Semantic Analysis

4.1-4.3, 1.6.2

Syntax vs semantics

- Programming languages have two sides
  - Syntax: the form of a valid program
  - Semantics: the meaning of a program
    - From a designer's view
      - Semantics allows to enforce rules (e.g., type consistency)
      - Semantics provides information for generating equivalent programs (say in lower level of languages) to the original ones

Syntax

- Usually the syntax of a language might not be fully described by context free grammar
  - No context free grammar can define the syntactical requirement that "the call to a subroutine must have the same number of parameters as the definition of the subroutine."
  - Some syntactical requirement like "every function must have one return" is too complex to be expressed as context free grammar.
- However, these can be checked easily in semantic analysis

Semantic analysis

- Static analysis (done at the compile time)
  - Enforces rules (semantic rules) that can not be captured by context free grammars, e.g.,
    - Every identifier has to be declared before use
    - Subroutine calls use the correct number of parameters
- Dynamic analysis (done at the runtime by the code produced by compiler)
    - Array subscript should be within the bounds of the array
    - Variables are never used unless they have been given a value

- Intermediate code generation
  - From a program in language X, generate a program in intermediate language Y

More on semantic rules

- Languages vary on their semantic rules
  - C allows operands of many types to appear in an expression while Ada does not
  - C requires no run time (dynamic) check while Java check as many as possible

Dynamic check/analysis

- Example

  Int a[100];

  For (i=1; i < n; i++) a[2*n]=a[2*n-1];

  – No run time check in C

  – Run time check in Java

    - Add in run time check code:

- Java allows programmer provide run time check!
  - E.g., assert denominator !=0;
- Some languages (e.g., Euclid and Eiffel) offer constructs to express properties of *invariants, preconditions, postconditions.* Code will be produced to check these properties at run time.

Static analysis

- Compile-time algorithms that predict run time behavior are known as *static analysis.*
- A static analysis is *precise* if it allows the compiler to determine whether a given program will always follow the (semantic) rules.

Annotated parse tree

- Both semantic analysis and intermediate code generation can be based on *annotation,* or *decoration* of a parse tree. Annotations are known as *attributes.*
- *Attribute grammars* provide a formal framework for the decoration of a parse tree.

Attribute grammars

- Attributes
  - Each symbol has one or more attributes
  - The value(s) of the attributes of the start symbol *S* is the meaning of the token string derived from *S.*

- Rules
  - Productions in CFG are extended with attributes and the expression of their relations.
  - Define how the attributes of the symbols in a production are related. They are of the form

    X.att := *semanticfunction(X1.attr,…, Xn.attr)*
  - When more than one symbols are used in the same production, we distinguish them by adding subscripts

Example

- An attribute grammar for arithmetic expressions

```
E → E + T   E1.val = E2.val + T.val
E → E – T   E1.val = E2.val - T.val
E → T       E.val  = T.val
T → T * F   T1.val = T2.val * F.val
T → T / F   T1.val = T2.val / F.val
T → F       T.val  = F.val
F → - F     F1.val = - F2.val
F → (E)     F.val  = E.val
F → const   F.val  = C.val
```

Parse tree and attributes evaluation

- Given an attribute grammar, and a user input program in this grammar, we are
  - not only able to construct a parse tree of the program,
  - but also able to label (find) the value (semantics or meaning) of the input program, i.e., evaluating attributes
    - The process of evaluating the attributes is called *annotation* or *decoration* of the parse tree.

Example: simple

- Example with first three rules in the previous slide and rule

  `T→const T.val = const.val`

  – Parse tree of 5 + 5 – 5

  – Attributes evaluation: value of each component of the expression 5+5-5 in the parse tree

Example: full

- Consider the full attribute grammar in the earlier slide, and expression (1+3)*2
  – What’s its parse tree?
  – What are the values of the attributes of the symbols in the parse tree?

Types of attributes

- Synthesized attributes
  - An attribute is *synthesized* if its value is calculated **only** for productions where the corresponding symbol is on the left side of the production
  - An AG is *S-attributed* if all attributes are synthesized.
  - Example: the previous example
- Inherited attributes

Inherited attributes

- An attribute is *inherited* if its values is calculated for some production where the corresponding symbol is on the **right** side of the production.

Inherited attributes – example

- Given a CFG grammar

```
<expr> -> const <expr_tail>
<expr_tail> -> - const <expr_tail> |  ε
```

  - write an attribute grammar such that we can know the value of an expression
    - What attributes we need?
      - <expr>: we want to know the value of the whole expression, so we introduce attribute val for <expr>.
      - <expr_tail> (decomposition approach can be used here!)
        - » st: the value of the expression before <expr_tail>
        - » value: the value of the whole expression.
    - How the attributes are related

Example (continued)

- Attribute grammar

  <expr> -> const <expr_tail>

  <expr_tail>.st := const.val

  <expr>.val := <expr_tail>.val

  <expr_tail> -> - const <expr_tail>

  <expr_tail>2.st := <expr_tail>1.st - const.val

  <expr_tail>1.val := <expr_tail>2.val

  <expr_tail> -> ε

  <expr_tail>.val := <expr_tail>.st

- Decoration of parse tree – parse tree of an expression and the evaluation of the attributes of symbols.

Algorithms to decorate a parse tree

- Problem: given an attribute grammar and a program, we will decorate a parse tree (i.e., produce a parse tree and evaluate the attributes of the symbols in the tree)
  - Oblivious algo (skip)
  - Dynamic algo (skip)
  - Static algo:

Static algorithm

- An AG is an *S-attributed grammar* if all its attributes are synthesized
- For S-attributed grammar, apply postorder tree traversal algo to the parse tree
- *L-attributed grammar* (see definition in the end of 3rd paragraph of P186)
- For L-attributed grammar, left to right depth first tree traverse algorithm is sufficient

Summary

- Context free grammar specifies a language
- Attribute grammar (AG)
  - Associate each symbol with some *attributes*
  - AG consists of productions (on symbols) and rules (on attributes)

- AG for semantic analysis
  - Produce a parse tree using productions of AG
  - Using rules to evaluate the attributes of symbols on the tree
  - Using the values of the attributes to check whether semantic rules are satisfied.
- AG for generating intermediate code
  - The value of the attribute *syntax tree* of the start symbol can be a base of intermediate code.

Appendix

More on attribute grammars

- Build a syntax tree from an input program
  - CFG
    - E -> E + T | E – T | T
      (i.e., three rules: E-> E+T E-> E-T T-> T)
    - T-> T * F | F
    - F -> (E) | const
  - *Syntax tree*: example of (1+3)*2
  - For E, we would like to have an attribute whose value is a syntax tree of E. In fact, for every symbol F, we introduce such an attribute whose value is a syntax tree of F.

– Attribute grammar
  - E -> E + T Δ E1.synt := binTree('+', E2.synt, T.synt)
  - E -> E – T Δ E1.synt := binTree('-', E2.synt, T.synt)
  - E -> T Δ E.synt := T.synt
  - T -> T * F Δ T1.synt := binTree('*', T2.synt, F.synt)
  - T -> F Δ T1.synt := F.synt
  - F -> (E) Δ F.synt := E.synt
  - F -> const Δ F.synt := leafnode(const.val)

– Example of (1+3)*2