Lecture 06 – Semantic Analysis

THEORY OF Compilation

Eran Yahav
You are here

Source text → Compiler

Lexical Analysis → Syntax Analysis

Executable code
You are here...

Back End

Source text → Process text input → Lexical Analysis → Syntax Analysis → AST → Sem. Analysis → Annotated AST → Intermediate code generation → IR Optimizations → IR Code generation → Target code optimization → Symbolic Instructions → Machine code generation → Write executable output → Executable code
What we want

Potato potato;
Carrot carrot;
x = tomato + potato + carrot

Lexical analyzer

{id,tomato>,<PLUS>,<id,potato>,<PLUS>,<id,carrot>,EOF

Parser

<table>
<thead>
<tr>
<th>symbol</th>
<th>kind</th>
<th>type</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>tomato</td>
<td>var</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>potato</td>
<td>var</td>
<td>Potato</td>
<td></td>
</tr>
<tr>
<td>carrot</td>
<td>var</td>
<td>Carrot</td>
<td></td>
</tr>
</tbody>
</table>

tomato is undefined
potato used before initialized
Cannot add Potato and Carrot
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
Contextual Analysis

- Identification
  - Gather information about each named item in the program
  - e.g., what is the declaration for each usage

- Context checking
  - Type checking
  - e.g., the condition in an if-statement is a Boolean
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

- A table containing information about identifiers in the program
- Single entry for each named item

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

<table>
<thead>
<tr>
<th>name</th>
<th>pos</th>
<th>type</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>month</td>
<td>1</td>
<td>RANGE[1..12]</td>
<td></td>
</tr>
<tr>
<td>month_name</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Not so fast...

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

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

- A struct field named i
- A struct variable named i
- Assignment to the “i” field of struct “i”
- Reading the “i” field of struct “i”
struct one_int {
    int i;
} i;

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

A struct field named i

A struct variable named i

Assignment to the “i” field of struct “i”

Reading the “i” field of struct “i”

int variable named “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

```
{  
  int the=1;  
  int fish=2;  
  Int thanks=3;  
  {  
    int x = 42;  
    int all = 73;  
    {  
      ...  
    }  
  }  
}
```
Scope and symbol table

- Scope x Identifier -> properties
  - Expensive lookup

- A better solution
  - hash table over identifiers
Hash-table based Symbol Table

Id.info

```
name   //
macro
decl   2 P

name   //
macro
decl   2 P

name   //
macro
decl   3 P
```

"x"

"thanks"

"so"
Scope info

Scope stack

(now just pointers to the corresponding record in the symbol table)
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

```java
string string
  “drive” + “drink”
  string

int string
  42 + “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

E1 : int     E2 : int
-------------
E1 + E2 : int

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

\[
\begin{align*}
A \vdash \text{true} : \text{boolean} & \quad A \vdash \text{false} : \text{boolean} \\
A \vdash \text{int-literall} : \text{int} & \quad A \vdash \text{string-literall} : \text{string}
\end{align*}
\]

\[
\begin{align*}
A \vdash E_1 : \text{int} & \quad A \vdash E_2 : \text{int} \\
A \vdash E_1 \ op \ E_2 : \text{int} & \quad \ op \in \{ +, -, /, *, \%\}
\end{align*}
\]

\[
\begin{align*}
A \vdash E_1 : \text{int} & \quad A \vdash E_2 : \text{int} \\
A \vdash E_1 \ rop \ E_2 : \text{boolean} & \quad \ rop \in \{ \leq, <, >, \geq\}
\end{align*}
\]

\[
\begin{align*}
A \vdash E_1 : T & \quad A \vdash E_2 : T \\
A \vdash E_1 \ rop \ E_2 : \text{boolean} & \quad \ rop \in \{ =, !=\}
\end{align*}
\]
And Even More Typing Rules

\[
\frac{A \vdash E_1 : \text{boolean} \quad A \vdash E_2 : \text{boolean}}{A \vdash E_1 \; \text{lop} \; E_2 : \text{boolean}} \quad \text{lop} \in \{ \&\& , |\| \} \\
\]

\[
\frac{A \vdash E_1 : \text{int}}{A \vdash -E_1 : \text{int}}
\]

\[
\frac{A \vdash E_1 : \text{boolean}}{A \vdash !E_1 : \text{boolean}}
\]

\[
\frac{A \vdash E_1 : T[\]}{A \vdash \text{E1.length} : \text{int}}
\]

\[
\frac{A \vdash E_1 : T[\]}{A \vdash E_1[E2] : T}
\]

\[
\frac{A \vdash E_1 : \text{int}}{A \vdash \text{new T[E1]} : T[\]}
\]

\[
\frac{A \vdash T \; \text{in C}}{A \vdash \text{new T()} : T}
\]

\[
\frac{id : T \in A}{A \vdash id : T}
\]
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

\[ 45 > 32 \land \neg \text{false} \]

\begin{align*}
\text{A} &\leftarrow \text{E1} : \text{boolean} \quad \text{A} \leftarrow \text{E2} : \text{boolean} \\
\text{A} &\leftarrow \text{E1} \ \text{lop} \ \text{E2} : \text{boolean} \\
\text{lop} &\in \{ \land, \lor \} \\
\text{A} &\leftarrow \neg \text{E1} : \text{boolean} \\
\text{A} &\leftarrow \text{E1} : \text{int} \quad \text{A} \leftarrow \text{E2} : \text{int} \\
\text{A} &\leftarrow \text{E1} \ \text{rop} \ \text{E2} : \text{boolean} \\
\text{rop} &\in \{ \leq, <, >, \geq \} \\
\text{A} &\leftarrow \text{false} : \text{boolean} \\
\text{A} &\leftarrow \text{int-literal} : \text{int}
\end{align*}
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)
```
Type Declarations

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

TYPE #type01_in_line_73 = ARRAY [Integer 1..42] OF Real;
Var a : #type01_in_line_73;
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 references 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 \( t_5 \) = RECORD \( c: \) Integer; \( p: \) POINTER TO \( t_5; \) END RECORD;
Type \( t_6 \) = RECORD \( c: \) Integer; \( p: \) POINTER TO \( t_6; \) END RECORD;
Type \( t_7 \) =
  RECORD
    c: Integer;
    p: POINTER TO
      RECORD
        c: Integer;
        p: POINTER to \( t_5; \)
      END RECORD;
  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?

```plaintext
float x = 3.141;
int y = x;
```
l-values and r-values

dst := 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)

\[ x := y + 1 \]
l-values and r-values (example)

\[
x := A[1]
\]

\[
x := A[A[1]]
\]
### l-values and r-values (examples)

<table>
<thead>
<tr>
<th>expression construct</th>
<th>resulting kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>rvalue</td>
</tr>
<tr>
<td>identifier (variable)</td>
<td>lvalue</td>
</tr>
<tr>
<td>identifier (otherwise)</td>
<td>rvalue</td>
</tr>
<tr>
<td>&amp;lvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>*rvalue</td>
<td>lvalue</td>
</tr>
<tr>
<td>V[rvalue]</td>
<td>V</td>
</tr>
<tr>
<td>V.selector</td>
<td>V</td>
</tr>
<tr>
<td>rvalue+rvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>lvalue := rvalue</td>
<td>rvalue</td>
</tr>
</tbody>
</table>
l-values and r-values

<table>
<thead>
<tr>
<th></th>
<th>expected</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lvalue</strong></td>
<td></td>
<td><strong>rvalue</strong></td>
</tr>
<tr>
<td>lvalue</td>
<td>-</td>
<td>deref</td>
</tr>
<tr>
<td>rvalue</td>
<td>error</td>
<td>-</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:
      
      ```
      Expr → Expr + Term
      Expr.type = Expr1.type when (Expr1.type == Term.type)
      Error otherwise
      ```
Indexed symbols

- Add indexes to distinguish repeated grammar symbols
- Does not affect grammar
- Used in semantic actions

- $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
  Becomes
- $\text{Expr} \rightarrow \text{Expr}_1 + \text{Term}$
Example

float \text{x, y, z}

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
<tr>
<td>L → L1, id</td>
<td>L1.in = L.in, addType(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addType(id.entry, L.in)</td>
</tr>
</tbody>
</table>
Dependencies

- A semantic equation \( a = b_1, ..., b_m \) requires computation of \( b_1, ..., b_m \) to determine the value of \( a \)

- The value of \( a \) depends on \( b_1, ..., b_m \)
  - 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

\[ E.S = T.i \]
\[ T.i = E.s + 1 \]
Attribute Evaluation

- Build the AST
- Build dependency graph
- Compute evaluation order using topological ordering
- Execute evaluation rules based on topological ordering

- Works as long as there are no cycles
Building Dependency Graph

- All semantic equations take the form
  
  \[ \text{attr1} = \text{func1}(\text{attr1.1}, \text{attr1.2}, \ldots) \]
  \[ \text{attr2} = \text{func2}(\text{attr2.1}, \text{attr2.2}, \ldots) \]

- 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 \( b = f(c_1, c_2, \ldots c_k) \) add edges of the form \((c_i, b)\)
Example

float x,y,z

<table>
<thead>
<tr>
<th>Prod.</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type = float</td>
</tr>
</tbody>
</table>
| L → L1, id | L1.in = L.in
addType(id.entry,L.in) |
| L → id  | addType(id.entry,L.in)                            |
Example

float x, y, z

Prod. | Semantic Rule
--- | ---
D → T L | L.in = T.type
T → int | T.type = integer
T → float | T.type = float
L → L1, id | L1.in = L.in
           | addType(id.entry, L.in)
L → id | addType(id.entry, L.in)
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
Example

float x, y, z

float 1

float 5

float 6

type

in

dmy

entry

ent1

ent2

ent3

float 2

float 3

float 4

float 7

float 9

float 8

float 10
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

Production | Semantic Rule
--- | ---
$D \rightarrow T L$ | $L.in = T.type$
$T \rightarrow int$ | $T.type = integer$
$T \rightarrow float$ | $T.type = float$
$L \rightarrow L_1, id$ | $L_1.in = L.in$
 | addType(id.entry, L.in)
$L \rightarrow id$ | addType(id.entry, L.in)

- **inherited**
- **synthesized**
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 → E ;</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>E → E₁ + T</td>
<td>E.val = E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>T → T₁ * F</td>
<td>T.val = T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>F → (E)</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val = digit.lexval</td>
</tr>
</tbody>
</table>
example
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