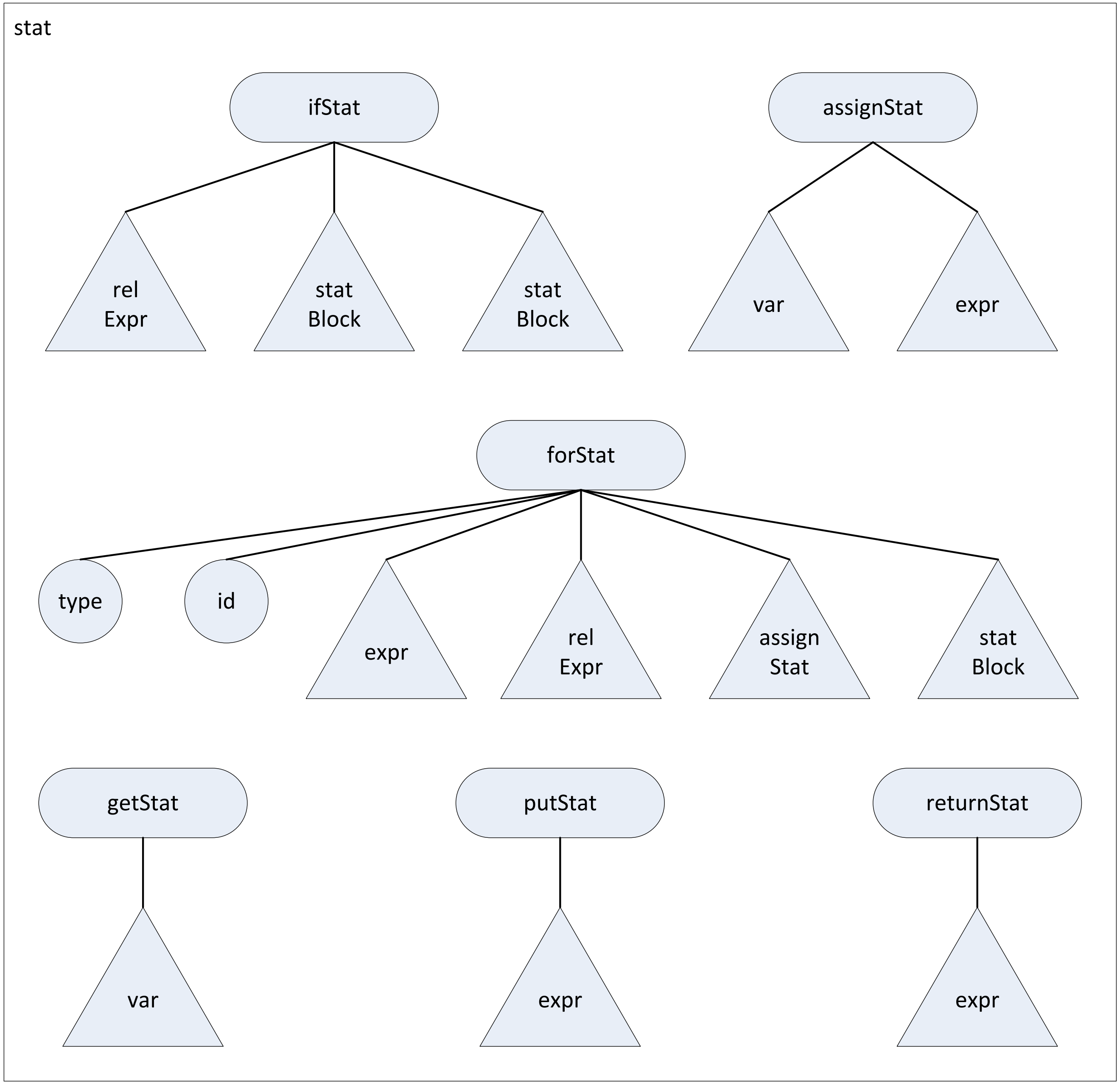


# Abstract Syntax Tree structural elements

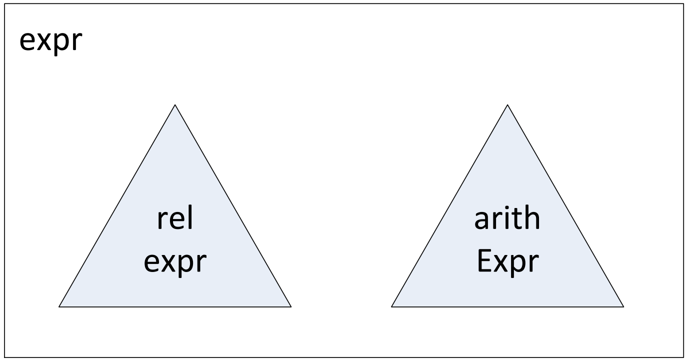


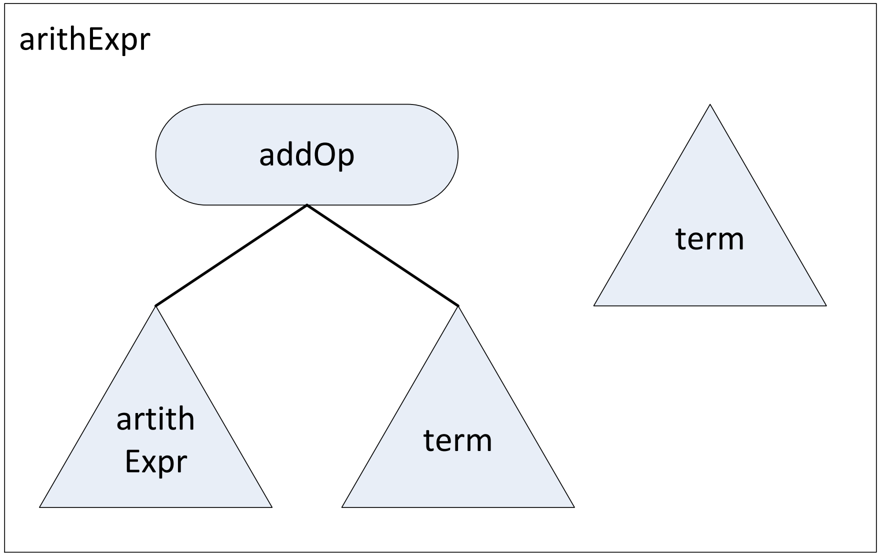



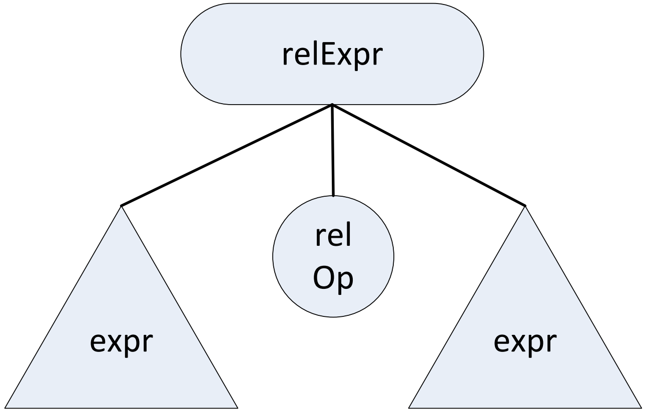
# Abstract Syntax Tree structural elements



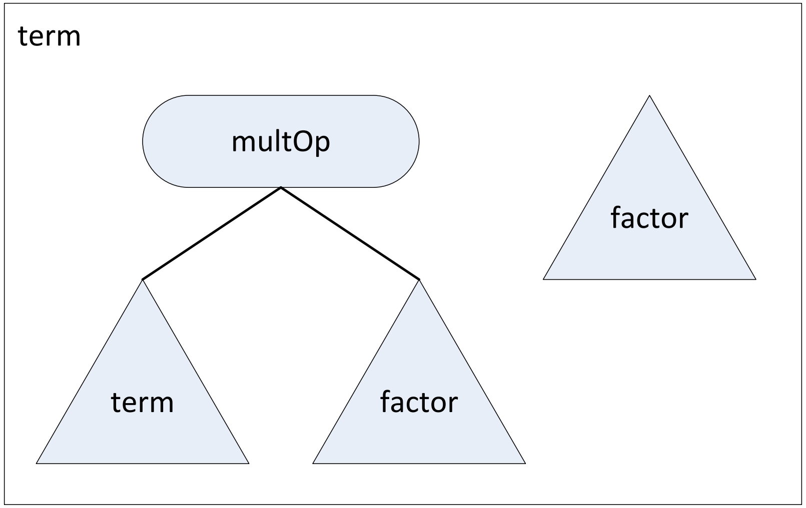
# Abstract Syntax Tree structural elements

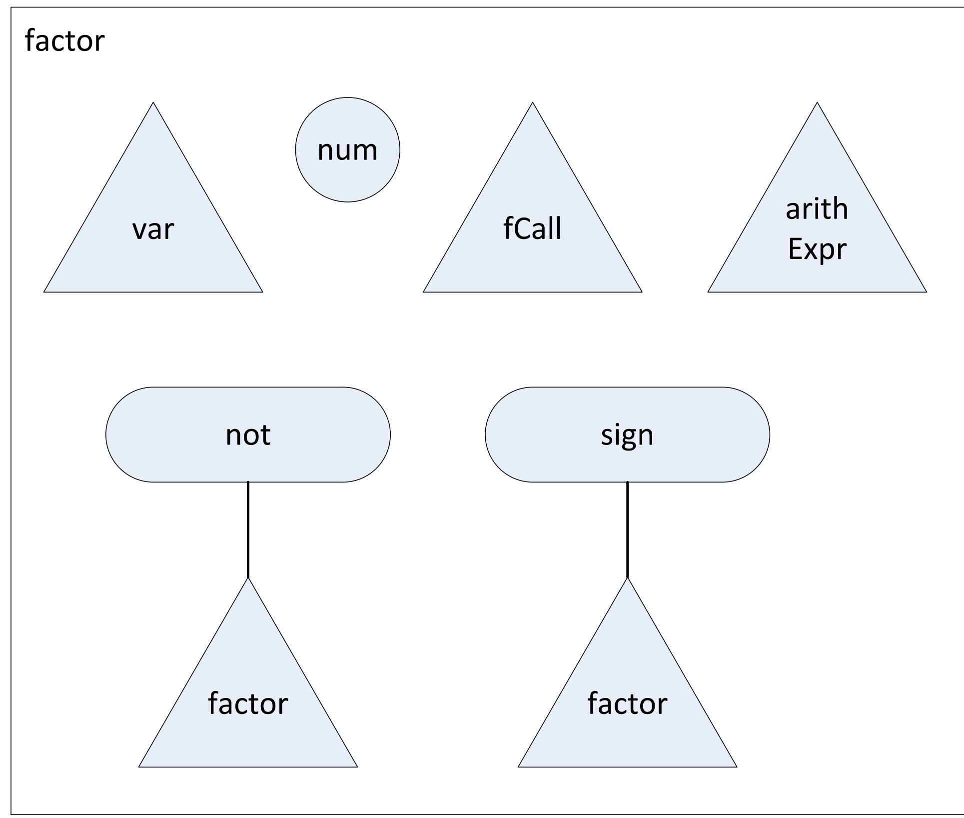


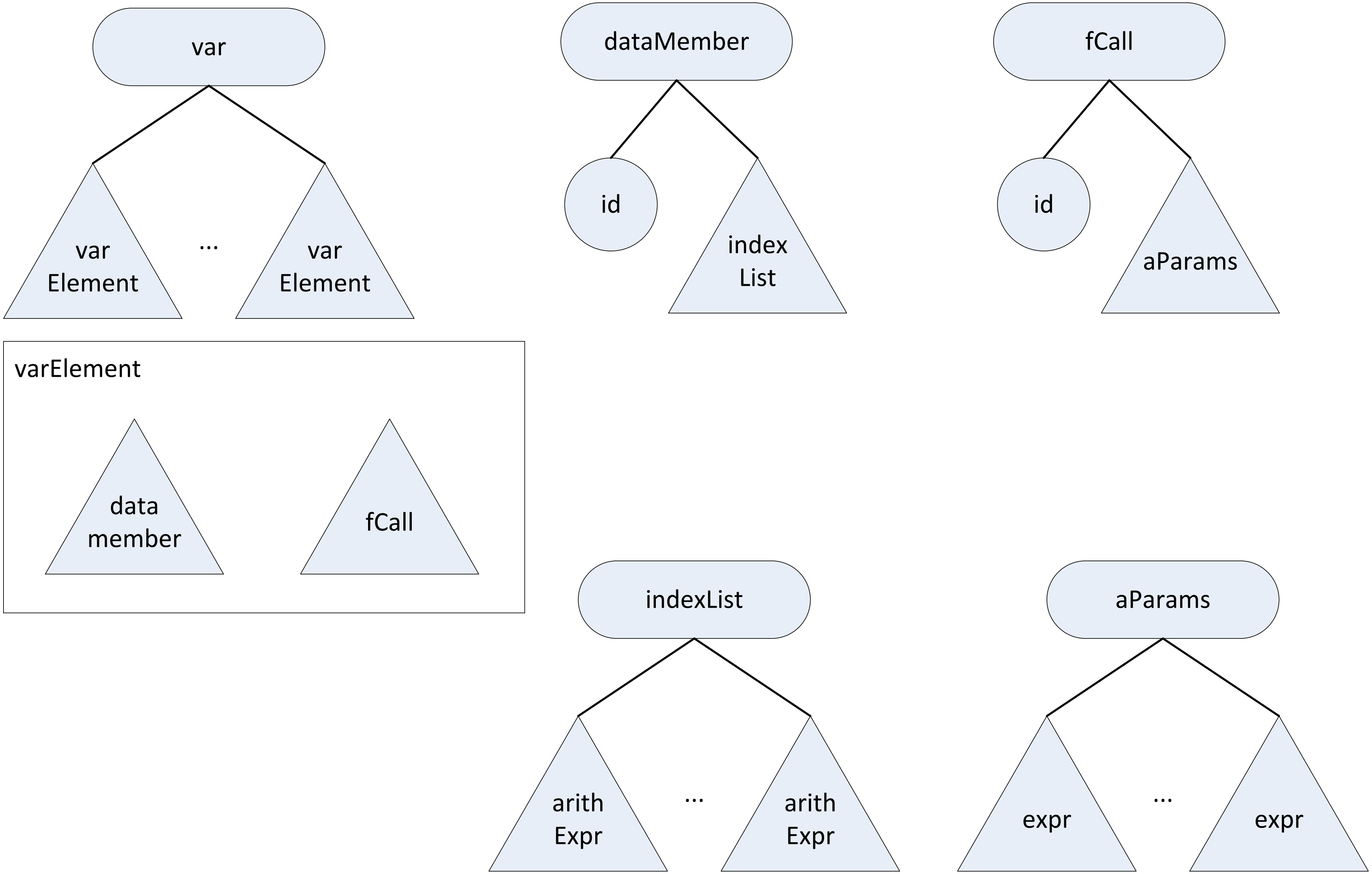



# Abstract Syntax Tree structural elements



# Abstract Syntax Tree structural elements



# References


- Wikipedia. Abstract Syntax Tree.
- C.N. Fischer, R.K. Cytron, R.J. LeBlanc Jr., Crafting a Compiler, Adison-Wesley, 2009. Chapter 7.