Implementing Type Checking

SLIDES BY MICHAEL WOLLOWSKI

SECOND AND LONG EXAMPLE FROM SLIDES BY MICHAEL D. BOND, COMPUTER SCIENCE & ENGINEERING, OHIO STATE UNIVERSITY

Overview

We will look at two means of implementing type checking:

- Ad-hoc syntax directed parsing and
- Attribute grammars

Ad-hoc syntax-directed translation

In *ad-hoc syntax-directed translation* the actions required for context sensitive analysis are incorporated into the process of parsing a context-free grammar.

This is in contrast to the attribute grammar approach where we modify the grammar.

In the end of the day, both approaches encode the same information.

Set-up

In this scheme, the compiler writer provides snippets of code that execute at parse time.

Each snippet, or *action,* is directly tied to a production in the grammar.

Each time the parser recognizes that it is at a particular place in the grammar, the corresponding action is invoked.

The compiler writer has complete control over when the actions execute.

To implement this in a recursive-descent parser, the compiler writer adds the appropriate code to the parsing routines.

Attribute Grammars

One formalism that has been proposed for performing context-sensitive analysis is the *attribute grammar,* or attributed context-free grammar.

It is a context-free grammar augmented with a set of rules.

We augment each symbol in the derivation (or parse tree) with a set of *attributes*

The rules specify how to compute a value for each attribute

The attributes are divided into two groups: synthesized and inherited.

- *Synthesized* attributes are the result of the attribute evaluation rules.
- *Inherited* attributes are passed down from parent nodes.

Attribute Grammar Example

Consider the following grammar.

It describes signed binary numbers.

```
Number    -> Sign List
Sign      -> +
           | -
List      -> List Bit
           | Bit
Bit       -> 0
           | 1
```

We would like to build an attribute grammar that annotates *Number*, the start symbol of this sample grammar with the value of the signed binary number that it represents.

Attributes

To build an attribute grammar from a context-free grammar, we must decide what attributes each node needs.

We will then elaborate the productions with rules that define values for these attributes.

We annotate our grammar with the following attributes:

Symbol	Attributes
Number	value
Sign	negative
List	position, value
Bit	position, value

Resulting attribute grammar

Production	Attribution Rules
`Number -> Sign List`	List.position <- 0 If Sign.negative then Number.value <- -List.value else Number.value <- List.value
`Sign -> +`	Sign.negative <- false
`Sign -> -`	Sign.negative <- true
`List -> Bit`	Bit.position <- List.position List.value <- Bit.value
$List_0$ -> $List_1$ Bit	$List_1$.position <- $List_0$.position + 1 Bit.position <- $List_0$.position $List_0$.value <- $List_1$.value + Bit.value
`Bit -> 0`	Bit.value <- 0
`Bit -> 1`	Bit.value <- $2^{Bit.position}$

Attribute Grammar Example

Subscripts are added to grammar symbols whenever a specific symbol appears multiple times in a single production.

This practice disambiguates reference to that symbol in the rules.

Rules add attributes to the parse tree nodes by their names.

An attribute mentioned in a rule must be instantiated for every occurrence of that kind of node.

A rule can pass information from the production's lhs side to its rhs.

A rule can also pass information in the other direction.

For example, the rules for production 4 pass information in both directions:

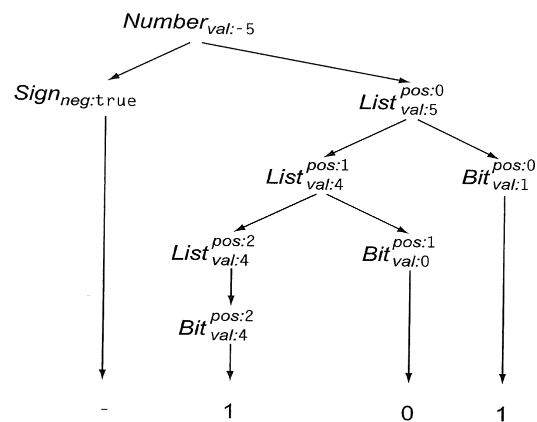
- The first rule sets *Bit.*position to *List.* Position.
- The second rule sets *List.value* to *Bit.value.*

Attribute Grammar Practice

Class exercise: Parse -101 to get a sense of the grammar in action.

Solution to Prior Exercise

The parse tree for -101:



Attribute Grammar Example

Each rule implicitly defines a set of dependences.

The attribute being defined depends on each argument to the rule.

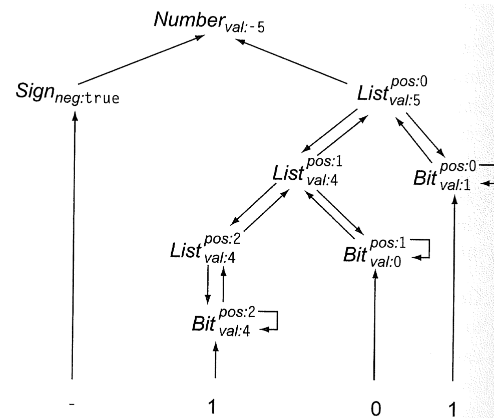
Taken over the entire parse tree, these dependences form an *attribute dependence graph.*

Edges in the graph follow the flow of values in the evaluation of a rule.

The edge from $node_i.field_j$ to $node_k.field_l$ indicates that the rule defining $node_k.field_l$ uses the value of $node_i.field_j$ as one of its inputs.

Attribute Grammar Example

The dependence graph for -101:



Attribute Grammar: Properties

An attribute grammar is *circular* if it can, for some inputs, create a cyclic dependence graph.

The dependence graph captures the flow of values that an evaluator must respect in evaluating an instance of an attributed tree.

If the grammar is noncircular, it imposes a partial order on the attributes.

This partial order determines when the rule defining each attribute can be evaluated.

Evaluation order is unrelated to the order in which the rules appear in the attribute grammar.

Type Checking Example

Consider the following program fragment.

We would like to check whether the variables are used correctly.

In particular, we will check the types of variables.

For nested blocks, we will use the innermost declaration.

```
begin
  bool i;
  int j;
  begin
    int x;
    int i;
    x := i + j
  end
end
```

Type Checking Example

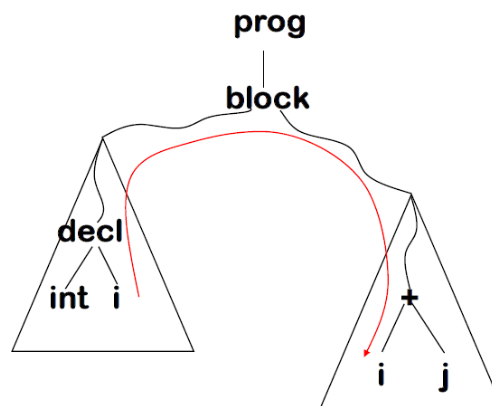
Here is the relevant part of the grammar:

```
<prog>     ::= <block>
<block>    ::= begin <declist>; <stmtlist> end
<declist>  ::= <decl>
             | <decl> ; <declist>
<decl>     ::= int <id>
             | bool <id>
<stmtlist> ::= <stmt>
             | <stmt> ; <stmtlist>
<stmt>     ::= <assign>
             | <block>
…
```

```
begin
  bool i;
  int j;
  begin
    int x;
    int i;
    x := i + j
  end
end
```

Type Checking Example

Information flow through the parse tree for this language:



Type Checking Example

We will manage the information flow with a stack of symbol tables

A symbol table is set of pairs (name, type)

Build a symbol table for the declarations in a begin-end block
- as a synthesized attribute `tbl`

Propagate a stack of symbol tables to statements
- propagate downwards on the parse tree as an inherited attribute `alltbl`

Type Checking Example

Visual representation of stack of symbol-tables:

```
begin
  bool i;
  int j;
  begin
    int x;
    int i;
    x := i + j;
  end
end
```


Type Checking Example

Augmenting to an attribute grammar:

```
<prog>      ::= <block>
                <block>.alltbl := emptystack

<block>     ::= begin <declist>; <stmtlist> end
                <stmtlist>.alltbl :=
                push(<declist>.tbl, <block>.alltbl)
```

Type Checking Example

Augmenting to an attribute grammar:

```
<declist>_1 ::= <decl>
                <declist>_1.tbl := <decl>.tbl
            | <decl> ; <declist>_2
              <declist>_1.tbl :=
              <decl>.tbl ∪ <declist>_2.tbl
              Cond: ids(<decl>.tbl)∩ids(<declist>_2.tbl)={}
```

"ids" is a function that returns the set of all keys in the table.

Type Checking Example

Augmenting to an attribute grammar:

```
<decl> ::= int <id>
           <decl>.tbl := { (id.lexval, INT) }
         | bool <id>
           <decl>.tbl := { (id.lexval, BOOL) }
```

Type Checking Example

Augmenting to an attribute grammar:

```
<stmtlist>1 ::= <stmt>
                 <stmt>.alltbl := <stmtlist>1.alltbl
              | <stmt> ; <stmtlist>2
                <stmt>.alltbl := <stmtlist>1.alltbl
                <stmtlist>2.alltbl :=
                   <stmtlist>1.alltbl
```

Type Checking Example

Augmenting to an attribute grammar:

```
<stmt>    ::= <assign>
               <assign>.alltbl := <stmt>.alltbl
             | <block>
               <block>.alltbl := <stmt>.alltbl
```

Type Checking Example

Class exercise of parsing the code sample. (Worksheet)

Type Checking Example (going further)

We now augment our grammar to handle assignment statements and expressions.

```
<assign> ::= <id> := <intexp>
           | <id> := <boolexp>
<boolexp> ::= true | false | <id>
<intexp> ::= <number>
            | <id>
            | <intexp> + <intexp>
```

Type Checking Example (going further)

Augmenting to an attribute grammar:

```
<assign> ::= <id> := <intexp>
             <intexp>.alltbl := <assign>.alltbl
             Cond: typeof(id.lexval,<assign>.alltbl) = INT
           | <id> := <boolexp>
             <boolexp>.alltbl := <assign>.alltbl
             Cond: typeof(id.lexval,<assign>.alltbl) = BOOL
```

To satisfy the condition, we look for the most recent occurrence of “id” on the symbol-table.

This implements scope properly and enables shadowing.

Type Checking Example (going further)

Augmenting to an attribute grammar:

```
<boolexp> ::= true | false | <id>
              Cond: typeof(id.lexval,<boolexp>.alltbl) = BOOL
<intexp>1 ::= <number>
            | <id>
              Cond: typeof(id.lexval,<intexp>1.alltbl) = INT
            | <intexp>2 + <intexp>3
              <intexp>2.alltbl := <intexp>1.alltbl
              <intexp>3.alltbl := <intexp>1.alltbl
```

Type Checking Example (going further)

We can now evaluate the assignment statement in:

```
begin
  bool i;
  int j;
  begin
    int x;
    int i;
    x := i + j;
  end
end
```

[
{("x",INT), ("i",INT)},
{("i",BOOL), ("j",INT)}
]

bottom of stack