Chapter 5: Semantic Actions

Constructing ASTs for expressions

- mknode(op, left, right)
- mkleaf(id, entry)
- mkleaf(num, val)
- A bottom-up translation scheme
  - the parser stack,
  - the semantic stack,
  - the semantic actions to allocate and link the tree nodes

Types

- At the machine level, different instructions (operations) may use different *machine-level representation* for their operands. E.g. *integer* arithmetic ops vs. *floating point* arithmetic operations.
- Therefore, at the machine level, data have *types*, which determine how they are stored in the *memory* and which *registers* can store them (*fixed point* vs. floating point, 32 bits vs. 64 bits, e.g.)
- Determination of a type (of a data item or an operation) is called *binding* (of the data item or the operation).

- A type can be determined during the program execution (*dynamic binding*, or *late binding*), or when the program is compiled (*static-binding*).

- Static binding usually makes the machine code much more efficient than late binding, but dynamic binding makes the source code more flexible.

- Types of *primitive operations* (such as arithmetic operations) are usually inferred from the operand types, without explicit type specification. This is called *overloading*.

- High-level operations, e.g. *function calls*, may also be overloaded, e.g. in C++.

- *High-level data types*, e.g. structures, make programs more structured.

Type Rules

- Whether every name (ID) must be declared explicitly.
- What names have predefined meanings (e.g. *intrinsic functions*.)
- What is the scope (*name scope*) in the program in which a declaration applies.
- *Type conformance*: what data types must the operands of an operation have. (Also, how many parameters for a particular function?)
- *Type coercision*: what are the implicit rules to transform a data type into another such that operands

are type-conformant.

- Can a data item be declared more than once in exactly the same scope?

What does the compiler need to do?

- In each *executable statement*, for every ID, determine whether it has been properly declared unless it is an intrinsic function name.
- In each *declaration statement*, make sure each programmer-defined type name has been properly declared.
- Make sure each function call has the proper number of parameters.
- If the type rules require so, then
  - make sure logical operations use integer operands,
  - make sure IF and WHILE conditions are integers, etc.

The *Symbol Table* is the information clearing-house for type-checking.

- Entries of type information are inserted when declaration statements are parsed.
- Entries are retrieved when executable statements are parsed.
- The top-level organization of the symbol table (how to deal with *nested blocks*):
  - A *central table*: A linked list for each ID.
  - A *tree* of sub-tables.
  - A *stack* of sub-tables.

* Disadvantage of using a stack of sub-tables: does not support code generation based on global program information. All right for code generation for individual blocks.

- Some entries can be inserted (initialized) before parsing, e.g. intrinsic functions.
- Each ID has a *class* attribute: a *variable*, a *type*, an *intrinsic function*?
- Each variable may have an *initial value*.
- A structure has an attribute of the *number of fields* and a sub-table for these fields.
- An intrinsic function has an attribute of the *num-*

ber of parameters and a sub-table for these parameters.

- The symbol table may be *sorted* by ID names or may use a *hash table*.