Semantic analysis

Semantic analysis

- Semantic analysis checks for the correctness of contextual dependences:
  - finds correspondence between declarations and usage of identifiers,
  - performs type checking/inference,
  - ...
- Syntax tree is decorated with typing- and other context dependent information.

Semantic analysis

- Semantic analysis checks restrictions imposed by a static semantics of the language.
- Sometimes it is possible to express semantic properties by context-free grammars, but usually this puts heavy restrictions to the language and/or complicates the grammar.
- Example – simple typed expressions:

$$
\begin{array}{lcl}
\mathrm{IntExp} & \to & \mathit{int} \mid \mathit{intVar} \\
 & \mid & \mathrm{IntExp} + \mathrm{IntExp} \\
\mathrm{BoolExp} & \to & \mathit{true} \mid \mathit{false} \\
 & \mid & \mathit{boolVar} \\
 & \mid & \mathrm{IntExp} \leq \mathrm{IntExp} \\
 & \mid & \mathit{not}\ \mathrm{BoolExp} \\
 & \mid & \mathrm{BoolExp}\ \&\ \mathrm{BoolExp}
\end{array}
$$

Semantic analysis

- At first glance, the grammar looks reasonable, but:
  - the grammar has two different (lexical) classes for variables;
  - additional types require new classes of variables;
  - most languages do not put restrictions to variable names based on their types;
  - moreover, usually one is allowed to use the same variable name for variables of different types in different context.

Attribute grammars

- Attribute grammars are generalization of context-free grammars, where:
  - each grammar symbol has an associated set of attributes;
  - each production rule has a set of attribute evaluation rules (or semantic rules).

- The goal is to find an evaluation of attributes which is consistent with the given semantic rules.

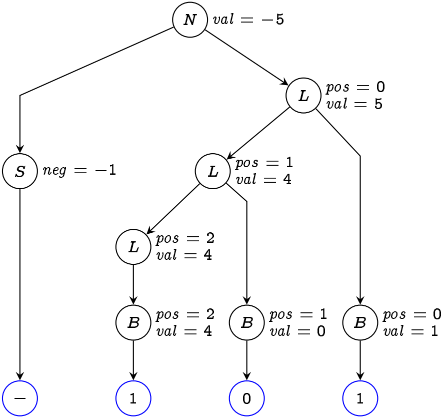
Attribute grammars

Example:

Productions	Semantic rules
$N \rightarrow S\ L$	$L.pos := 0$ $N.val := S.neg * L.val$
$S \rightarrow +$	$S.neg := 1$
$S \rightarrow -$	$S.neg := -1$
$L \rightarrow L_1\ B$	$L_1.pos := L.pos + 1$ $B.pos := L.pos$ $L.val := L_1.val + B.val$
$L \rightarrow B$	$B.pos := L.pos$ $L.val := B.val$
$B \rightarrow 0$ $B \rightarrow 1$	$B.val := 0$ $B.val := 2^{B.pos}$

Attribute grammars

Example:



Attribute grammars

- Semantic rules associated with a production $A \to \alpha$ are in the form $y = f(x_1, \ldots, x_n)$, where y and x_i are attributes associated with symbols in the production, and f is a function.
- There are two kinds of attributes:
  - synthesized attributes: y is an attribute associated with the non-terminal A;
  - inherited attributes: y is an attribute associated with some symbol in α.
- Synthesized attributes depend only attribute values of the subtrees.
- Inherited attributes may depend from the values of parent node and siblings.

Attribute grammars

Example:

Productions	Semantic rules
$N \rightarrow S\ L$	$L.pos := 0$ $N.val := S.neg * L.val$
$S \rightarrow +$	$S.neg := 1$
$S \rightarrow -$	$S.neg := -1$
$L \rightarrow L_1\ B$	$L_1.pos := L.pos + 1$ $B.pos := L.pos$ $L.val := L_1.val + B.val$
$L \rightarrow B$	$B.pos := L.pos$ $L.val := B.val$
$B \rightarrow 0$ $B \rightarrow 1$	$B.val := 0$ $B.val := 2^{B.pos}$

synthesized attributes

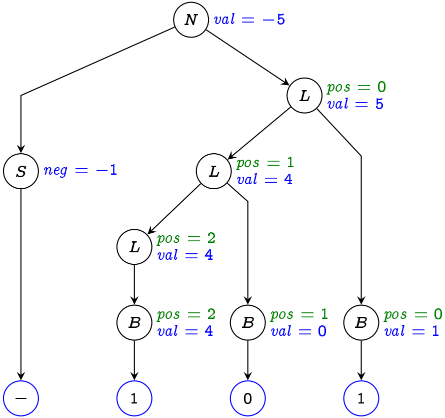
inherited attributes

Attribute grammars

- An attribute a depends from b if the evaluation of a requires the value of b.
- Dependencies between attributes define a dependency graph:
  - an directed graph, where edges show the dependencies between attributes;
  - describes the data flow during the attribute evaluation.
- Synthesized attributes have edges pointing upwards.
- Inherited attributes have edges pointing downwards and/or sidewise.

Attribute grammars

Example:



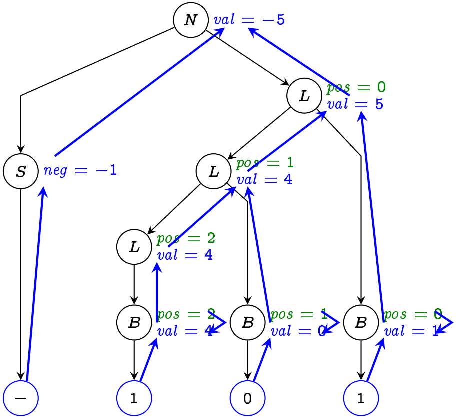
dependency graph

synthesized attributes

inherited attributes

Attribute grammars

Example:



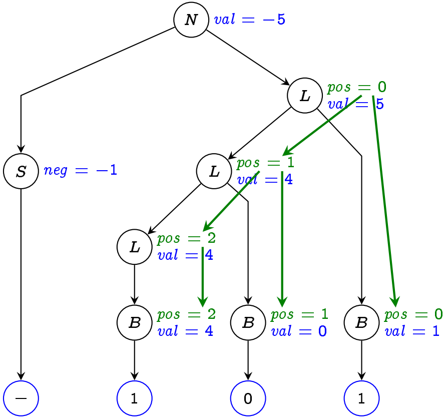
dependency graph

synthesized attributes

inherited attributes

Attribute grammars

Example:



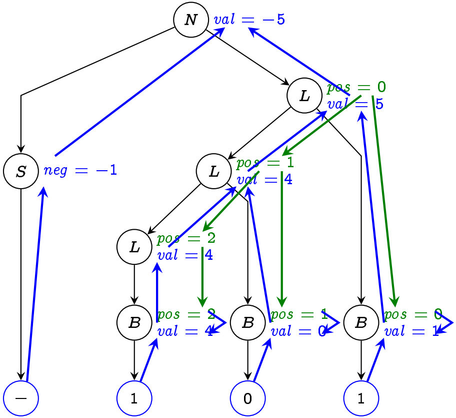
dependency graph

synthesized attributes

inherited attributes

Attribute grammars

Example:



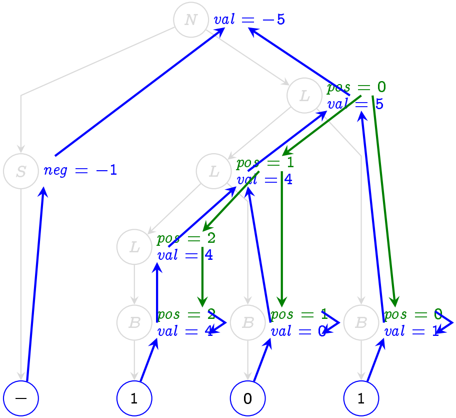
dependency graph

synthesized attributes

inherited attributes

Attribute grammars

Example:



dependency graph

synthesized attributes

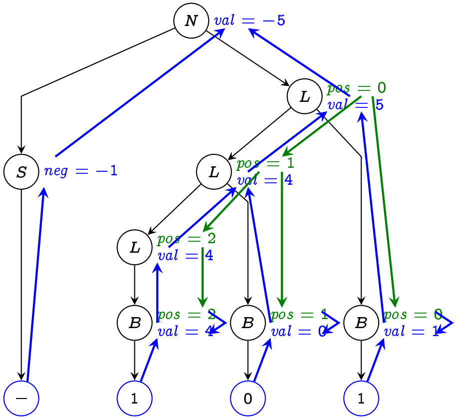
inherited attributes

Attribute grammars

- Topological sorting of a directed acyclic graph is a process of finding a linear ordering of its nodes, st., each node comes before all nodes to which it has outbound edges.
- Topological sorting of the dependency graph gives a valid evaluation ordering for attributes.
- NB! In the case of cyclic dependency graphs a valid ordering may not exist.

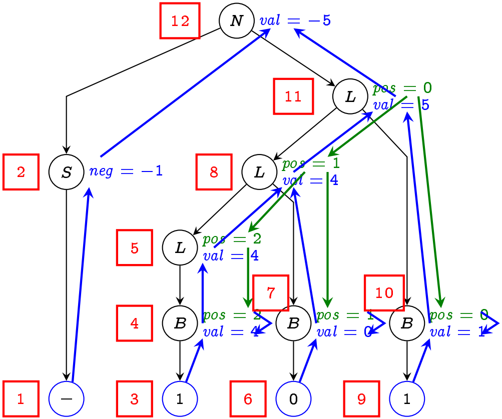
Attribute grammars

Example:



Attribute grammars

Example:



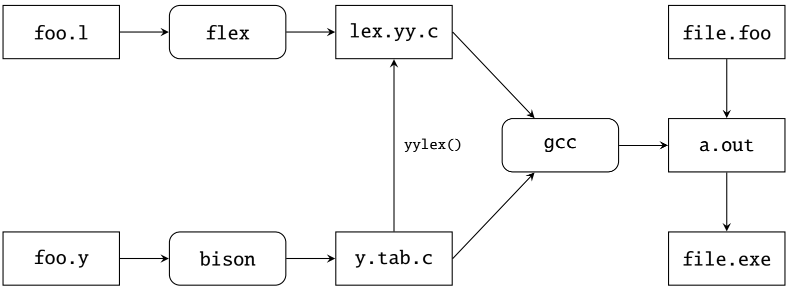
Attribute grammars

- S-attribute grammar is an AG where all attributes are synthesized.
- S-attribute grammars interact well with LR(k)-parsers since the evaluation of attributes is bottom-up.
- The values of attributes can be kept together with the associated symbol in the stack.
- Before reduction by production $A \rightarrow \alpha$, attributes corresponding to symbols of α are available in top of the stack.
- Hence, all the information for evaluating synthesized attributes of A are available, and these can be computed during reduction.

Attribute grammars

- L-attribute grammar is an AG where for all productions $A \to X_1 X_2 \ldots X_n$ inherited attributes of symbol X_i $(1 \leq i \leq n)$ depend only from inherited attributes of A and from attributes of symbols X_j $(j < i)$.
- NB! Each S-attribute grammar is also a L-attribute grammar.
- L-attribute grammars support the evaluation of attributes in depth-first left-to-right order.
- Interacts well with LL(k) parsers (both table driven and recursive decent).

Parser generator Bison



Parser generator Bison

Format of the input file:

- An input file of Bison has three parts:

```
definitions
%%
rules
%%
user code
```

- The same general structure as Flex has.

Parser generator Bison

```
%{
#include <stdio.h>
%}
%token INTEGER
%%
program:
    | program expr '\n'  { printf("%d\n", $2); }
    ;
expr:
    INTEGER              { $$ = $1; }
    | expr '+' expr      { $$ = $1 + $3; }
    | expr '-' expr      { $$ = $1 - $3; }
    ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

Parser generator Bison

```
%{
#include <stdio.h>
%}
%token INTEGER
%%
```

The definitions part consists of:

- C code surrounded by %{ and %}, which is copied verbatim into the generated file;
- Bison declarations:
  - %token list of terminal symbols (used in production rules, but also in the scanner);
  - %start declaration of the start symbol (if absent, the first non-terminal is the start symbol);
  - %union, %left, %right, ...

Parser generator Bison

```
program:
    | program expr '\n'  { printf("%d\n", $2); }
    ;
expr:
    INTEGER              { $$ = $1; }
    | expr '+' expr      { $$ = $1 + $3; }
    | expr '-' expr      { $$ = $1 - $3; }
    ;
```

- The second part consists of production rules.
- Must contain at least one rule.
- RHS of a rule is built of terminal and non-terminal symbols, and may also contain actions.
- Terminal symbols may be the ones declared before or singleton characters.

Parser generator Bison

```
program:
    | program expr '\n'  { printf("%d\n", $2); }
    ;
expr:
    INTEGER              { $$ = $1; }
    | expr '+' expr      { $$ = $1 + $3; }
    | expr '-' expr      { $$ = $1 - $3; }
    ;
```

- Actions are C code fragments surrounded by curly braces.
- They correspond to semantic rules of attribute grammars.
- Can refer to attribute values of grammar symbols appearing in the rule:
  - $$ corresponds to the value of LHS symbol;
  - $1 corresponds to the value of the first symbol in RHS;
  - ...

Parser generator Bison

```
program:
    | program expr '\n'  { printf("%d\n", $2); }
    ;
expr:
    INTEGER              { $$ = $1; }
    | expr '+' expr      { $$ = $1 + $3; }
    | expr '-' expr      { $$ = $1 - $3; }
    ;
```

- Actions are usually at the end of RHS, and they are executed when the rule is reduced.
- They may appear also in between the symbols, in which case it's equivalent of having an extra non-terminal in the place; this non-terminal has the action as RHS of its rule (and is otherwise empty).
- If an action is missing, then the default action is

$$\{\$\$ = \$1;\}$$

Parser generator Bison

```
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}
int main(void) {
    yyparse();
    return 0;
}
```

The third part of the specification is a C code which will be copied into the generated file verbatim.

- Most important functions:
  - main() calls yyparse();
  - yyerror() reports of syntax errors;
  - yylex() recognizes tokens (usually defined in a Flex generated scanner).
- The third part may be absent in what case the separator line may also be missing.

Parser generator Bison

- The function `yyparse()` uses the function `yylex()` to get the next token.
- The interface between scanner and parser is specified in Bison:
  - terminal symbols are declared with the command `%token`;
  - single character terminal symbols may be left undeclared;
  - the return value of `yylex()` is either a declared terminal symbol or a single character.
- Scanner should include the header file `"*.tab.h"`
  - can be generated automatically by Bison with the argument option -d.
- An alternative is to include the file of scanner `"lex.yy.c"` in the third part of the parser specification file.

Parser generator Bison

- Attribute values of non-terminals are in the variable `yylval`.
- Attributes are of type `YYSTYPE`, which by default is `int`.
- If attributes of different non-terminals have different types, one has to:
  - declare all types with the command `%union`

    ```
    %union {
        type1 name1;
        type2 name2;
        ...
    }
    ```

  - specify the types of terminal and non-terminal symbols

    ```
    %token <name> TOKEN
    %type <name> non-terminal
    ```

- An attribute value corresponding to a terminal symbol (and assigned by the scanner) should be a field of the union (`yylval.name`).

Parser generator Bison

- Shift-reduce actions are decided using one symbol lookahead.
- Conflicts are resolved by precedence and default rules:
  - commands `%left`, `%right` and `%nonassoc` are used to determine associativity and precedence of symbols;
  - the precedence is determined implicitly by textual ordering (later one have higher precedence);
  - the precedence of a rule is the same as the last nonterminal in RHS (can be changed using the command `%prec`);
  - shift/reduce conflict is solved by preferring shift action;
  - reduce/reduce conflict is solved by using textually the first production rule.