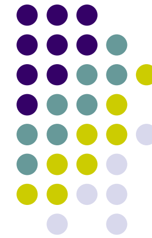
CSc 453
Semantic Analysis

Saumya Debray
The University of Arizona
Tucson



Need for Semantic Analysis

- Not all program properties can be represented using context-free grammars.
  E.g.: "*variables must be declared before use*" is not a context-free property.
- Parsing context-sensitive grammars is expensive.
- As a pragmatic measure, compilers combine context-free and context-sensitive checking:
  - Context-free parsing used to check "code shape;"
  - Additional rules used to check context-sensitive aspects.

Syntax-Directed Translation

- *Basic Idea*:
  - Associate information with grammar symbols using *attributes*.

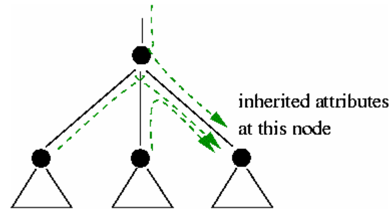
    An attribute can represent any reasonable aspect of a program, e.g., character string, numerical value, type, memory location, etc.
  - Use *semantic rules* associated with grammar productions to compute attribute values.
- A parse tree showing attribute values at each node is called an *annotated parse tree*.
- *Implementation*: Add code to parser to compute and propagate attribute values.

Example: Attributes for an Identifier

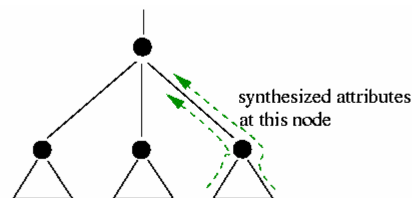
- *name*: character string (from scanner)
- *scope*: global, local, …
  - if local: whether or not a formal parameter
- *type*:
  - integer
  - array:
    - no. of dimensions
    - upper and lower bound for each dimension
    - type of elements
  - struct:
    - name and type of each field
  - function:
    - number and type of arguments (in order)
    - type of returned value
    - entry point in memory
    - size of stack frame
- …

Types of Attributes

- *Inherited attributes*: An attribute is *inherited* at a parse tree node if its value is computed at a parent or sibling node.



- *Synthesized attributes*: An attribute is *synthesized* at a parse tree node if its value is computed at that node or one of its children.



Example: A Simple Calculator

Production	***Semantic Rule***
$E \to E_1 + E_2$	$E.val = E_1.val \oplus E_2.val$
$E \to E_1 * E_2$	$E.val = E_1.val \otimes E_2.val$
$E \to (E_1)$	$E.val = E_1.val$
$E \to$ **intcon**	$E.val =$ **intcon**.*val*

Symbol Tables

- *Purpose*: To hold information (i.e., attribute values) about identifiers that get computed at one point and used later.

  E.g.: type information:
  - computed during parsing;
  - used during type checking, code generation.
- *Operations*:
  - create, delete a symbol table;
  - insert, lookup an identifier
- *Typical implementations*: linked list, hash table.

Semantic Actions in Yacc

- Semantic actions are embedded in RHS of rules.

  An action consists of one or more C statements, enclosed in braces **{** … **}**.

- *Examples*:

```
ident_decl : ID    { symtbl_install( id_name ); }

type_decl : type  { tval = … } id_list;
```

Semantic Actions in Yacc: cont'd

Each nonterminal can return a value.

- The value returned by the ith symbol on the RHS is denoted by **$i**.
- An action that occurs in the middle of a rule counts as a "symbol" for this.
- To set the value to be returned by a rule, assign to **$$**.

  *By default, the value returned by a rule is the value of the first RHS symbol, i.e., **$1**.*

Yacc: Declaring Return Value Types

- Default return value for symbols is **int**.
- We may want other types of return values, e.g., symbol table pointers, syntax tree nodes.

Declare the various kinds of values that may be returned:

```
%union {
   symtab_ptr   st_ptr;
   idlist_ptr   idents;
   tree_node    tn_ptr;
   int          val;
}
```

Specify return type for each grammar symbol:

```
/* tokens: */
%token <val> INTCON;

/* nonterminals: */
%type <st_ptr> ident;
%type <tn_ptr> expr;
```

Semantic Actions in Yacc: *Example 1*

func : [1] *type* [2] { ret_type = $1; }

[3] **ID** [4] { this_fn = symtbl_install(id_name); }

[5] '(' [6] { scope = LOCAL; }

[7] *formals*

[8] ')'

[9] '{'

[10] *decls*

[11] *stmt* [12] { this_fn→body = $11; }

[13] '}' [14] { scope = GLOBAL; }

Semantic Actions in Yacc: *Example 2*

A simple calculator in Yacc:

```
E :   E '+' E      { $$ = $1 + $3; }

E :   E '*' E      { $$ = $1 * $3; }

E :  '(' E ')'     { $$ = $2; }

E :   intcon       { $$ = $1.val; }
```

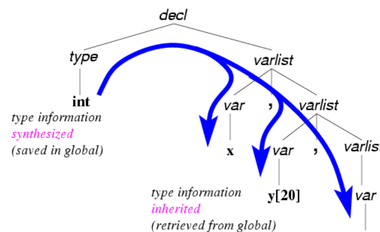
Managing Scope Information

- When looking up a name in a symbol table, we need to find the "appropriate" declaration.
  The scope rules of the language determine what is "appropriate."
- Often, we want the *most deeply nested* declaration for a name.
- *Implementation*: for each new scope: push a new symbol table on entry; pop on exit (*stack*).
  - implement symbol table stack as a linked list of symbol tables;
    newly declared identifiers go into the topmost symbol table.
  - lookup: search the symbol table stack from the top downwards.

Processing Declarations

Production	Semantic Rule
decl → type varlist ';'	varlist.*tval* = type.*tval*;
varlist → var ',' $varlist_1$	var.*type* = varlist.*type*; $varlist_1$.*type* = varlist.*type*;

xxx : inherited
yyy : synthesized



Processing Declarations: cont'd

```
decl : type { tval = $1; } varlist ;

varlist : var varlist
        | var ;

var : ID opt_subscript { symtbl_insert($1, $2); } ;
```


Static Checking

Static checking aims to ensure, at compile time, that syntactic and semantic constraints of the source language are obeyed. E.g.:

- *Type checks*: operators and operands must have compatible types.
- *Flow-of-control checks*: control transfer statements must have legitimate targets (e.g., break/continue statements).
- *Uniqueness checks*: a language may dictate unique occurrences in some situations, e.g., variable declarations, case labels in switch statements.

These checks can often be integrated with parsing.

Data Types and Type Checking

- A data type is a set of values together with a set of operations that can be performed on them.
- Type checking aims to verify that operations in a program code are, in fact, permissible on their operand values.
- Reasoning about types:
  - The language provides a set of base types and a set of type constructors;
  - The compiler uses type expressions to represent types definable by the language.

Type Constructors and Type Expressions

A *type expression* denotes (i.e., is a syntactic representation of) the type of a program entity:

- A base type is a type expression (e.g., boolean, char, int, float);
- A type name is a type expression;
- A type constructor applied to type expressions is a type expression, e.g.:
  - *arrays*: if T is a type expression then so is array(T);
  - *records*: if $T_1, \ldots, T_n$ are type expressions and $f_1, \ldots, f_n$ is a list of (unique) identifiers, then $record(f_1{:}T_1, \ldots, f_n{:}T_n)$ is a type expression;
  - *pointers*: if T is a type expression then so is ptr(T);
  - *functions*: if $T, T_1, \ldots, T_n$ are type expressions, then so is $(T_1, \ldots, T_n) \rightarrow T$.

Why use Type Expressions?

```
char *X;                 main()
char **f()               {
{                           printf("%c\n", (*f())[2]);
   X = "GOTCHA!"                                   /* legal??? */
   return &X;            }
}
```

Program Code	Type Expression	Rule
`f`	`()→ptr(ptr(char))`	symbol table lookup
`f()`	`ptr(ptr(char))`	**if** $e : T_1 \rightarrow T_2$ and $e_1 : T_1$ **then** $e(e_1) : T_2$
`*f()`	`ptr(char)`	**if** $e : ptr(T)$ **then** $*e : T$
`*f()`	`array(char)`	**if** $e : ptr(T)$ **then** $e : array(T)$
`(*f())`	`array(char)`	**if** $e : T$ **then** $(e) : T$
2	`int`	base type
`(*f())[2]`	`char`	**if** $e_1 : array(T)$ and e_2 : `int` **then** $e_1[e_2] : T$

What about:

```
qsort((void **)lptr,0,k,(int (*)(void*,void*))(num ? ncmp : strcmp));
```

CSc 453: Semantic Analysis 19

Notions of Type Equivalence

1. *Name equivalence*:
   In some languages (e.g., Pascal), types can be given names.
   Name equivalence views distinct type names as distinct types: two types are name equivalent if and only if they have the same type name.
2. *Structural equivalence*:
   Two type expressions are structurally equivalent if they have the same structure, i.e., if both apply the same type constructor to structurally equivalent type expressions.

E.g.: in the Pascal fragment

```
type p = ↑node;
     q = ↑node;
var x : p;
    y : q;
```

`x` and `y` are structurally equivalent, but not name-equivalent.

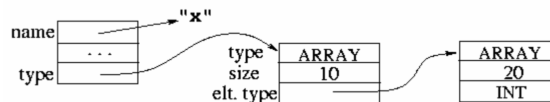
CSc 453: Semantic Analysis 20

Representing Type Expressions

Type graphs: A graph-structured representation of type expressions:

- Basic types are given predefined “internal values”;
- Named types can be represented via pointers into a hash table.
- A composite type expression $f(T_1,\dots,T_n)$ is represented as a node identifying the constructor f and with pointers to the nodes for $T_1, \dots, T_n$.

  E.g.: **`int x[10][20]:`**



Type Checking Expressions

Production	***Semantic Rule***	***Yacc Code***
E → **id**	E.*type* = **id**.*type*	{ $$ = symtab_lookup(id_name); }
E → **intcon**	E.*type* = INTEGER	{ $$ = INTEGER; }
E → E_1 + E_2	E.*type* = result_type(E_1.*type*, E_2.*type*)	{ $$ = result_type($1, $3); }

```
/* arithmetic type conversions */
Type result_type(Type t1, Type t2)
{
  if (t1 == error || t2 == error) return error;
  if (t1 == t2) return t1;
  if (t1 == double || t2 == double) return double;
  if (t1 == float || t2 == float) return float;
  …
}
```

Return types:

- currently: the type of the expression
- down the road:
  - type
  - location
  - code to evaluate the expression

Type Checking Expressions: cont'd

Arrays:

E → **id**[E_1] { t_1 = **id**.*type*;
 if (t_1 == ARRAY ∧ E_1.*type* == INTEGER)
 E.*type* = **id**.*element_type*;
 else
 E.*type* = *error*;
}

Type Checking Expressions: cont'd

Function calls:

E → **id** '**(**' expr_list '**)**'
{ **if** (**id**.*return_type* == VOID)
 E.*type* = *error*;
 else if (chk_arg_types(**id**, expr_list)) /* actuals match formals in number, type */
 E.*type* = **id**.*return_type*;
 else
 E.*type* = *error*;
}

Type Checking Statements

Different kinds of statements have different type requirements. E.g.:

- **if**, **while** statements may require boolean conditiona;
- LHS of an assignment must be an "l-value", i.e., something that can be assigned.
- LHS and RHS of an assignment must have "compatible" types. If they are of different types, conversion will be necessary.

Operator Overloading

- *Overloading* refers to the use of the same syntax to refer to different operations, depending on the operand types.
  E.g.: in Java, '+' can refer to integer addition, floating point addition, or string concatenation.
- The compiler uses operand type information to r*esolve* the overloading, i.e., figure out which operation is actually referred to.
  If there is insufficient information to resolve overloading, the compiler may give an error.