CA4003 - Compiler Construction

Semantic Analysis

David Sinclair

Semantic Actions

A compiler has to do more than just recognise if a sequence of characters forms a valid sentence in the language. It must do something useful with the parsed sentence.

The *semantic actions* of a parser perform useful operations.

- Build an abstract parse tree.
- Type checking.
- Evaluation in the case of an interpreter.
- Begin the translation process in the case of a compiler.

In some compiler constructors (such as JavaCC and Yacc) the semantic actions are attached to the production rules.

In other compiler constructors (JJTree and SableCC for example) the syntax tree are automatically generated.

Semantic Actions [2]

Each symbol, terminal or non-terminal, may have its own type of semantic value.

$$\begin{array}{lcl} exp & \rightarrow & \mathrm{INT} \\ exp & \rightarrow & exp\ \mathrm{PLUS}\ exp \\ exp & \rightarrow & exp\ \mathrm{MINUS}\ exp \\ exp & \rightarrow & exp\ \mathrm{MUL}\ exp \\ exp & \rightarrow & exp\ \mathrm{DIV}\ exp \\ exp & \rightarrow & \mathrm{MINUS}\ exp \end{array}$$

- In a simple calculator grammar above *exp* and INT may have the type of `int`.
- Other symbols may have a different semantic type (like `String` or not semantic type at all.
- A rule $W \rightarrow XYZ$ must return a semantic type associated with the symbol W, but may use the values associated with the symbols X, Y and Z.

Abstract Syntax Trees

A *parse tree* is a data structure that is used to separate parsing (lexical and syntactical analysis) from semantics (type checking and translation to machine code) .

- It is possible to write the compiler so that every thing is done by the semantic actions of the parsing phase, but
  - it constrains the compiler to analyse the input exactly in the order in which it is parsed; and
  - it is very difficult to maintain.

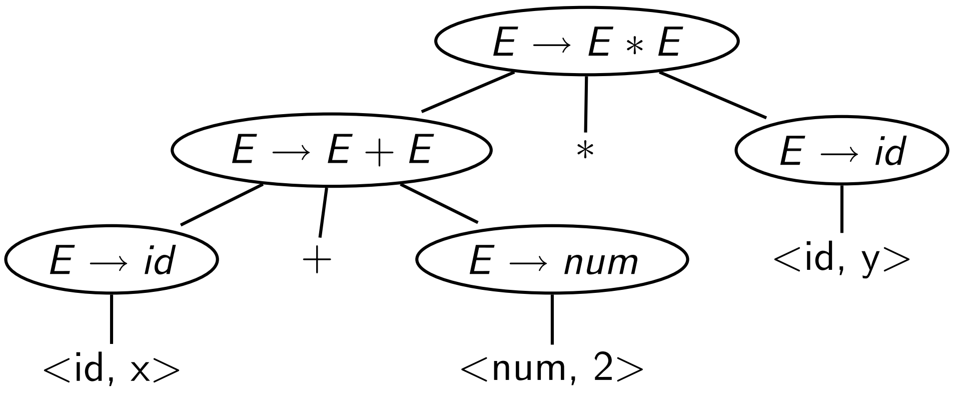
A *concrete parse tree* is a parse tree that has

- exactly one terminal node (leaf node) for each token in the input; and
- one internal node for each production rule reduced during the parse.

Abstract Syntax Trees [2]

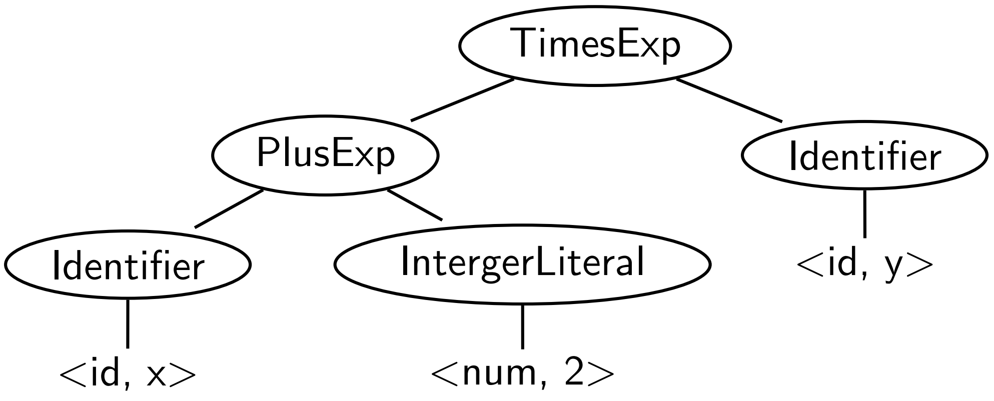
Concrete parse trees contain lots of information, such as punctuation and operator tokens, that while useful during parsing, either redundant or contain no useful information for semantic analysis.

An *abstract syntax tree* is a clean interface between parsing and semantic analysis (and other later phases of a compiler). The *abstract syntax tree* captures the phrase structure of the input.



Abstract Syntax Trees [3]

A corresponding *abstract syntax tree* is:



Abstract Syntax Trees [4]

The following example code shows how an *abstract syntax tree* for expressions could be built using JavaCC.

```
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, e2; }
{
  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; }
  )
}
```

Abstract Syntax Trees [5]

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

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 TimesExp extends Exp
{
  private Exp e1, e2;
  public TimesExp (Exp a1, Exp a2) { e1 = a1, e2 = a2; }
  public int eval ( ) { return (e1.eval ( ) * e2.eval ( )); }
}
...
public class Identifer extends Exp
{
  private String s;
  public Identifier (String n) { s = n; }
  public int eval ( ) { return lookup (s); }
}
...
public class IntegerLiteral extends Exp
{
  private String s;
  public IntegerLiteral (String n) { s = n; }
  public int eval ( ) { return Integer.parseInt (s); }
}
```

Positions

In a single pass complier where lexical analysis, parsing and semantic analysis are all done simultaneously, then the *current position* of the lexical analyser is a reasonable approximation of the source of the error.

In a multi-pass compiler, which uses an *abstract syntax tree*, the current position of the lexical analyser (i.e. the end of the file) is not useful if we find an error during semantic analysis of the *abstract syntax tree*.

Instead we need to extend the data structures used in the *abstract syntax tree* to include a `pos` field that indicates the position, in the original source file, of the characters from which the abstract syntax structures were derived.

Visitor Pattern

In the previous example, the use of the `eval` method was an example of the *object-oriented* style of programming.

However, there are advantages in separating the processing of the abstract syntax trees from creating/definition of the abstract syntax trees. This style of programming is called *syntax separate from interpretation*.

In programming languages we have different *kinds* of objects (i.e. compound statements, assignment statements, expressions, etc) and different *interpretations* of these objects (i.e. type check, translate to Pentium, translate to Sparc, optimise, etc).

Whether we choose the *object-oriented* style of programming or the *syntax separate from interpretation* style of programming will affect the modularity of the code.

Visitor Pattern [2]

If we choose the *object-oriented* style of programming each *interpretation* is just a method in the class that defined the *kind*. Adding a new *kind* is just adding a new class with a method of each *interpretation*. Adding a new *interpretation* means editing each class that represents a *kind*.

If we choose the *syntax separate from interpretation* style of programming then one method is written for each *interpretation* with clauses for each *kind*. Usually these *interpretation* are gathered together. Adding a new *interpretation* just means adding a new method. Adding a new *kind* means editing all the methods that represent *interpretations*.

Visitor Pattern [3]

Which we should use depends on which tends to change more frequently?

- In graphical user interfaces, the number and type of objects tend to change whereas the operations on then do not. Hence, the *object-oriented* style of programming is better.
- In compilers it makes more sense to fix the syntax (Java is Java) and then provide different interpretations. Hence, the *syntax separate from interpretation* style of programming is better.

How do we avoid using the `instanceof` operator and having each *interpretation* method looking like a morass of clauses?

We use the Visitor pattern.

Visitor Pattern [4]

A visitor implements an *interpretation*. It is an object that contains a method for each *kind*, i.e each class of syntax tree node.

- Each syntax tree class has an `accept` method that is called by the Visitor.
- The `accept` method acts as a hook for all interpretations.
- The `accept` method's only task is the call the visitor, parameterised by the `accept` method's own type. This ensures that the correct visitor method is invoked.

Visitor Pattern [5]

```
public interface Visitor
{
  public int visit (PlusExp ex);
  public int visit (MinusExp ex);
  public int visit (TimesExp ex);
  public int visit (DivideExp ex);
  public int visit (Identifier ex);
  public int visit (IntegerLiteral ex);
}

public class Interpreter implements Visitor
{
  public int visit (PlusExp ex)
  {
    return (ex.e1.accept(this) + ex.e2.accept(this));
  }

  public int visit (MinusExp ex)
  {
    return (ex.e1.accept(this) - ex.e2.accept(this));
  }

  public int visit (TimesExp ex)
  {
    return (ex.e1.accept(this) * ex.e2.accept(this));
  }

  public int visit (DivideExp ex)
  {
    return (ex.e1.accept(this) / ex.e2.accept(this));
  }
```

Visitor Pattern [6]

```
  public int visit (Identifier ex)
  {
    return (lookup (ex.s));
  }

  public int visit (IntegerLiteral ex)
  {
    return (Integer,parseInt (ex.s));
  }
}


public abstract class Exp
{
  public abstract class int accept (Visitor v);
}

public class PlusExp extends Exp
{
  private Exp e1, e2;
  public PlusExp (Exp a1, Exp a2) { e1 = a1, e2 = a2; }
  public int accept (Visitor v) { return (v.visit (this)); }
}
...
public class IntegerLiteral extends Exp
{
  private String s;
  public IntegerLiteral (String n) { s = n; }
  public int accept (Visitor v) { return (v.visit (this)); }
}
```

Symbol Tables

A *symbol table*, also called an *environment*, maps identifiers to their types and locations. As types, variables and functions are declared they are added to the *symbol table*. When an identifier is used, it is looked up in the *symbol table*.

Identifiers in modern languages typically have a *scope* in which they are visible.

Environments are a set of *bindings*. $a \mapsto x$ denotes a is bound to x. For example, $\sigma_0 = \{g \mapsto \texttt{string}, a \mapsto \texttt{int}\}$ denotes an *environment* σ_0 where g is a `string` variable and a is an integer variable.

Symbol Tables [2]

Consider the following code fragment:

`class C`	assume current environment is σ_0
`{`	
`  int a; int b, int c;`	$\sigma_1 = \sigma_0 + \{a \mapsto \texttt{int}, b \mapsto \texttt{int}, c \mapsto \texttt{int}\}$
`   public void m ( )`	
`  {`	
`    System.out.println (a+ c);`	a and c are looked up in σ_1
`    int j = a + b;`	$\sigma_2 = \sigma_1 + \{j \mapsto \texttt{int}\}$
`    String a = "hello";`	$\sigma_3 = \sigma_2 + \{a \mapsto \texttt{String}\}$
`    System.out.println (a);`	
`    System.out.println (j);`	
`    System.out.println (b);`	
`  }`	discard σ_3 and σ_2 and use σ_1
`}`	discard σ_1 and use σ_0

Symbol Tables [3]

This example raises a couple of issues:

- How does the $+$ operator for environments work when both environments contain different bindings for the same symbol?
  - We would like the later binding to take precedence. So, in our example, $a \mapsto \texttt{String}$ in σ_3.
- How do we efficiently implement symbol tables so that:
  - We find the correct binding.
  - We can efficiently discard environments.

A Hash Table with external chaining and an *undo* stack can do the required tasks efficiently.

Symbol Tables [4]

Hash Table with external chaining.

- A *hash function* converts the symbol into an index in a table/array.
    - *Hash function* should be capable of being evaluated quickly.
    - The resulting indices should be randomly distributed across the table so it is unlikely that two different symbols will result in the same hash value (or index).
- Each entry in the hash table is a linked list of bindings.
    - When a binding is added to the hash table, it is inserted at the front of the linked list associated with the symbol's hash value.
    - When looking up a symbol in the hash table the linked list associated with the symbol's hash value is searched from the front. That way the search will always find the most recent binding that involves that symbol.
    - To improve lookup time and make use of locality, you move a symbol that you have successfully searched for to the front of the list.

Symbol Tables [5]

Undo stack.

- Whenever a binding is added to the symbol table, the binding is also pushed onto the undo stack.
- When a new scope is created, for example by the { symbol in Java, a special marker is pushed onto the undo stack.
- When a scope is being destroyed, for example by the } symbol in Java, the entries on the undo stack are popped off and removed from the symbol table until the special marker is popped off the undo stack.

Symbol Tables [6]

Symbol table bindings associates names with attributes.

Names may have different attributes depending on their meaning:

variables:	type, procedure level, frame offset
types:	type descriptor, data size/alignment
constants:	type, value
procedures:	formals (names/types), result type, block information (local decls.), frame size

Type Checking

One of the major semantic analysis operations is type checking.

Type expressions are a textual representation for types:

1. basic types: *boolean*, *char*, *integer*, *real*, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   - i arrays: $array(I, T)$ denotes array of elements of type T, index type I e.g., $array(1..10, integer)$
   - ii products: $T_1 \times T_2$ denotes the Cartesian product of type expressions T_1 and T_2
   - iii records: fields have names e.g., $record((\mathtt{a} \times \mathtt{integer}), (\mathtt{b} \times \mathtt{real}))$
   - iv pointers: $pointer(T)$ denotes the type "pointer to an object of type T"
   - v functions: $D \rightarrow R$ denotes type of a function mapping domain type D to range type R e.g., $integer \times integer \rightarrow integer$

Type Compatibility

Type checking needs to determine type equivalence.

There are two approaches:

Name equivalence: each type name is a distinct type.

Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names).

- $s \equiv t$ iff. s and t are the same basic types.
- $array(s_1, s_2) \equiv array(t_1, t_2)$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$.
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$.
- $pointer(s) \equiv pointer(t)$ iff. $s \equiv t$.
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$.

Type Compatibility: Example

Consider:

```
type  link  =  ↑cell;
var   next  :  link;
      last  :  link;
      p     :  ↑cell;
      q, r  :  ↑cell;
```

Under name equivalence:

- `next` and `last` have the same type.
- `p`, `q` and `r` have the same type.
- `p` and `next` have different types.

Under structural equivalence all variables have the same type.

Ada/Pascal/Modula-2 are somewhat confusing as they treat distinct type definitions as distinct types. So `p` has different type from `q` and `r`.

Overloading

Most languages have type overloading:

- If nothing else, for integers and floats.
- Equality/assignment overloaded for almost anything.
- In languages with dynamic types (Lisp, Smalltalk), decision on what to do depends on type check at run-time.
- Can be very inefficient for integers/floats.
- Can be resolved at compile-time by *type inference*.
- Type inference is usually done bottom-up:
  - Say we have f can be either $int \to int$ or $float \to float$,
  - then $f(42)$ has only one valid typing: *int*

Polymorphic Functions

Polymorphism = many shapes

- Ad-hoc polymorphism: on a case-by-case basis (overloading).
- Parametric polymorphism: can take a type as an argument.
  - Templates
  - “True” parametric polymorphism:
    - function length(L) = if null(L) then 0 else 1+length tail(L)
      length: $List(\alpha) \to int$
    - function first(L) = head(L)
      first: $List(\alpha) \to \alpha$
    - function reverse(l) = ...
      reverse: $List(\alpha) \to List(\alpha)$
  - Often combined with type inference.