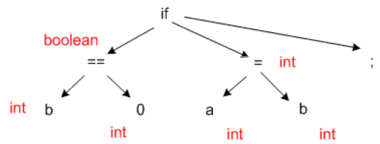
Semantic Analysis

- Check semantics
- Error reporting
- Disambiguate overloaded operators
- Type coercion
- Static checking
  - Type checking
  - Control flow checking
  - Uniqueness checking
  - Name checks



Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language $\{waw \mid w \in \Sigma^*\}$
  - This language is not context free

Beyond syntax …

- Example 1
  string x; int y;
  y = x + 3
  the use of x is a type error
- int a, b;
  a = b + c
  c is not declared
- An identifier may refer to different variables in different parts of the program
- An identifier may be usable in one part of the program but not another

Compiler needs to know?

- Whether a variable has been declared?
- Are there variables which have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?
- Inheritance relationship
- Classes not multiply defined
- Methods in a class are not multiply defined
- The exact requirements depend upon the language

How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases

How to … ?

- Use formal methods
  - Context sensitive grammars
  - Extended attribute grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between !!!
  - Use attributes
  - Do analysis along with parsing
  - Use code for attribute value computation
  - However, code is developed in a systematic way

Why attributes ?

- For lexical analysis and syntax analysis formal techniques were used.
- However, we still had code in form of actions along with regular expressions and context free grammar
- The attribute grammar formalism is important
  - However, it is very difficult to implement
  - But makes many points clear
  - Makes "ad-hoc" code more organized
  - Helps in doing non local computations

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - high level specifications
    - hides implementation details
    - explicit order of evaluation is not specified
  - Translation schemes
    - indicate order in which semantic rules are to be evaluated
    - allow some implementation details to be shown

- Conceptually both:
  - parse input token stream
  - build parse tree
  - traverse the parse tree to evaluate the semantic rules at the parse tree nodes
- Evaluation may:
  - generate code
  - save information in the symbol table
  - issue error messages
  - perform any other activity

Example

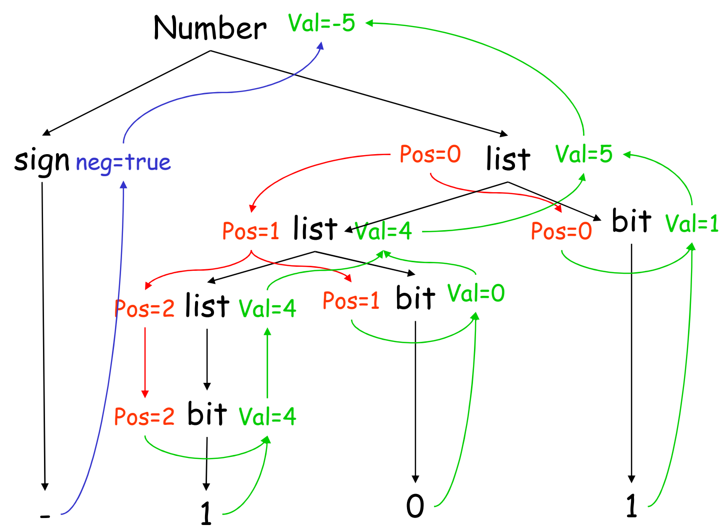
- Consider a grammar for signed binary numbers

  Number → sign list
  sign → + | -
  list → list bit | bit
  bit → 0 | 1

- Build attribute grammar that annotates Number with the value it represents
- Associate attributes with grammar symbols

symbol	**attributes**
Number	value
sign	negative
list	position, value
bit	position, value

Parse tree and the dependence graph



production	Attribute rule
number → sign list	list.position ← 0
	if sign.negative
	then number.value ← - list.value
	else number.value ← list.value
sign → +	sign.negative ← false
sign → -	sign.negative ← true
list → bit	bit.position ← list.position
	list.value ← bit.value
$list_0$ → $list_1$ bit	$list_1$.position ← $list_0$.position + 1
	bit.position ← $list_0$.position
	$list_0$.value ← $list_1$.value + bit.value
bit → 0	bit.value ← 0
bit → 1	bit.value ← $2^{bit.position}$

Attributes …

- attributes fall into two classes: *synthesized* and *inherited*
- value of a synthesized attribute is computed from the values of its children nodes
- value of an inherited attribute is computed from the sibling and parent nodes

Attributes …

- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form

  $b = f\ (c_1, c_2, \ldots, c_k)$

  where f is a function, and either

  - b is a synthesized attribute of A
    OR
  - b is an inherited attribute of one of the grammar symbols on the right
- attribute b depends on attributes $c_1, c_2, \ldots, c_k$

Synthesized Attributes

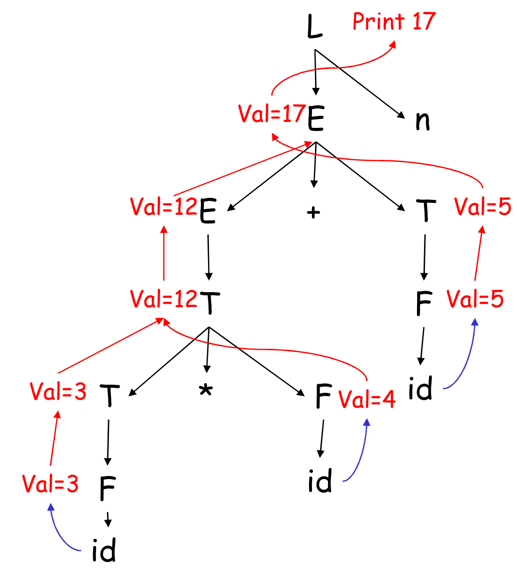
- a syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

Syntax Directed Definitions for a desk calculator program

$L \rightarrow E\ n$	Print (E.val)
$E \rightarrow E + T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ digit	F.val = digit.lexval

- terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer
- start symbol does not have any inherited attribute

Parse tree for 3 * 4 + 5 n

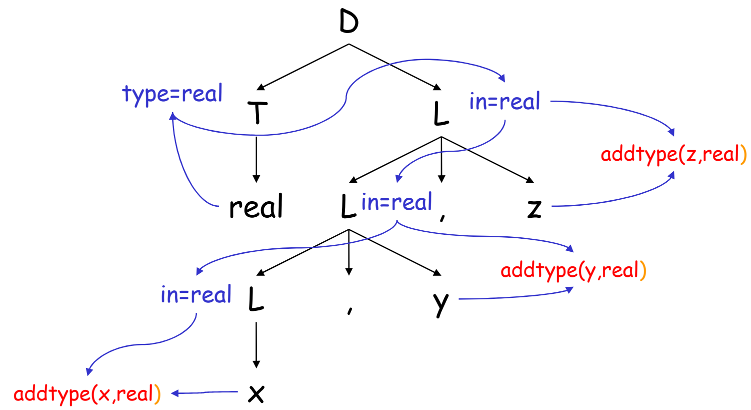


Inherited Attributes

- an inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears
- possible to use only S-attributes but more natural to use inherited attributes

$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ real	T.type = real
$T \rightarrow$ int	T.type = int
$L \rightarrow L_1$, id	L_1.in = L.in; addtype(id.entry, L.in)
$L \rightarrow$ id	addtype (id.entry,L.in)

Parse tree for real x, y, z



Dependence Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph

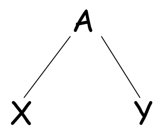
Algorithm to construct dependency graph

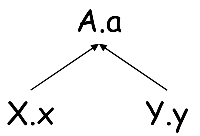
for each node **n** in the parse tree do
  for each attribute **a** of the grammar symbol do
    construct a node in the dependency graph
      for **a**

for each node n in the parse tree do
  for each semantic rule b = f (c_1, c_2 , …, c_k) do
  { associated with production at n }
    for i = 1 to k do
      construct an edge from c_i to b

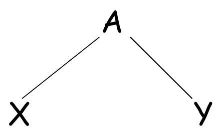
Example

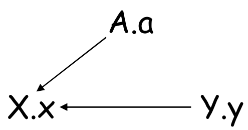
- Suppose A.a = f(X.x , Y.y) is a semantic rule for $A \rightarrow X\ Y$





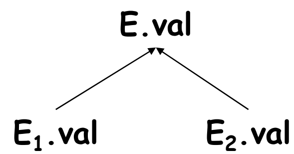
- If production $A \rightarrow X\ Y$ has the semantic rule X.x = g(A.a, Y.y)





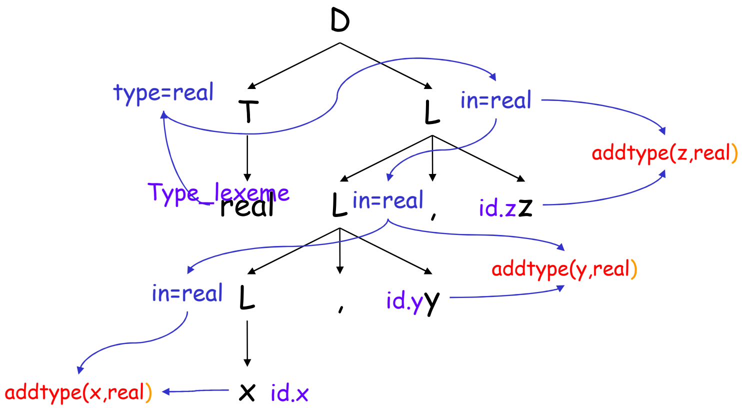
Example

- Whenever following production is used in a parse tree
  $E \rightarrow E_1 + E_2$ $E.val = E_1.val + E_2.val$
  we create a dependency graph



Example

- dependency graph for real id1, id2, id3
- put a dummy node for a semantic rule that consists of a procedure call



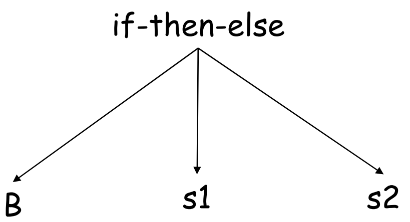
Evaluation Order

- Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

a4 = real
a5 = a4
addtype(id3.entry, a5)
a7 = a5
addtype(id2.entry, a7)
a9 := a7
addtype(id1.entry, a9)

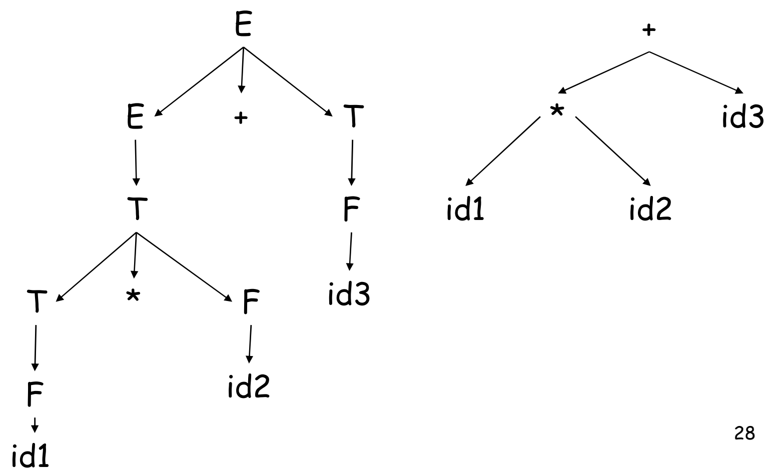
Abstract Syntax Tree

- Condensed form of parse tree,
- useful for representing language constructs.
- The production S → if B then s1 else s2 may appear as



Abstract Syntax tree …

- Chain of single productions may be collapsed, and operators move to the parent nodes



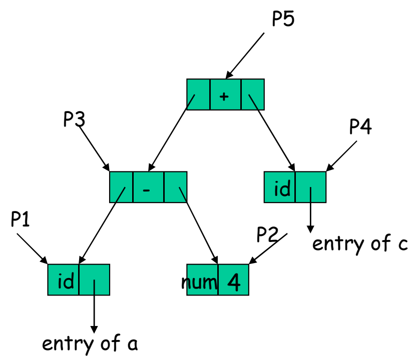
Constructing Abstract Syntax tree for expression

- Each node can be represented as a record
- *operators*: one field for operator, remaining fields ptrs to operands
  mknode(op,left,right)
- *identifier*: one field with label id and another ptr to symbol table
  mkleaf(id,entry)
- *number*: one field with label num and another to keep the value of the number
  mkleaf(num,val)

Example

the following sequence of function calls creates a parse tree for a- 4 + c

P_1 = mkleaf(id, entry.a)
P_2 = mkleaf(num, 4)
P_3 = mknode(-, P_1, P_2)
P_4 = mkleaf(id, entry.c)
P_5 = mknode(+, P_3, P_4)



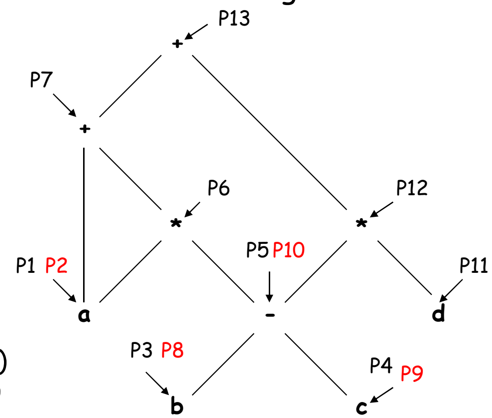
A syntax directed definition for constructing syntax tree

$E \rightarrow E_1 + T$	E.ptr = mknode(+, E_1.ptr, T.ptr)
$E \rightarrow T$	E.ptr = T.ptr
$T \rightarrow T_1 * F$	T.ptr := mknode(*, T_1.ptr, F.ptr)
$T \rightarrow F$	T.ptr := F.ptr
$F \rightarrow (E)$	F.ptr := E.ptr
$F \rightarrow$ id	F.ptr := mkleaf(id, entry.id)
$F \rightarrow$ num	F.ptr := mkleaf(num,val)

DAG for Expressions

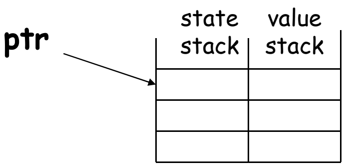
Expression a + a * (b – c) + (b - c) * d
make a leaf or node if not present, otherwise return pointer to the existing node

P_1 = makeleaf(id,a)
P_2 = makeleaf(id,a)
P_3 = makeleaf(id,b)
P_4 = makeleaf(id,c)
P_5 = makenode(-,P_3,P_4)
P_6 = makenode(*,P_2,P_5)
P_7 = makenode(+,P_1,P_6)
P_8 = makeleaf(id,b)
P_9 = makeleaf(id,c)
P_{10} = makenode(-,P_8,P_9)
P_{11} = makeleaf(id,d)
P_{12} = makenode(*,P_{10},P_{11})
P_{13} = makenode(+,P_7,P_{12})



Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also



- The current top of stack is indicated by ptr top

- Suppose semantic rule A.a = f(X.x, Y.y, Z.z) is associated with production A → XYZ
- Before reducing XYZ to A, value of Z is in val(top), value of Y is in val(top-1) and value of X is in val(top-2)
- If symbol has no attribute then the entry is undefined
- After the reduction, top is decremented by 2 and state covering A is put in val(top)

Example: desk calculator

L → En	print(val(top))
E → E + T	val(ntop) = val(top-2) + val(top)
E → T	
T → T * F	val(ntop) = val(top-2) * val(top)
T → F	
F → (E)	val(ntop) = val(top-1)
F → digit	

Before reduction ntop = top - r +1
After code reduction top = ntop

INPUT	STATE	Val	PRODUCTION
3*5+4n			
*5+4n	digit	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T*	3 –	
+4n	T*digit	3 – 5	
+4n	T*F	3 – 5	F → digit
+4n	T	15	T → T * F
+4n	E	15	E → T
4n	E+	15 –	
n	E+digit	15 – 4	
n	E+F	15 – 4	F → digit
n	E+T	15 – 4	T → F
n	E	19	E → E +T

L-attributed definitions

- When translation takes place during parsing, order of evaluation is linked to the order in which nodes are created
- A natural order in both top-down and bottom-up parsing is depth first-order
- L-attributed definition: where attributes can be evaluated in depth-first order

L attributed definitions …

- A syntax directed definition is L-attributed if each inherited attribute of X_j ($1 \le j \le n$) at the right hand side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \ldots X_{j-1}$ and
  - Inherited attribute of A
- Consider translation scheme

$A \rightarrow LM$ $L.i = f_1(A.i)$
$M.i = f_2(L.s)$
$A_s = f_3(M.s)$

$A \rightarrow QR$ $R_i = f_4(A.i)$
$Q_i = f_5(R.s)$
$A.s = f_6(Q.s)$

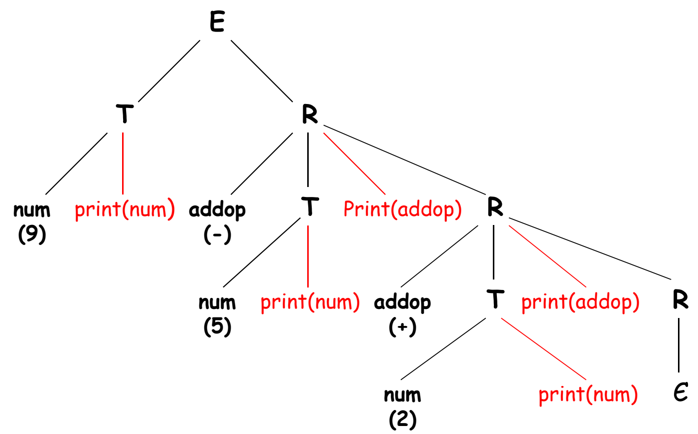
Translation schemes

- A CFG where semantic actions occur within the rhs of production
- A translation scheme to map infix to postfix

E→ T R
R→ addop T {print(addop)} R | ε
T→ num {print(num)}

parse tree for 9 – 5 + 2

Parse tree for 9-5+2

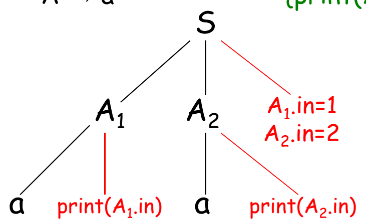


- Assume actions are terminal symbols
- Perform depth first order traversal to obtain 9 5 – 2 +
- When designing translation scheme, ensure attribute value is available when referred to
- In case of synthesized attribute it is trivial (why ?)

- In case of both inherited and synthesized attributes
- An inherited attribute for a symbol on rhs of a production must be computed in an action before that symbol

$S \rightarrow A_1\ A_2 \qquad \{A_1.in = 1, A_2.in = 2\}$

$A \rightarrow a \qquad \{print(A.in)\}$



depth first order traversal gives error *undefined*

- A synthesized attribute for non terminal on the lhs can be computed after all attributes it references, have been computed. The action normally should be placed at the end of rhs

Example: Translation scheme for EQN

$S \rightarrow B$	B.pts = 10 S.ht = B.ht
$B \rightarrow B_1\ B_2$	B_1.pts = B.pts B_2.pts = B.pts B.ht = max(B_1.ht,B_2.ht)
$B \rightarrow B_1$ sub B_2	B_1.pts = B.pts; B_2.pts = shrink(B.pts) B.ht = disp(B_1.ht,B_2.ht)
$B \rightarrow$ text	B.ht = text.h * B.pts

after putting actions in the right place

$S \rightarrow$ {B.pts = 10} B
{S.ht = B.ht}

$B \rightarrow$ {B_1.pts = B.pts} B_1
{B_2.pts = B.pts} B_2
{B.ht = max(B_1.ht,B_2.ht)}

$B \rightarrow$ {B_1.pts = B.pts} B_1 sub
{B_2.pts = shrink(B.pts)} B_2
{B.ht = disp(B_1.ht,B_2.ht)}

$B \rightarrow$ text {B.ht = text.h * B.pts}

Top down Translation

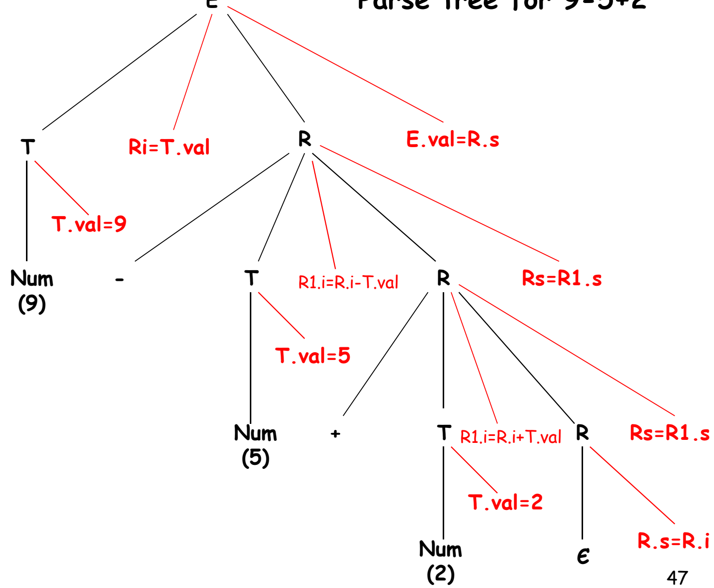
Use predictive parsing to implement L-attributed definitions

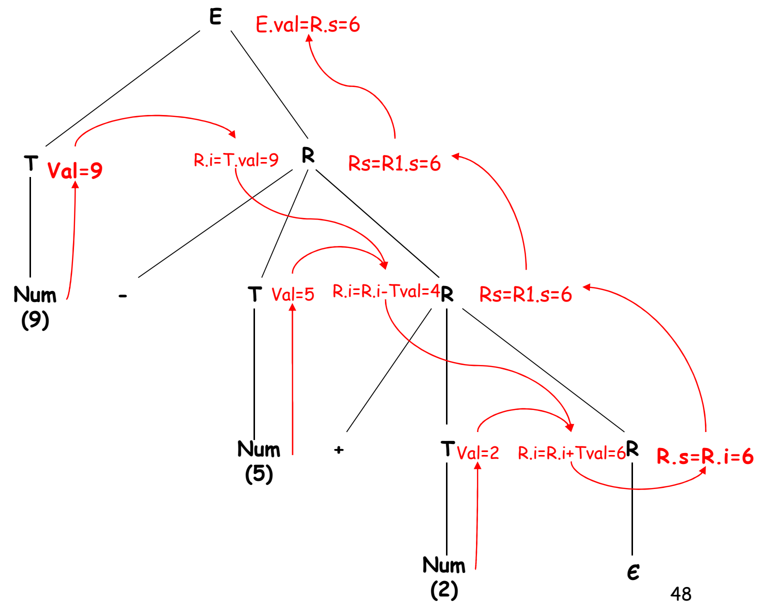
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val := E_1.val - T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow (E)$	$T.val := E.val$
$T \rightarrow num$	$T.val := num.lexval$

Eliminate left recursion

$E \rightarrow$	T	{R.i = T.val}
	R	{E.val = R.s}
$R \rightarrow$	+	
	T	{R_1.i = R.i + T.val}
	R_1	{R.s = R_1.s}
$R \rightarrow$	-	
	T	{R_1.i =R.i – T.val}
	R_1	{R.s = R_1.s}
$R \rightarrow$	ε	{R.s = R.i}
$T \rightarrow$	(E)	{T.val = E.val}
$T \rightarrow$	num	{T.val = num.lexval}

Parse tree for 9-5+2





Removal of left recursion

Suppose we have translation scheme:

$A \rightarrow A_1 Y$ $\{A = g(A_1, Y)\}$
$A \rightarrow X$ $\{A = f(X)\}$

After removal of left recursion it becomes

$A \rightarrow X$ $\{R.in = f(X)\}$
R $\{A.s = R.s\}$
$R \rightarrow Y$ $\{R_1.in = g(Y,R)\}$
R_1 $\{R.s = R_1.s\}$
$R \rightarrow \varepsilon$ $\{R.s = R.i\}$

Bottom up evaluation of inherited attributes

- Remove embedded actions from translation scheme
- Make transformation so that embedded actions occur only at the ends of their productions
- Replace each action by a distinct marker non terminal M and attach action at end of $M \rightarrow \varepsilon$

Therefore,

$E \rightarrow T\ R$
$R \rightarrow +\ T$ {print (+)} R
$R \rightarrow -\ T$ {print (-)} R
$R \rightarrow \epsilon$
$T \rightarrow$ num {print(num.val)}

transforms to

$E \rightarrow T\ R$
$R \rightarrow +\ T\ M\ R$
$R \rightarrow -\ T\ N\ R$
$R \rightarrow \epsilon$
$T \rightarrow$ num {print(num.val)}
$M \rightarrow \epsilon$ {print(+)}
$N \rightarrow \epsilon$ {print(-)}

Inheriting attribute on parser stacks

- bottom up parser reduces rhs of $A \rightarrow XY$ by removing XY from stack and putting A on the stack
- synthesized attributes of Xs can be inherited by Y by using the copy rule Y.i=X.s

Example :take string real p,q,r

$D \rightarrow T$ {L.in = T.type}
L

$T \rightarrow$ int {T.type = integer}
$T \rightarrow$ real {T.type = real}

$L \rightarrow$ $\{L_1.in = L.in\}$ L_1 ,
id $\{addtype(id.entry, L_{in})\}$

$L \rightarrow$ id $\{addtype(id.entry, L_{in})\}$

State stack	INPUT	PRODUCTION
	real p,q,r	
real	p,q,r	
T	p,q,r	$T \to$ real
Tp	,q,r	
TL	,q,r	$L \to$ id
TL,	q,r	
TL,q	,r	
TL	,r	$L \to$ L,id
TL,	r	
TL,r	-	
TL	-	$L \to$ L,id
D	-	$D \to$TL

Every time a string is reduced to L, T.val is just below it on the stack

Example …

- Every tine a reduction to L is made value of T type is just below it
- Use the fact that T.val (type information) is at a known place in the stack
- When production $L \to$ id is applied, id.entry is at the top of the stack and T.type is just below it, therefore,

  addtype(id.entry, L.in) $\Leftrightarrow$ addtype(val[top], val[top-1])

- Similarly when production $L \to L_1$, id is applied id.entry is at the top of the stack and T.type is three places below it, therefore,

  addtype(id.entry, L.in) $\Leftrightarrow$ addtype(val[top],val[top-3])

Example …

Therefore, the translation scheme becomes

$D \to T\ L$	
$T \to$ int	val[top] =integer
$T \to$ real	val[top] =real
$L \to$ L,id	addtype(val[top], val[top-3])
$L \to$ id	addtype(val[top], val[top-1])

Simulating the evaluation of inherited attributes

- The scheme works only if grammar allows position of attribute to be predicted.
- Consider the grammar

  $S \to aAC$ $\qquad C_i = A_s$
  $S \to bABC$ $\qquad C_i = A_s$
  $C \to c$ $\qquad C_s = g(C_i)$

- C inherits A_s
- there may or may not be a B between A and C on the stack when reduction by rule $C \to c$ takes place
- When reduction by $C \to c$ is performed the value of C_i is either in [top-1] or [top-2]

Simulating the evaluation …

- Insert a marker M just before C in the second rule and change rules to

$S \rightarrow aAC$ $\quad C_i = A_s$
$S \rightarrow bABMC$ $\quad M_i = A_s;\ C_i = M_s$
$C \rightarrow c$ $\quad C_s = g(C_i)$
$M \rightarrow \varepsilon$ $\quad M_s = M_i$

- When production $M \rightarrow \varepsilon$ is applied we have $M_s = M_i = A_s$

- Therefore value of C_i is always at [top-1]

Simulating the evaluation …

- Markers can also be used to simulate rules that are not copy rules

$S \rightarrow aAC$ $\quad C_i = f(A.s)$

- using a marker

$S \rightarrow aANC$ $\quad N_i = A_s;\ C_i = N_s$
$N \rightarrow \varepsilon$ $\quad N_s = f(N_i)$

General algorithm

- **Algorithm:** Bottom up parsing and translation with inherited attributes
- **Input**: L attributed definitions
- **Output**: A bottom up parser
- Assume every non terminal has one inherited attribute and every grammar symbol has a synthesized attribute
- For every production $A \rightarrow X_1 \ldots X_n$ introduce n markers $M_1 \ldots M_n$ and replace the production by
  $A \rightarrow M_1\ X_1\ \ldots\ M_n\ X_n$
  $M_1 \ldots M_n \rightarrow \epsilon$
- Synthesized attribute $X_{j,s}$ goes into the value entry of X_j
- Inherited attribute $X_{j,i}$ goes into the value entry of M_j

Algorithm …

- If the reduction is to a marker M_j and the marker belongs to a production

  $A \rightarrow M_1\ X_1 \ldots M_n X_n$ then

  A_i is in position top-2j+2
  $X_{1.i}$ is in position top-2j+3
  $X_{1.s}$ is in position top-2j+4

- If reduction is to a non terminal A by production $A \rightarrow M_1\ X_1 \ldots M_n X_n$ then compute A_s and push on the stack

Space for attributes at compile time

- Lifetime of an attribute begins when it is first computed
- Lifetime of an attribute ends when all the attributes depending on it, have been computed
- Space can be conserved by assigning space for an attribute only during its lifetime

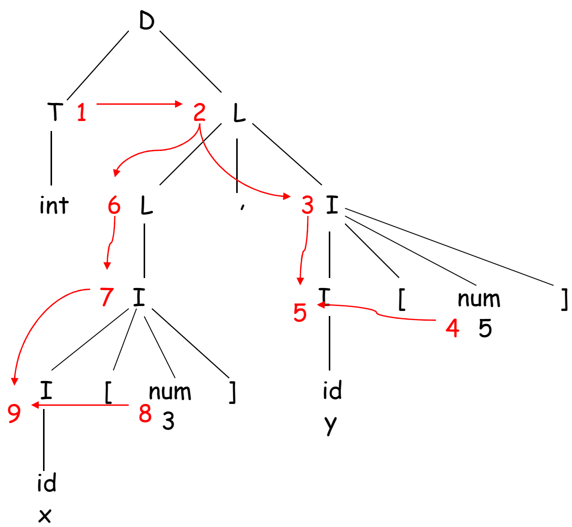
Example

- Consider following definition

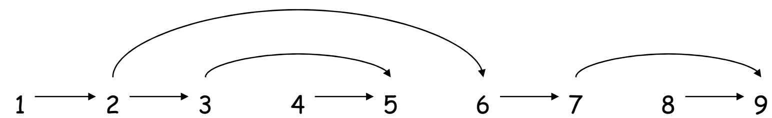
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ real	T.type := real
$T \rightarrow$ int	T.type := int
$L \rightarrow L_1, I$	L_1.in :=L.in; I.in=L.in
$L \rightarrow I$	I.in = L.in
$I \rightarrow I_1$[num]	I_1.in=array(numeral, I.in)
$I \rightarrow$ id	addtype(id.entry,I.in)

Consider string int x[3], y[5]

its parse tree and dependence graph



Resource requirement



Allocate resources using life time information

R1 R1 R2 R3 R2 R1 R1 R2 R1

Allocate resources using life time and copy information

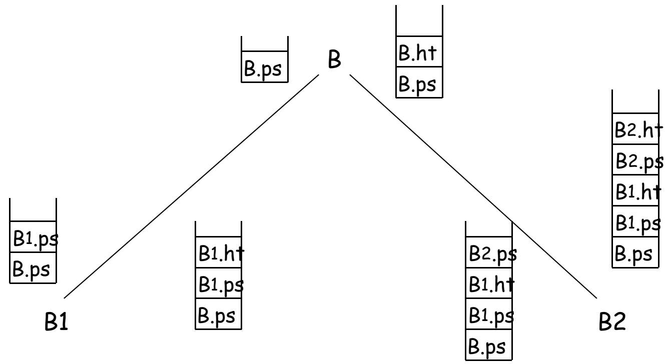
R1 =R1 =R1 R2 R2 =R1 =R1 R2 R1

Space for attributes at compiler Construction time

- Attributes can be held on a single stack. However, lot of attributes are copies of other attributes
- For a rule like $A \rightarrow B\ C$ stack grows up to a height of five (assuming each symbol has one inherited and one synthesized attribute)
- Just before reduction by the rule $A \rightarrow B\ C$ the stack contains I(A) I(B) S(B) I (C) S(C)
- After reduction the stack contains I(A) S(A)

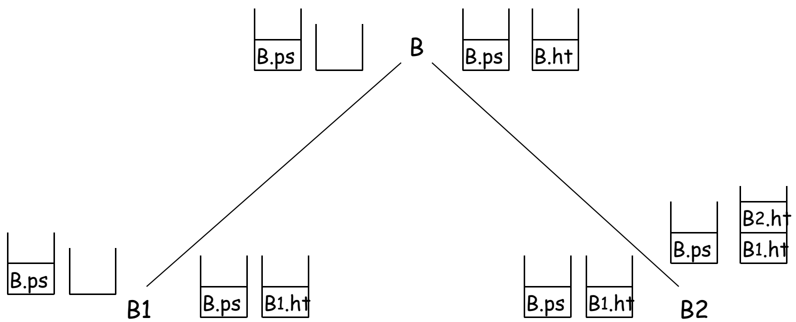
Example

- Consider rule $B \rightarrow B1\ B2$ with inherited attribute ps and synthesized attribute ht
- The parse tree for this string and a snapshot of the stack at each node appears as



Example …

- However, if different stacks are maintained for the inherited and synthesized attributes, the stacks will normally be smaller



Type system

- A type is a set of values
- Certain operations are legal for values of each type
- A language's type system specifies which operations are valid for a type
- The aim of type checking is to ensure that operations are used on the variable/expressions of the correct types

Type system …

- Languages can be divided into three categories with respect to the type:
  - "untyped"
    - No type checking needs to be done
    - Assembly languages
  - Statically typed
    - All type checking is done at compile time
    - Algol class of languages
    - Also, called strongly typed
  - Dynamically typed
    - Type checking is done at run time
    - Mostly functional languages like Lisp, Scheme etc.

Type systems …

- Static typing
  - Catches most common programming errors at compile time
  - Avoids runtime overhead
  - May be restrictive in some situations
  - Rapid prototyping may be difficult
- Most code is written using static types languages
- In fact, most people insist that code be strongly type checked at compile time even if language is not strongly typed (use of Lint for C code, code compliance checkers)

Type System

- A type system is a collection of rules for assigning type expressions to various parts of a program
- Different type systems may be used by different compilers for the same language
- In Pascal type of an array includes the index set. Therefore, a function with an array parameter can only be applied to arrays with that index set
- Many Pascal compilers allow index set to be left unspecified when an array is passed as a parameter

Type system and type checking

- If both the operands of arithmetic operators +, -, x are integers then the result is of type integer
- The result of unary & operator is a pointer to the object referred to by the operand.
  - If the type of operand is *X* the type of result is ***pointer to X***
- **Basic types:** integer, char, float, boolean
- **Sub range type:** 1 … 100
- **Enumerated type:** (violet, indigo, red)
- **Constructed type:** array, record, pointers, functions

Type expression

- Type of a language construct is denoted by a type expression
- It is either a basic type
  or
  it is formed by applying operators called *type constructor* to other type expressions
- A type constructor applied to a type expression is a type expression
- A basic type is type expression. There are two other special basic types:
  - *type error*: error during type checking
  - *void*: no type value

Type Constructors

- Array: if T is a type expression then array(I, T) is a type expression denoting the type of an array with elements of type T and index set I

  var A: array [1 .. 10] of integer

  A has type expression array(1 .. 10, integer)

- Product: if T1 and T2 are type expressions then their Cartesian product T1 **x** T2 is a type expression

Type constructors …

- Records: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
             addr :  integer;
             lexeme :  array [1 .. 15] of char
           end;
```

  var table: array [1 .. 10] of row;

  The type row has type expression

  record ((addr **x** integer) **x** (lexeme **x** array(1 .. 15, char)))

  and type expression of table is array(1 .. 10, row)

Type constructors …

- Pointer: if T is a type expression then pointer(T) is a type expression denoting type pointer to an object of type T
- Function: function maps domain set to range set. It is denoted by type expression D → R
  - For example mod has type expression int **x** int → int
  - function f(a, b: char) : ^ integer; is denoted by

    char **x** char → pointer(integer)

Specifications of a type checker

- Consider a language which consists of a sequence of declarations followed by a single expression

$P \rightarrow D ; E$

$D \rightarrow D ; D \mid id : T$

$T \rightarrow char \mid integer \mid array\ [\ num]\ of\ T \mid \hat{}\ T$

$E \rightarrow literal \mid num \mid E\ mod\ E \mid E\ [E] \mid E\ \hat{}$

Specifications of a type checker …

- A program generated by this grammar is

  key : integer;
  key mod 1999

- Assume following:
  - basic types are char, int, type-error
  - all arrays start at 1
  - array[256] of char has type expression array(1 .. 256, char)

Rules for Symbol Table entry

D → id : T	addtype(id.entry, T.type)
T → char	T.type = char
T → integer	T.type = int
T → ^T_1	T.type = pointer(T_1.type)
T → array [num] of T_1	T.type = array(1..num, T_1.type)

Type checking of functions

E → E_1 (E_2)　　E. type = if E_2.type == s and
E_1.type == s → t
then t
else type-error

Type checking for expressions

E → literal	E.type = char
E → num	E.type = integer
E → id	E.type = lookup(id.entry)
E → E_1 mod E_2	E.type = if E_1.type == integer and E_2.type==integer then integer else type_error
E → E_1[E_2]	E.type = if E_2.type==integer and E_1.type==array(s,t) then t else type_error
E → E_1^	E.type = if E_1.type==pointer(t) then t else type_error

Type checking for statements

- Statements typically do not have values. Special basic type *void* can be assigned to them.

S → id := E	S.Type = if id.type == E.type then void else type_error
S → if E then S1	S.Type = if E.type == boolean then S1.type else type_error
S → while E do S1	S.Type = if E.type == boolean then S1.type else type_error
S → S1 ; S2	S.Type = if S1.type == void and S2.type == void then void else type_error

Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types
  - or these are formed by applying same constructor to equivalent types
- Name equivalence: types can be given names
  - Two type expressions are equivalent if they have the same name

Function to test structural equivalence

```
function sequiv(s, t) :  boolean;
  If s and t are same basic types
    then return true
      elseif s == array(s1, s2) and t == array(t1, t2)
        then return sequiv(s1, t1) && sequiv(s2, t2)
          elseif s == s1 x s2  and  t == t1 x t2
            then return sequiv(s1, t1) && sequiv(s2, t2)
              elseif s == pointer(s1)  and  t == pointer(t1)
                then return sequiv(s1, t1)
                  elseif s == s1→s2  and  t == t1→t2
                    then return sequiv(s1,t1)  &&  sequiv(s2,t2)
                      else return false;
```


Efficient implementation

- Bit vectors can be used to represent type expressions. Refer to: A Tour Through the Portable C Compiler: S. C. Johnson, 1979.

Basic type	Encoding
Boolean	0000
Char	0001
Integer	0010
real	0011

Type constructor	encoding
pointer	01
array	10
function	11

Efficient implementation …

Type expression	encoding
char	000000 0001
function(char)	000011 0001
pointer(function(char))	000111 0001
array(pointer(function(char)))	100111 0001

This representation saves space and keeps track of constructors

Checking name equivalence

- Consider following declarations

  type link = ^cell;
  var next, last : link;
  p, q, r : ^cell;

- Do the variables next, last, p, q and r have identical types ?
- Type expressions have names and names appear in type expressions.
- Name equivalence views each type name as a distinct type

Name equivalence …

variable	type expression
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

- Under name equivalence next = last and p = q = r , however, next ≠ p
- Under structural equivalence all the variables are of the same type

Name equivalence …

- Some compilers allow type expressions to have names.
- However, some compilers assign implicit type names to each declared identifier in the list of variables.
- Consider

  type link = ^ cell;
  var next : link;
  last : link;
  p : ^ cell;
  q : ^ cell;
  r : ^ cell;

- In this case type expression of p, q and r are given different names and therefore, those are not of the same type

Name equivalence …

The code is similar to

```
type link  =  ^ cell
     np = ^ cell;
     nq = ^ cell;
     nr = ^ cell;
var  next : link;
     last : link;
     p : np;
     q : nq;
     r : nr;
```


Cycles in representation of types

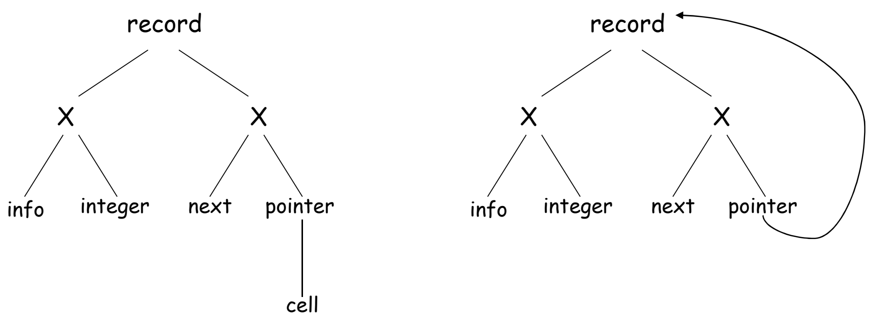
- Data structures like linked lists are defined recursively
- Implemented through structures which contain pointers to structures
- Consider following code

```
type link = ^ cell;
     cell = record
               info : integer;
               next : link
             end;
```

- The type name cell is defined in terms of link and link is defined in terms of cell (recursive definitions)

Cycles in representation of …

- Recursively defined type names can be substituted by definitions
- However, it introduces cycles into the type graph



Cycles in representation of …

- C uses structural equivalence for all types except records
- It uses the acyclic structure of the type graph
- Type names must be declared before they are used
  - However, allow pointers to undeclared record types
  - All potential cycles are due to pointers to records
- Name of a record is part of its type
  - Testing for structural equivalence stops when a record constructor is reached

Type conversion

- Consider expression like x + i where x is of type real and i is of type integer
- Internal representations of integers and reals are different in a computer
  - different machine instructions are used for operations on integers and reals
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.

Type conversion …

- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:

  x + i ⇨ x real+ inttoreal(i)
- Type conversion is called implicit/coercion if done by compiler.
- It is limited to the situations where no information is lost
- Conversions are explicit if programmer has to write something to cause conversion

Type checking for expressions

E → num	E.type = int
E → num.num	E.type = real
E → id	E.type = lookup(id.entry)
E → E_1 op E_2	E.type = if E_1.type == int && E_2.type == int then int elseif E_1.type == int && E_2.type == real then real elseif E_1.type == real && E_2.type == int then real elseif E_1.type == real && E_2.type==real then real

Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In maths + is overloaded; used for integer, real, complex, matrices
- In Ada () is overloaded; used for array, function call, type conversion
- Overloading is resolved when a unique meaning for an occurrence of a symbol is determined

Overloaded functions and operators …

- In Ada standard interpretation of * is multiplication
- However, it may be overloaded by saying

  function "*" (i, j: integer) return complex;
  function "*" (i, j: complex) return complex;

- Possible type expression for " * " are

  integer × integer → integer
  integer × integer → complex
  complex × complex → complex

Overloaded function resolution

- Suppose only possible type for 2, 3 and 5 is integer and Z is a complex variable
  - then 3*5 is either integer or complex depending upon the context
  - in 2*(3*5)

    3*5 is integer because 2 is integer
  - in Z*(3*5)

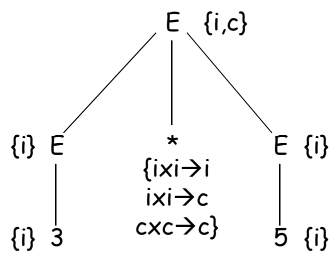
    3*5 is complex because Z is complex

Type resolution

- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type
- Overloading can be resolved in two passes:
  - Bottom up: compute set of all possible types for each expression
  - Top down: narrow set of possible types based on what could be used in an expression

Determining set of possible types

E' → E	E'.types = E.types
E → id	E.types = lookup(id)
E → $E_1(E_2)$	E.types = { t | there exists an s in E_2.types and s→t is in E_1.types}



Narrowing the set of possible types

- Ada requires a complete expression to have a unique type
- Given a unique type from the context we can narrow down the type choices for each expression
- If this process does not result in a unique type for each sub expression then a type error is declared for the expression

Narrowing the set of …

$E' \rightarrow E$	E'.types = E.types E.unique = if E'.types=={t} then t else type_error
$E \rightarrow id$	E.types = lookup(id)
$E \rightarrow E_1(E_2)$	E.types = { t | there exists an s in E_2.types and $s \rightarrow t$ is in E_1.types} t = E.unique S = {s | s∈E2.types and ($s \rightarrow t$)∈E1.types} E_2.unique = if S=={s} then s else type_error E_1.unique = if S=={s} then $s \rightarrow t$ else type_error

Polymorphic functions

- A function can be invoked with arguments of different types
- Built in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic
- Extend type expressions to include expressions with type variables
- Facilitate the implementation of algorithms that manipulate data structures (regardless of types of elements)
  - Determine length of the list without knowing types of the elements

Polymorphic functions …

- Strongly typed languages can make programming very tedious
- Consider identity function written in a language like Pascal
  function identity (x: integer): integer;
- This function is the identity on integers
  identity: int $\rightarrow$ int
- In Pascal types must be explicitly declared
- If we want to write identity function on char then we must write
  function identity (x: char): char;
- This is the same code; only types have changed. However, in Pascal a new identity function must be written for each type

Type variables

- Variables can be used in type expressions to represent unknown types
- Important use: check consistent use of an identifier in a language that does not require identifiers to be declared
- An inconsistent use is reported as an error
- If the variable is always used as of the same type then the use is consistent and has lead to type inference
- Type inference: determine the type of a variable/language construct from the way it is used
  - Infer type of a function from its body

- Consider

```
function deref(p);
  begin
        return p^
  end;
```

- When the first line of the code is seen nothing is known about type of p
  - Represent it by a type variable
- Operator ^ takes pointer to an object and returns the object
- Therefore, p must be pointer to an object of unknown type α
  - If type of p is represented by β then β=pointer(α)
  - Expression p^ has type α
- Type expression for function deref is
  for any type α pointer(α) $\rightarrow$ α
- For identity function   for any type α $\alpha \rightarrow \alpha$