Semantic Analysis

- What can we do with semantic information for identifier $x$?
  - What kind of value is stored in $x$?
  - How big is $x$?
  - Who is responsible for allocating space for $x$?
  - Who is responsible for initializing $x$?
  - How long must the value of $x$ be kept?
  - If $x$ is a procedure, what kinds of arguments does it take and what kind of return value does it have?

- Storage layout for local names
Introduction

- A source program should follow both the syntactic and semantic rules of the source language.

- Some rules can be checked *statically* during compile time and other rules can only be checked *dynamically* during run time.

- Static checking includes the syntax checks performed by the parser and semantic checks such as type checks, flow-of-control checks, uniqueness checks, and name-related checks.

- Here we focus on type checking.
Use of Type

- Virtually all high-level programming languages associate types with values.
- Types often provide an implicit context for operations.
  - In C the expression $x + y$ will use integer addition if $x$ and $y$ are int's, and floating-point addition if $x$ and $y$ are float's.
- Types can catch programming errors at compile time by making sure operators are applied to semantically valid operands.
  - For example, a Java compiler will report an error if $x$ and $y$ are String's in the expression $x \times y$. 
Types

- **Basic types** are atomic types that have no internal structure as far as the programmer is concerned.
  - They include types like `integer`, `real`, `boolean`, and `character`.
  - Subrange types like `1..10` in Pascal and enumerated types like `(violet, indigo, blue, green, yellow, orange, red)` are also basic types.

- **Constructed types** include `arrays`, `records`, `sets`, and `structures` constructed from the basic types and/or other constructed types.
  - **Pointers** and **functions** are also constructed types.
Type Expressions

- **Type Expressions** denote the type of a language construct
  - It is either a basic type or formed from other type expressions by applying an operator called a *type constructor*.
    - Example: a function from an integer to an integer
    - A type constructor applied to a type expression is a type expression.

- Here we use type expressions formed from the following rules:
  - A basic type is a type expression. Other basic type expressions are **type-error** to signal the presence of a type error and **void** to signal the absence of a value.

- If a type expression has a name then the name is also a type expression.
Type Constructors

- **Arrays.** If $T$ is a type expression and $I$ is the type expression of an index set then $array (I, T)$ denotes an array of elements of type $T$.

- **Products.** If $T_1$ and $T_2$ are type expressions, then their Cartesian product, $T_1 \times T_2$, is a type expression.
  - For example if the arguments of a function are two reals followed by an integer then the type expression for the arguments is: `real x real x integer`.

- **Records.** The fields in a record (or structure) have names which should be included in the type expression of the record. The type expression of a record with $n$ fields is:

  \[
  record (F_1 \times F_2 \times \ldots \times F_n)
  \]

  where if the name of field $i$ is `name_i` and the type expression of field $i$ is $T_i$ then $F_i$ is:

  \[
  (name_i \times T_i)
  \]
Type Constructors

- **Pointers.** If $T$ is a type expression then $\text{pointer}(T)$ denotes a pointer to an object of type $T$.

- **Functions.** A function maps elements from its *domain* to its *range*. The type expression for a function is: $D \rightarrow R$ where $D$ is the type expression for the domain of the function and $R$ is the type expression for the range of the function. For example, the type expression of the `mod` operator in Pascal is: $\text{integer} \times \text{integer} \rightarrow \text{integer}$ because it divides an integer by an integer and returns the integer remainder.

- The type expression for the domain of a function with no arguments is $\text{void}$ and the type expression for the range of a function with no returned value is $\text{void}$: e.g., $\text{void} \rightarrow \text{void}$ is the type expression for a procedure with no arguments and no returned value.
Type Systems

- A type system is a set of rules for assigning type expressions to the syntactic constructs of a program and for specifying
  - *type equivalence* - when the types of two values are the same,
  - *type compatibility* - when a value of a given type can be used in a given context
  - *type inference* - rules that determine the type of a language construct based on how it is used.
Type Equivalence

- Forms of type equivalence
  - Name equivalence: two types are equivalent iff they have the same name.
  - Structural equivalence: two types are equivalent iff they have the same structure.
- To test for structural equivalence, a compiler must encode the structure of a type in its representation. A tree (or type graph) is typically used.
Type Checker

- Most all programming languages insist that the type of an ID token be declared before it can be used.
- A type checker makes sure that a program obeys the type-compatibility rules of the language.
- We can think about types in several different ways:
  - Denotational: a type is a set of values called a domain.
  - Constructive: a type is either a primitive type or a composite type created by applying a type constructor to simpler types.
  - Abstraction-based: a type is an interface consisting of a set of operations with well-defined and mutually consistent semantics.
Typing in Programming Languages

• The type system of a language determines whether type checking can be performed at compile time (statically) or at run time (dynamically).

• A statically typed language is one in which all constructs of a language can be typed at compile type.
  • C, ML, and Haskell are statically typed.

• A dynamically typed language is one in which some of the constructs of a language can only be typed at run time.
  • Perl, Python, and Lisp are dynamically typed.

• A strongly typed language is one in which the compiler can guarantee that the programs it accepts will run without type errors.
  • ML and Haskell are strongly typed.

• A type-safe language is one in which the only operations that can be performed on data in the language are those sanctioned by the type of the data.
Type Inference Rules

- Type inference rules specify for each operator the mapping between the types of the operands and the type of the result.
- E.g., result types for $x + y$:

<table>
<thead>
<tr>
<th>Operator</th>
<th>int</th>
<th>float</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>int</td>
<td>int</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>

- Operator and function overloading
  - In Java the operator $+$ can mean addition or string concatenation depending on the types of its operands.
  - We can choose between two versions of an overloaded function by looking at the types of their arguments.
Type Inference Rules - Functions

- Compiler must check that the type of each actual parameter is compatible with the type of the corresponding formal parameter.
- It must check that the type of the returned value is compatible with the type of the function.
- The *type signature* of a function specifies the types of the formal parameters and the type of the return value.
- Example: strlen in C
  - Function prototype in C:
    ```c
    unsigned int strlen(const char *s);
    ```
  - Type expression:
    ```c
    strlen: const char * → unsigned int
    ```
A polymorphic function allows a function to manipulate data structures regardless of the types of the elements in the data structure.

Example: an ML program for the length of a list

```ml
fun length(x) =
    if null(x) then 0 else length(tl(x)) + 1;
```
Type Conversions

- Implicit type conversions
  - In an expression such as \( f + i \) where \( f \) is a float and \( i \) is an integer, a compiler must first convert the integer to a float before the floating point addition operation is performed. That is, the expression must be transformed into an intermediate representation like
    \[
    t1 = \text{INTTOFLOAT} \ i \\
    t2 = x \ \text{FADD} \ t1
    \]

- Explicit type conversions
  - In C, explicit type conversions can be forced ("coerced") in an expression using a unary operator called a cast. E.g., \( \text{sqrt}((\text{double}) \ n) \) converts the value of the integer \( n \) to a double before passing it on to the square root routine \( \text{sqrt} \).
Example Type Checking
Simple Type Checker

- A type checker has two kinds of actions:
  - (1) when processing declarations it stores the appropriate type expressions in the symbol table entries of ID tokens;
  - (2) when processing statements it checks that all ID tokens, constants, etc., are of the proper types.

- Here we describe a translation scheme for treating declarations in the project grammar.
Simple Type Checker

- The type expression for an array has three attributes:
  - \( Typ \)
    - the type of the array (Boolean array, Integer array, or Real array);
  - \( low \)
    - a pointer to the symbol entry of the lowest index of the array; and
  - \( high \)
    - a pointer to the symbol entry of the highest index of the array.

- For consistency, the type expression for a scalar also has three attributes but \( low \) and \( high \) are set to the NULL value.

- The translation scheme for the \textit{type} and \textit{standard_type} nonterminals is shown below (it uses the \textbf{ChangeToArray} function to change a scalar type to an array type and the \textbf{ChkInt} function to report an error if \textit{attributes} does not point to an integer constant.)
**Simple Type Checker**

```
type-->
standard_type { type.typ := standard_type.typ ;
    type.low := NULL ; type.high := NULL ; }
```

```
type-->
ARRAYTOK LBRK { ChkInt() ; type.low := attributes ; } NUM
    DOTDOT { ChkInt() ; type.high := attributes ; } NUM
    RBRK OF TOK standard_type
    { type.typ := ChangeToArray(standard_type.typ) ; }
```

```
standard_type-->
INTTOK { standard_type.typ := integer ; }
```

```
standard_type-->
REALTOK { standard_type.typ := real ; }
```

```
standard_type-->
BOOLTOK { standard_type.typ := boolean ; }
```
Declarations of Scalars and Arrays

- A declaration of scalars or arrays uses the following productions:

  \[
  \text{declaration} \rightarrow \text{ID} \text{ declaration}_{\text{rest}} \\
  \text{declaration}_{\text{rest}} \rightarrow \text{COMMA ID} \text{ declaration}_{\text{rest}} \mid \text{COLON type}
  \]

What is the parse tree for…

\[
\text{id1, id2 : real}
\]
Declarations of Scalars and Arrays

- The *type* node is at the bottom of a chain of *declaration_rest* nodes so it's a simple matter to move the synthesized attributes, *typ*, *low*, and *high*, up the chain and insert them into the symbol table entries of the ID tokens.

- **InsertType** is a function that inserts *typ*, *low*, and *high*, into the appropriate fields of a symbol table entry.
Declarations of Scalars and Arrays

- A subroutine in the source program may declare a local variable with the same name as a global variable so a new symbol table entry must be created for the local variable.

- The translation scheme calls a function, ChkScope, to create such a new entry whenever it is needed.

- ChkScope checks the scope field of the ID-entry that attributes points to:
  - If the scope field of the entry equals CurrentScope then the entry was newly created by the lexical analyzer. The lexeme of the entry was never seen before so there is no conflict with any global variable and ChkScope simply returns a pointer to that entry.

  - If the scope field of the entry doesn't equal CurrentScope then the entry is really for a previously-declared global variable. To prevent a conflict with the global variable, ChkScope creates a new ID entry in the symbol table with the same lexeme as the old entry but with its scope field set to CurrentScope. ChkScope then returns a pointer to the new entry.
Declarations of Scalars and Arrays

- The `parameter_list` nonterminal uses `declaration` to declare the formal parameters of a subroutine and to generate the Cartesian product of all the formal parameters.
- One declaration may declare multiple formal parameters so a fourth synthesized attribute, `prod`, is added - `declaration.prod` is the Cartesian product of all parameter types declared by the declaration.
- **Cartesian** is a function that returns the Cartesian product of two type expressions –
  - if type expressions are character strings then the Cartesian product is simply the concatenation of the two character strings.

- The translation scheme for declarations is:…
Declarations of Scalars and Arrays

declaration --> { idptr := ChkScope() ; } ID declaration_rest

{ declaration.prod := declaration_rest.prod ; InsertType(idptr, declaration_rest.typ ;
declaration_rest.low ; declaration_rest.high) ; } 

declaration_rest --> COMMA { idptr := ChkScope() ; } ID declaration_rest1

{ declaration_rest.typ := declaration_rest1.typ ;
declaration_rest.low := declaration_rest1.low ;
declaration_rest.high := declaration_rest1.high ;
declaration_rest.prod :=
Cartesian( declaration_rest1.prod, declaration_rest.typ ) ; InsertType( idptr, declaration_rest.typ,
declaration_rest.low, declaration_rest.high) ; } 

declaration_rest --> COLON type { declaration_rest.typ := type.typ ;
declaration_rest.low := type.low ;
declaration_rest.high := type.high ;
declaration_rest.prod := type.typ ; }
Declarations of Procedures and Functions

- The type expression of a function or a procedure specifies the number and types of its formal parameters (arguments) with a Cartesian product.

- The project grammar defines the syntax of the formal parameter list with:

  \[ \text{parameter\_list} \rightarrow \text{declaration} \]
  \[ \quad | \text{parameter\_list \ SEMICOL \ declaration} \]

- When left recursion is eliminated we obtain:

  \[ \text{parameter\_list} \rightarrow \text{declaration \ plistrest} \]
  \[ \quad \text{plistrest} \rightarrow \text{SEMICOL} \ \text{declaration \ plistrest} \quad | \ \epsilon \]

- The \textit{parameter\_list} node should return the Cartesian product of the arguments with a synthesized attribute, \textit{prod}.

- The following translation scheme can be used:
Declarations of Procedures and Functions

parameter_list --> declaration plistrest { parameter_list.prod :=
    Cartesian(declaration.prod, plistrest.prod) ; }

plistrest --> SEMICOL declaration plistrest₁ { plistrest.prod :=
    Cartesian(declaration.prod, plistrest₁.prod) ; }

plistrest --> ε { plistrest.prod := void
    /* the empty string if type expressions are character strings */ ; }

/* the empty string if type expressions are character strings */
Arguments

- The *arguments* nonterminal has one synthesized attribute, *arguments.typ*, which is the type expression for the formal parameters followed by the "->" string.

- The translation scheme for this nonterminal is:

  \[
  \begin{align*}
  \text{arguments} & \rightarrow \text{LPAR parameter_list RPAR} \\
  & \{ \text{arguments.typ} := \text{Cartesian( parameter_list.prod, "}\text{->}\text{" ) ;} \\
  \text{arguments} & \rightarrow \varepsilon \{ \text{arguments.typ} := "}\text{->}\text{" ;} \\
  \end{align*}
  \]
Procedures

- The declaration of a procedure uses the following production of the project grammar:
  sub_head--> PROC ID arguments SEMICOL

- The ID token in this production is the name of the procedure being defined:
  - it must be a global symbol so other program units can call it.
  - Any arguments following the name are local variables so a semantic action is needed to increment CurrentScope between the ID token and the arguments.
  - The type expression for the name of the procedure is arguments.typ so the translation scheme for this production is:

```plaintext
sub_head--> PROC { idptr := attributes ; }
ID { CurrentScope++ ; }
arguments { InsertType( idptr, arguments.typ, NULL, NULL ) ; }
SEMICOL
```
Functions

• The declaration of a function uses the following production of the project grammar:

sub_head --> FUNC ID arguments COLON standard_type SEMICOL
Functions

- Pascal has no `return` statement to indicate what value a defined function should return to the caller.
- Instead the compiler declares a local variable with the same name as the function:
  - The body of the defined function sets that local variable to the proper value before returning.
  - For example, the following Pascal function computes the factorial function of any positive integer:

```pascal
function factorial( n : integer ) : integer ;
begin
  if n = 1 then
    factorial := 1
  else
    factorial := n * factorial(n-1)
end;
```
Functions

- Note that in the else-clause of this function, \textit{factorial} on the left side of the assignment operator refers to the local integer but \textit{factorial} on the right-side refers to the global function.

- While compiling the body of a defined function, the compiler must differentiate between calls to execute the function and assignments of values to the returned value of the function.
Functions

- One way to handle this problem is as follows:

- Add a second entry to the symbol table for the returned value.

- Declare two globals in the compiler:
  - FCallPtr to point to entry of the function itself;
  - and FRetValPtr to point to the entry of the returned value.

- Statements in the grammar will compare the pointer of every ID entry to these compiler globals to change the pointer when necessary.

- FCallPtr and FRetValPtr are given NULL values except when compiling the body of a function.
Functions

- The translation scheme uses the INSERT function to add the second entry to the symbol table:

```
sub_head---> FUNC { FCallPtr := attributes ; } ID { CurrentScope++ ;
    FRetValPtr := INSERT( FCallPtr.lexeme, ID ) ; }
arguments COLON standard_type
    { InsertType( FRetValPtr, standard_type.typ, NULL, NULL ) ;
    InsertType( FCallPtr, Cartesian( arguments.typ,
        standard_type.typ ), NULL, NULL ) ; }
SEMICOL
```
The End of A Subroutine

- Nonterminal `subroutine` in the project grammar defines the syntax of subroutine:

  `subroutine --> sub_head declarations block`

- Local symbols are only valid until the end of a subroutine so a semantic action is needed at that point to negate all `scope` fields in the symbol table that equal `CurrentScope` (as a debugging aid for project 2 this semantic action could also list the lexemes and type expressions of all entries it invalidates.)

- After that semantic action `CurrentScope` should be decremented and compiler globals `FCallPtr` and `FRetValPtr` set to NULL values.
The End of A Subroutine

- The translation scheme looks like:

```
subroutine --> sub_head declarations block
  { negate all scope fields that equal CurrentScope ;
    CurrentScope-- ; FCallPtr := NULL ; FRetValPtr := NULL ;
  }
```
Type Checking Statements

- Left-factoring the productions for the `statement` nonterminal in the project grammar produces the following:

  - `statement` --> ID `stmt_rest`
  - `statement` --> `BEGIN TOK` `block_rest`
  - `statement` --> `IFTOK expr THENTOK` `ELSE TOK` `statement`
  - `statement` --> `WHILE TOK expr DOTOK` `statement`
  - `stmt_rest` --> `ASSIGNOP expr`
  - `stmt_rest` --> `LBRK expr RBRK ASSIGNOP expr`
  - `stmt_rest` --> `LPAR expr_list RPAR`
  - `stmt_rest` --> ε

- Other nonterminals on the right-sides of these productions are `block_rest`, `expr` and `expr_list` but `block_rest` needs no semantic actions so we ignore it.
Type Checking Statements

- **expr:**
  - The parent of an `expr` node in the parse tree needs to know both the lexeme and the type of the expression so the `expr` nonterminal has a synthesized attribute, `expr.ptr`, that points to the symbol table entry of the expression.
  - In project 2 the only productions for `expr` are:

    expr--> NUM
    expr--> BCONST

- A translation scheme for `expr` in project 2 is simply:

    expr--> {expr.ptr := attributes ; } NUM
    expr--> {expr.ptr := attributes ; } BCONST

- Note that we place the semantic actions before the tokens in these productions as a reminder that `attributes` should be read before the tokens are matched.
Type Checking Statements

- **expr_list:**
  - The productions for `expr_list` are:

    - `expr_list --> expr`
    - `expr_list --> expr_list COMMA expr`

  - But these productions must be modified to eliminate left recursion:

    - `expr_list --> expr elistrest`
    - `elistrest --> COMMA expr elistrest`
    - `elistrest --> ε`
Type Checking Statements

- The `expr_list` nonterminal returns the Cartesian product of all expressions in the list as a synthesized attribute, `expr_list.typexpr`.

- We assume there is a `GetType` function that accepts a pointer to a symbol table entry and returns the type expression of that entry.

- A translation scheme for these productions is:

  expr_list---> expr elistrest { expr_list.typexpr :=
    Cartesian( GetType( expr.ptr), elistrest.typexpr ) ; }
  elistrest---> COMMA expr elistrest1 { elistrest.typexpr :=
    Cartesian( GetType( expr.ptr), elistrest1.typexpr ) ; }
  elistrest---> ε  { elistrest.typexpr := void
    /* the empty string if type expressions are character strings */ ; }
Type Checking Statements

- **stmt_rest:** The `stmt_rest` nonterminal accepts a pointer to the symbol table entry of an ID token in an inherited attribute, `stmt_rest.idptr`.

- We assume the type system described [here](#).

- Type checking in the four productions for `stmt_rest` is described in the following paragraphs (\(t1\) and \(t2\) are used as temporary placeholders of type expressions.)
Type Checking Statements

- The first production assigns the value of an expression to a scalar variable so there is a type-error if `stmt_rest.idptr` does not point to a scalar.

- Integer-to-real and real-to-integer type conversions are allowable so the only other type-errors that can occur are when a boolean is assigned to a non-boolean or a non-boolean is assigned to a boolean:

```
stmt_rest--> ASSIGNOP { t1 := GetType(stmt_rest.idptr) ;
    if t1 != 'b' and t1 != 'i' and t1 != 'r' then
        type-error ; }
expr { t2 := GetType(expr.ptr) ;
    if (t1 != 'b' and t2 == 'b') or
        (t1 == 'b' and t2 != 'b') then
        type-error ; }
```
Type Checking Statements

- The second production assigns the value of an expression to an element of an array so there is a type-error if `stmt_rest.idptr` does not point to an array.

- Also there is a type-error if the expression for the index is not an integer.

- Integer-to-real and real-to-integer type conversions are allowable so the only other type-errors that can occur are when a boolean is assigned to a non-boolean or a non-boolean is assigned to a boolean:

```plaintext
stmt_rest --> LBRK { t1 := GetType(stmt_rest.idptr) ;
if t1 != 'B' and t1 != 'I' and t1 != 'R' then
  type-error ; }
  expr1 { if GetType(expr1.ptr) != 'i' then type-error ; }
RBRK ASSIGNOP
  expr2 { t2 := GetType(expr2.ptr) ;
if (t1 != 'B' and t2 == 'b') or
  (t1 == 'B' and t2 != 'b') then type-error ; }
```
Type Checking Statements

- The third production calls a procedure with one or more arguments. The type expression of \textit{stmt\_rest.idptr} should equal \textit{expr\_list.typexpr} with a '>' character appended to it:

\begin{verbatim}
stmt_rest---> LPAR { t1 := GetType(stmt_rest.idptr) ;
    expr_list { t2 := Cartesian(expr_list.typexpr, "">") ;
    if t1 != t2 then type-error ; } RPAR
\end{verbatim}
Type Checking Statements

• The fourth production calls a procedure with no arguments. The type expression of `stmt_rest.idptr` should simply be the '=>' character:

```
stmt_rest--> ε { if GetType(stmt_rest.idptr) != "->" then type-error ; }
```

• Note that intermediate code generation adds other semantic actions to all four productions for `stmt_rest`.
Type Checking Statements

- The third and fourth productions should check that \( expr \) is a boolean. Note that intermediate code generation adds other semantic actions to these two productions:

\[
\begin{align*}
\text{statement} & \rightarrow \text{IFTOK expr} \\
& \quad \{ \text{if GetType(expr.ptr) } \neq 'b' \text{ then type-error ; } \} \\
& \quad \text{THEN TOK statement ELSE TOK statement}
\end{align*}
\]

\[
\begin{align*}
\text{statement} & \rightarrow \text{WHILE TOK expr} \\
& \quad \{ \text{if GetType(expr.ptr) } \neq 'b' \text{ then type-error ; } \} \\
& \quad \text{DOTOK statement}
\end{align*}
\]
Semantic Rules for Type Checking

\[ P \rightarrow D; S \]
\[ D \rightarrow D; D \]
\[ D \rightarrow \text{id} : T \quad \text{addvar(id.value, T.type)} \]
\[ T \rightarrow \text{char} \quad \text{T.type} = \text{char} \]
\[ T \rightarrow \text{integer} \quad \text{T.type} = \text{integer} \]
\[ T \rightarrow \text{\^T}_1 \quad \text{T.type} = \text{pointer(T}_1.\text{type)} \]
\[ T \rightarrow \text{array[num] of T}_1 \quad \text{T.type} = \text{array(num, T}_1.\text{type)} \]
\[ S \rightarrow \text{id := E} \quad \text{if lookup(id).type} <> \text{E.type} \text{err} \]
\[ S \rightarrow \text{if E then S1} \quad \text{if E.type} <> \text{boolean} \text{err} \]
\[ E \rightarrow \text{id} \quad \text{E.type} = \text{lookup(id)} \]
\[ E \rightarrow E_1 \text{ relop E}_2 \quad \text{if } E_1 \text{ & } E_2 \text{ bool } E.\text{type} = \text{bool else err} \]
\[ E \rightarrow E_1 \text{ op E}_2 \quad \text{if } E_1.\text{type} == E_2.\text{type} \quad E.\text{type} = E_1.\text{type} \]
\[ \quad \text{if types(float, int) } E.\text{type} = \text{float ...} \]
Type Checking in YACC

/* Lex spec */
[0-9]+                   yylval.ival = atoi(yytext); return ICONST
[0-9]+"."[0-9]*          yylval.fval = atof(yytext); return FCONST

/* YACC spec*/
struct Info {  int intval; float floatval; int type; }
/* Definition for YYLVAL, this struct will get passed on the parse stack */

%union{
  int ival;
  float fval;
  struct Info info;
}
%token <ival> ICONST
%token <fval> FCONST
Type Checking in YACC

%%

e: e ' + ' e { if ($<info.type>1 == 1 && $<info.type>3 == 1) {
    $<info.type>$ = 1;
    $<info.ival>$ = $<info.ival>1 + $<info.ival>3;
}
    if ($<info.type>1 == 2 && $<info.type>3 == 2) {
    $<info.type>$ = 2;
    $<info.fval>$ = $<info.fval>1 + $<info.fval>3;
}
}

| e: ICONST   { $<info.ival>$ = $1; $<info.type>$ = 1; }
| FCONST     { $<info.fval>$ = $1; $<info.type>$ = 2; }