

## Abstract Syntax Tree (AST)

- Parse tree vs. AST  
e.g.,  $b*b + 4*a*c$   
Parse tree describes the input exactly  
AST captures structural info, with all parsing issues resolved
  - Can build “semantic actions” into the process!
    - could interpret semantic actions directly as we parse – may require detailed understanding of parsing process
      - (T->T\*F ok, T->TT', T'->\*FT' ...)
    - with parser gens. you can specify “semantic actions” of parsers
      - recursive computations using abstract syntax
      - parallel stack of semantic values
- ```
exp: INT          (INT)
    exp TIMES exp (exp1 * exp2)
```

## Abstract Syntax

```
struct Absyn = struct
type pos=int and symbol=Symbol.symbol

datatype var =
  SimpleVar of symbol*pos
  | ...
and exp =
  VarExp of var
  | NilExp
  | IntExp of int
  | OpExp of {left:exp, oper:oper, right:exp, pos:pos}
  | CallExp of {func:symbol, args:exp list, pos:pos}
  ...
and open = PlusOp | MinusOp | TimesOp...
end ty =
  NameTy of symbol*pos
  | ...
end
```

```
%term INT of int | ID of string | PLUS | MINUS...
%nonterm exp of Absyn.exp | var of Absyn.var | ...
%precedence and other parser decls.
```

```
prog: exp          (exp)

exp:  NIL          (Absyn.NilExp)
     | INT         (Absyn.IntExp INT)
     | STRING      (Absyn.StringExp(STRING, STRINGleft))
     | exp PLUS exp (Absyn.OpExp{left=exp1,
                                oper=Absyn.PlusOp,
                                right=exp2,
                                pos=PLUSleft})
     | ID LPAREN arg_list RPAREN
       (Absyn.CallExp{func = Symbol.symbol ID,
                      args=arg_list,
                      pos=IDleft})
     | ...
```

## Symbols

- names of labels, variables, functions, types...
- the meaning of the name is related to the scope in which it appears
- A *symbol table* is a data structure that associates *names* with *information about the objects* that are denoted by the names.
- Symbol vs. string: objective is fast operations
  - keys in hash tables or search trees (cannot use string as index)
- Symbol table has to provide fast lookup, enter, delete
- Need efficient representation of scopes

- Nested scopes – update or create new symbol table?
  - don't want to copy
  - need to maintain 'old' values
  - stack of tables
  - table of stacks

- Imperative style SymbolTbls
  - enter\_scope(), exit\_scope()
    - destructive – earlier bindings no longer valid
    - must restore values upon exit
  - enter\_symbol/get\_symbol
  - need to provide access to global scopes, etc.
- Functional style SymbolTbls
  - persistent
  - balanced trees: hash name to get symbol, use symbol as index in balanced tree
- For project order imposed by symbol is most order-of-appearance.