### Lecture 06 – Semantic Analysis

**THEORY OF COMPILATION**

Eran Yahav

---

**You are here...**

- Source text
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Representation (IR)
- Code Generation

---

**What we want**

Potato potato;
Carrot carrot;
\( x = \text{tomato} + \text{potato} + \text{carrot} \)

Lexical analyzer:

Parser:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{tomato}</td>
<td>\text{}</td>
</tr>
<tr>
<td>\text{PLUS}</td>
<td>\text{}</td>
</tr>
<tr>
<td>\text{PLUS}</td>
<td>\text{}</td>
</tr>
<tr>
<td>\text{carrot}</td>
<td>\text{}</td>
</tr>
</tbody>
</table>

- \text{tomato} is undefined
- \text{potato} used before initialized
- Cannot add \text{Potato} and \text{Carrot}

---

**You are here**

Compiler

- Source text
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Representation (IR)
- Code Generation

Executable code
Contextual Analysis

- Often called “Semantic analysis”

- Properties that cannot be formulated via CFG
  - Type checking
  - Declare before use
  - Identifying the same word “w” re-appearing – wbw
  - Initialization
  - ...

- Properties that are hard to formulate via CFG
  - “break” only appears inside a loop
  - ...

- Processing of the AST

Identification

```
month : integer RANGE [1..12];
...
month := 1;
while (month <= 12) {
  print(month_name[month]);
  month := month + 1;
}
```

- Forward references?
- Languages that don't require declarations?

Symbol table

```
month : integer RANGE [1..12];
...
month := 1;
while (month <= 12) {
  print(month_name[month]);
  month := month + 1;
}
```

- A table containing information about identifiers in the program
- Single entry for each named item
Not so fast...

```c
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
}
```

Not so fast...

```c
struct one_int {
    int i;
} i;

main() {
    i.i = 42;
    int t = i.i;
    printf("%d", t);
    
    int i = 73;
    printf("%d", i);
}
```

Scopes

- Typically stack structured scopes
- Scope entry
  - push new empty scope element
- Scope exit
  - pop scope element and discard its content
- Identifier declaration
  - identifier created inside top scope
- Identifier Lookup
  - Search for identifier top-down in scope stack

Scope-structured symbol table
Scope and symbol table

- Scope x Identifier -> properties
  - Expensive lookup

- A better solution
  - hash table over identifiers

Hash-table based Symbol Table

Scope info

Remember lexing/parsing?

- How did we know to always map an identifier to the same token?
Semantic Checks

- Scope rules
  - Use symbol table to check that
  - Identifiers defined before used
  - No multiple definition of same identifier
  - Program conforms to scope rules

- Type checking
  - Check that types in the program are consistent
  - How?

Types

- What is a type?
  - Simplest answer: a set of values
  - Integers, real numbers, booleans, ...

- Why do we care?
  - Safety
    - Guarantee that certain errors cannot occur at runtime
    - Abstraction
      - Hide implementation details
    - Documentation
    - Optimization

Type System (textbook definition)

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute”

-- Types and Programming Languages / Benjamin C. Pierce

Type System

- A type system of a programming language is a way to define how “good” program behave
  - Good programs = well-typed programs
  - Bad programs = not well typed

- Type checking
  - Static typing – most checking at compile time
  - Dynamic typing – most checking at runtime

- Type inference
  - Automatically infer types for a program(or show that there is no valid typing)
Static typing vs. dynamic typing

- Static type checking is conservative
  - Any program that is determined to be well-typed is free from certain kinds of errors
  - May reject programs that cannot be statically determined as well typed
  - Why?

- Dynamic type checking
  - May accept more programs as valid (runtime info)
  - Errors not caught at compile time
  - Runtime cost

Type Checking

- Type rules specify
  - Which types can be combined with certain operator
  - Assignment of expression to variable
  - Formal and actual parameters of a method call

- Examples
  - `string + string`
  - "drive" + "drink"
  - `string`
  - `int + "the answer"
  - ERROR

Type Checking Rules

- Specify for each operator
  - Types of operands
  - Type of result

- Basic Types
  - Building blocks for the type system (type rules)
  - E.g., int, boolean, (sometimes) string

- Type Expressions
  - Array types
  - Function types
  - Record types / Classes

Typing Rules

If E1 has type int and E2 has type int, then E1 + E2 has type int

<table>
<thead>
<tr>
<th>E1 : int</th>
<th>E2 : int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>E1 + E2 : int</td>
<td></td>
</tr>
</tbody>
</table>

(Generally, also use a context A)
More Typing Rules (examples)

A = true : boolean
A = false : boolean
A = int-literal : int
A = string-literal : string

A = E1 : int
A = E2 : int
A = E1 op E2 : int

A = E1 : int
A = E2 : int
A = E1 rop E2 : boolean

A = E1 : boolean
A = E2 : boolean
A = E1 rop E2 : boolean

A = E1 : int
A = E2 : int
A = E1 [E2] : T

A = T in C
A = new T() : T
A = id : T

And Even More Typing Rules

A = E1 : boolean
A = E2 : boolean
A = E1 op E2 : boolean

A = E1 : int
A = E2 : int
A = E1[0] : boolean

A = E1 : boolean
A = E2 : boolean
A = E1 [E2] : T

A = new T[1][1] : T
A = new T[1][2] : T
A = new T[1] : boolean

A = id : T

Type Checking

- Traverse AST and assign types for AST nodes
  - Use typing rules to compute node types
- Alternative: type-check during parsing
  - More complicated alternative
  - But naturally also more efficient

Example

A = E1 : boolean
A = E2 : boolean
A = E1 op E2 : boolean

A = E1 : boolean
A = E2 : boolean
A = E1 [E2] : boolean

A = false : boolean
A = string-literal : string
Type Declarations

- So far, we ignored the fact that types can also be declared

    TYPE Int_Array = ARRAY [Integer 1..42] OF Integer;  (explicitly)

    Var a : ARRAY [Integer 1..42] OF Real;  (anonymously)

Forward References

    TYPE Ptr_List_Entry = POINTER TO List_Entry;
    TYPE List_Entry =
        RECORD
            Element : Integer;
            Next : Ptr_List_Entry;
        END RECORD;

- Forward references must be resolved
  - A forward reference is added to the symbol table as forward reference, and later updated when type declaration is met
  - At the end of scope, must check that all forward references have been resolved
  - Check must be added for circularity

Type Table

- All types in a compilation unit are collected in a type table

- For each type, its table entry contains:
  - Type constructor: basic, record, array, pointer, ...
  - Size and alignment requirements
    - to be used later in code generation
  - Types of components (if applicable)
    - e.g., types of record fields
Type Equivalence: Name Equivalence

Type \( t_1 = \text{ARRAY}[\text{Integer}] \text{ OF Integer} \);  
Type \( t_2 = \text{ARRAY}[\text{Integer}] \text{ OF Integer} \);  
\( t_1 \) not (name) equivalence to \( t_2 \)

Type \( t_3 = \text{ARRAY}[\text{Integer}] \text{ OF Integer} \);  
Type \( t_4 = t_3 \);  
\( t_3 \) equivalent to \( t_4 \)

Type Equivalence: Structural Equivalence

Type \( t_5 = \text{RECORD} \ c : \text{Integer}; \ p : \text{POINTER TO} t_5; \ \text{END RECORD}; \);
Type \( t_6 = \text{RECORD} \ c : \text{Integer}; \ p : \text{POINTER TO} t_6; \ \text{END RECORD}; \);
Type \( t_7 = \text{RECORD} \ c : \text{Integer}; \ p : \text{POINTER TO} t_5; \ \text{END RECORD}; \);  
\( t_5, t_6, t_7 \) are all (structurally) equivalent

In practice

- Almost all modern languages use name equivalence
- why?

Coercions

- If we expect a value of type \( T_1 \) at some point in the program, and find a value of type \( T_2 \), is that acceptable?

\[
\begin{align*}
\text{float } x = 3.141; \\
\text{int } y = x;
\end{align*}
\]
1-values and r-values

\[ \text{dst} := \text{src} \]

- What is dst? What is src?
  - dst is a memory location where the value should be stored
  - src is a value
- “location” on the left of the assignment called an l-value
- “value” on the right of the assignment is called an r-value

l-values and r-values (example)

\[ x := \text{y} + 1 \]

expression construct | resulting kind
--- | ---
constant | rvalue
identifier (variable) | value
identifier (otherwise) | rvalue
&lvalue | rvalue
*rvalue | value
V[\text{value}] | V
V.selector | V
rvalue+rvalue | rvalue
value := rvalue | value
l-values and r-values

<table>
<thead>
<tr>
<th>found</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvalue</td>
<td>-</td>
</tr>
<tr>
<td>rvalue</td>
<td>error</td>
</tr>
</tbody>
</table>

So far...

- Static correctness checking
  - Identification
  - Type checking
- Identification matches applied occurrences of identifier to its defining occurrence
- Type checking checks which type combinations are legal
- Each node in the AST of an expression represents either an l-value (location) or an r-value (value)

How does this magic happen?

- We probably need to go over the AST?
- how does this relate to the clean formalism of the parser?

Syntax Directed Translation

- Semantic attributes
  - Attributes attached to grammar symbols
- Semantic actions
  - (already mentioned when we did recursive descent)
  - How to update the attributes
- Attribute grammars
Attribute grammars

- Attributes
  - Every grammar symbol has attached attributes
    - Example: Expr.type
- Semantic actions
  - Every production rule can define how to assign values to attributes
    - Example:
      \[
      \text{Expr} \rightarrow \text{Expr} + \text{Term} \\
      \text{Expr.type} = \text{Expr1.type} \text{ when } (\text{Expr1.type} \equiv \text{Term.type}) \\
      \text{Error otherwise}
      \]

Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions
  - Example:
    \[
    \text{Expr} \rightarrow \text{Expr1} + \text{Term} \\
    \text{Becomes} \\
    \text{Expr} \rightarrow \text{Expr11} + \text{Term}
    \]

Example

```
float x, y, z
```

Dependencies

- A semantic equation \( a = b_1, \ldots, b_m \) requires computation of \( b_1, \ldots, b_m \) to determine the value of \( a \)
  - The value of \( a \) depends on \( b_1, \ldots, b_m \)
    - We write \( a \leftarrow b_i \)
Attribute Evaluation

- Build the AST
- Fill attributes of terminals with values derived from their representation
- Execute evaluation rules of the nodes to assign values until no new values can be assigned
  - In the right order such that
    - No attribute value is used before its available
    - Each attribute will get a value only once

Cycles

- Cycle in the dependence graph
- May not be able to compute attribute values

\[ E.S = T_i \]
\[ T_i = E.S + 1 \]

Building Dependency Graph

- All semantic equations take the form
  \[ \text{attr1} = \text{func1(attr1.1, attr1.2, ...)} \]
  \[ \text{attr2} = \text{func2(attr2.1, attr2.2, ...)} \]
- Actions with side effects use a dummy attribute
- Build a directed dependency graph \( G \)
  - For every attribute \( a \) of a node \( n \) in the AST create a node \( n.a \)
  - For every node \( n \) in the AST and a semantic action of the form \( b = f(c_1, c_2, ... c_k) \) add edges of the form \( (c_i, b) \)
Topological Order

- For a graph $G=(V,E)$, $|V|=k$

- Ordering of the nodes $v_1, v_2, \ldots, v_k$ such that for every edge $(v_i, v_j) \in E$, $i < j$

Example topological orderings: 1 4 3 2 5, 4 1 3 5 2
But what about cycles?

- For a given attribute grammar hard to detect if it has cyclic dependencies
  - Exponential cost
- Special classes of attribute grammars
  - Our “usual trick”
  - Sacrifice generality for predictable performance

Inherited vs. Synthesized Attributes

- Synthesized attributes
  - Computed from children of a node
- Inherited attributes
  - Computed from parents and siblings of a node
- Attributes of tokens are technically considered as synthesized attributes

Example

S-attributed Grammars

- Special class of attribute grammars
- Only uses synthesized attributes (S-attributed)
- No use of inherited attributes
- Can be computed by any bottom-up parser during parsing
- Attributes can be stored on the parsing stack
- Reduce operation computes the (synthesized) attribute from attributes of children
S-attributed Grammar: example

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow E$</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>$E \rightarrow E1 + T$</td>
<td>E.val = E1.val + T.val</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>$T \rightarrow T1 * F$</td>
<td>T.val = T1.val * F.val</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>

L-attributed grammars

- L-attributed attribute grammar when every attribute in a production $A \rightarrow X_1...X_n$ is
  - A synthesized attribute, or
  - An inherited attribute of $X_j$, $1 \leq j \leq n$ that only depends on
    - Attributes of $X_1...X_{j-1}$ to the left of $X_j$, or
    - Inherited attributes of $A$

Summary

- Contextual analysis can move information between nodes in the AST
  - Even when they are not “local”
- Attribute grammars
  - Attach attributes and semantic actions to grammar
- Attribute evaluation
  - Build dependency graph, topological sort, evaluate
- Special classes with pre-determined evaluation order: S-attributed, L-attributed
The End