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 a 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         semantic rule
  \(<A> ::= <B> <C>\) \(A.a := ...; B.a := ...; C.a := ...\)

- 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

Production

```
production                      semantic rule
<E_1> ::= <E_2> + <T>        E_1.val := E_2.val + T.val
<E_1> ::= <E_2> - <T>        E_1.val := E_2.val - T.val
<E>  ::= <T>                 E.val := T.val
<T_1> ::= <T_2> * <F>        T_1.val := T_2.val * F.val
<T_1> ::= <T_2> / <F>        T_1.val := T_2.val / F.val
<T>  ::= <F>                 T.val := F.val
<F_1> ::= - <F_2>            F_1.val := -F_2.val
<F>  ::= ( <E> )             F.val := E.val
<F>  ::= unsigned_int        F.val := unsigned_int.val
```
Decorated Parse Trees

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

- Example decorated parse tree of 
  \((1+3)\times2\) 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**

- `<E>` ::= `<T>` `<TT>`
- `<TT_1>` ::= + `<T>` `<TT_2>`
- `<TT>` ::= ε

**Semantic Rule**

- `TT.st := T.val; E.val := TT.val`
- `TT_2.st := TT_1.st + T.val; TT_1.val := TT_2.val`
- `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
<E> ::= <T> <TT>

semantic rule
TT.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

\[
<TT_1> ::= + <T> <TT_2>
\]

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

\[
<TT_1> ::= + <T> <TT_2>
\]

#### Semantic Rule

\[
TT_1.val := TT_2.val
\]
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  semantic rule
<E> ::= <T> <TT>    E.val ::= TT.val
```

```
E ::= T TT.
T ::= ...
TT ::= ...
```

![Diagram showing attribute flow through a parse tree with production and semantic rule]

10/7/16
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>` ::= `ε`
- `<T>` ::= `<F> `<FT>`
- `<FT>` ::= `<F> `<FT>`
- `<FT>` ::= `ε`
- `<F>` ::= `- `<F>`
- `<F>` ::= `( <E> )`
- `<F>` ::= `unsigned_int`

**semantic rule**

- `TT.st := T.val; E.val := TT.val`
- `TT2.st := TT1.st + T.val; TT1.val := TT2.val`
- `TT2.st := TT1.st - T.val; TT1.val := TT2.val`
- `TT.val := TT.st`
- `FT.st := F.val; T.val := FT.val`
- `FT2.st := FT1.st * F.val; FT1.val := FT2.val`
- `FT2.st := FT1.st / F.val; FT1.val := FT2.val`
- `FT.val := FT.st`
- `F1.val := -F2.val`
- `F.val := E.val`
- `F.val := 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

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

semantic rule

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

procedure E()

\[
\begin{align*}
  Tval & := T(); \\
  Eval & := TT(Tval); \\
  \text{return } & \text{ Eval;}
\end{align*}
\]

procedure TT(TTst)

\[
\begin{align*}
  \text{case (input_token())} \\
  & \text{of ' + ': match('+');} \\
  & \quad Tval := T(); \\
  & \quad TTval := TT(TTst + Tval); \\
  & \text{of ' - ': match('-');} \\
  & \quad Tval := T(); \\
  & \quad TTval := TT(TTst - Tval); \\
  & \text{otherwise: } \quad TTval := TTst; \\
  & \text{return } \quad TTval;
\end{align*}
\]
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>(&lt;E&gt;) ::= (&lt;T&gt;) (&lt;TT&gt;)</td>
<td>(TT).st := (T).ptr; (E).ptr := (TT).ptr</td>
</tr>
<tr>
<td>(&lt;TT_1&gt;) ::= (+) (&lt;T&gt;) (&lt;TT_2&gt;)</td>
<td>(TT_2).st := (mk_bin_op(&quot;+&quot;, TT_1.st, T.ptr); TT_1.ptr := TT_2.ptr</td>
</tr>
<tr>
<td>(&lt;TT_1&gt;) ::= (-) (&lt;T&gt;) (&lt;TT_2&gt;)</td>
<td>(TT_2).st := (mk_bin_op(&quot;-&quot;, TT_1.st, T.ptr); TT_1.ptr := TT_2.ptr</td>
</tr>
<tr>
<td>(&lt;TT&gt;) ::= (\varepsilon)</td>
<td>(TT).ptr := TT.st</td>
</tr>
<tr>
<td>(&lt;T&gt;) ::= (&lt;F&gt;) (&lt;FT&gt;)</td>
<td>(FT).st := (F).ptr; (T).ptr := FT.ptr</td>
</tr>
<tr>
<td>(&lt;FT_1&gt;) ::= (*) (&lt;F&gt;) (&lt;FT_2&gt;)</td>
<td>(FT_2).st := (mk_bin_op(&quot;*&quot;, FT_1.st, F.ptr); FT_1.ptr := FT_2.ptr</td>
</tr>
<tr>
<td>(&lt;FT_1&gt;) ::= (/) (&lt;F&gt;) (&lt;FT_2&gt;)</td>
<td>(FT_2).st := (mk_bin_op(&quot;/&quot;, FT_1.st, F.ptr); FT_1.ptr := FT_2.ptr</td>
</tr>
<tr>
<td>(&lt;FT&gt;) ::= (\varepsilon)</td>
<td>(FT).ptr := FT.st</td>
</tr>
<tr>
<td>(&lt;F_1&gt;) ::= (-) (&lt;F_2&gt;)</td>
<td>(F_1).ptr := (mk_un_op(&quot;-&quot;, F_2.ptr)</td>
</tr>
<tr>
<td>(&lt;F&gt;) ::= ((&lt;E&gt;))</td>
<td>(F).ptr := (E).ptr</td>
</tr>
<tr>
<td>(&lt;F&gt;) ::= (\text{unsigned_int})</td>
<td>(F).ptr := (mk_leaf)(unsigned_int.val)</td>
</tr>
</tbody>
</table>
Example Decorated Parse Tree with AST

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

![Decorated parse tree of \((1+3)*2\) with AST]
Putting it all Together for LALR Parsing with Yacc/Bison

- Define an attributed grammar in Yacc/Bison
  - Define productions for non-terminals
  - Define attributes for (non-)terminals
  - Define semantic actions for productions to update attributes

- Define a lexical analyzer in Lex/Flex
  - Define regular expressions for tokens (grammar terminals)
  - Define actions to execute when RE patterns match

- Define syntax error handler to display errors
  - For example, `int yyerror(char *s)` for Bison

- Invoke the parser in main program
  - E.g. `yyparse()` in Bison
Constructing a Compiler with Lex/Flex and Yacc/Bison

- yacc specification
  - `yacc.y`

- Lex specification
  - `lex.l`
  - and token definitions
  - `y.tab.h`

- input stream

- `lex.yy.c`

- Yacc or Bison compiler
  - `y.tab.c`
  - `y.tab.h`

- Lex or Flex compiler
  - `lex.yy.c`

- C compiler
  - `a.out`

- `a.out`

- output stream
Yacc Specification of Attributed LALR Grammar

- A yacc specification consists of three parts:
  - yacc declarations, and C declarations within % {   % }
  - translation rules
  - user-defined auxiliary procedures

- The translation rules are productions with actions:
  - \( \text{production}_1 \)  \{  \text{semantic action}_1 \}  
  - \( \text{production}_2 \)  \{  \text{semantic action}_2 \}  
  - …
  - \( \text{production}_n \)  \{  \text{semantic action}_n \}  

Synthesized Attributes in Yacc Specifications

- Semantic actions refer to synthesized attributes of tokens and non-terminals in a production:
  \[ X : Y_1 Y_2 Y_3 \ldots Y_n \{ \text{semantic action} \} \]
  - \$\$ refers to the value of the single attribute of \( X \)
  - \$i \) refers to the value of the single attribute of \( Y_i \)

- For example
  \[
  \text{factor} : \text{'}(\text{expr}\text{')} \{ \$\$ = \$2; \}
  \]
Example Yacc Specification

 %{  
#include <stdio.h>  
#define YYSTYPE double  
%}

%token NUMBER  
%left '+' '-'  
%left '*' '/'  
%right UMINUS  
%%

lines : lines expr '\n' { printf("= %g\n", $2); }  
| lines '\n'  
| /* empty */  
;  
expr : expr '+' expr { $$ = $1 + $3; }  
| expr '-' expr { $$ = $1 - $3; }  
| expr '*' expr { $$ = $1 * $3; }  
| expr '/' expr { $$ = $1 / $3; }  
| '(' expr ')' { $$ = $2; }  
| '-' expr %prec UMINUS { $$ = -$2; }  
| NUMBER  
;  

Yystype: double for attributes and yylval  
Token definition for Lex  
Operator associativity and precedence (disambiguation)
Main Program (Added as Third Part in Yacc Specification)

```c
%%
int main()
{
    if (yyparse() != 0)
        fprintf(stderr, "Syntax error\n");
    return 0;
}

int yyerror(char *s)
{
    fprintf(stderr, "Error: %s\n", s);
}
```

Main program invokes parser

Syntax error reporting
Lex Specification for our Example

{%
#define YYSTYPE double
#include "y.tab.h"
%

number [0-9]+\.?|[0-9]*\.?[0-9]+ Regular definition for numbers

%%
[ ]
{ /* skip blanks */ }
{number} 
{ sscanf(yytext, "%lf", &yylval);
  return NUMBER;
}

\n|.
{ return yytext[0]; }
Set **yylval** is the synthesized attribute of tokens and has type **YYSTYPE**

Lexer returns tokens that are numeric codes
Compile and Run the Example

```bash
bison -d -y example.y
flex example.l
gcc y.tab.c lex.yy.c
./a.out

1+2
= 3
2*(3+4)
= 14
```