Overview

- Static semantics
- Dynamic semantics
- Attribute grammars
- Abstract syntax trees
Static Semantics

- **Syntax** concerns the form of a valid program, while **semantics** concerns its meaning
  - Context-free grammars are not powerful enough to describe certain rules, e.g. checking variable declaration with variable use

- **Static semantic** rules are enforced by a compiler at compile time
  - Implemented in semantic analysis phase of the compiler

**Examples:**
- Type checking
- Identifiers are used in appropriate context
- Check subroutine call arguments
- Check labels
Dynamic Semantics

- Dynamic semantic rules are enforced by the compiler by generating code to perform the checks at run-time.

Examples:
- Array subscript values are within bounds
- Arithmetic errors
- Pointers are not dereferenced unless pointing to valid object
- A variable is used but hasn't been initialized

- Some languages (Euclid, Eiffel) allow programmers to add explicit dynamic semantic checks in the form of assertions, e.g.
  ```
  assert denominator not= 0
  ```

- When a check fails at run time, an exception is raised.
Attribute Grammars

- An attribute grammar “connects” syntax with semantics
- Each grammar production has a *semantic rule with actions* (e.g. assignments) to modify values of *attributes* of (non)terminals
  - A (non)terminal may have any number of attributes
  - Attributes have values that hold information related to the (non)terminal
- General form:

  production:  
  `<A> ::= <B> <C>`

  semantic rule:  
  `A.a := ...; B.a := ...; C.a := ...`

- Semantic rules are used by a compiler to enforce static semantics and/or to produce an abstract syntax tree while parsing tokens
- Can also be used to build simple language interpreters
Example Attributed Grammar

- The $val$ attribute of a (non)terminal holds the subtotal value of the subexpression
- Nonterminals are indexed in the attribute grammar to distinguish multiple occurrences of the nonterminal in a production

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;E_1&gt;$ ::= $&lt;E_2&gt;$ + $&lt;T&gt;$</td>
<td>$E_1.val := E_2.val + T.val$</td>
</tr>
<tr>
<td>$&lt;E_1&gt;$ ::= $&lt;E_2&gt;$ - $&lt;T&gt;$</td>
<td>$E_1.val := E_2.val - T.val$</td>
</tr>
<tr>
<td>$&lt;E&gt;$ ::= $&lt;T&gt;$</td>
<td>$E.val := T.val$</td>
</tr>
<tr>
<td>$&lt;T_1&gt;$ ::= $&lt;T_2&gt;$ * $&lt;F&gt;$</td>
<td>$T_1.val := T_2.val * F.val$</td>
</tr>
<tr>
<td>$&lt;T_1&gt;$ ::= $&lt;T_2&gt;$ / $&lt;F&gt;$</td>
<td>$T_1.val := T_2.val / F.val$</td>
</tr>
<tr>
<td>$&lt;T&gt;$ ::= $&lt;F&gt;$</td>
<td>$T.val := F.val$</td>
</tr>
<tr>
<td>$&lt;F_1&gt;$ ::= - $&lt;F_2&gt;$</td>
<td>$F_1.val := -F_2.val$</td>
</tr>
<tr>
<td>$&lt;F&gt;$ ::= ( $&lt;E&gt;$ )</td>
<td>$F.val := E.val$</td>
</tr>
<tr>
<td>$&lt;F&gt;$ ::= unsigned_int</td>
<td>$F.val :=$ unsigned_int.val</td>
</tr>
</tbody>
</table>
Decorated Parse Trees

- A parser produces a parse tree that is *decorated* with the attribute values

- Example decorated parse tree of \((1+3)*2\) with the val attributes
Synthesized Attributes

- Synthesized attributes of a node hold values that are computed from attribute values of the child nodes in the parse tree and therefore information flows upwards.

Production:

\[
<E_1> ::= <E_2> + <T>
\]

Semantic Rule:

\[
E_1.val := E_2.val + T.val
\]
Inherited Attributes

- *Inherited attributes* of *child* nodes are set by the *parent* node and therefore information flows *downwards*

**production**

\[
\text{\(<E>\)} := \text{\(<T>\) \text{\(<TT>\)}>}
\]

\[
\text{\(<TT_1>\)} := + \text{\(<T>\) \text{\(<TT_2>\)>}
\]

\[
\text{\(<TT>\)} := \varepsilon
\]

**semantic rule**

\[
\text{\(<TT>\).st := \text{T.val}; \text{\(<TT>\).val := \text{TT.val}>}
\]

\[
\text{\(<TT_1>\).val := \text{TT_1.st + T.val}; \text{\(<TT_1>\).val := \text{TT_2.val}>}
\]

\[
\text{\(<TT>\).val := \text{TT.st>}
\]
Attribute Flow

- An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set (write) and use (read) dependencies (an attribute must be set before it is used)

```
production  semantic rule
<E> ::= <T> <TT>        TT.st := T.val
```

```
E
  /  
 T   TT
  \
   +
   T
   T
   ε
```
Attribute Flow

- An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used).

production

\(<TT_1> ::= + <T> <TT_2>\)

semantic rule

\(TT_2.st := TT_1.st + T.val\)
Attribute Flow

- An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used).

```
production
<TT> ::= ε

semantic rule
TT.val := TT.st
```
Attribute Flow

- An *attribute flow algorithm* propagates attribute values through the parse tree by traversing the tree according to the *set* (write) and *use* (read) dependencies (an attribute must be set before it is used)

```
production
<TT₁> ::= + <T> <TT₂>

semantic rule
TT₁.val := TT₂.val
```
Attribute Flow

- An attribute flow algorithm propagates attribute values through the parse tree by traversing the tree according to the set (write) and use (read) dependencies (an attribute must be set before it is used)

```
production
<E> ::= <T> <TT>

semantic rule
E.val := TT.val
```
S- and L-Attributed Grammars

- A grammar is called *S-attributed* if all attributes are synthesized.
- A grammar is called *L-attributed* if the parse tree traversal to update attribute values is always left-to-right and depth-first:
  - Synthesized attributes always OK
  - Values of inherited attributes must be passed down to children from left to right
  - Semantic rules can be applied immediately during parsing and parse trees do not need to be kept in memory
  - This is an essential grammar property for a one-pass compiler
- An S-attributed grammar is a special case of an L-attributed grammar
Example L-Attributed Grammar

- Implements a calculator

production

\[<E> ::= <T> <TT>\]
\[<TT> ::= + <T> <TT>\]
\[<TT> ::= - <T> <TT>\]
\[<TT> ::= \epsilon\]
\[<T> ::= <F> <FT>\]
\[<FT> ::= * <F> <FT>\]
\[<FT> ::= / <F> <FT>\]
\[<FT> ::= \epsilon\]
\[<F> ::= - <F>\]
\[<F> ::= ( <E> )\]
\[<F> ::= \text{unsigned\_int}\]

semantic rule

\[TT\text{.st} := T\text{.val}; E\text{.val} := TT\text{.val}\]
\[TT_2\text{.st} := TT_1\text{.st} + T\text{.val}; TT_1\text{.val} := TT_2\text{.val}\]
\[TT_2\text{.st} := TT_1\text{.st} - T\text{.val}; TT_1\text{.val} := TT_2\text{.val}\]
\[TT\text{.val} := TT\text{.st}\]
\[FT\text{.st} := F\text{.val}; T\text{.val} := FT\text{.val}\]
\[FT_2\text{.st} := FT_1\text{.st} * F\text{.val}; FT_1\text{.val} := FT_2\text{.val}\]
\[FT_2\text{.st} := FT_1\text{.st} / F\text{.val}; FT_1\text{.val} := FT_2\text{.val}\]
\[FT\text{.val} := FT\text{.st}\]
\[F_1\text{.val} := \text{-}F_2\text{.val}\]
\[F\text{.val} := E\text{.val}\]
\[F\text{.val} := \text{unsigned\_int\_val}\]
Example Decorated Parse Tree

- Fully decorated parse tree of \((1+3)*2\)
Recursive Descent Parsing with L-Attributed Grammars

- Semantic rules are added to the bodies of the recursive descent functions and placed appropriately between the function calls.

- Inherited attribute values are input arguments to the functions.
  - Argument passing flows downwards in call graphs.

- Synthesized attribute values are returned by functions.
  - Return values flow upwards in call graphs.
**Example**

**production**

\[
\begin{align*}
  <E> & ::= <T> <TT> \\
  <TT_1> & ::= + <T> <TT_2> \\
  <TT_1> & ::= - <T> <TT_2> \\
  <TT> & ::= \varepsilon
\end{align*}
\]

**semantic rule**

\[
\begin{align*}
  TT.st & ::= T.val; E.val ::= TT.val \\
  TT_2.st & ::= TT_1.st + T.val; TT_1.val ::= TT_2.val \\
  TT_2.st & ::= TT_1.st - T.val; TT_1.val ::= TT_2.val \\
  TT.val & ::= TT.st
\end{align*}
\]

**procedure**

```plaintext
procedure E()
  Tval = T();
  Eval = TT(Tval);
  return Eval;

procedure TT(TTst)
  case (input_token())
  of '+' : match('+');
      Tval = T();
      TTval = TT(TTst + Tval);
  of '-' : match('-');
      Tval = T();
      TTval = TT(TTst - Tval);
  otherwise: TTval = TTst;
  return TTval;
```
Constructing Abstract Syntax Trees with Attribute Grammars

- Three operations to create nodes for an AST tree that represents expressions:
  - `mk_bin_op(op, left, right)`: constructs a new node that contains a binary operator `op` and AST sub-trees `left` and `right` representing the operator’s operands and returns pointer to the new node
  - `mk_un_op(op, node)`: constructs a new node that contains a unary operator `op` and sub-tree `node` representing the operator’s operand and returns pointer to the new node
  - `mk_leaf(value)`: constructs an AST leaf that contains a value and returns pointer to the new node
An L-Attributed Grammar to Construct ASTs

- Semantic rules to build up an AST

<table>
<thead>
<tr>
<th>production</th>
<th>semantic rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E ::= T \ TT$</td>
<td>$TT.st := T.ptr; \ E.ptr := TT.ptr$</td>
</tr>
<tr>
<td>$TT1 ::= + T \ TT2$</td>
<td>$TT2.st := \text{mk_bin_op}(&quot;+&quot;, \ TT1.st, T.ptr); \ TT1.ptr := TT2.ptr$</td>
</tr>
<tr>
<td>$TT1 ::= - T \ TT2$</td>
<td>$TT2.st := \text{mk_bin_op}(&quot;-&quot;, \ TT1.st, T.ptr); \ TT1.ptr := TT2.ptr$</td>
</tr>
<tr>
<td>$TT ::= \epsilon$</td>
<td>$TT.ptr := TT.st$</td>
</tr>
<tr>
<td>$T ::= F \ FT$</td>
<td>$FT.st := F.ptr; \ T.ptr := FT.ptr$</td>
</tr>
<tr>
<td>$FT1 ::= \ast F \ FT2$</td>
<td>$FT2.st := \text{mk_bin_op}(&quot;\ast&quot;, \ FT1.st, F.ptr); \ FT1.ptr := FT2.ptr$</td>
</tr>
<tr>
<td>$FT1 ::= / F \ FT2$</td>
<td>$FT2.st := \text{mk_bin_op}(=&quot;/&quot;, \ FT1.st, F.ptr); \ FT1.ptr := FT2.ptr$</td>
</tr>
<tr>
<td>$FT ::= \epsilon$</td>
<td>$FT.ptr := FT.st$</td>
</tr>
<tr>
<td>$F1 ::= - F2$</td>
<td>$F1.ptr := \text{mk_un_op}(&quot;-&quot;, \ F2.ptr)$</td>
</tr>
<tr>
<td>$F ::= ( E )$</td>
<td>$F.ptr := E.ptr$</td>
</tr>
<tr>
<td>$F ::= \text{unsigned_int}$</td>
<td>$F.ptr := \text{mk_leaf}(\text{unsigned_int.val})$</td>
</tr>
</tbody>
</table>
Example Decorated Parse Tree with AST

- Decorated parse tree of \((1+3)*2\) with AST