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 syntactic construct occurring in the source code.
- The syntax is "abstract" i.e. it does not represent every detail appearing in the *concrete syntax* used in the source code, or some of the non-terminals in the grammar that do not directly represent syntactical/semantic constructs.
- Such details are removed because they do not convey any form of meaning and are thus superfluous for further processing.

- For instance:
  - punctuation such as commas, semicolons, and grouping parentheses are removed
  - syntactic construct like an `if-then-else` may be denoted by means of a single node with three branches
  - non-terminals that were introduced as accessory to grammar transformations are removed

- This distinguishes *abstract syntax trees* from *concrete syntax trees*, which are traditionally designated as *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: Goals

- **Goals**:

(1) to *aggregate* information gathered during the parse in order to get a broader understanding of the meaning of *whole syntactic constructs* in a single subtree.

(2) to represent the entire program in a data structure that can later be *repeatedly traversed* for further analysis and translation steps.

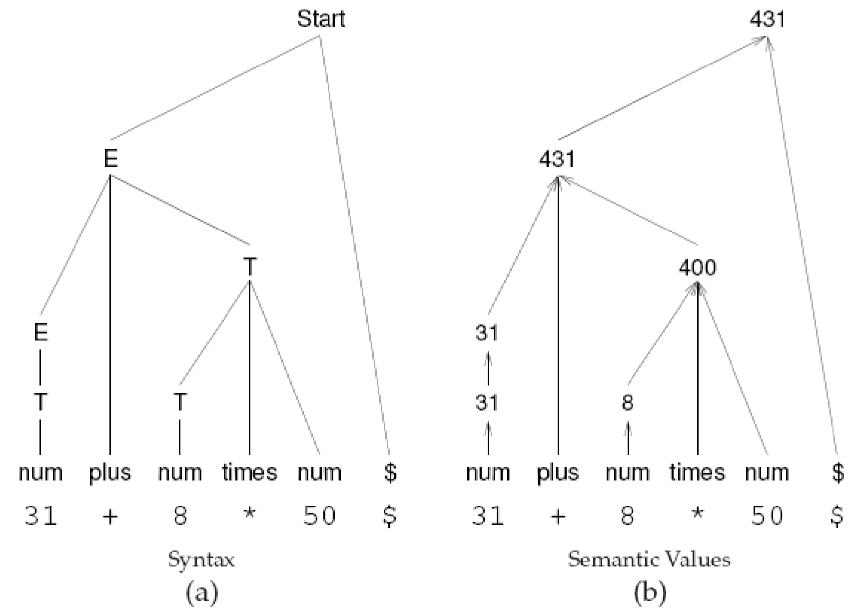


Figure 7.1: (a) Parse tree for the displayed expression; (b) Synthesized attributes transmit values up the parse tree toward the root.

- At the leaves of the tree is fine-grained syntactical concepts/information.
- Intermediate nodes represent higher-level constructs created by aggregation of the information conveyed by its branches' subtrees.
- The root node has direct access to all the information for an entire syntactical construct and its composing constructs.

AST data structure: requirements, design, implementation

Abstract Syntax Tree: data structure requirements

- The AST structure is constructed bottom-up:
  - A set of siblings nodes is generated and each is pushed on a *semantic stack* through the operation of a *semantic action*.
  - The elements are later popped from the semantic stack and adopted by a parent node, which is then pushed onto the stack through the operation of another semantic action.

- Some AST nodes require a fixed number of children, e.g.
  - Arithmetic operators
  - `if-then-else` statement
- Some AST nodes require zero or more number of children
  - Parameter lists, array dimension lists
  - Statements in a statement block
  - Members of a class
- In order to be generally applicable, an AST node data structure should allow for <u>any number of children</u>.

Abstract Syntax Tree: data structure requirements/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 in a statement block.
    - Link to right sibling (thus creating a 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 the linked list of children, which are each other's siblings).

Abstract Syntax Tree: data structure design (example)

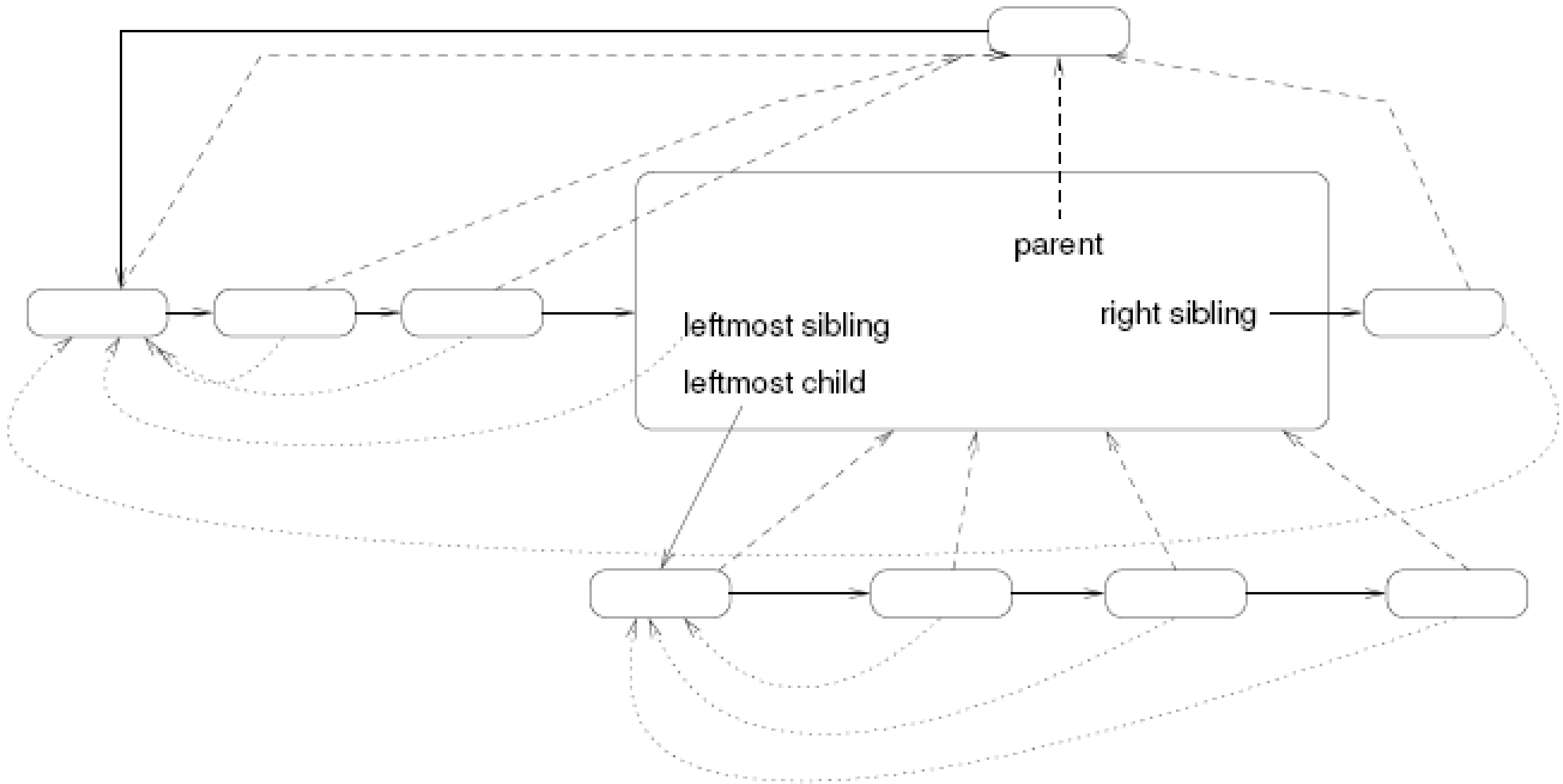


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

Abstract Syntax Tree: data structure implementation

- `makeNode(t)`

  A *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(composite c)`: 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 (example)

- **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**.

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

- One such function exists to create each kind of sub-tree, or one single variadic function.
- Some (many) programming languages do not allow variadic functions.

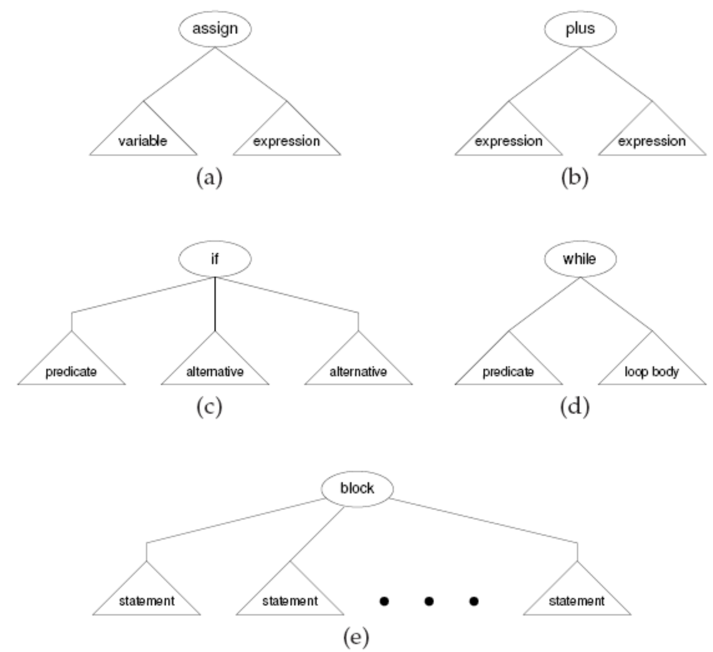


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

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, 22).

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

Example: parse tree

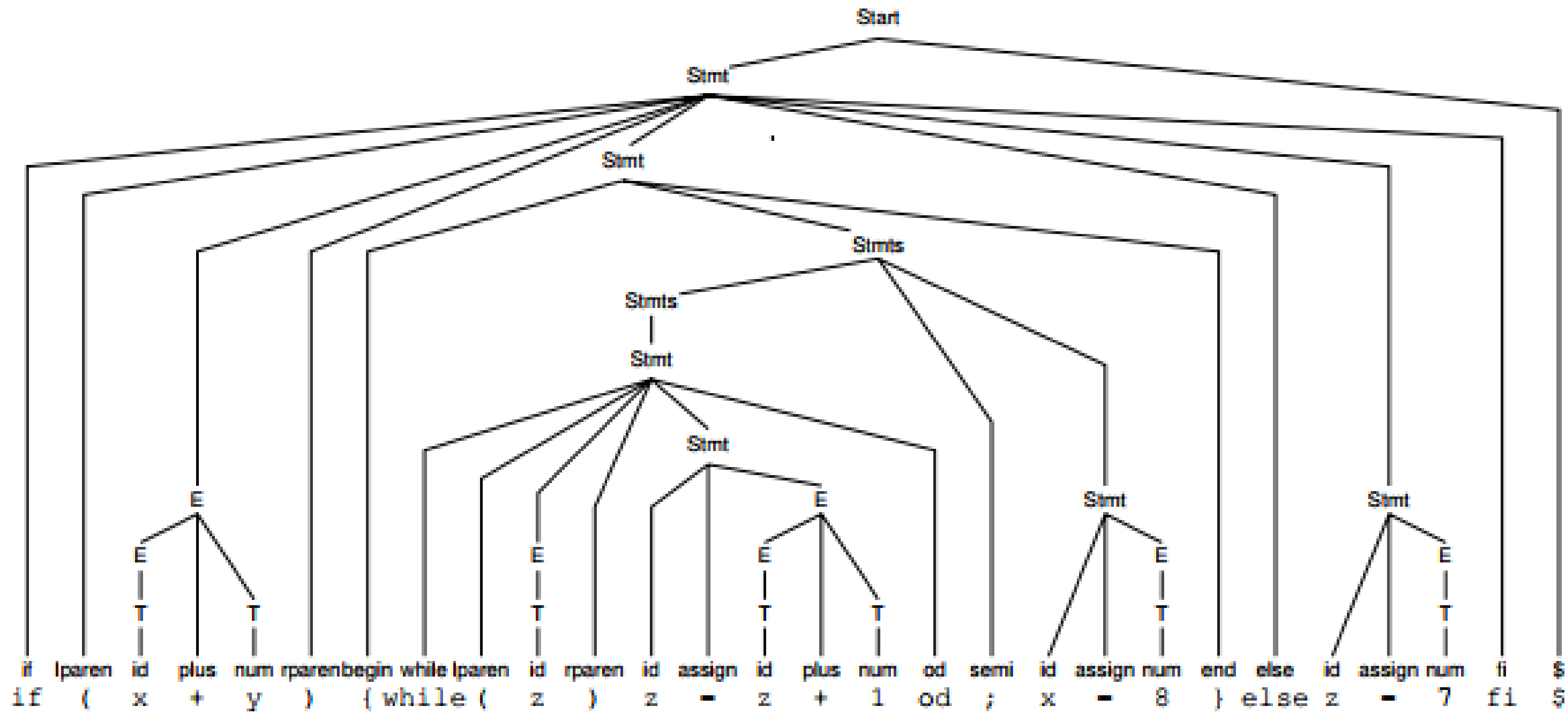


Figure 7.18: Concrete syntax tree.

Example: corresponding AST

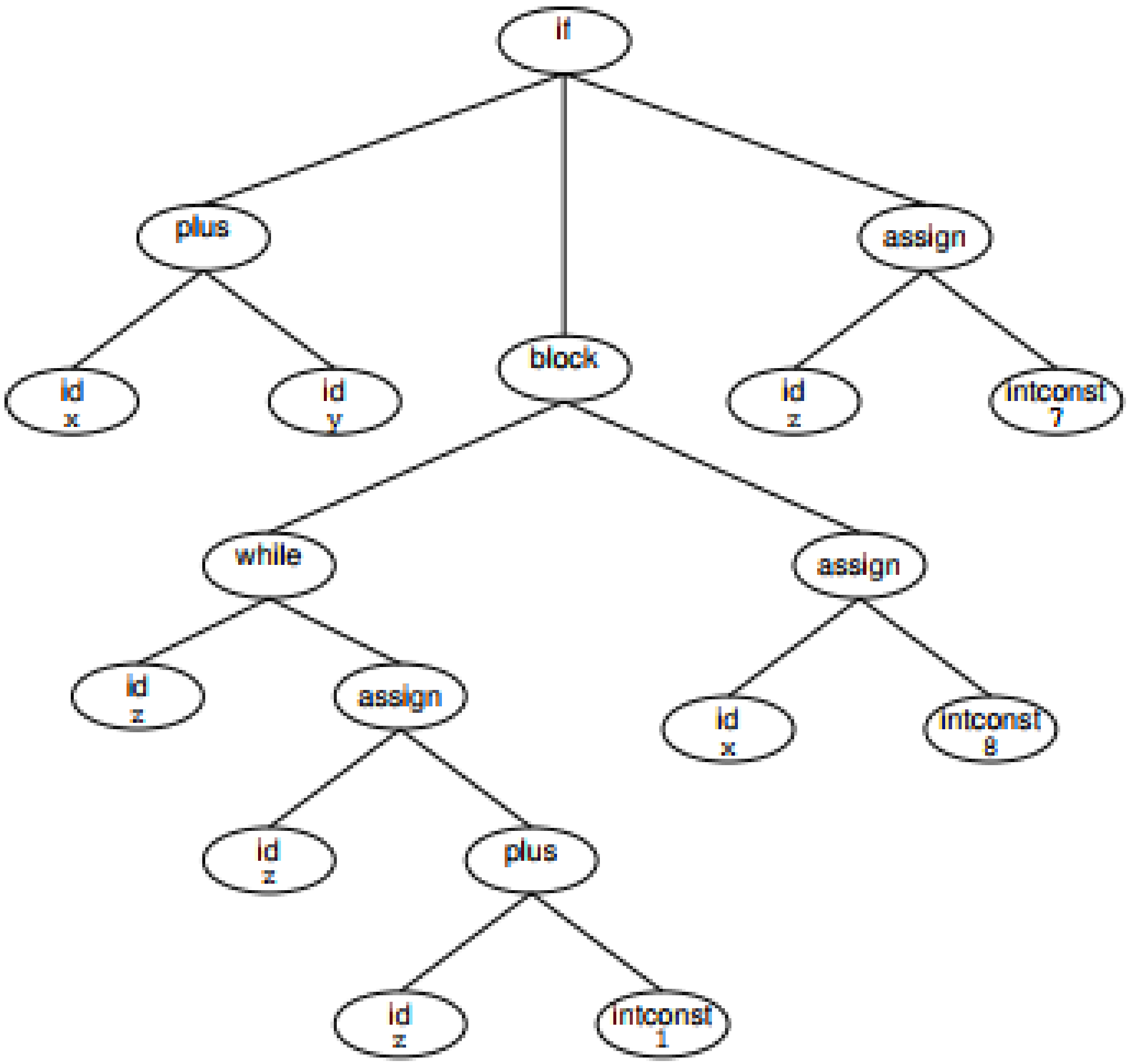


Figure 7.19: AST for the parse tree in Figure 7.18.

AST generation using Syntax-Directed Translation

- A language's ***Semantic Concept*** is a building block of the ***meaning*** of a program
  - literal value, variable, function definition, class and/or data structure, statement, expression, etc.

- In an AST, each concept is represented by a ***node*** and possibly a ***subtree***.

- Atomic concepts (***Ca***) are represented by ***AST leaf nodes*** (***CaN***).
  - Literal value, identifier, etc.

- Composite concepts (***Cc***) represent higher-level concepts that aggregate n subordinate concepts (***Cs***).

- Composite concepts are represented by an ***AST subtree*** (***CcN***) with n AST subtrees as children.
  - class, function definition, statement, expression, etc.

AST generation using Syntax-Directed Translation

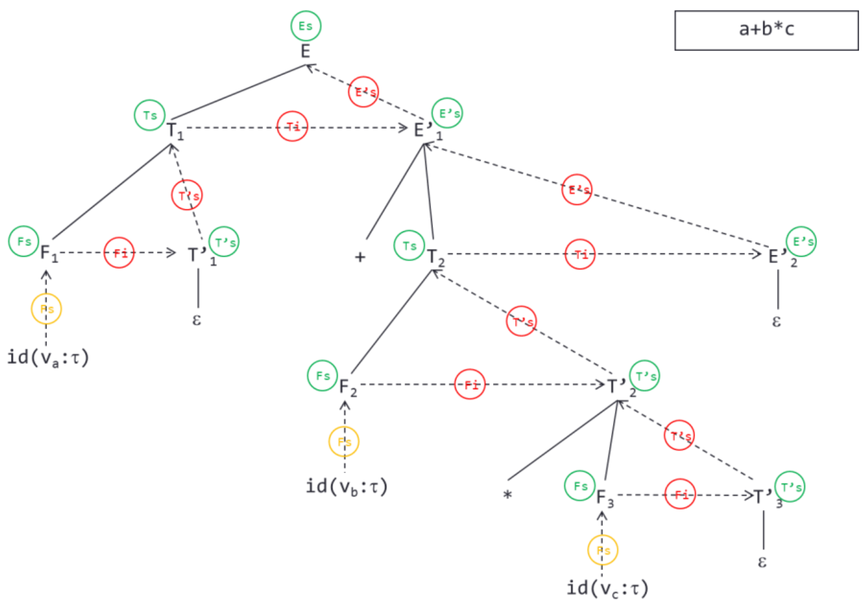
- General Procedure:
  - Upon reaching a parse tree leaf node where semantic information for atomic concept ***Ca*** is present
    - call **`makeNode(Ca)`** to generate an AST node ***CaN*** for atomic concept ***Ca***
    - put the semantic information in ***CaN***
    - push ***CaN*** on the semantic stack
  - As soon as a parsing subtree has gathered all necessary semantic information for composite concept ***Cc***
    - call **`makeNode(Cc)`** to generate an AST node ***CcN*** for composite concept ***Cc***
    - for each subordinate concept ***Cs*** of ***Cc***
      - pop the top of the semantic stack, yielding a node ***CsN*** representing ***Cs***
      - make ***CsN*** a child of ***CcN***
    - push ***CcN*** onto the semantic stack
  - When the parse finishes, the semantic stack should contain only one node representing the full AST of the parsed program structure.

AST creation in a recursive-descent predictive parser – using parameters for migration.

AST generation: recursive-descent predictive parser -- using parameters

```
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);
}
```

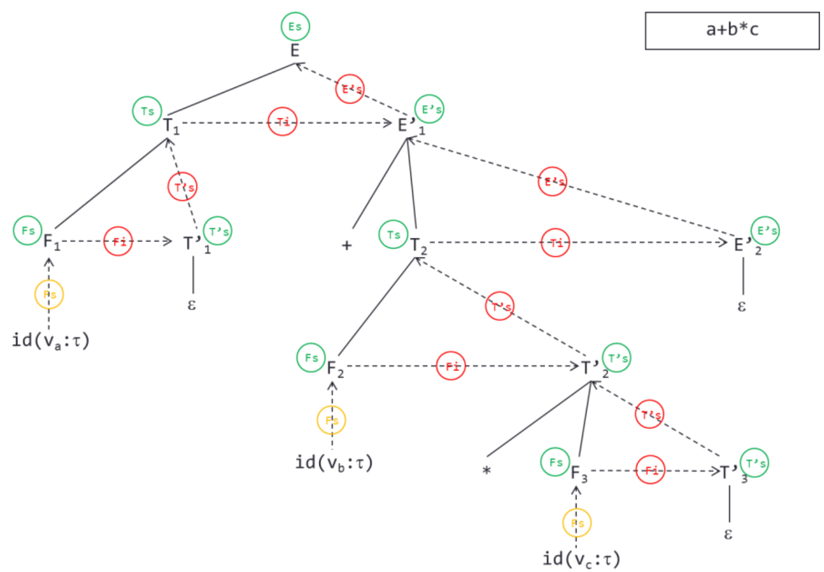
- **AST** variables represents tree nodes that are created, migrated and grafted/adopted in order to construct an abstract syntax tree.



AST generation: recursive-descent predictive parser -- using parameters

```
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 necessarily all of them) defines its own AST nodes used locally that represents its own subtree.
- **Ts,E's** are ASTs produced/used by the **T()** and **E'()** functions and returned by them to the **E()** function.



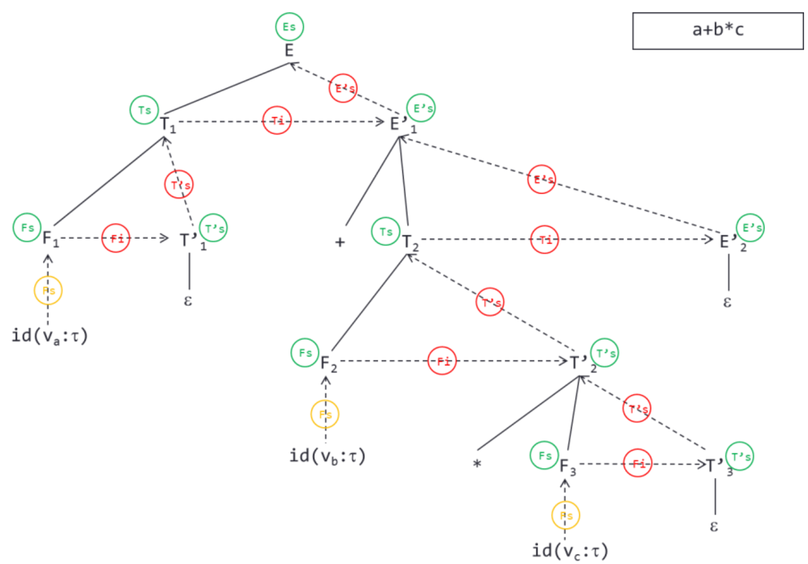
AST generation: recursive-descent predictive parser -- using parameters

```
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: recursive-descent predictive parser -- using parameters

```
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)
}
```



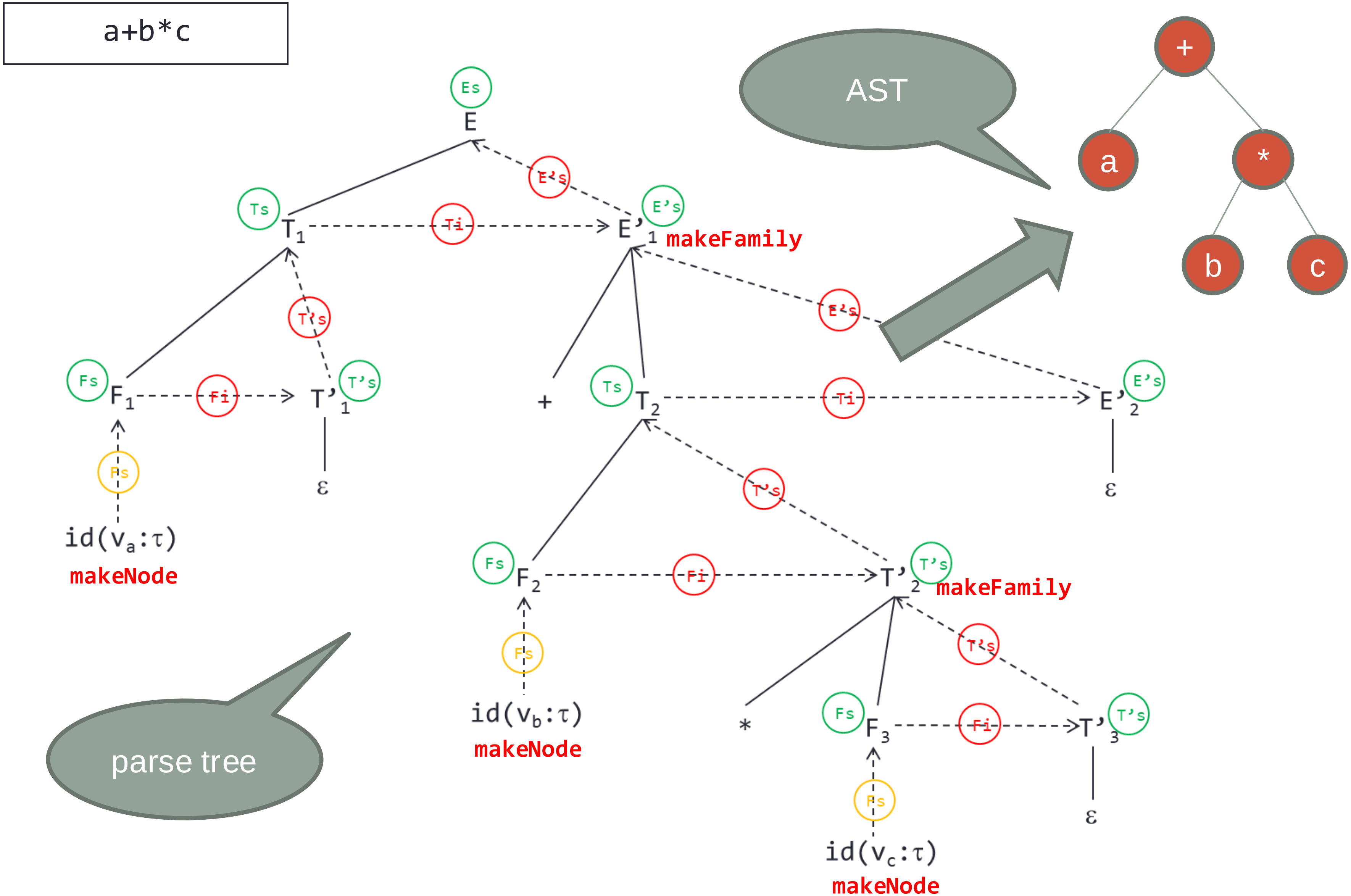
AST generation: recursive-descent predictive parser -- using parameters

```
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: recursive-descent predictive parser -- using parameters

```
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



AST creation in a recursive-descent predictive parser – using a stack for migration.

AST generation: recursive-descent predictive parser -- using stack

```
Parse(){
  ASTnode Es                       // empty semantic record created
  push(Es)                         // and push before the call to E()

                                   // any parsing function that needs
                                   // to hold new semantic information
                                   // creates semantic records and pushes
                                   // them on the stack

  lookahead = NextToken()
  if (E();Match('$'))              // parsing functions expect the
                                   // semantic record on the stack
                                   // and will use it
    return(true);
  else
    return(false);
}
```

AST generation: recursive-descent predictive parser -- using stack

```
E(){
  ASTnode Es = pop(Es)          // will work on Es provided by calling
                                // function, so pop it.
  if (lookahead is in [0,1,(])
    ASTnode TsE’s               // this right hand side needs two new
    push(E’s)                   // semantic records: Ts and E’s
    push(Ts)                    // to fill-in the Es that its
                                // calling function needs
    if (T();E'();)              // T() will pop Ts,
                                // fill it in and push it back for E’()
                                // to use it along with the empty E’s
      write(E->TE')
      Es = pop(E’s)             // the E’s we got from E’() is sent up
      push(Es)                  // the tree as Es
      return(true)
    else
      return(false)
  else
    return(false)
}
```

AST generation: recursive-descent predictive parser -- using stack

```
E'(){
  ASTnode Ti = pop(Ts)                          // get Ti we got by way of T() as Ts
  ASTnode E's = pop(E's)                        // get the empty E’s sent from the
                                                // calling function
  if (lookahead is in [+])
    ASTnode Ts,E'2s                             // we will need T() and E’()
    push(E'2s)                                  // to process Ts and E’2s, so
    push(Ts)                                    // we create and push them.
    if (Match('+');T();E'())                    // T() will pop Ts, fill it in
                                                // and push it back for E’() to
                                                // pop along with the empty E’2s.
                                                // E’() will then push an Es
      write(E'->TE')
      E's = makeFamily(+,pop(Ti),pop(E’2s))     // create subtree
      push(E's)                                 // send up
      return(true)
    else
      return(false)
  else if (lookahead is in [$,)])
    write(E'->epsilon)
    E's = Ti                                    // synth. attr. is inherited
    push(E's)                                   // from T (sibling, not child)
                                                // and sent up
    return(true)
  else
    return(false)
}
```

AST generation: recursive-descent predictive parser -- using stack

```
T(){
  ASTnode Ts = pop(Ts)              // the empty Ts pushed by the
                                    // calling function for T() to fill in
  if (lookahead is in [0,1,(])
    ASTnode Fs, T's                 // we need F() and T’() to process
    push(T's)                       // Fs and T’s, so we create it and
    push(Fs)                        // push it on the stack for them
    if (F();T'();)                  // F will pop the empty Fs, fill it in
                                    // and push it back. T’ will pop it
                                    // along with the empty T’s, create a
                                    // T’s and push it.
      write(T->FT')
      Ts = pop(T's)                 // the T’s pushed by T’()
      push(Ts)                      // is popped and pushed as Ts
      return(true)
    else
      return(false)
  else
    return(false)
}
```

AST generation: recursive-descent predictive parser -- using stack

```
T'(){
  ASTnode T's = pop(T's)            // pop the empty T's was pushed by the
  ASTnode Fi  = pop(Fs)             // calling function. pop the Fs that
                                    // was pushed by the calling function
                                    // which is our inherited attribute from F (Fi)
  if (lookahead is in [*])
    ASTnode Fs, T's2                // we will need F() and T'()
    push(T's2)                      // to process Fs and T'2s, so
    push(Fs)                        // we create and push them.
    if (Match('*');F();T'())        // F() will pop Fs, fill it in
                                    // and push it back for T'() to
                                    // pop along with the empty T's2.
                                    // E'() will then push an Es
      write(T'->*FT')
      T's = makeFamily(*,pop(Fi),pop(T'2s)) // make subtree
      push(T's)                              // send up
      return(true)
    else
      return(false)
  else if (lookahead is in [+,$,)])
    write(T'->epsilon)
    T's = Fi                        // synthetized attribute is
    push(T's)                        // inherited from F sibling
                                     // and sent up the tree

    return(true)
  else
    return(false)
}
```

AST generation: recursive-descent predictive parser -- using stack

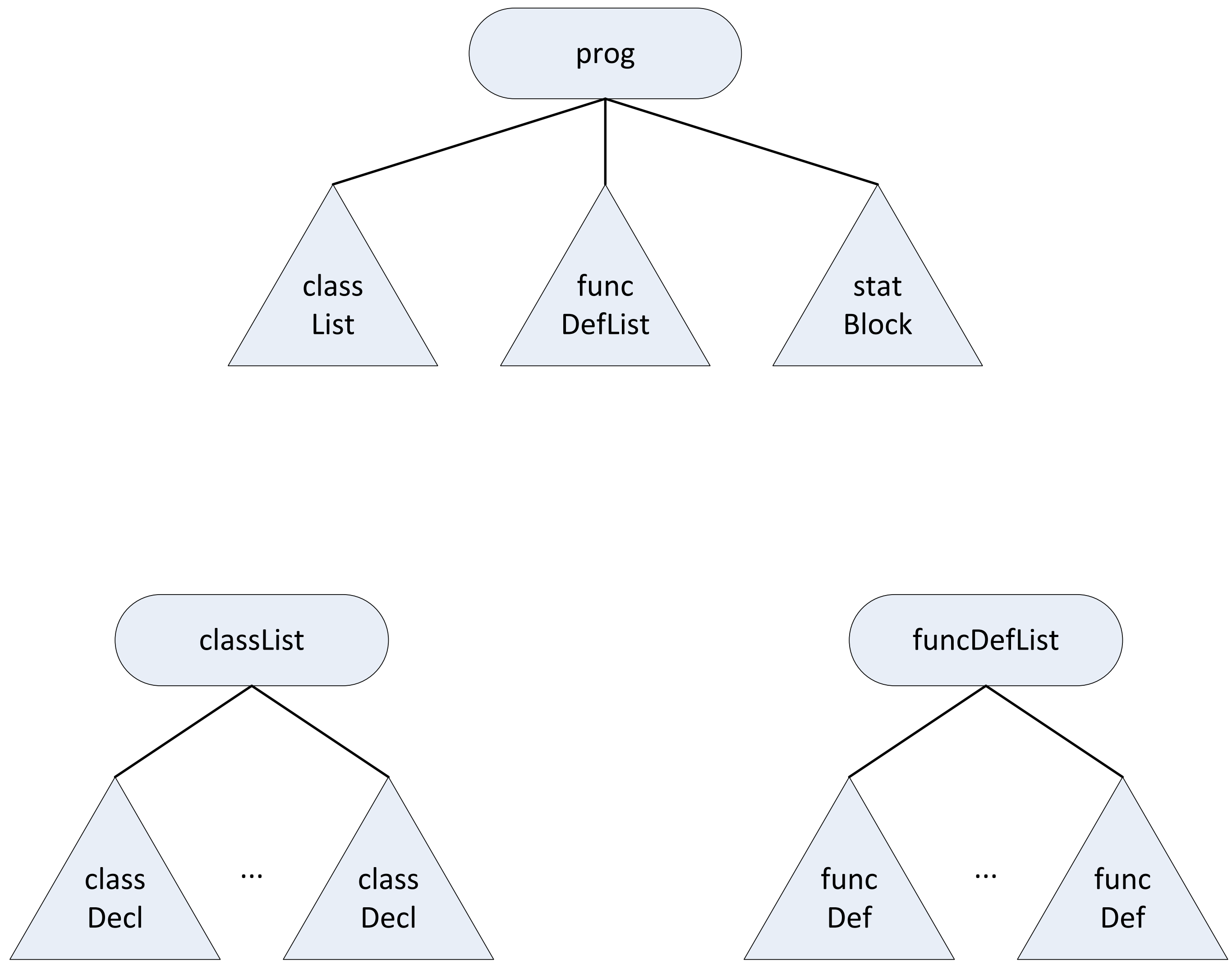
```
F(){
  ASTnode Fs = pop(Fs)                    // get empty Fs from calling function
  if (lookahead is in [id])
    if (Match('id'))
      write(F->id)
      Fs = makeNode(id)                   // create leaf node
      push(Fs)                            // send up
      return(true)
    else
      return(false)
  else if (lookahead is in [(])
    semrec Es                             // we need E() to process an Es
    push(Es)                              // so we create an empty Es and
                                          // send it to E() to fill in
    if (Match('(');E();Match(')'))
      write(F->(E))
      Fs = pop(Es)                        // Es pushed by E()
                                          // sent up the tree as Fs
      return(true)
    else return(false)
  else return(false)
}
```

Abstract Syntax Tree structural elements

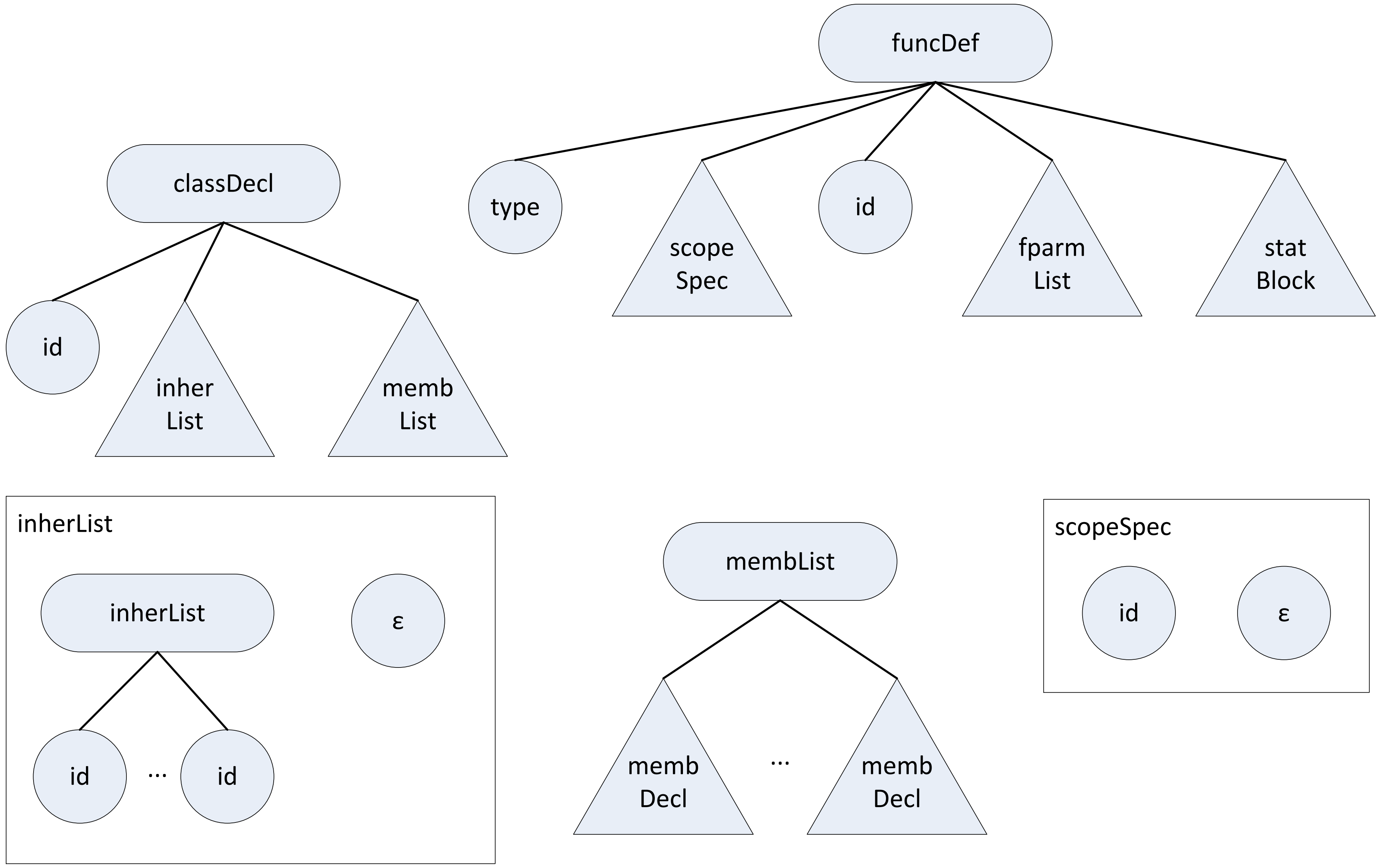
Example grammar (slightly different from project)

```
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'
```

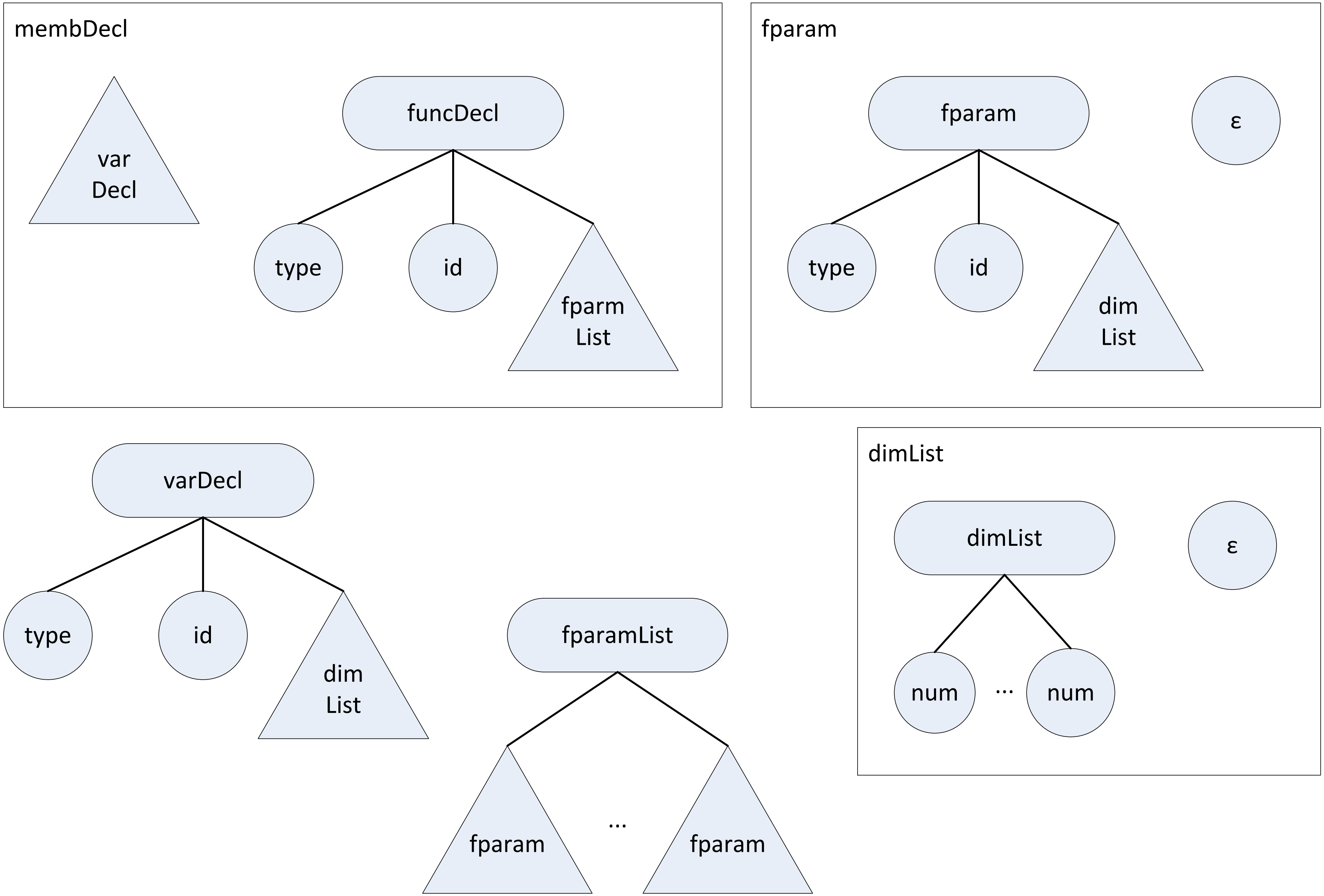
Abstract Syntax Tree structural elements



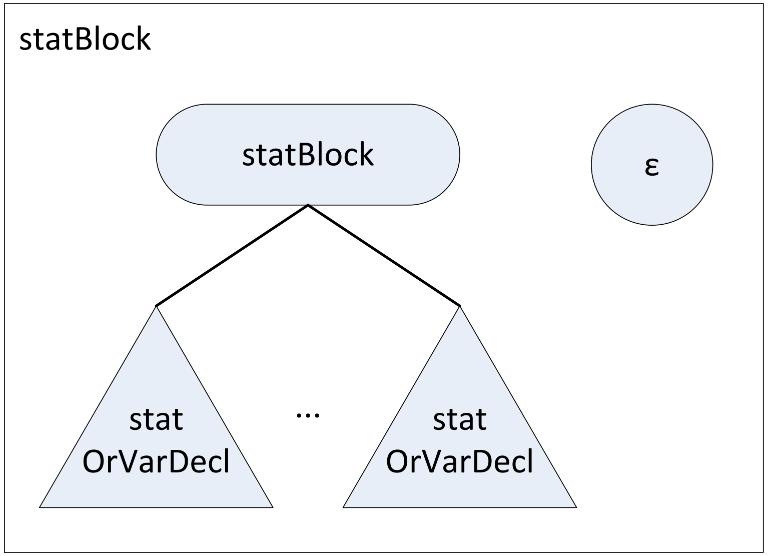
Abstract Syntax Tree structural elements

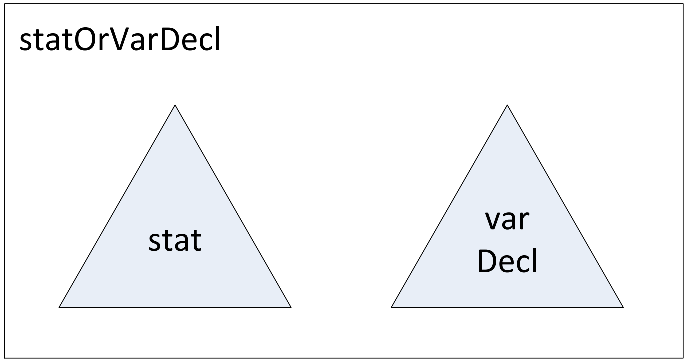


Abstract Syntax Tree structural elements

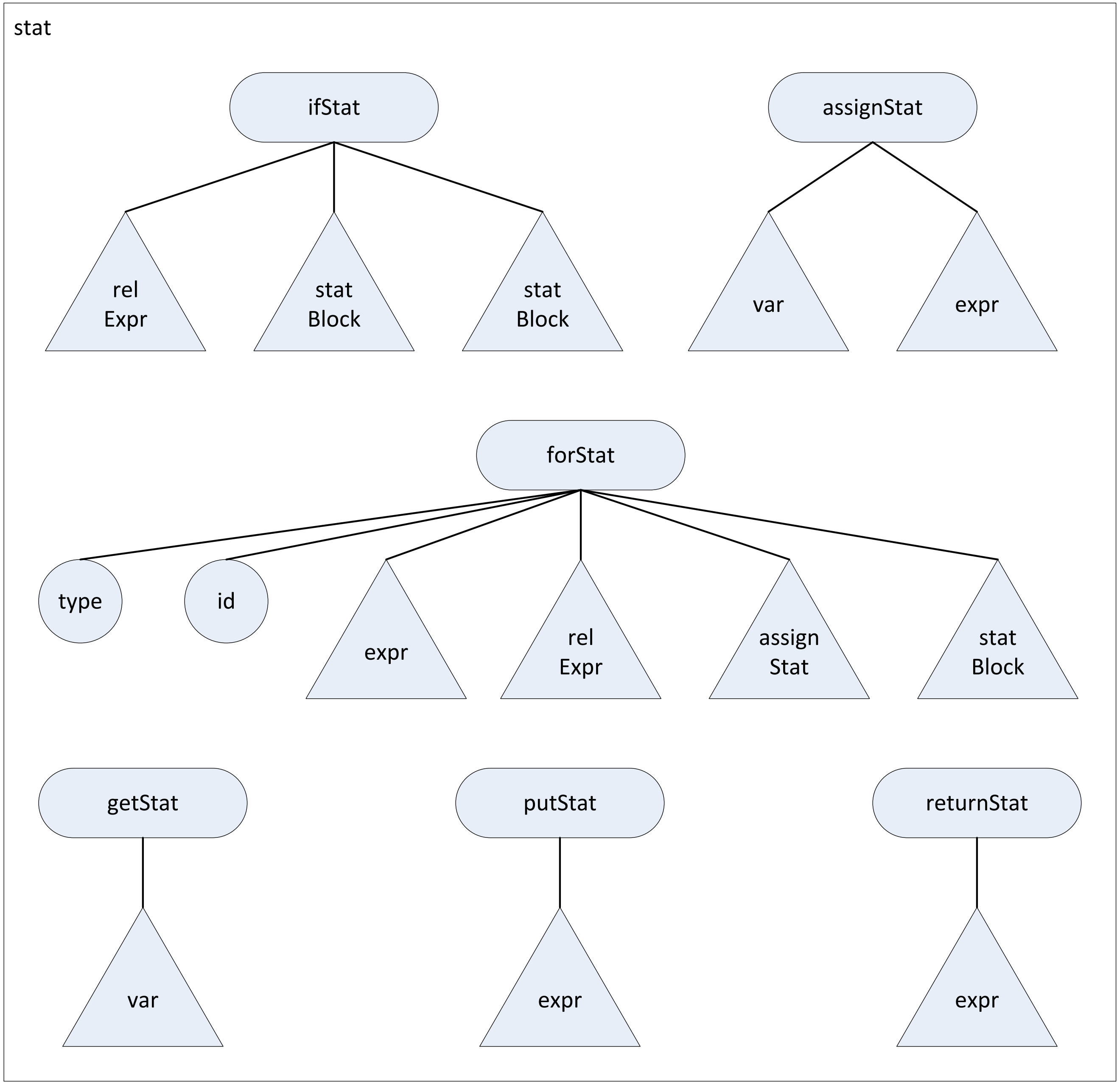


Abstract Syntax Tree structural elements

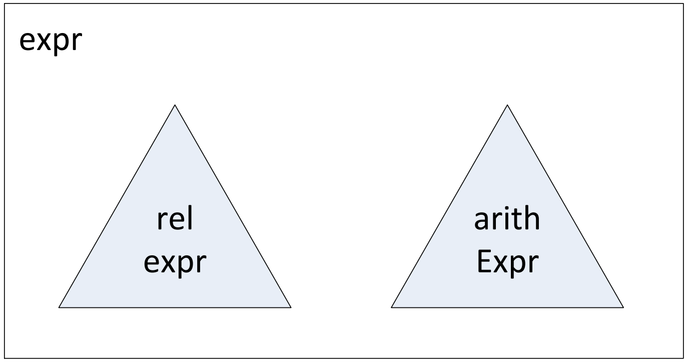


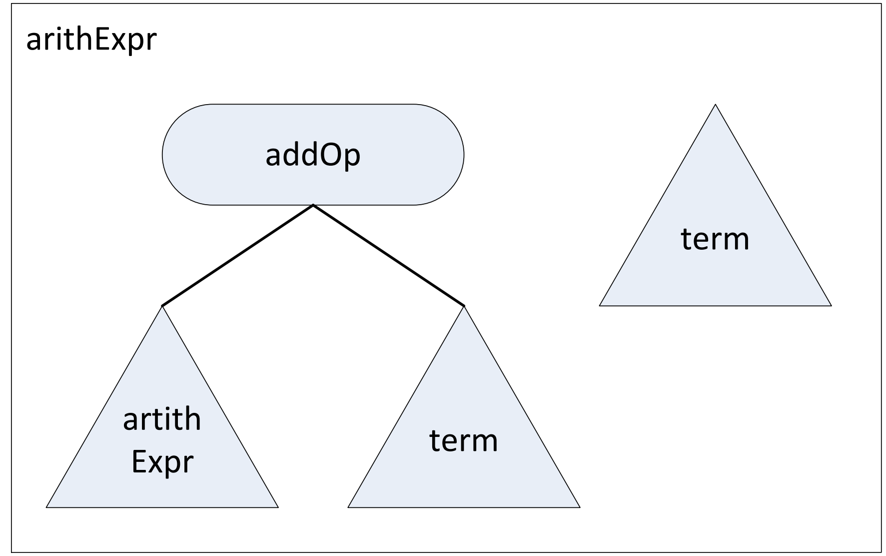


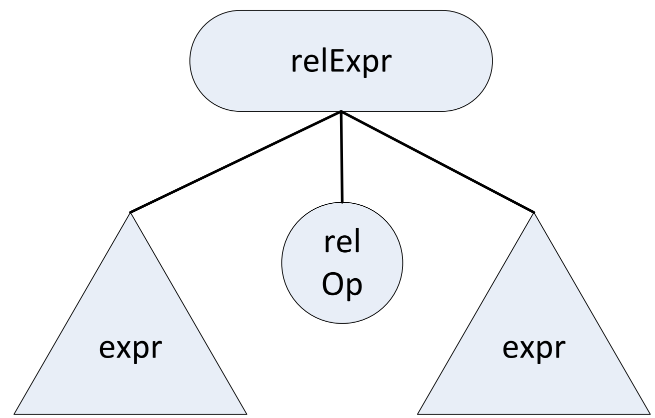
Abstract Syntax Tree structural elements



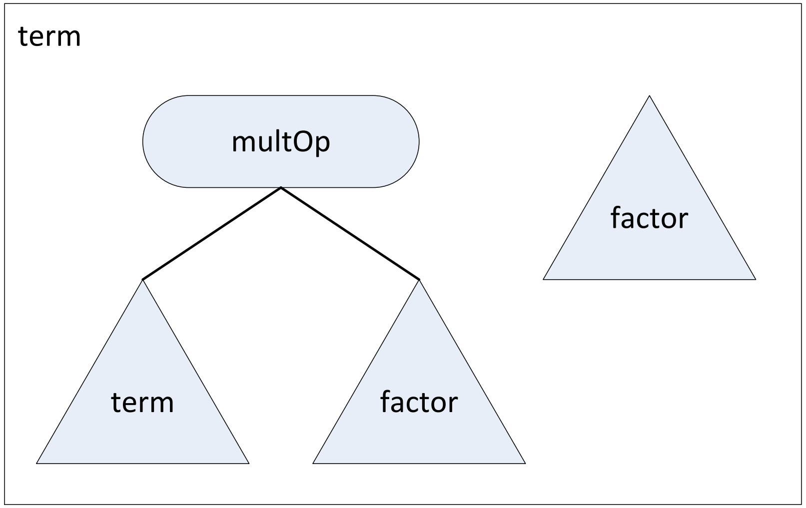
Abstract Syntax Tree structural elements

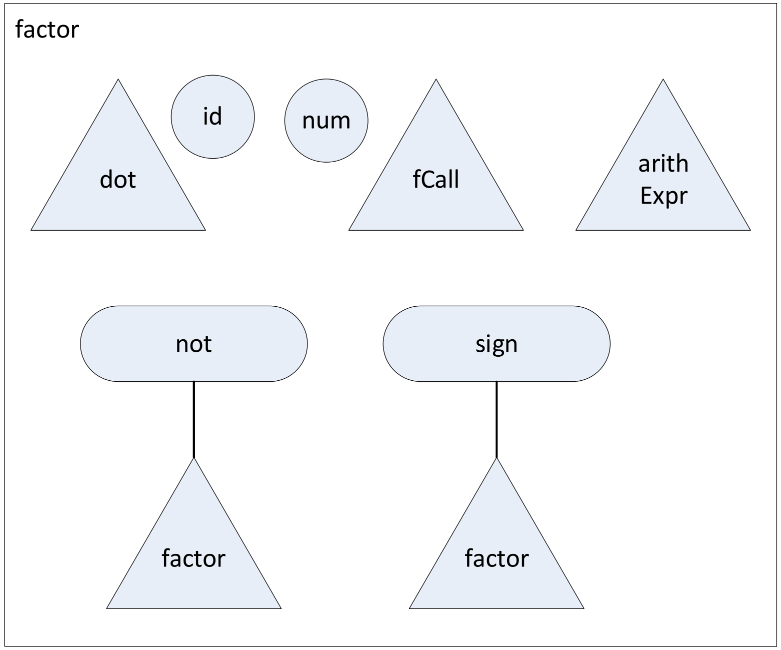




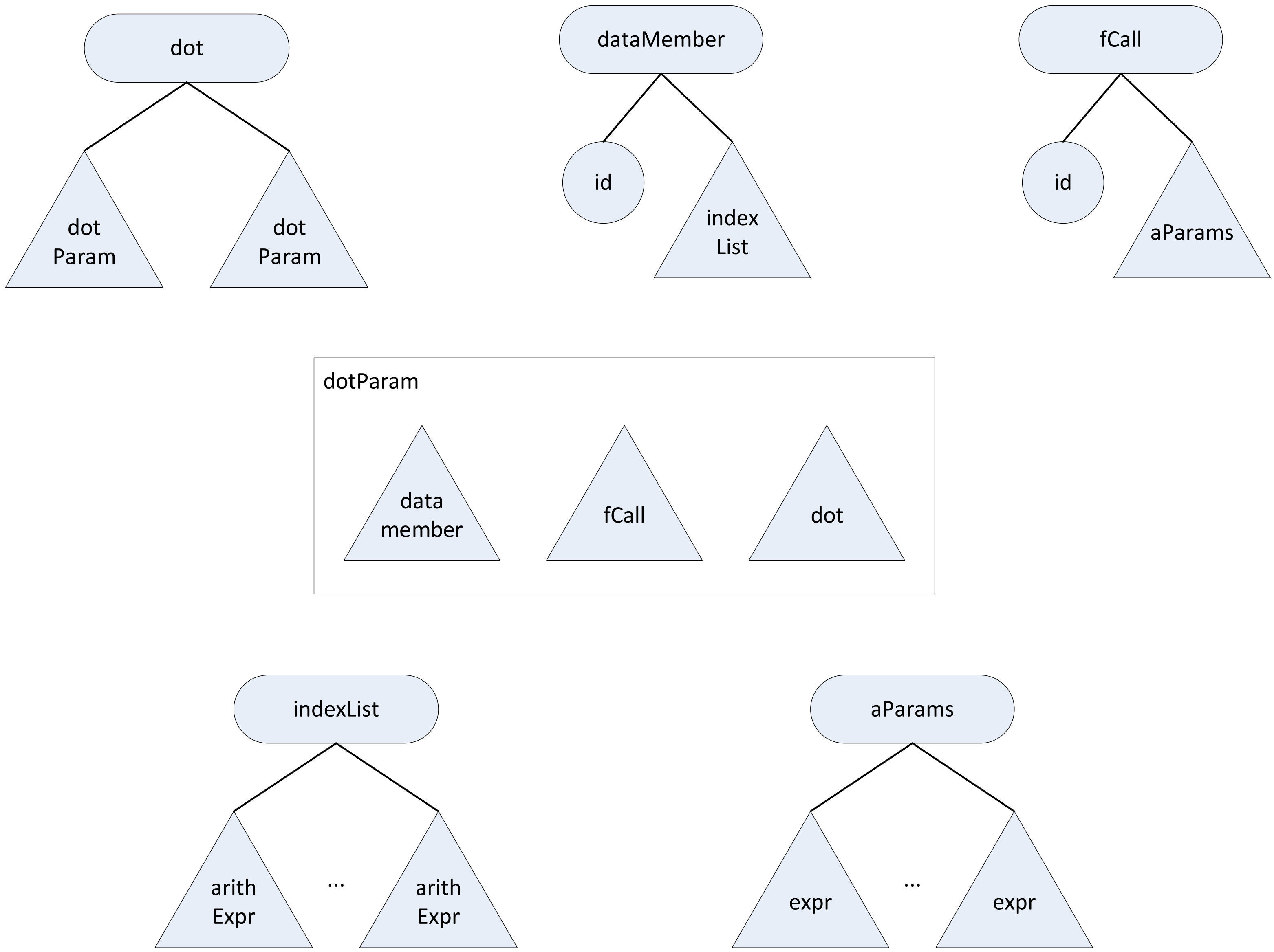


Abstract Syntax Tree structural elements





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.