Now the Specifics of Semantic Action Routines
A Common Compiler Structure: Semantic Actions Generate ASTs

• In many compilers, the sequence of semantic actions generated by the parser build an abstract syntax tree (AST, or simply syntax tree.)
• After this step, many compiler passes operate on the syntax tree.
Tree Traversals

After the AST has been built, it is traversed several times, for

- testing attributes of the tree (e.g., type checking)
- testing structural information (e.g., number of subroutine parameters)
- optimizations
- output generation.
Semantic Actions and LL/LR Parsers

- Actions are called either by parsing routines or by the parser driver. Both need provisions for semantic record parameter passing.

  Example:

  ```
  <if-stmt> → IF <expr> #start-if THEN <stmt-list> ENDIF #finish-if
  ```

- For LL parsers, semantic actions are a perfect fit, thanks to their predictive nature.

- In LR parsers, productions are only recognized at their end. It may be necessary to split a production, generating “semantics hooks”

  ```
  <if-stmt> → <begin-if> THEN <stmt-list> ENDIF #finish-if
  <begin-if> → IF <expr> #start-if
  ```
Semantic Records

• Associated with both nonterminals and terminals.
• Can also be empty (e.g., for “;” or <stmt-list>).
• Control statements often have 2 or more actions.
• Typically semantic record information is generated by actions at symbols and is passed to actions at the end of productions.

A good organization of the semantic records is the semantic stack.
Semantic Stack Example

• consider \( a := b + 1 \)

• sequence of parse actions invoked:
  - process_id, process_id, process_op, process_lit, gen_infix, gen_assign

```
process_id  process_id  process_op  process_lit  gen_infix  gen_assign
a            b           +           b+1          a           a
```
Action-Controlled Semantic Stack

- Action routines can push/pop semantic records directly onto/from the stack. This is called *action-controlled* stack.
  - Disadvantage: stack management has to be implemented in action routines.
LR Parser-Controlled Stack

• Every shift operation pushes a semantic record onto the semantic stack, describing the token.

• At a reduce operation, the production produces a semantic record and replaces all RHS records on the stack with it.

The action procedures don’t see the stack. They see the semantic records as parameters.

Example: YACC (use $1, $2, etc. as args)
LL Parser-Controlled Stack

Remember: the parse stack contains predicted symbols, not the symbols already parsed.

• Entries for all RHS symbols (left-to-right) are also pushed onto the semantic stack and gradually filled in.

• When a production is matched: the RHS symbols are popped, the LHS symbol remains.

• Keep pointers to left, right, current symbol, and top for each production in progress. Recursively store these values in a EOP (end of production) symbol as nonterminals on the RHS are parsed.

  – Algorithm and example on pages 238-241.
Runtime Storage Organization
(overview of Chapter 9, which we don’t really cover)

• Activation records (will be discussed later)
• Heap allocation
  – explicit malloc, free
  – implicit heap allocation (e.g., Lisp)
  – Explicit alloc, implicit free (Java)
• Program layout in memory
• Procedure parameters (“this” pointer, other parameters) go onto stack
Symbol Tables

Operations on Symbol Tables:

• create table
• delete table
• enterId(tab,string) returns: entryId, exists
• find(tab,string) returns: entryId, exists
• deleteEntry(tab,entryId)
• addAttributes(entryId,attributes)
• getAttributes(entryId) returns: attributes
Implementation Aspects of Symbol Tables

- Dynamic size is important. Space requirements can be from a few to tens of thousands of entries.
  
  Both should be provided:
  - dynamic growth for large programs (one common scheme is to double size each growth up to some maximum, then add a constant amount)
  - speed for small programs
Implementation Schemes

• Linear list
  - can be ordered or unordered
  - works for toy programs only (n items, k lookups, O(n/k))

• Binary search trees (Knuth vol. 3 for details.)
  - usually good solution. However, trees can be unbalanced, especially if alphabetical keys are used
  - N items, k lookups, O(k log N) time

• Hash tables (See Knuth Vol. 3 for gory details)
  - best variant. More complex. Good schemes exist
  - dynamic extension unclear (Java allows resize – expensive – basically requires copy of table.)
  - issues: clustering, deletion, chaining.
  - N items, k lookups, O(k) time
  - Languages such as Java and C++ provide these!
Dealing with Long Identifiers

• can be a waste of space if each table entry is big enough to hold largest identifier

• one solution is to store strings in a separate string array

```
i1.exp.the_weather_forecast_of_tomorrow.i.the_weather_forecast_of_today. ...... 
```

<table>
<thead>
<tr>
<th>name length = 2</th>
<th>name length = 3</th>
<th>name length =32</th>
<th>name length = 1</th>
<th>name length =29</th>
</tr>
</thead>
<tbody>
<tr>
<td>other attributes</td>
<td>other attributes</td>
<td>other attributes</td>
<td>other attributes</td>
<td>other attributes</td>
</tr>
</tbody>
</table>

...
Symbol Table Issues

• Symbol tables can be one per program block
  – size can be smaller
  – issue of dynamic size still remains
  – deletion in hash tables is less of a problem
  – For multipass, need to keep after block visit

• Overloading (same name used for different identifiers)
  – keep symbols together. Context will choose between them
  – name “mangling” in C++ -- makes context part of the name. Signature, class & classloader serves same purpose in Java
The symbol table project step

• Implement the semantic actions associated with variable declaration. The symbol table entry object has an identifier name field and a type field.

• Every block that allows declarations will have a symbol table associated with it. When you enter such a block, create a symbol table.
  – When an integer or float variable declaration is encountered, create an entry whose type field is integer (or “float”) and its return type to N/A. *Functions declarations do not need to be handled at this time.* The string corresponding to the identifier name can either be part of the identifier entry in the symbol table, or can be part of an external string table that is pointed to by the symbol table entry.
  – When you exit the block, call a routine to emit a listing of the symbol table entries, with each line containing the variable name and its type.
Symbol Table Attributes

• Examples:
  – Identifier and TypeDescriptor in Pascal (textbook p. 321/322)
  – In C, need length (for records)
  – In Fortran, need to know if it’s “save” or can be recreated on each invocation
  – Other info: alias sets, dimensionality (for array), static, final, public, private, type, number arguments (for func), …
Processing Declarations
(overview)

- Attributes and implementation techniques of symbol tables and type descriptors
- Action routines for simple declarations
  - semantic routines for processing declarations and creating symbol table entries
- Action Routines for advanced features -- green features subsumed by classes in OO languages.
  - constant declarations
  - enumeration types
  - subtypes
  - array types
  - variant records
  - pointers
  - packages and modules
Processing Declarations
(overview)

• constant declarations:
  – needs a field that can take the constant value.
  – in general the constant value may not be known until run time (static final int i = f+1)
  – At compile time we allow the initialization expression to assign into the variable, and then evaluate the expression at run time

• enumeration types
  – color in (red, blue, green)
  – There is an entry for the enumeration type, and an entry for each enumeration member. Entries are typically linked.
  – Processing of the enum declaration production sets the enum counter to lower bound (typically 0)
  – Each enum assigned next value, or possibly an assigned value (C)
Processing Declarations
(overview)

• Subtypes
  – E.g. ranges in Pascal (e.g. 1..2)
  – Need an entry for the type and the bounds of the range.

• array types
  – Dynamic array sizes not known until runtime – store *dope vector* on the activation record for the block/procedure (symbol table entry has info on dope vector size)
    • Consider `int a[M][N];`
    • Allows known offsets for stack entries
  – Array can be allocated at the top of the stack at runtime, and dope vector filled in
  – Static (i.e. fixed size) arrays are created on the activation record for the block/procedure they are alive-created in.
Processing Declarations
(overview)

• variant records (a type in Pascal, emulated with unions of structs in C)
  – Symbol table needs to hold list of variants identified with a name
  – Need info about the tag field that decides which variant is being used
  – Need information about the make-up of each variant
  – Each type of record must record offsets for each member

• Pointers
  – Type information and length of what it points to (for pointer arithmetic purposes) is necessary.
  – Need to be able to handle forward references – i.e. the thing being pointed to may not be declared yet.
Processing Declarations
(overview)

• packages and modules
  – Need entry for the package or module itself
  – Need information about private and global members of the package – pointers to their symbol table entry
  – Need information about which members are private and global -- may be kept in the member entry or package/module entry

• Shadow variable entries (not in the book)
  – Used to express non-obvious relationships between variables
  – One example if Fortran Common in which different arrays can live in the same storage. Shadow variable gives a way of saying that these two entities are aliased. Useful in more subtle areas.
Processing Expression and Data Structure References

• Simple identifiers and literal constants
• Expressions
• Record and array references
  \[ A[i,j] \rightarrow A + i \times \text{dim}_1 \times \text{element_size} + j \times \text{element_size} \] (if row major)
  \[ \text{dim}_1 \text{ and length of element are in the symbol table and/or dope vector} \]
  \[ R.f \rightarrow R + \text{offset}(f) \]
  \[ \text{Offset}(f) \text{ is in the symbol table} \]
• Strings – generated code depends on what we need and representation of strings, i.e. \( \text{length}(S1) + \text{length}(S2) \) may be an add of two values or a constant (PLI, Java) or require function calls (C).
• Advanced features (multi-dimensional arrays, variant records, etc.)
Simple identifiers and literals

- Need to do different things if we have a name used in a declaration and in an expression.
- `<decl>` ➔ ID: int; #decl_id
- `<primary>` ➔ ID; #id
Handling declarations

<decl> ➔ ID: int; #decl_id

decl_id

• Is passed the type token and string for the id
• check if id is already in the symbol table
• Add if not present, and an error if it is.
• Note: symbol table should be accessed through symbol table routines, not directly through action routines.
Use of identifiers

<primary> → ID; #id

#id
• Locate ID in symbol table, error if not there
• Extract the symbol table info and create a semantic record (e.g. dataobject) for the <primary>.
• Since primaries are used as operands, their semantic records must be typed.
• Having a semantic record that points to a symbol table routine for either a declared id or an identifier for a temporary variable will provide the needed information
Use of a literal

<primary> -> INTLITERAL #intliteral
#intliteral

• Put the string representation of the literal into string pool
• Create a semantic record for the primary. All primaries should create the same semantic record.
• Literal initially is a string of ASCII characters representing a number
• At some point it will need to be a two’s complement representation of the number.
• For integers, conversion time should be done early (allows constant folding). For floating point may wait until runtime because of rounding issues