Compiler Theory

(Semantic Analysis and Run-Time Environments)

005

Semantic Actions

- A compiler must do more than recognise whether a sentence belongs to the language of a grammar – it must do something useful with that sentence !!

- The semantic actions of a parser can do useful things with the phrases that are parsed.

- In a recursive-descent parser, semantic action code is interspersed with the control flow of the parsing actions. In JavaCC, semantic actions are fragments of Java program code attached to the grammar productions.

Semantic Actions (ii)

- For a rule A -> B C D, the semantic action must return a value whose type is the one associated with the non-terminal A.
- It will build it's value from the values associated with the matched terminals and non-terminals B,C and D.
- It is possible to construct an entire compiler that fits within the semantic actions phrases of JavaCC, however such a compiler would be difficult to read and maintain. Hence abstract syntax trees ...

Abstract Syntax Tree (i)

- To improve modularity it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code)
- One way to do this is for the compiler to produce a parse tree then an abstract syntax tree– a data structure that later phases of the compiler will traverse.
- Parse tree has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.

Abstract Syntax Trees (ii)

- Factoring, elimination of left recursion and ambiguity should be confined to the parsing phase.

- The abstract syntax tree conveys the phrase structure of the source program, with all parsing issues resolved but without any semantic interpretation.

- Punctuation tokens may be removed since they convey no information in an abstract syntax tree.

Abstract Syntax of expressions

- E -> E + E
- E -> E – E
- E -> E * E
- E -> E / E
- E -> id
- E -> num

- Note that this grammar is completely impractical for parsing. The grammar is ambiguous since the precedence of the operators is not specified.

- The semantic analysis phase takes this abstract syntax tree; it is not bothered by the ambiguity of the grammar, since it already has the parse tree.

Data structures for Abstract Syntax Trees (let us look at some code ... !!)

- Compiler needs to represent and manipulate abstract syntax trees as data structures.
- Typically a (Java) compiler would have an abstract class for each non-terminal and a sub-class for each production ...
- The next slide gives an implementation of the abstract class Exp together with some of it's productions.
- On the slide after that one, there's the JavaCC specification file which generates the abstract syntax tree !!

Code for Exp class

```
public abstract class Exp {
   public abstract int evala();
}

public class PlusExp extends Exp {
   private Exp e1, e2;
   public PlusExp(Exp a1, Exp a2) { e1 = a1; e2 = a2; }
   public int eval() {
     return e1.eval()-e2.eval();
   }
}

public class Identifier extends Exp {
   private String f0;
   public Identifier(String n0) { f0 = n0; }
   public int eval() {
      return lookup(f0);
   }
}

public class IntegerLiteral extends Exp {
   private String f0;
   public IntegerLiteral(String n0) {f0 = n0; }
   public int eval() {
      return Integer.parserInt(f0);|
   }
}
```

JavaCC code to construct Abstract Syntax Tree

```
Exp Start() :
  { Exp e; }
  { e = Exp() { return e; }
  }

Exp Exp() :
  { Exp e1, e2; }
  { e1=Term()
      ( "+" e2=Term() {e1=new PlusExp(e1,e2); }
      | "-" e2=Term() {e1=new MinusExp(e1,e2); }
      )*
    { return e1; }
  }

Exp Term() :
  { Exp e1, e1; }
  { e1 = Factor()
       ( "*" e2=Factor() { e1 = new TimesExp(e1,e2); }
       | "/" e2=Factor() { e1 = new DivideExp(e1,e2); }
       )*
    { return e1; }

Exp Factor() :
  { Token t; Exp e; }
  { ( t=<IDENTIFIER>        {return new Identifier(t.image); } |
      t=<INTEGER_LITERAL> {return new IntegerLiteral(t.image); } |
      "(" e=Exp() ")"     {return e; } )
  }
```

Semantic Analysis (i)

- The semantic analysis phase of a compiler
  - connects variable definitions to their uses,
  - checks that each expression has a correct type, and
  - translates the abstract syntax into a simpler representation suitable for generating machine code.
- This phase is characterised by the maintenance of the symbol tables !!

Semantic Analysis (ii)

- Each local variable in a program has a *scope* in which it is visible.

- In a typical programming language, in a method *m*, all formal parameters and local variables declared in *m* are visible only until the end of *m*.

- As the semantic analysis reaches the end of each scope, the identifier bindings local to that scope are discarded.

Semantic Analysis - Environments(iii)

- An environment is a set of bindings denoted by the ├ symbol (should be an arrow |->)
- For example, we could say that the environment σ0 (sigma 0) contains the bindings { g ├ string, a├ int }, meaning that the identifier a is an integer and g is a string variable.
- Consider the small Java program in the next slide. The environment of the program changes from one set to another … σ1 = σ0 + { a├ int, b├ int, c├ int } then σ2 = σ1 + { j├ int } and then σ3 = σ2 + { a├ String }

Java Sample

```
class C {
  int a; int b; int c;
  public void m() {
    System.out.println(a+c);
    int j = a+b;
    String a = "hello";
    System.out.println(a);
    System.out.println(j);
    System.out.println(b);
  }
}
```

Precedence in scoping tables

- In the previous example we wanted {a├String} to take precedence.
- One very simple strategy is to say that bindings in the right of the table override those on the left.
- Note that at the end of the previous method we need to discard $\sigma 3$ and go back to $\sigma 1$.
- And at the end of the program we go back to $\sigma 0$.

How do we implement the symbol table? (i)

- In the *imperative* style we modify $\sigma 1$ until it becomes $\sigma 2$. In a way it is a destructive (destroys $\sigma 1$) update ...
- We need a way of undoing changes so that from $\sigma 2$ we can go back to $\sigma 1$
- A single global variable s becomes at different times $\sigma 0$, $\sigma 1$, $\sigma 2$, $\sigma 3$, $\sigma 1$, $\sigma 0$.
- We use an "undo stack" with enough information to remove the destructive updates.

How do we implement the symbol table? (ii)

- Imperative-style environments are usually implemented using hash tables (because they are very efficient)
- The idea is to have a hashtable (possibly per symbol table) in which the keys are the variable names and the values point to an ordered list (stack like) with the different scope bindings.
- Insert : $\sigma' = \sigma + \{a \vdash \tau\}$ is implemented by inserting τ in the hash table with key a.
- At the end of a's scope we need to restore σ, with a call to pop(a).
- Note that this is a very simple implementation !!

Multiple Symbol Tables !!

- Check out this Java code …
- There can be several active environments at once.
- $\sigma 1 = \{ a \vdash int \}$
- $\sigma 2 = \{ E \vdash \sigma 1 \}$
- $\sigma 3 = \{ b \vdash int, a \vdash int \}$
- $\sigma 4 = \{ N \vdash \sigma 3 \}$
- $\sigma 5 = \{ d \vdash int \}$
- $\sigma 6 = \{ D \vdash \sigma 5 \}$
- $\sigma 7 = \sigma 2 + \sigma 4 + \sigma 6$

```
package M;
class E {
    static int a = 5;
}

class N {
    static int b = 10;
    static int a = E.a + b;
}

class D {
    static int d = E.a + N.a;
}
```

Symbol Table Content (i)

- With what should a symbol table be filled – that is, what is a binding?
- It should contain all declared type information
  - Each variable name and formal-parameter name should be bound to its type;
  - Each method name should be bound to its parameters, result type, and local variables; and
  - Each class should be bound to its variable and method declarations.

Symbol Table Contents (ii)

- B and C are mapped to two tables for fields and methods
- Each method is then mapped to both its result type, tables with formal parameters and local variables

```
class B {
    C f; int[] j; int q;
    public int start(int p, int q) {
        int ret; int a;
         /*.....*/
        return ret;
    }
    public boolean stop(int p) {
        /*.....*/
        return false;
    }
}

class C {
    /*.....*/
}
```

Type Checking ...

- Two phase process
  - First finish off building the symbol table,
  - Then type-check statements and expressions

- It is best (for example in Java) to first build the symbol table because in the code we would normally have classes which are mutually recursive.
- So we want everything to be in the symbol table before we start type checking.

Type Checking (ii) ...

- Can take two forms
  - ***Type Synthesis*** builds up the type of an expression from the types of its sub-expressions. It requires names to be declared before they are used.
  - ***Type Inference*** determines the type of a language construct from the way it is used. e.g. In ML

    A typical rule for type inference has the form

    if f(x) is an expression,

    then for some α and β, f has type α -> β and x has type α

Type Synthesis and Conversions

- Suppose that in our language integers are converted to floats when necessary,
- We can use rules to type check and if necessary convert an int to a float
- For e.g. For an expression E = E1 + E2
  - If (E1.type = integer and E2.type = integer) E.type = integer
  - Else if (E1.type = float and E2.type = integer) ...
  - ....

Run-time Environment – Stack Allocation of Space

- Each time a procedure is called, space for its local variables is *pushed* onto a stack.
- When the procedure terminates, that space is *popped* off the stack.
- Note that this arrangement only works for procedure calls whose duration do not overlap in time.
- We shall refer to procedure calls as activations.

Recursive procedure calls - Activation Trees

- e.g. a quicksort implementation
- Procedure activations are nested in time, i.e. If an activation of procedure p calls procedure q, then that activation of q must end before the activation of p can end.
- If the activation of q terminates normally, then control resumes just after the point of p at which the call to q was made.
- We can represent the activation of procedures during the running of an entire program by a tree called an ***activation tree***.

Activation Tree

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 7.3: Possible activations for the program of Fig. 7.2

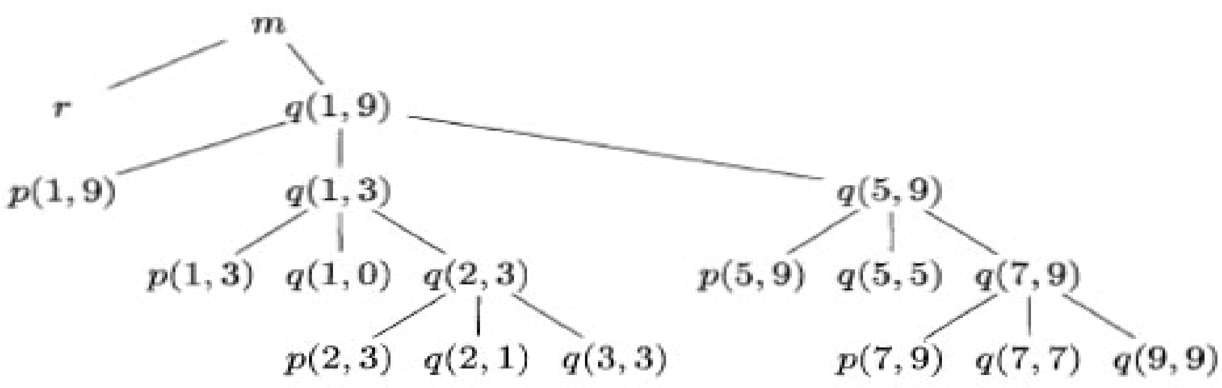

Figure 7.4: Activation tree representing calls during an execution of *quicksort*

Activation Records (i)

- We know that functions may have local variables that are created upon entry to a function.
- We also know that several invocations of the same function (method) may exist at the same time.
- Each invocation must have its own *instantiations* of local variables

Activation Records (ii)

- A new instantiation of x is created (and initialized by f's caller) each time that f is called.
- Because of recursion, many of these x's exist simultaneously
- Similarly, a new instantiation of y is created each time the body of f is entered.

```
int f(int x) {
  int y = x+x;
  if (y < 10)
     return f(y);
  else
     return y-1;
}
```

Activation Records (iii)

- Each live activation (function or procedure) has an activation record (sometimes called a frame) located in the stack which stores local variables, parameters, return addresses and other temporary data.

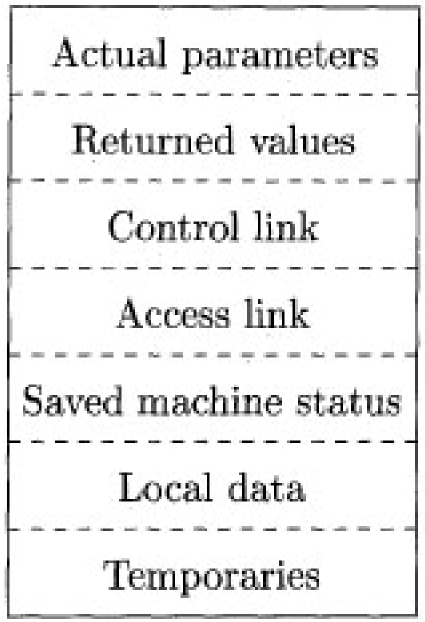

Figure 7.5: A general activation record

Activation Records – Contents (iv)

- *Saved machine status* stores info about
  - State of the machine just before the call to the procedure – return address (value of program counter, to which the called procedure must return)
  - Contents of registers prior to call so that they are restored when the procedure returns
- *Control Link*
  - Pointing to the activation record of the caller
- *Local Data* – belonging to the procedure whose activation record this is.

Run-time updates on stack of activation records for quicksort

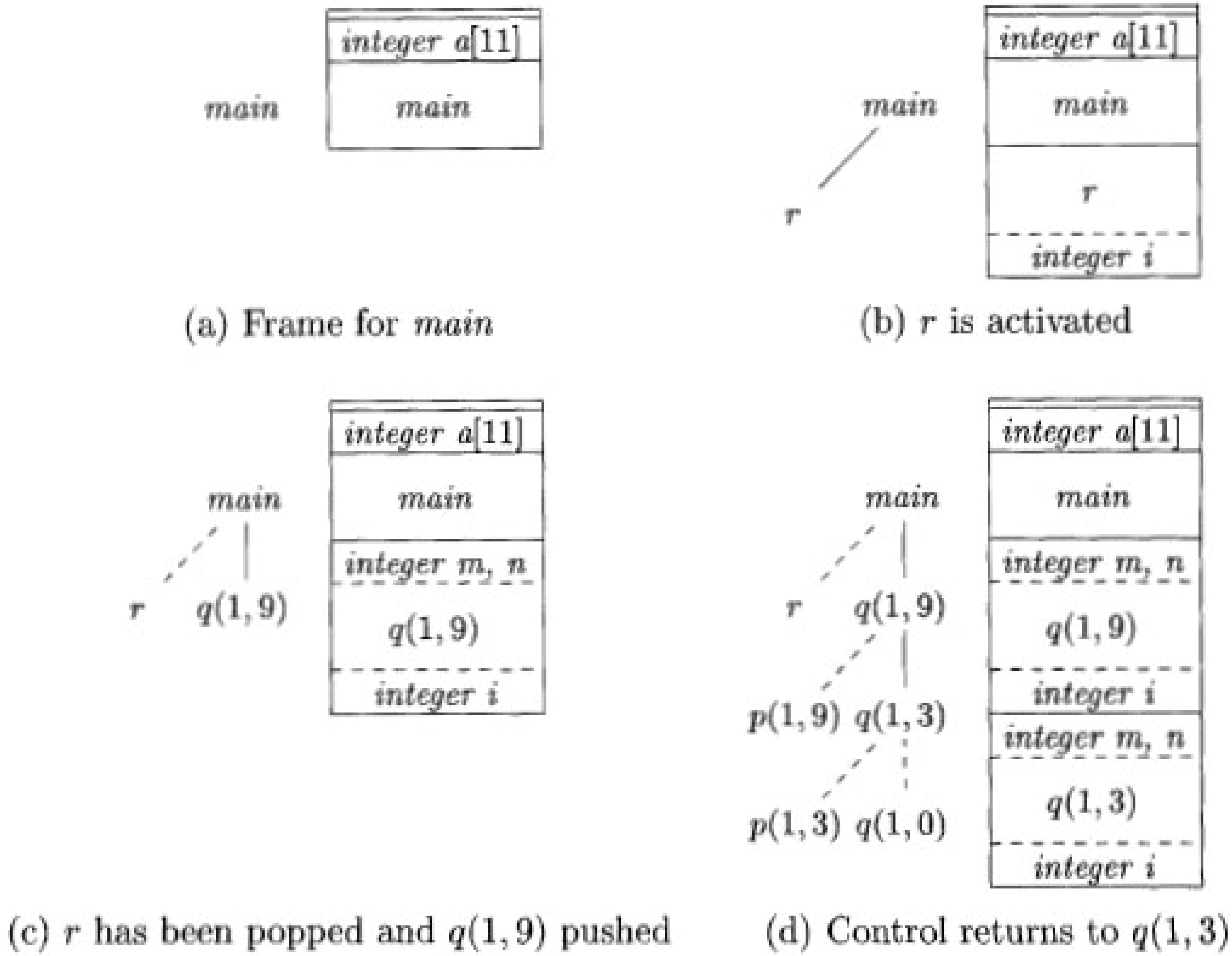

Figure 7.6: Downward-growing stack of activation records

Explanation of previous slide !

- Procedure r (readArray) is activated first ... it's activation record (AR) is pushed onto the stack,
- When control returns its AR is popped, leaving just the record for main on the stack,
- Control then goes to q (quicksort) with parameters 1 and 9. An AR for this call is placed on the top of the stack,
- Several activations occur between the last two snapshots
  - A recursive call to q(1,3) was made
  - p(1,3) and q(1,0) have begun and ended during the lifetime of q(1,3)
- The last snapshot shows control returning to q(1,3)