Semantic Analysis

Overview

- What is semantic analysis?
- Dynamic and static checks
- Attribute grammars
- Synthesized and inherited grammars
- Explicit AST construction

Introduction

- Syntax: refers to form, structure
- Semantics: meaning
- Relevance:
  - Allows to enforce rules
  - Provides information to produce equivalent programs
- Why we need it?
  - Example: args → args , one_arg | one_arg
  - Rule provides structure of list
  - Cannot determine length of list by rule alone
  - Function definition and calling requires specific number of parameters

Introduction

- Rule of thumb:
  - Anything that needs counting
  - Accumulating
  - Nestedness
  - Putting together things that appear separated in time/space
- Semantic rules divided between:
  - Static rules: add code for checking, array bounds checking
  - Dynamic rules: a division by zero, accessing valid array positions
- Line between one and the other could be fuzzy (per language, per implementation)

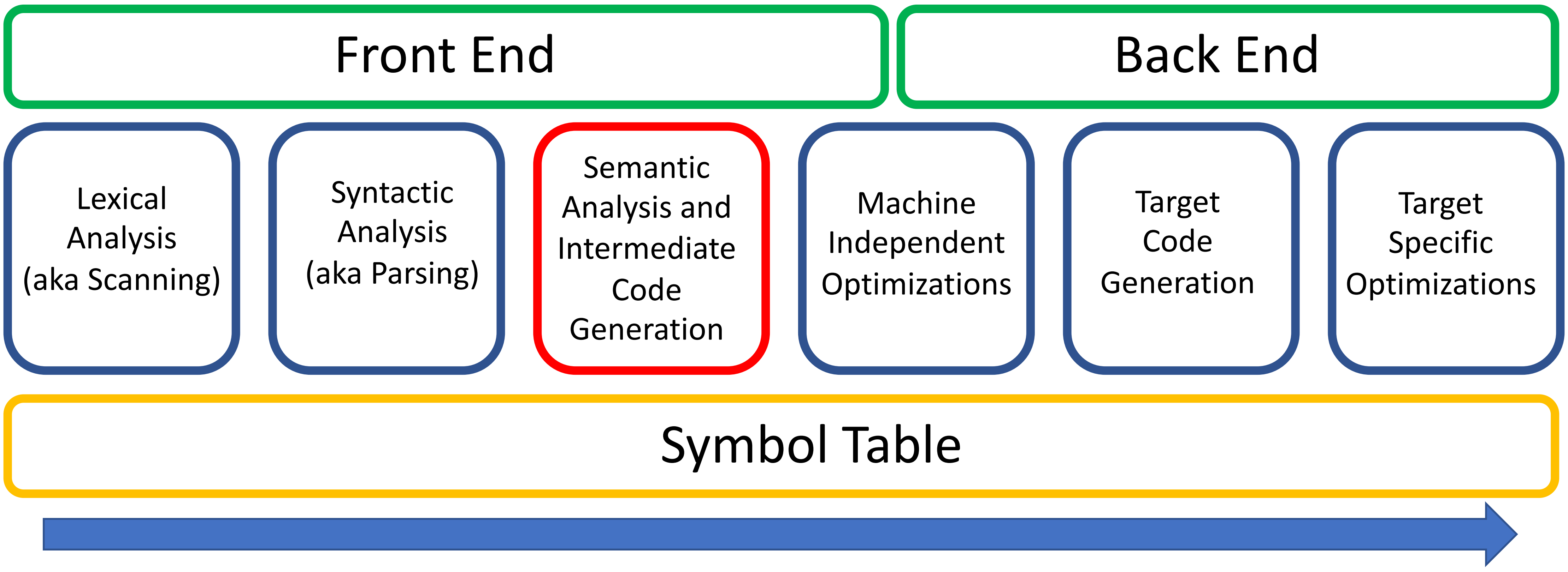
Introduction

- Semantic analysis and code generation can be described in terms of *annotations* / *decorations* of the parse / syntax tree
- Ability to:
  - Gather information during parsing
  - Structure information (as in a data structure)
  - Associate information (the data structure) to specific part of the parse tree and other compiler parts
- Bits of information: *attributes*
- Here we cover *attribute grammars* and mechanisms to enforce static rules
- Heavily rely on the structure of the CFG
- Use grammar to pass information along

Role of Semantic Analyzer

- Semantics vary a lot with language
- Lisp dialects: mixed-mode arithmetic, automatic promotion from integer to rational to floating point or “bignum”
- Ada: assigns specific type to numeric variable
- C: no checks at all ☺
- Java: checks a lot (out of bounds, integer overflow, proper variable initialization)
- Semantic Analyzer and Intermediate Code Generator mark the end of the front-end of a typical compiler

Compilation Overview



Input (*.c file)

Output
(for instance, a binary file)

Dynamic Checks

- The compiler will generate code to perform some checks
- Checks vary a lot per language
- Languages may also provide mechanisms for users to implement their own checks, which then can be disabled prior to production:
  - C/C++ assertions: assert (x > 0 && "This should not happen");
- Need for dynamic checks could also vary a lot across languages:
  - 3 + "four" → Concatenate or Add?
  - 3 + "4.5" → Concatenate or Add? If adding, as integer or float?
- Invariants: Logic conditions that should be met at runtime (e.g. a loop condition such as i < N)

Static Analysis

- Compile-time algorithms that predict run-time behavior
- Example: pointer analysis attempts to find when are pointers safe, being used, not being used, initialized, etc
- Analysis is precise if it allows the compiler to determine when something follows the rule or not; eq. imprecise if there's a chance that the algorithm may fail
- Some languages (e.g. Ada, ML): will enforce variables are always appropriately used according to their type
- Lisp, Python, Ruby: type-safe, added dynamic overhead for various checks
- Example: *definite assignments* in Java and C# force variables to be initialized

Static Analysis

- *Escape analysis*: references limited to context? If so, can be allocated in the stack
- Other:
  - *out of order optimizations*: is it safe/correct to re-arrange the order of computations?
  - *thread safety*: can some instructions be executed in parallel?
- *Unsafe optimizations*: may lead to incorrect results
- *Speculative*: sort of no guarantees of the result when doing something; can also refers to something that can be undone (e.g. data prefetching)
- *Conservative*: analysis or optimization guaranteeing minimum requirement of results in terms of safety and/on effectiveness
- *Optimistic:* similar to conservative but with a "maybe" flavor

Attribute Grammars

1. E → E + T
2. E → E – T
3. E → T
4. T → T * F
5. T → T / F
6. T → F
7. F → - F
8. F → (E)
9. F → const

- For every symbol S, we have some set of attributes: S.attr1, S.attr2 …
- Attributes can vary depending on the symbol type
- Not all symbols need to have the same set of attributes
- Examples:
  - For a list we could associate a C++ stdd::vector<sometype>
  - For some arithmetic expressions we could have int or float field types

Attribute Grammars

1. $E_1 \rightarrow E_2 + T$
2. $E_1 \rightarrow E_2 - T$
3. $E \rightarrow T$
4. $T_1 \rightarrow T_2 * F$
5. $T_1 \rightarrow T_2 / F$
6. $T \rightarrow F$
7. $F_1 \rightarrow - F_2$
8. $F \rightarrow (E)$
9. $F \rightarrow const$

1. $val(E_1) := val(E_2) + val(T)$
2. $val(E_1) := val(E_2) - val(T)$
3. $val(E) := val(T)$
4. $val(T_1) := val(T_2) * val(F)$
5. $val(T_1) := val(T_2) / val(F)$
6. $val(T) := val(F)$
7. $val(F_1) := - val(F_2)$
8. $val(F) := val(E)$
9. $val(F) := val(const)$

a. Copy rules
b. Semantic functions

Attribute Grammars

- Usual type of rules:
  - Copy rules
  - Semantic rules
  - Small fragments of code to be executed in specific parts of the parsing process → semantic actions

1. L → **id** LT
2. LT → , L
3. LT → ε

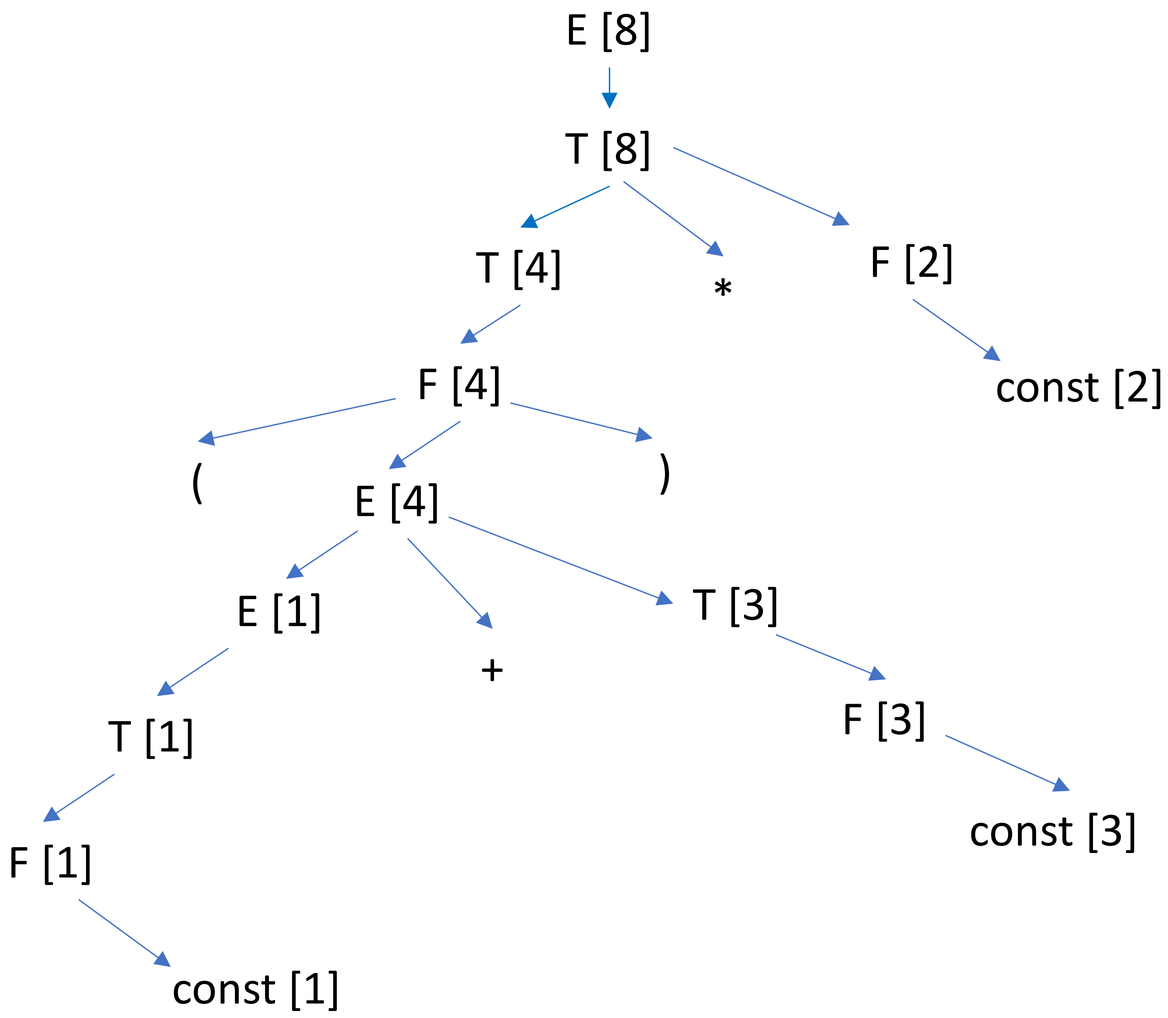
1. c(L) := 1 + c(LT)
2. c(LT) := c(L)
3. c(LT) := 0

Attribute Grammars

Attributes of tree nodes might include

- for an identifier: a reference to its symbol table entry (maybe)
- for an expression, it’s type
- for a statement or expression, a reference to the code in the compiler’s intermediate form
- for practically all constructs, information relating to filename, line, column, source code position
- internal code, list of semantic errors in the subtree below

Evaluating Attributes



Information flow with attribute grammar for string: (1 + 3) * 2

Attributes

- They come in two flavors:
  - Synthesized: compute somewhere in the tree, and push up to parent
  - Inherited: some node computes it, and then passes it to its children or younger sibling

Synthesized Attributes

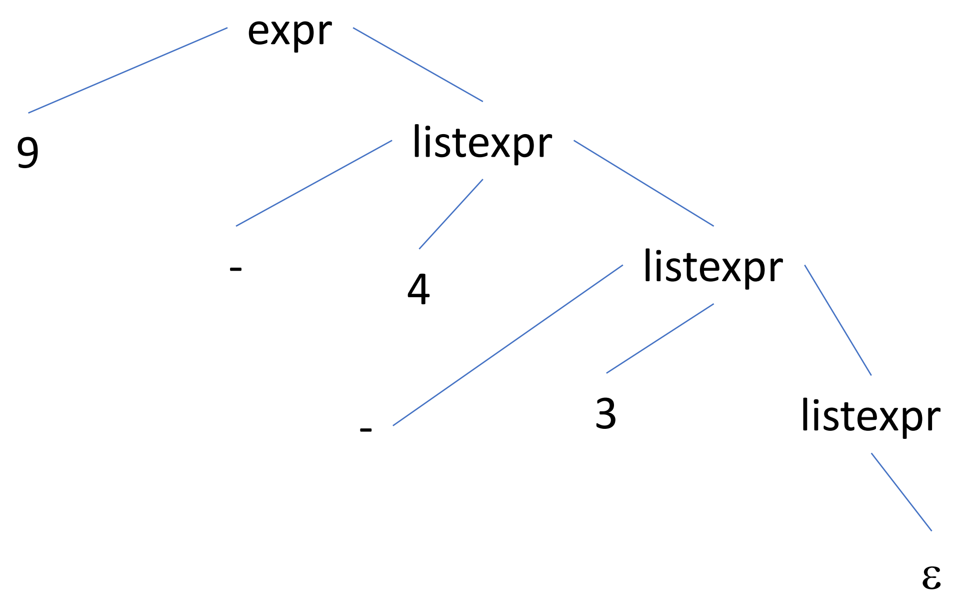
- Values are calculated along the way
- Only in productions when the symbol appears on the left-hand side
- Attribute flow: information boiling up, from leaves to root
- Attribute grammars where all are attributes are synthesized are S-attributed
- In S-attributed grammars:
  - Arguments to semantic functions can only be from the right-hand side (We’ll come back to this)
  - Result always goes to attribute on left-hand side symbol
- Attributes of tokens can only be initialized by information coming from scanner

Inherited Attributes

- *<u>Inherited attributes</u>* are attributes whose values are calculated when their symbol appears on the right-hand side of the current production
- Key difference: Information flows from parent to child or among siblings; synthesized attributes flow from from children to parents
- In some compilers, symbol table information passed via inherited attributes

LL(1) grammar:

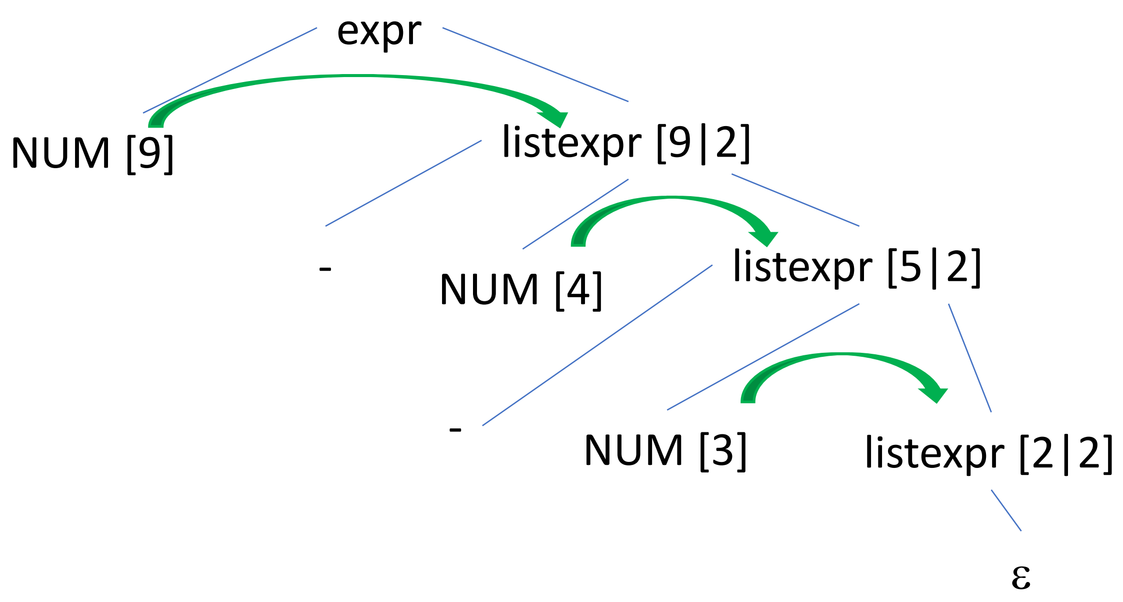
- expr → **NUM** listexpr
- listexpr → - **NUM** listexpr | ε



Inherited Attributes

- An LL(1) grammar (top-down, left-left) is natural for inherited attributes
- Synthesized attributes do not match the order in which the tree is discovered
- To support an S-attribute grammar, we would need the ability to store an explicit representation of listexpr (a right-hand subtree) → defeats the purpose

LL(1) grammar:
- expr → NUM listexpr
- listexpr → - NUM listexpr | ε



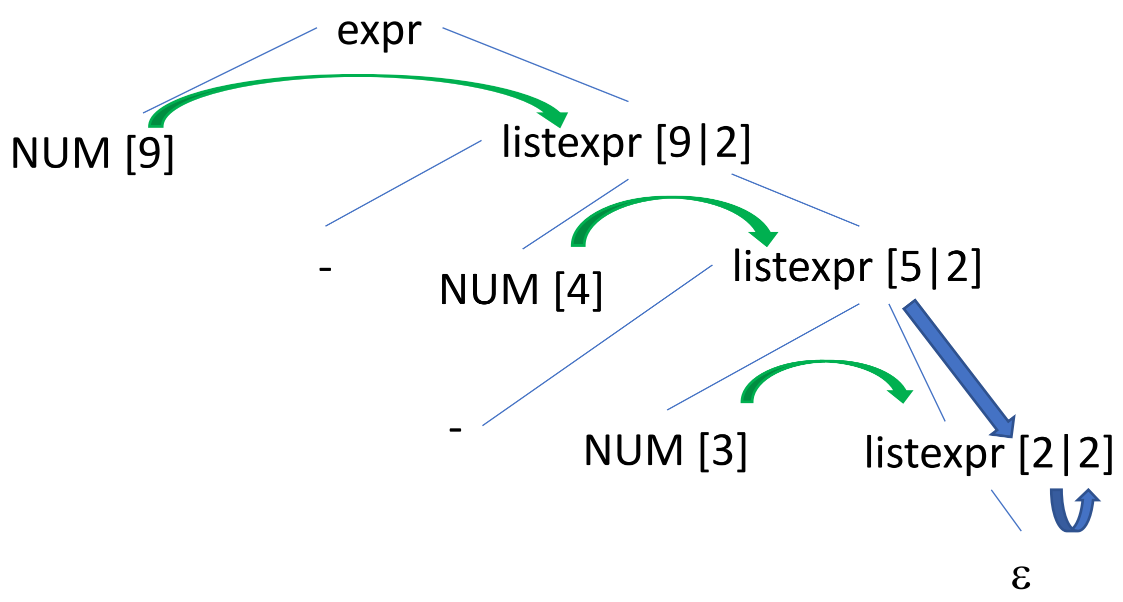
Inherited Attributes

Attribute rules:

- expr → NUM listexpr
  - ➢ subtot(listexpr) := val(NUM)
  - ➢ val(expr) := val(listexpr)
- listexpr_1 → - NUM listexpr_2
  - ➢ $\text{subtot}(\text{listexpr}_2) := \text{subtot}(\text{listexpr}_1) - \text{val}(\text{NUM})$
  - ➢ $\text{val}(\text{listexpr}_1) := \text{val}(\text{listexpr}_2)$
- listexpr → ε
  - ➢ val(listexpr) := subtot(listexpr)

LL(1) grammar:

- expr → NUM listexpr
- listexpr → - NUM listexpr | ε



Inherited Attributes

LL(1) grammar with inherit attributes:

Main source of problem: left associative operators

Left and right operands appear in separate productions

```
1. E → T TT
•  st(TT) := val(TT)
•  val(E) := val(TT)
2. TT1 → + T TT2
•  st(TT2) := st(TT1) + val(T)
•  val(TT1) := val (TT2)
3. TT1→ - T TT2
•  st(TT2) := st(TT1) – val(T)
•  val(TT1) := val(TT2)
4. TT → ε
•  val(TT) := st(TT)
5. T → F FT
•  st(FT) := val(F)
•  val(T) := val(FT)
6. FT1 → * F FT2
•  st(FT2) := st(FT1) * val(F)
•  val(FT1) := val(FT2)
```

```
7. FT1 → / F FT2
•  st(FT2) := st(FT1) /
   val(F)
•  val(FT1) := val(FT2)
8 FT → ε
•  val(FT) := st(FT)
9. F1 → - F2
•  val(F1) :=  - val(F2)
10. F → ( E )
•  val(F) := val(E)
11. F → const
•  val(F) := val(const)
```

Attribute Flow

- Attribute grammars do not specify the order in which attribute rules are or should be invoked
- Annotations are declarative: define set of trees, but not how to annotate them
- S-attribute grammars:
  - Strictly bottom-up
  - Evaluation of attributes follows visiting order of parse tree (of an LR-parser)
  - Attributes can be evaluated on the fly, interleaving semantic and syntactic analysis (and lexical analysis)

Attribute Flow

L-attribute (Left—ttributed) grammars (formally):

a. Synthesized attributes of left-hand side (lhs) symbols depend only on: i) symbol's own inherited attributes or ii) on attributes (synthesized or inherited) of symbols on the right-hand side (rhs) of the production

b. Inherited attributes of rhs symbols depend only on inherited attributes of the lhs or on attributes (any kind) of symbols to its left in the rhs of the production

```
syn(sym_lhs)  :=  f_inh(sym_lhs)
```

~~syn(sym_lhs) := fsyn(sym_lhs)~~

$$\texttt{syn}(\texttt{sym}_{\texttt{lhs}}) := \texttt{f}(\{\texttt{inh}(\texttt{rhs}_1,\texttt{rhs}_2,\ldots\texttt{rhs}_M),\ \texttt{syn}(\texttt{inh}(\texttt{rhs}_1,\texttt{rhs}_2,\ldots\texttt{rhs}_M)\})$$

$$\texttt{inh}(\texttt{rhs}_i) := \texttt{f}(\{\texttt{inh}(\texttt{sym}_{\texttt{lhs}}),\ \texttt{syn}(\texttt{rhs}_{j<i}),\texttt{inh}(\texttt{rhs}_{j<i})\})$$

- Every S-attribute grammar is also an L-attribute grammar
- Compiler that interleaves semantic analysis and code generation is a one-pass compiler (not too common nowadays)
- Avoids the need for explicit construction and representation of the parse tree

Explicit Syntax Tree Construction

- Often, we want to construct an explicit representati on of the syntax tree
- Useful in multi-pass compilers
- Example of bottom-up parsing →

1. $E_1 \rightarrow E_2 + T$
2. $E_1 \rightarrow E_2 - T$
3. $E \rightarrow T$
4. $T_1 \rightarrow T_2 * F$
5. $T_1 \rightarrow T_2 / F$
6. $T \rightarrow F$
7. $F_1 \rightarrow - F_2$
8. $F \rightarrow (E)$
9. $F \rightarrow$ const

⇨

1. ptr(E_1) := make_bin_op(“+”,ptr(E_2), ptr(T))
2. ptr(E_1) := make_bin_op(“-”,ptr(E_2), ptr(T))
3. ptr(E) := ptr(T)
4. ptr(T_1) := make_bin_op(“*”,ptr(T_2), ptr(F))
5. ptr(T_1) := make_bin_op(“/”,ptr(T_2), ptr(F))
6. ptr(T) := ptr(F)
7. ptr(F_1) := make_un_op(ptr(F_2))
8. ptr(F) := ptr(E)
9. ptr(F) := make_leaf(val(const))

Inherited Attributes

```
1. E → T TT
•  st(TT) := ptr(T)
•  ptr(E) := ptr(TT)
2. TT₁ → + T TT₂
•  st(TT₂) := make_bin_op("+", st(TT₁), ptr(T))
•  ptr(TT₁) := ptr (TT₂)
3. TT₁→ - T TT₂
•  st(TT₂) := make_bin_op("-", st(TT₁), ptr(T))
•  ptr(TT₁) := ptr (TT₂)
4. TT → ε
•  val(TT) := st(TT)
5. T → F FT
•  st(FT) := val(F)
•  val(T) := val(FT)
6. FT₁ → * F FT₂
•  st(FT₂) := make_bin_op("*", st(FT₁), ptr(F))
•  ptr(FT₁) := ptr (FT₂)
```

```
7. FT₁ → / F FT₂
•  st(FT₂) := make_bin_op("/",
   st(FT₁), ptr(F))
•  ptr(FT₁) := ptr (FT₂)
8 FT → ε
•  ptr(FT) := st(FT)
9. F₁ → - F₂
•  ptr(F₁) := make_un_op("-
   ",ptr(F₂))
10. F → ( E )
•  ptr(F) := ptr(E)
11. F → const
•  ptr(F) := make_leaf(val(const))
```

Action Routines

- Commonly used in parsing-driven translation
- An action is a semantic function invoked during specific points of the parsing process
- Compiler designer decided what to call when, for instance, adding new variables to the symbol table
- We will normally have several semantic actions in each production
- During the compiler design and implementation process, we might notice that we want to change the structure of the productions / grammar