Semantic Analysis

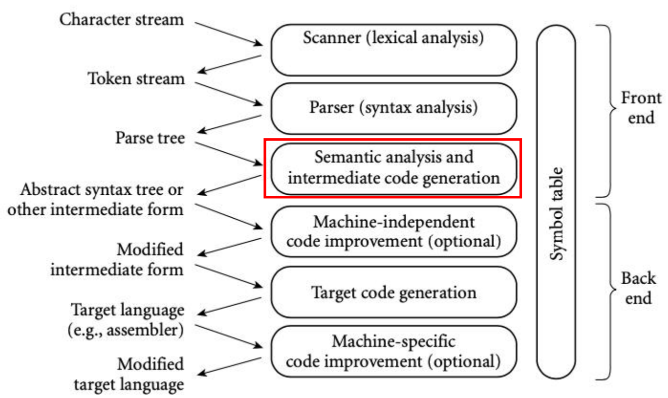
204315: OPL

Scott, M. L. (2016). Programming Language Pragmatics.
Tutorialspoint, https://www.tutorialspoint.com/, accessed on 11/22/2023

Outline

- Semantic Analysis
- Role of Semantic Analysis
- Parse tree decoration
- Attribute Grammar

Compilation Process Overview (Recap)



Scott, M. L. (2016). *Programming Language Pragmatics*.

Why we need Semantic Analysis?

- The plain **parse-tree** constructed in syntax analysis phase **does not carry any information of how to evaluate the tree**.
- The **productions of context-free grammar**, which makes the rules of the language, **do not accommodate how to interpret them**.
- **Semantics**
  - provide meaning to a language constructs, like tokens and syntax structure.
  - Semantics help interpret symbols, their types, and their relations with each other.
  - **Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not**.

Syntactic vs Semantic Errors

```
int a = “value”;
```

- The above example should not issue an error in lexical and syntax analysis phase, as **it is lexically and structurally correct**,
- **but it should generate a semantic error** as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Semantic Analysis Overview

- Following parsing, the next two phases of the "typical" compiler are
  - semantic analysis
  - (intermediate) code generation
- The principal job of the semantic analyzer is to enforce static semantic rules
  - constructs a decorated syntax tree
  - information gathered is needed by the code generator
- Semantic analysis, and intermediate code generation are interleaved.

Role of Semantic Analysis

- To recognize Semantic Errors
  - Type mismatch
  - Undeclared variable
  - Reserved identifier misuse.
  - Multiple declaration of variable in a scope.
  - Accessing an out of scope variable.
  - Actual and formal parameter mismatch.
- [Class participation] Let’s give some examples of the above errors

Decorating a parse tree

- Both semantic analysis and (intermediate) code generation can be described in terms of annotation, or "decoration" of a parse tree
- ATTRIBUTE GRAMMARS provide a formal framework for decorating such a tree

Attribute Grammar

- a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information
- each attribute has well-defined domain of values, such as integer, float, character, string, and expressions
- a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language
- Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

Context-Free Grammars

- LR (bottom-up) grammar for arithmetic expressions made of constants, with precedence and associativity:

```
E → E + T
E → E − T
E → T
T → T * F
T → T / F
T → F
F → - F
F → (E)
F → const
```

- This says nothing about what the program MEANS

CFGs with Attribute Grammar

- CFGs can be turned into an attribute grammar as follows:

```
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
```


Understanding Attribute Grammar

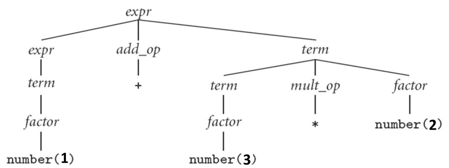
E → E + T { E1.value = E2.value + T.value }

- **Interpretation**: the attributes (values) of non-terminals E and T are added together and the result is copied to the attribute of non-terminal E (on the left-hand-side).
- Attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment.
- Attributes can be broadly divided into two groups:
  - **Synthesized attributes** are the result of the attribute evaluation rules.
  - **Inherited attributes** are passed down from parent nodes.

Synthesized attributes

- The process of evaluating attributes is called annotation, or DECORATION, of the parse tree.
- **Example:** decoration of a parse tree for **(1 + 3) * 2**

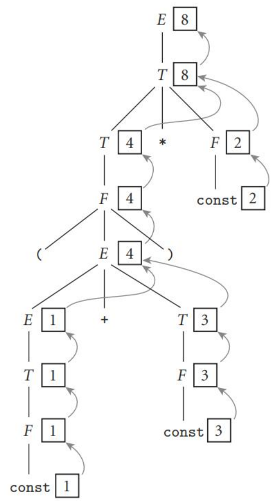
E → E + T	E1.val = E2.val + T.val
E → E − T	E1.val = E2.val - T.val
E → T	E.val = T.val
T → T * F	T1.val = T2.val * F.val
T → T / F	T1.val = T2.val / F.val
T → F	T.val = F.val
F → - F	F1.val = - F2.val
F → (E)	F.val = E.val
F → const	F.val = C.val



Parse Tree: (1+3) * 2

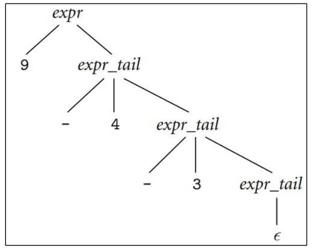
CFGs: LR Parsing

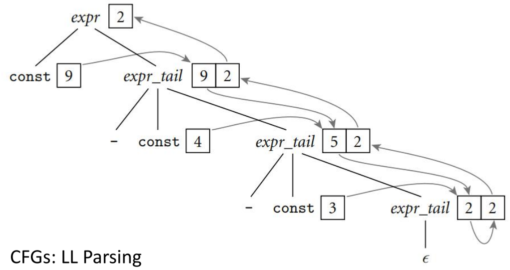
Synthesized attributes



Inherited Attributes

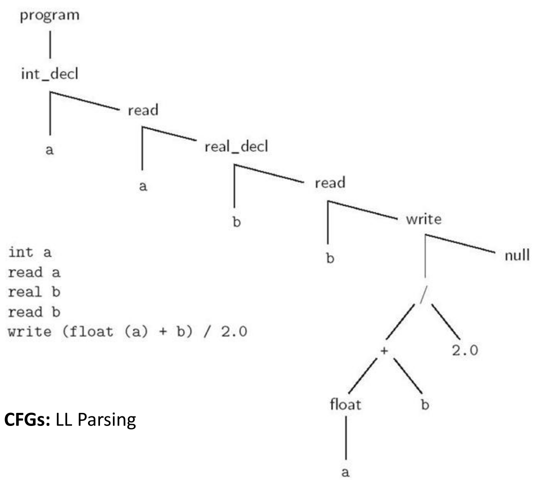
- CFG and parse tree for subtraction
  - **expr → const expr tail**
  - **expr tail → - const expr tail | ε**
- If we want to create an attribute grammar with synthesized attributes, we have a problem: because subtraction is left associative, we cannot summarize the right subtree of the root with a single numeric value.
- Parse Tree Example: **9 - 4 - 3**





CFGs: LL Parsing

Example of a syntax tree




```
int a
read a
real b
read b
write (float (a) + b) / 2.0
```

CFGs: LL Parsing

Syntax tree for a simple calculator program

Example: Attribute grammar for type checking

- A .t attribute encodes data type. This grammar allows for both declaration and type inference.

dcl	
int dlist	dlist.t = **int**
real dlist	dlist.t = **real**
dlist	
ident	ident.t := dlist.t
dlist , ident	$dlist_1$.t := $dlist_0$.t
	ident.t := dlist.t
assign	
var := expr	var.t = expr.t
var	
ident	var.t = ident.t
expr	
const	expr.t = const.t
sum	expr.t = sum.t
sum	
var	sum.t = var.t
sum + var	sum_0.t = sum_1.t = var.t
const	
num	const.t = **int**
num . num	const.t = **real**