Semantic actions for declarations and expressions
Semantic actions

- **Semantic actions** are routines called as productions (or parts of productions) are recognized.

- Actions work together to build up intermediate representations.

  
  ```
  <if-stmt> → IF <expr> #startif THEN <stmts> END #endif
  ```

- Semantic action for `#startif` needs to pass a **semantic record** to `#endif`.

- For LL parsers, semantic actions work easily, because they are predictive.

- For LR parsers, do not know which production is used until reduce step; need to place semantic actions at end of production.

  
  ```
  <if-stmt> → <begin-if> THEN <stmts> END #endif
  <begin-if> → IF <expr> #startif
  ```
Semantic Records

- Data structures produced by semantic actions
- Associated with both non-terminals (code structures) and terminals (tokens/symbols)
- Do not have to exist (e.g., no action associated with “;”)
- Control statements often require multiple actions (see <if-stmt> example on previous slide)
- Typically: semantic records are produced by actions associated with terminals, and are passed to actions associated with non-terminals
- Standard organization: *semantic stack*
Example of semantic stack

- Consider following grammar:

  assign  →  ID := expr
  expr    →  term addop term
  term    →  ID | LIT
  addop   →  + | −

- And now annotated with semantic actions:

  assign  →  ID #process_id := expr #gen_assign
  expr    →  term addop term #gen_infix
  term    →  ID #process_id | LIT #process_id
  addop   →  + #process_p | − #process_m
Example of semantic stack

• Consider \( a := b + 1; \)

• Sequence of semantic actions invoked:

  process_id, process_id, process_op, process_lit, gen_infix, gen_assign
How do we manipulate stack?

- Action-controlled: actions directly manipulate stack (call push and pop)
- Parser-controlled: parser automatically manipulates stack
LR-parser controlled

• Shift operations push semantic records onto stack (describing the token)

• Reduce operations pop semantic records associated with symbols off stack, replace with semantic record associated with production

• Action routines do not see stack. Can refer to popped off records using handles
  • e.g., in yacc/bison, use $1, $2 etc. to refer to popped off records
LL-controlled

- Parse stack contains predicted productions, not matched productions
- Push empty semantic records onto stack when production is predicted
- Fill in records as symbols are matched
- When non-terminal is matched, pop off records associated with RHS, use to fill in the record associated with LHS (leave LHS record on stack)
Overview of declarations

• Symbol tables
• Action routines for simple declarations
• Action routines for advanced features
  • Constants
  • Enumerations
• Arrays
• Structs
• Pointers
Symbol Tables

- Table of declarations, associated with each scope
- One entry for each variable declared
  - Store declaration *attributes* (e.g., name and type) – will discuss this in a few slides
- Table must be dynamic (why?)
- Possible implementations
  - Linear list (easy to implement, only good for small programs)
  - Binary search trees (better for large programs, but can still be slow)
  - Hash tables (best solution)
- BSTs and Hash tables can be difficult to implement, but languages like C++ and Java provide implementations for you
Managing symbol tables

• Maintain list of all symbol tables
• Maintain stack marking “current” symbol table
• Whenever you see a program block that allows declarations, create a new symbol table
  • Push onto stack as “current” symbol table
• When you see declaration, add to current symbol table
• When you exist a program block, pop current symbol table off stack
Handling declarations

- Declarations of variables, arrays, functions, etc.
- Create entry in symbol table
- Allocate space in *activation record*
  - Activation record stores information for a particular function call (arguments, return value, local variables, etc.)
  - Need to have space for all of this information
- Activation record stored on program stack
- We will discuss these in more detail when we get to functions
Simple declarations

• Declarations of simple types

    INT x;

    FLOAT f;

• Semantic action should

  • Get the type and name of identifier
  
  • Check to see if identifier is already in the symbol table

      • If it isn’t, add it, if it is, error
Simple declarations (cont.)

- How do we get the type and name of an identifier?
  
  \[ \text{var\_decl} \rightarrow \text{var\_type\ id;} \]
  
  \[ \text{var\_type} \rightarrow \text{INT \mid FLOAT} \]
  
  \[ \text{id} \rightarrow \text{IDENTIFIER} \]

- Where do we put the semantic actions?
Simple declarations (cont.)

• How do we get the type and name of an identifier?
  var_decl → var_type id; #decl_id
  var_type → INT #int_type | FLOAT #float_type
  id → IDENTIFIER #id

• Where do we put the semantic actions?
  • When we process #int_type and #id, can store the type and identifier name and pass them to #decl_id
  • When creating activation record, allocate space based on type (why?)
Constants and ranges

- Constants
  - Symbol table needs a field to store constant value
  - In general, the constant value may not be known until runtime (`static final int i = 2 + j;`)
  - At compile time, we create code that allows the initialization expression to assign to the variable, then evaluate the expression at run-time

- Range types (like in Pascal)
  - `Type alpha = ‘a’ .. ‘z’`
  - Need an entry for the type as well as the upper and lower bounds
Enums

• Enumeration types:

```plaintext
enum days {mon, tue, wed, thu, fri, sat, sun};
```

• Create an entry for the enumeration type itself, and an entry for each member of the enumeration

• Entries are usually linked

• Processing enum declaration sets the “enum counter” to lower bound (usually 0)

• Each new member seen is assigned the next value and the counter is incremented

• In some languages (e.g., C), enum members may be assigned particular values. Should ensure that enum value isn’t reused
Arrays

• Fixed size (static) arrays
  ```c
  int A[10];
  ```
  • Store type and length of array
  • When creating activation record, allocate enough space on stack for array

• What about variable size arrays?
  ```c
  int A[M][N]
  ```
  • Store information for a *dope vector*
    • Tracks dimensionality of array, size, location
    • Activation record stores dope vector
    • At runtime, allocate array at top of stack, fill in dope vector
Structs/classes

- Can have variables/methods declared inside, need extra symbol table
- Need to store visibility of members
- Complication: can create multiple instances of a struct or class!
- Need to store offset of each member in struct
Pointers

• Need to store type information and length of what it points to

  • Needed for pointer arithmetic

    ```
    int * a = &y;
    z = *(a + 1);
    ```

• Need to worry about forward declarations

  • The thing being pointed to may not have been declared yet

    ```
    Class Foo;
    Foo * head;
    Class Foo { ... };
    ```
Abstract syntax trees

- Tree representing structure of the program
- Built by semantic actions
- Some compilers skip this
- AST nodes
  - Represent program construct
  - Store important information about construct

Diagram:
- binary_op operator: +
- identifier "x"
- literal "10"
ASTs for References
Referencing identifiers

- Different behavior if identifier is used in a declaration vs. expression
  - If used in declaration, treat as before
  - If in expression, need to:
    - Check if it is symbol table
    - Create new AST node with pointer to symbol table entry
    - Note: may want to directly store type information in AST (or could look up in symbol table each time)
Referencing Literals

- What about if we see a literal?
  
  primary → INTLITERAL | FLOATLITERAL

- Create AST node for literal

- Store string representation of literal
  
  • “155”, “2.45” etc.

- At some point, this will be converted into actual representation of literal
  
  • For integers, may want to convert early (to do constant folding)
  
  • For floats, may want to wait (for compilation to different machines). Why?
More complex references

• Arrays
  • $A[i][j]$ is equivalent to $A + i*\text{dim}_1 + j$
  • Extract $\text{dim}_1$ from symbol table or dope vector

• Structs
  • $A.\text{f}$ is equivalent to $&A + \text{offset}(\text{f})$
  • Find $\text{offset}(\text{f})$ in symbol table for declaration of record

• Strings
  • Complicated—depends on language
Expressions

• Three semantic actions needed
  • eval_binary (processes binary expressions)
    • Create AST node with two children, point to AST
      nodes created for left and right sides
  • eval_unary (processes unary expressions)
    • Create AST node with one child
  • process_op (determines type of operation)
    • Store operator in AST node
Expressions example

• $x + y + 5$
Expressions example

- $x + y + 5$
Expressions example

- \( x + y + 5 \)
Expressions example

- $x + y + 5$
Expressions example

- $x + y + 5$
Expressions example

- \( x + y + 5 \)
Generating three-address code

• For project, will need to generate three-address code
  • `op A, B, C // C = A op B`
  • Can do this directly or after building AST
Generating code from an AST

• Do a post-order walk of AST to generate code, pass generated code up

```cpp
data_object generate_code() {
    data_object lcode = left.generate_code();
    data_object rcode = right.generate_code();
    return generate_self(lcode, rcode);
}
```

• Important things to note:
  • A node generates code for its children before generating code for itself
  • Data object can contain code or other information
  • Code generation is context free
    • What does this mean?
Generating code directly

• Generating code directly using semantic routines is very similar to generating code from the AST
• Why?
• Because post-order traversal is essentially what happens when you evaluate semantic actions as you pop them off stack
  • LL parser: evaluate left child before right child
  • LR parser: evaluate right child before left child
• AST nodes are just semantic records
L-values vs. R-values

- L-values: addresses which can be stored to or loaded from
- R-values: data (often loaded from addresses)
  - Expressions operate on R-values
- Assignment statements:
  L-value := R-value
- Consider the statement a := a
  - the a on LHS refers to the memory location referred to by a and we store to that location
  - the a on RHS refers to data stored in memory location referred to by a so we will load from that location to produce the R-value
Temporaries

• Can be thought of as an unlimited pool of registers (with memory to be allocated at a later time)

• Need to declare them like variables

• Name should be something that cannot appear in the program (e.g., use illegal character as prefix)

• Memory must be allocated if address of temporary can be taken (e.g. `a := &b`)

• Temporaries can hold either L-values or R-values
Data objects

- Records various important info
- The temporary storing the result of the current expression
- Flags describing value in temporary
  - Constant, L-value, R-value
- Code for expression
Simple cases

- Generating code for constants/literals
  - Store constant in temporary
  - Optional: pass up flag specifying this is a constant
- Generating code for identifiers
  - Generated code depends on whether identifier is used as L-value or R-value
    - Do we load from it? Or store to it?
    - One solution (may be inefficient): store address in temporary, let next level decide what to do with it
    - Set flag specifying this is an L-value
Generating code for expressions

- Create a new temporary for result of expression
- Examine data-objects from subtrees
- If temporaries are L-values, load data from them into new temporaries
  - Generate code to perform operation
- If temporaries are constant, can perform operation immediately
  - No need to perform code generation!
- Store result in new temporary
  - Is this an L-value or an R-value?
- Return code for entire expression
Generating code for assignment

- Store value of temporary from RHS into address specified by temporary from LHS

- Why does this work?

- Because temporary for LHS holds an address
  - If LHS is an identifier, we just stored the address of it in temporary
  - If LHS is complex expression

```c
int *p = &x
*(p + 1) = 7;
```

It *still* holds an address, even though the address was computed by an expression