CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107S in the
Winter 2012/2013 term at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

Reading Assignment

Fischer, Cytron and LeBlanc

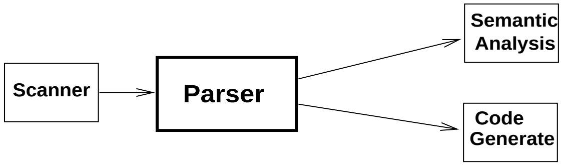
Chapter 7
Omit 7.7

Emphasize AST use and structure

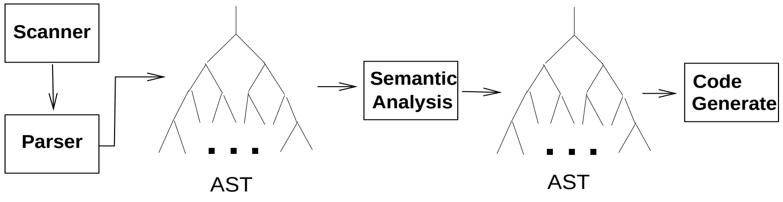
Syntax Directed Translation

- In one pass compilers (e.g. Slide 19) the most common organization is for the syntax analysis phase (parser) to direct the entire compilation
- The parser calls the scanner as require to get tokens.
  At various points during the parse the parser calls the semantic analysis and code generation phases to perform various actions.
- The semantic and code generation actions are often described using a grammar for the programming language.
- In multi-pass compilers (e.g. Slide 21) the parser builds some data structure (usually an **A**bstract **S**yntax **T**ree) that represents the program being compiled.
  Semantic analysis and code generation are implemented as algorithms that traverse this data structure.
- If the language being processed is context sensitive (e.g. C) then some semantic analysis must be done during parsing to allow parsing to proceed.

Single Pass Compiler



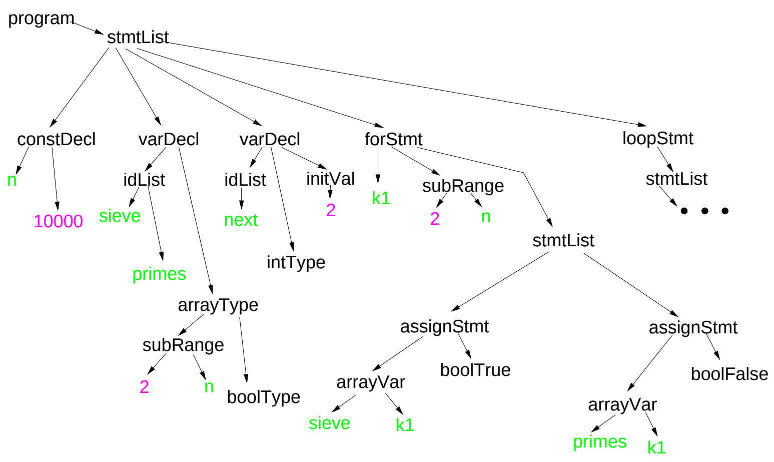
Multi Pass Compiler



Abstract Syntax Trees

- Abstract Syntax Trees (ASTs) are designed to capture all of the essential structural information about a program being compiled.
- Bottom Up syntax analysis is a good match with AST building. The child nodes to the tree are always built before the parent node.
- The basic process for *building* an AST is very simple:
  - the leaf nodes are typically constants and identifiers.
    Build a node to represent each of these entities.
  - Interior nodes are build as needed to represent particular language constructs.
    Typically interior nodes contain links to one or more child nodes.
- The *processing* of AST nodes for semantic analysis is also simple:
  - Process the AST *depth first*. This guarantees that child nodes are processed before parent nodes.
  - At each parent node do any processing required using information from already processed child nodes.

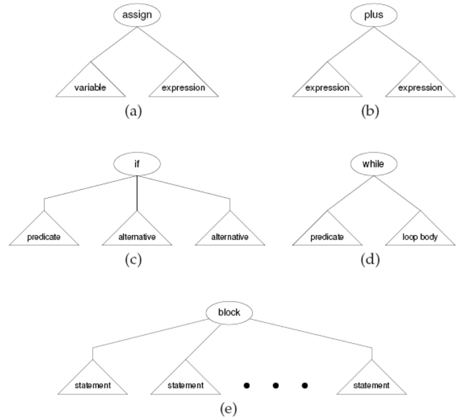
Partial[a] Abstract Syntax Tree for Running Example (Slide 37)



[a] Compare this to the full parse tree in Slide 67

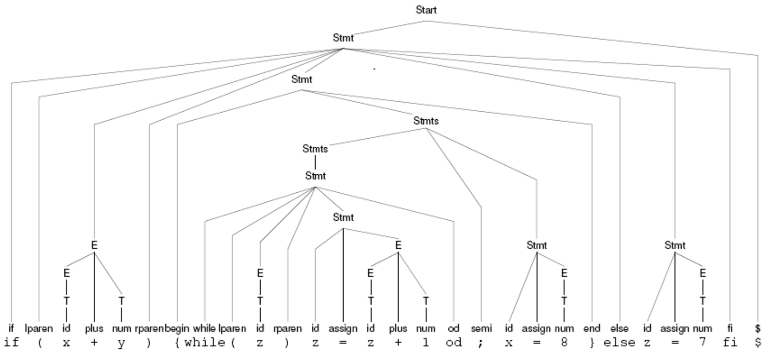
ASTs Are Language Specific

Typically AST notes are custom designed for each of the major constructs in the language. Some Examples (*Fischer, Cytron, LeBlanc* Figure 7-15)[a]:



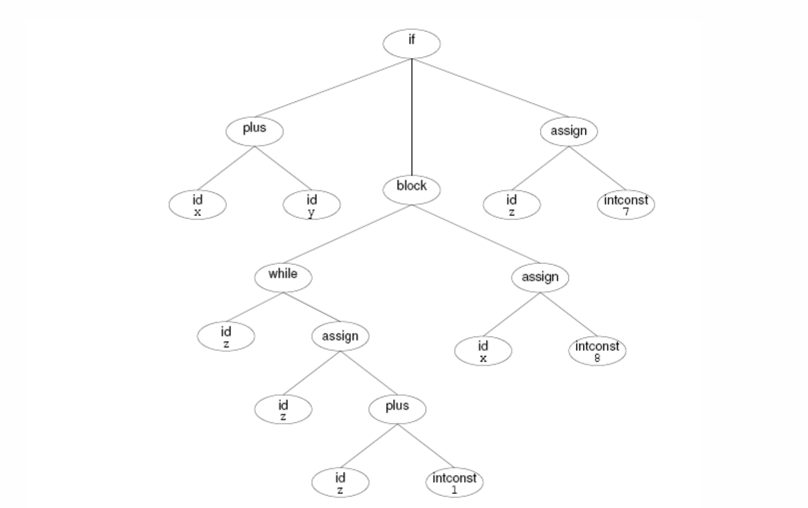
[a] For CSC488S Winter 2012/2013 examples see the AST classes: `AssignStmt`, `BinaryExpn`, `IfStmt`, `LoopStmt`, and `Scope`

Complete Syntax Tree for Program Fragment[a]


Start
Stmt
Stmt
Stmts
Stmts
Stmt
Stmt
E
E
T
T
E
T
E
E
T
T
Stmt
E
T
Stmt
E
T
if lparen id plus num rparen begin while lparen id rparen id assign id plus num od semi id assign num end else id assign num fi $
if (x + y) { while (z) z = z + 1 od ; x = 8 } else z = 7 fi $

[a] *Fischer, Cytron, LeBlanc* Figure 7-18

Abstract Syntax Tree for Program Fragment[a]



[a] *Fischer, Cytron, LeBlanc* Figure 7-19

A Very General Abstract Syntax Tree Node[a]

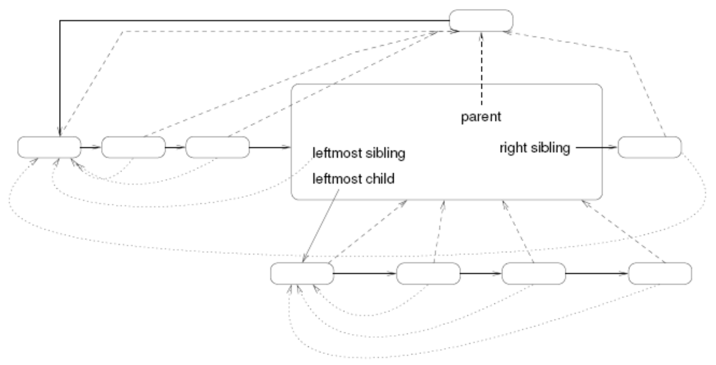
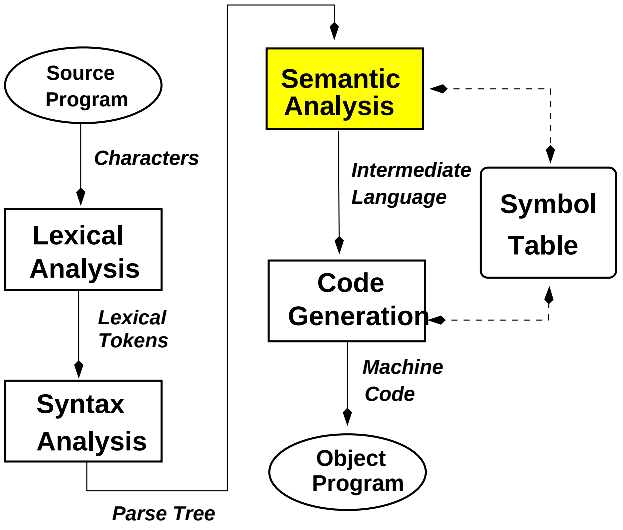


Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

[a] *Fischer, Cytron, LeBlanc* Figure 7-12



Source Program

Characters

Lexical Analysis

Lexical Tokens

Syntax Analysis

Parse Tree

Semantic Analysis

Intermediate Language

Symbol Table

Code Generation

Machine Code

Object Program

Reading Assignment

```
Fischer, Cytron and LeBlanc

Sections  8.5, 8.6, 8.7, 8.8, 8.9

Sections  9.1.0, 9.2, 9.3
```

Semantic Analysis

- Validation of *non-syntactic* language constraints

  Static semantic analysis - during compilation

  Dynamic semantic analysis - run time checks

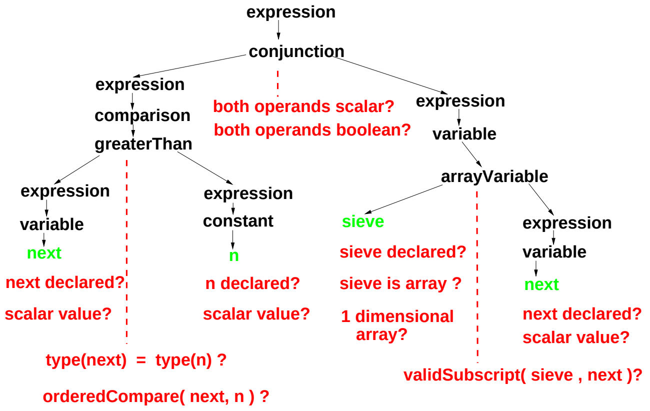
- Semantic analysis
  - Visibility and Accessibility Analysis
  - Type Checking
  - Proper Usage Analysis
  - Range and Value Analysis
  - Range and Value Propagation
- The compilers symbol table is usually built during semantic analysis as a side effect of declaration processing.

Semantic Analysis Example

	X	:=	Y		
Visibility	declared(X)?		declared(Y)?		
	visible(X)?		visible(Y)?		
Accessibility	access(X)?		access(Y)?		
	write(X)		read(Y)?		
Usage	variable(X)?		variable(Y)?	constant(Y)?	function(Y)?
Type	type(X)?		type(Y)?	type(Y)?	type(Y)?
	scalar(X)?		scalar(Y)?	scalar(Y)?	scalar(Y)?
Usage	assign(X)?				parameters(Y)?
Type		compat(X,Y)?			
Value/Range	range(X)?		range(Y)?	value(Y)?	range(Y)?
		inRange(X,Y)?			

Semantic Analysis – Running Example (Slide 37)

next > n or sieve[next]



Type Equivalence, Compatibility, Suitability

- Every language definition includes rules about when objects of different types can be used together in various constructs.
- Many languages have several rules concerning types:
  - *Type Equivalence Conditions under which objects of two different types are considered to be* equivalent*.*

    Equivalence is usually required when the addresses *of data objects are being manipulated.*
  - *Type Compatibility Conditions under which objects of two different types are considered to be* compatible*.*

    Compatibility is usually required for assignments.
  - *Type suitability Conditions under which objects of two different types are considered* suitable *to be used together.*

    Suitability is usually required for operands in expressions.
- The two most widely used Type Equivalence Rules are *structural equivalence* and *name equivalence*

Type Equivalence Rules

- Define: Name Type Equivalence
  - Two types are name equivalent iff they ultimately derive from a common definition.
  - *ultimately derive* allows type renaming, e.g. **type** S : T
  - In implementation terms, two named types are equivalent if they refer to the same type table entry.
- Define: Structural Type Equivalence
  - Type types are structurally equivalent if their definitions have the same structure and corresponding values are equal.
  - Structural equivalence is isomorphism for types.
  - In implementation terms a parallel walk of two type trees is required to establish structural equivalence.

Algorithm for Structural Equivalence

```
function isEquivalent( type $T_1$ , type $T_2$ ) : Boolean {
    /* Test types $T_1$ and $T_2$ for structural equivalence */
    if $T_1$ . typeKind  not = $T_2$ . typeKind  then
        return  false                  /* node mismatch */
    for each value $field_i$ in $T_1$ , $T_2$
        if $T_1$ . $field_i$  not $=_{lang}$ $T_2$ . $field_i$  then
            return  false              /* value mismatch */
    for each $subtree_i$ of $T_1$ , $T_2$
        if  not isEquivalent( $T_1$ . $subtree_i$ , $T_2$ . $subtree_i$ )  then
            return  false              /* subtree mismatch */
    return  true         /* all values and subtrees match */
end /* isEquivalent */
```

Note use of language specific test in comparing value fields

Type Equivalence Example

```
typedef A =                 typedef F =
  struct                       struct
    B : integer;                 G : integer;
    C : char;                    H : char;
  end;                         end;

 typedef D = A;
```

A and F are structurally equivalent but not name equivalent.

D and A are name equivalent and thus structurally equivalent.

Type Equivalence Checking

- Type equivalence checking is used to guarantee that the type of data object that a pointer points at is *the same* as the type that was declared for the pointer.
- This check is necessary to ensure that when the pointer is used to access parts of the object that access will yield the correct result.
- Type equivalence implies **memory image equivalence** i.e. the two types have identical memory images.
- Usually type equivalence checking is done in two cases
  - When the address of a data object is assigned to a pointer.
  - When a variable is passed as a reference (i.e. address) parameter.
- Memory image equivalence guarantees that when an address is used to access a type, the internal parts of the type (e.g. fields in a structure) will be accessed correctly

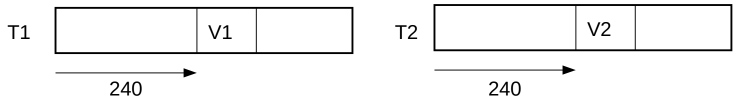
Memory Equivalence Example

Given:

```
type T1 = record ... V1 : integer ... end record
type T2 = record ... V2 : integer ... end record
```

assume that T1 and T2 are *structurally equivalent*.

Then structural equivalence guarantees that for all corresponding fields F_i in T1 and T2 the address of the field relative to the start of the type is equal.



Implementing memory equivalence constrains the way that record fields are allocated within a record/structure. See Slides 272 and 275

Why memory equivalence matters. Consider:

```
var A , *AP: T1           /* AP is pointer to T1 */
procedure P( var X : T2 , Y : T2 ) {
    ...
    X . V2 := Y . V2
    ...
} /* end P */
/* Assume A and AP are given values here */
```

If T1 and T2 are memory equivalent then the following calls are legal

```
P( A , *AP )
P( AP, A )
```

as would be the cast

```
AP := ( * T1) ADDR( Y )
```

Type Compatibility and Suitability Checking

- Type compatibility checking is used to determine when a value of some given type is compatible with the type of some variable.
- Compatibility checking is typically used to check
  - That the expression on the right side of an assignment statement may be legally assigned to the variable on the left side.
  - That an expression being passed as a value parameter to a routine[a] can be legally assigned to the corresponding formal parameter variable.
- Type suitability checking is used to determine when one or more values can be used together or with an operator.
- Typical instances of suitability checking include
  - checking that the operand of a unary operator is of a suitable type for the operator.
  - checking that both operands of a binary operator are of suitable types for the operator and for each other.

[a] In these slides we use *routine* as a synonym for *function or procedure*.

Visibility and Accessibility

- Visibility analysis determines whether a given reference to an identifier at some point it the program is legal according to the scope rules (Slide 169) of the language.
- The perform visibility analysis, the compiler must keep track of declared symbols behaves (logically at least) in a way that is consistent with the scope rules of the language.
- Semantic analyzer must also track the scope structure of the program as it is being processed.
- Accessibility analysis determines whether a visible identifier can be accessed in a given way at some point in a program. Examples

  bind in Turing restricts access to bound to identifiers
  const in C prevents variables from being assigned to
  or having their address assigned to non-const pointers.

Usage Analysis

- Appropriate use of constants, variables, types, procedures, functions.
  Is a give *use* of an identifier *consistent with* the information that the compiler knows (i.e. from declarations) about the identifier?
- Enforcing language and implementation limits.
- Detection of potential arithmetic faults.
- Detection of potential run-time faults.
  Emitting code for dynamic semantic analysis.
- Enforcing anti-aliasing and other safety constraints.

Usage Analysis Example

Assume

```
type i : 1 .. 23
var x : integer
    y : i
    a : array [ 1 .. 10 ] of char
    function f ( p : integer ) : integer
```

Statement	Error
i := 17	Assignment to a type
f(3) := x	Assignment to a function
x := a	Assign array to scalar variable
x[y] := 31	Subscript on a scalar variable
x := y.q	Field selection on a scalar variable
a := f(a[1], y)	Assign integer to char array
	Wrong number and type of parameters to f

Value and Range Analysis

- Range checking of array subscripts.
- Checking assignment to subrange variables.
- Enforce language/implementation limits.
- Range propagation to minimize run time checking.
- Range Propagation
  - Each constant and variable has an inherent range.
  - For each arithmetic operation on range variables there is an expression that gives the range of the result.
  - Propagate ranges through expressions to determine range of final result.
  - Use propagated range information to minimize run time checking and to detect potential run time arithmetic faults (e.g. divide by zero, overflow).

Given two subrange variables:

A : minA .. maxA
B : minB .. maxB

then the range of values produced by arithmetic can be computed

A + B	minA + minB .. maxA + maxB
A - B	minA - maxB .. maxA - minB

Range Propagation Example

Assume

var i,j : 1 .. 100
m,n : **integer**
k : -50 .. 40

Statement	RHS Range	Checks
i := i + 1	2 .. 101	check i $\leq$ 99
i := n + 1	minInt+1 .. maxInt+1	check n $<$ maxInt check 1 $\leq$ n+1 $\leq$ 100
n := i * j	1 .. 10000	
m := k * j - 10*i + 40	k*j -5000 .. 4000 k*j-10*i -6000 .. 3990 -5960 .. 4030	10*i 10 .. 1000 40 40 .. 40

Arithmetic Fault Detection Example

Assume

var i,j,k : **integer**
m : 0 .. 100
n : -100 .. 100
/* Assume 16-bit integers */
/* minInt = -32768   maxInt = 32767 */

Expression	Potential fault
i := i / n	divide by zero
k := k + 1	overflow
j := - i	overflow
i := 33 * m * n	underflow, overflow

Programming Language Influences

- Definition of the programming language being compiled can have a major influence on the way semantic analysis is implemented.
- Some issues:
  - Languages that do not require *declaration before use* (e.g. PL/I, C) will usually require a separate semantic analysis pass (or a lot of kludgery).
  - Languages with a context sensitive syntax (e.g. C, Fortran) will require feedback from the semantic analysis pass to the parser or scanner.
  - Language with a weak or non-existent declaration structure (.e.g. Lisp, Icon, APL, Prolog) require that most semantic analysis be done dynamically.
  - Object-Oriented languages (e.g. Smalltalk, C++, Java) that allow dynamic object binding may require extensive run time checking.
  - The presence of dynamically sized objects in a language (e.g. arrays whose bounds are determined at run time) may require more run time checking.
  - Compilers for *student* languages should do very thorough static and dynamic semantic analysis.

Managing Semantic Analysis Data

- The semantic analysis phase will typically require some data structures to keep track of semantic information as it processes the program.
- Due to the *embedding* that occurs in most programming languages (e.g. statements within statement, expressions within expression) stack-like data structures are usually required.
- Toy, prototype and student compilers often use fixed sized stacks for various internal data structures. Production compilers often allocate data structures dynamically on demand and use linked lists instead of stacks.
- In single pass compilers, a commonly used approach is to use several stacks that operate *in parallel* with the parse stack.
  For each item on the parse stack, the corresponding parallel stack entries might hold type, symbol, value, address or other information.
  Use right recursive definitions to capture an entire construct (e.g a case statement) in the parse stack.

Example - Stacks parallel to parse stack

Type stores type information

Symbol Stores symbol table pointers

ValType Stores type of value field

Value Arbitrary value

Parse	Type	Symbol	ValType	Value
then				
0	integer		intConst	0
=	relOp			
identifier	integer	syt(I)	VarAdr	200(4)
if			CodeAdr	2420
statements			CodeAdr	1456

Semantic Analysis of Declarations

- The canonical declaration associates one or more identifiers with type, structure and size attributes. The syntax of the language determines how these associations are made. Examples

  C **int** i, ia[10], ig(), *ip ;

  PL/I DECLARE (I, IA(0:9)) FIXED BINARY(31,0) ;

  DECLARE IP POINTER,

  IG ENTRY RETURNS(FIXED BINARY(31,0));

- Declaration processing involves collecting attribute information and applying defaults for missing attributes.
- Language may allow the user to specify the default attributes for incompletely declared variables. PL/I has a particularly complex DEFAULT statement.
- Essentially declaration processing involves filling in a symbol table entry for each declared item. e.g. the Pascal symbol table in Slides 189 **..** 193

Basic Declaration Processing

- Accumulate list of identifiers being declared

  int i, j, k, l ;

  var a, b, c, d, e, f, g, h, i, j, k, l : **real**

- Lookup each identifier in the current scope to check for multiple declaration in the current scope.
- Enter each symbol in the symbol table for the current scope.
- Associate attributes from declaration with each identifier.

  Apply language-specific defaults as required.

  int a, b[10], *c, *d[], **e, f(), *g(), *(h()) ;

  DECLARE (I, J STATIC, (K, L)(0:19)) FIXED DECIMAL(3) ;

- Process initial value if one is present.

  int i, j = 3 ;

Declaration Processing Example - Arrays

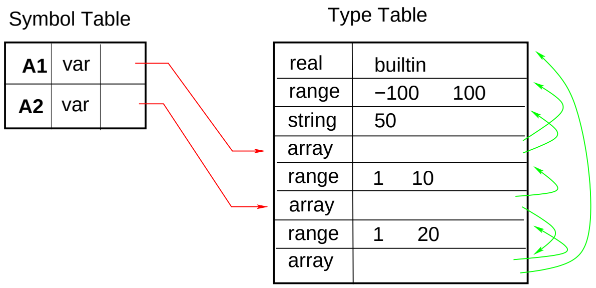
- Determine dimensionality

  Determine range and type of each index.

  Determine type of array elements.

  var A1 **array** –100 **..** 100 **of string**(50)

  var A2 **array** 1 **..** 10 **,** 1 **..** 20 **of real**



Symbol Table

A1	var	
A2	var	

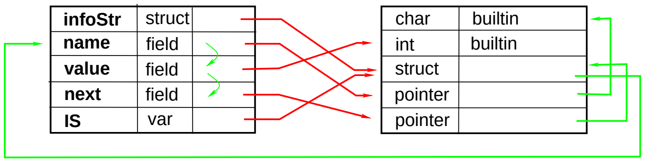
Type Table

real	builtin
range	−100 100
string	50
array	
range	1 10
array	
range	1 20
array	

Declaration Processing Example - Records

- Determine number of fields in the record.
  Build list of symbol table entries for fields.
  Recursively determine attributes for each field.
  Determine structure of variant records/unions.

```
        struct infoStr {
                char * name ;
                int value ;
                struct infoStr * next ;
        }  IS  ;
```

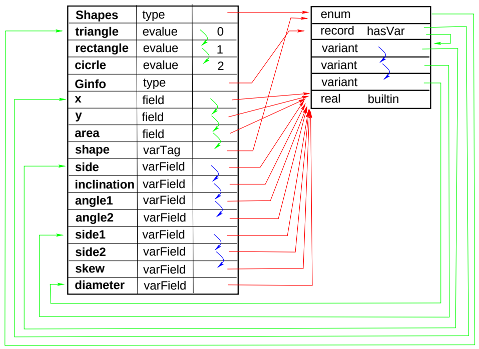


infoStr	struct	
name	field	
value	field	
next	field	
IS	var	

char	builtin
int	builtin
struct	
pointer	
pointer	

Declaration Processing Example - Variant record/Union

```
type Shapes =
   ( triangle , rectangle , circle ) ;
type Ginfo =
   record
      x, y, area : real ;
      case shape : Shapes of
      triangle :
         ( side, inclination : real ;
         angle1, angle2 : real )
      rectangle :
         ( side1, side2 : real ;
         skew : real )
      circle:
         ( diameter : real )
   end ;
```



Shapes	type	
triangle	evalue	0
rectangle	evalue	1
cicrle	evalue	2
Ginfo	type	
x	field	
y	field	
area	field	
shape	varTag	
side	varField	
inclination	varField	
angle1	varField	
angle2	varField	
side1	varField	
side2	varField	
skew	varField	
diameter	varField	

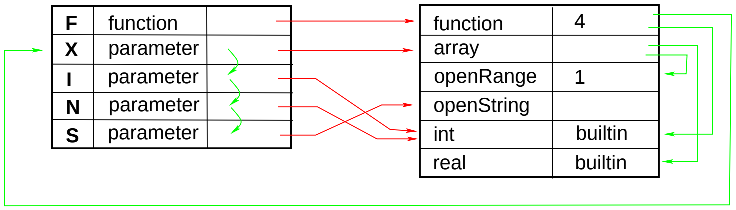
enum	
record	hasVar
variant	
variant	
variant	
real	builtin

Each variant is treated as an anonymous record.

Declaration Processing Example - Function

- Determine number and type of formal parameters
  Build list of symbol table entries for parameters
  Determine return type of function.

```
        function F ( var X : array 1 .. * of real ,
                I, N : int ,
                S : (string) ( * )      ) : int
                    . . .
        end F
```



F	function	
X	parameter	
I	parameter	
N	parameter	
S	parameter	

function	4
array	
openRange	1
openString	
int	builtin
real	builtin

Processing Variables and Expressions

- A major part of semantic analysis is concerned with the processing of variables and expressions.
- Each use of a variable must be checked for correct visibility, accessibility and usage.
- Each expression must be checked for correct usage according to language-specific rules.
- Since expressions and variable references can contain embedded expressions and variable references, semantic analysis is inherently a recursive process.
- Stacks (e.g. type stack, symbol stack) are often used to keep track of information during expression and variable processing.
- Processing variable references is facilitated by the DAG-like symbol and type table structures. Processing a variable reference .e.g. `A[I].B.C.D[J]` typically involves a simultaneous traversal of the DAG defining the type of the variable.

Variable Processing Example

- Regular expression for the general form of *variable reference*
  $(modName.)^*\ ident\ (\ '('\ expnList\ ')'\ \mid [\ expnList\ ] \mid .\ fieldId \mid \uparrow)^*$
- This form allows
  - Multiple module name qualification ($(\ modName.\ )^*$)
  - Arbitrary array subscripts ($[\ expnList\ ]$)
  - Record/union field selection ($.\ fieldId$)
  - Pointer dereferencing ($\uparrow$)
  - Function invocation ($'('\ expnList\ ')'$)
  - Any arbitrary sequence of subscripting, field selection, pointer dereferencing and function invocation.

Note the embedding of variable reference inside expnList.

Variable Processing Actions

$modName.ident$	Lookup $modName$ in symbol table
	Verify that $modName$ is a module
	Obtain type table entry for $modName$
	Get symbol table entry for list of module exported symbols
	Verify $ident$ is exported from the module
	Get symbol table entry for $ident$
	Get type table entry for $ident$
$ident$	Lookup $ident$ in symbol table
	Get type table entry for $ident$
$[\ expnList\ ]$	Verify that preceding thing is an array
	Get type table entry for array dimensions
	Verify length of $expnList$ against dimensionality of array
	Verify that type of each expression in $expnList$ is compatible with array declaration
	Get type table entry for array element

$.\ fieldId$	Verify that preceding thing is record
	Get symbol table entry for list of record fields
	Verify $fieldId$ is a field in the record
	Get type table entry for $fieldId$
$\uparrow$	Verify that preceding thing is pointer.
	Get type table entry for thing being pointed at
$'('\ expnList\ ')'$	Verify that preceding thing is procedure or function.
	Get symbol table entry for list of parameters
	Verify length of $expnList$ against number of parameters
	Check type and accessibility of each expn in $expnList$ against corresponding formal parameter type
	Get type table entry for return type of function/procedure .

Variable Processing Example

Example variable reference $M.X[r.i+1].f\uparrow$

Expression	Saved Attributes
$M.X[r.i+1].f\uparrow$	module M
$X[r.i+1].f\uparrow$	X exported from module M
$[r.i+1].f\uparrow$	1 dim array exported from M
$.i+1].f\uparrow$	local record variable, 1 dim array exported from M
$+1].f\uparrow$	integer field of local record, 1 dim array exported from M
$].f\uparrow$	integer expression, 1 dim array exported from M
$.f\uparrow$	field of 1 dim array of records exported from M
$\uparrow$	reference to boolean
	boolean variable

Expression Processing

- Semantic analysis expression processing involves type and usage checking of expressions. It assumes that references to identifiers (e.g. variables, named constants, etc.) are processed as described above.
- Semantic analysis of expressions can be combined with value and range analysis
- Due to the embedding of expressions within expressions, stacks are often used to save type and symbol information during expression processing.
- Conceptually expression processing is a depth-first walk of the parse tree for each expression.
- Frequently expression processing if operator driven, i.e. the arithmetic operator at each node in the expression tree determines what checks are performed on the operands attached to that node.

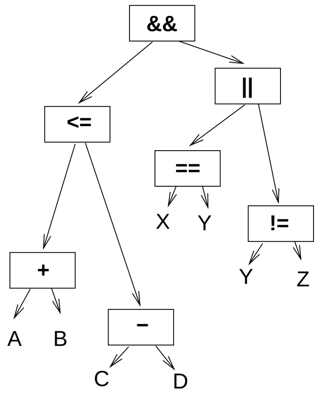
Expression Processing

Walk the parse tree for an expression in a depth-first fashion.
Process the operands attached to each node before processing the node.

Operator(s)	Actions
constant	Create node containing value of literal constant
	Tag node with type of constant
variable	Create node for variable
	Tag node with type of variable
$+, -, *, /$	Verify left and right operands are of a suitable arithmetic type.
	Record arithmetic result type of operator
$<, <=, ==, != , >=, >$	Verify operands are of a comparable type
	Verify comparison is legal
	Record result type is boolean
& , | , **not**	Check operands are boolean type
	Record result type is boolean

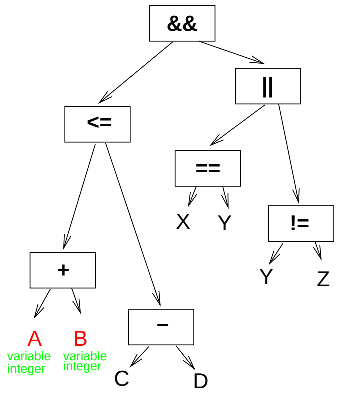
Expression Processing Example

(A + B) <= (C − D) && (X == Y || Y != Z)



Expression Processing Example

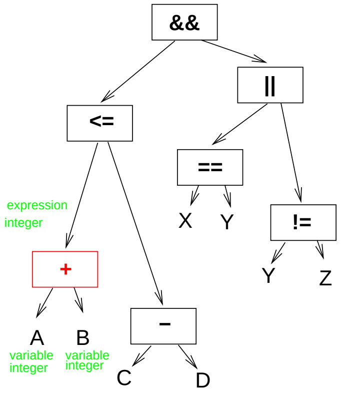
(A + B) <= (C − D) && (X == Y || Y != Z)



Process variables

Expression Processing Example

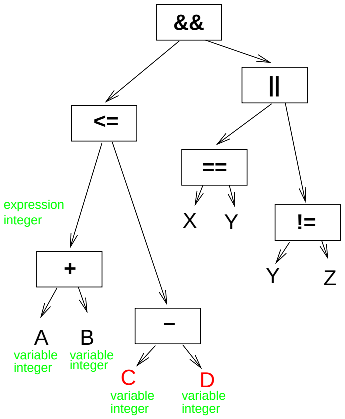
(A + B) <= (C − D) && (X == Y || Y != Z)



Process arithmetic node

Expression Processing Example

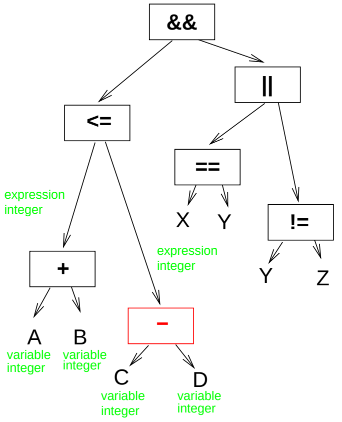
(A + B) <= (C − D) && (X == Y || Y != Z)



Process variables

Expression Processing Example

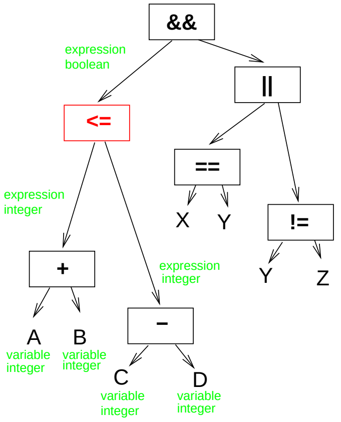
(A + B) <= (C − D) && (X == Y || Y != Z)



Process arithmetic node

Expression Processing Example

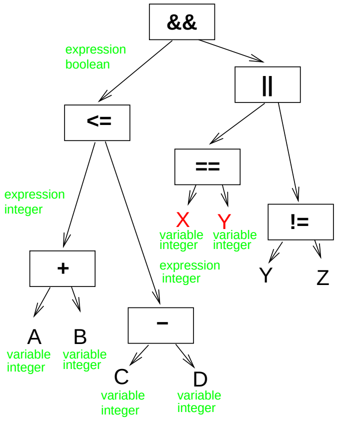
(A + B) <= (C − D) && (X == Y || Y != Z)



Process comparison node

Expression Processing Example

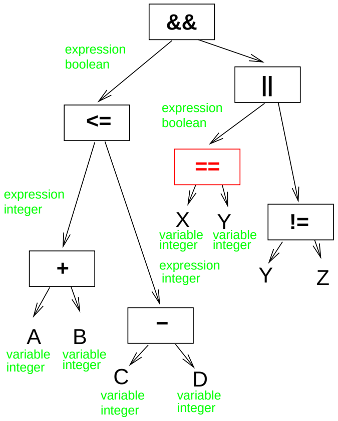
(A + B) <= (C − D) && (X == Y || Y != Z)



Process variables

Expression Processing Example

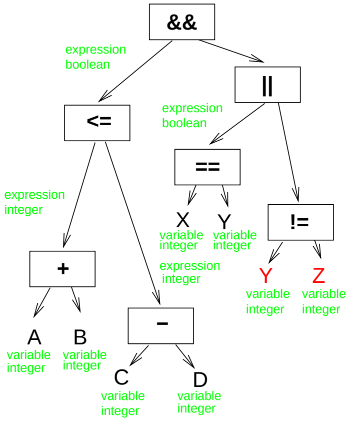
(A + B) <= (C − D) && (X == Y || Y != Z)



Process comparison node

Expression Processing Example

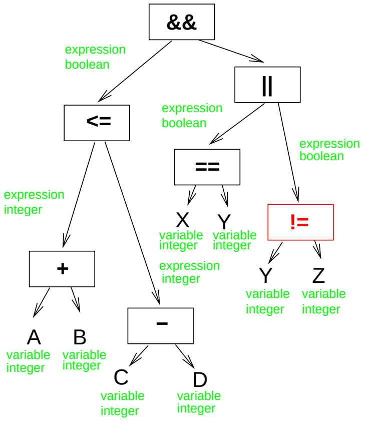
(A + B) <= (C − D) && (X == Y || Y != Z)



Process variables

Expression Processing Example

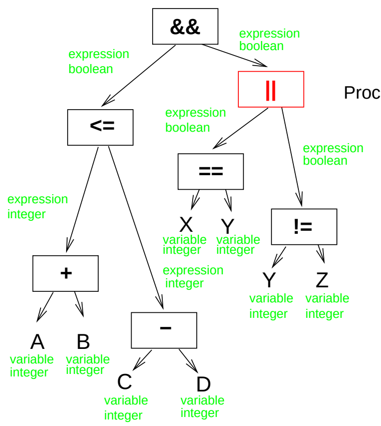
(A + B) <= (C − D) && (X == Y || Y != Z)



Process comparison node

Expression Processing Example

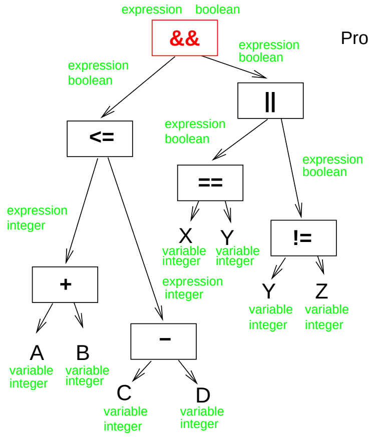
(A + B) <= (C − D) && (X == Y || Y != Z)



Process boolean node

Expression Processing Example

(A + B) <= (C − D) && (X == Y || Y != Z)



Process boolean node