Chapter 6

Semantic Analysis & Intermediate Code Generation

Semantic Analysis & Intermediate Code Generation

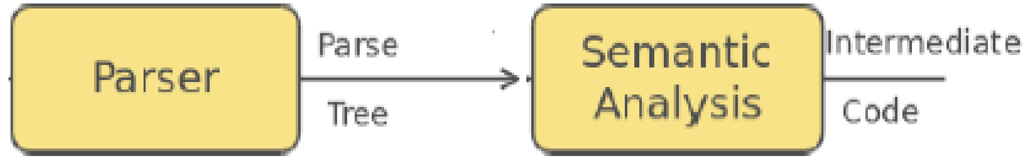
- Semantic Analysis
- Syntax-Directed Definition
- Syntax-Directed Translation
- Translation Schemes
- Intermediate Code Generation
- Generating Abstract Stack Machine Code
- Intermediate Code Generation:
  Three-address Code

Semantic Analysis

- The semantics of the language defines what its programs mean, what each program does when it executes.

- Semantic Analyzer adds semantic information to the parse tree (syntax directed translation), checks the source program for semantic errors and collects information for the code generation.

- The parser constructs parse trees in the syntax analysis phase. The parse tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree.



Semantic Analysis

- The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.
- For example, the production E → E + T, has no semantic rule associated with it, and it cannot help in making any sense of the production.
- Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
- For example, int a = “value”; 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.
- Meaning of statements (semantic) can be achieved by **Syntax-Directed Definition and Translation Schemes.**

Syntax-Directed Definition(1)

- Each Production Has a Set of Semantic Rules
- Each Grammar Symbol Has a Set of Attributes
- For the Following Example, String Attribute "*t*" is Associated With Each Grammar Symbol

expr **→** ***expr*** **–** ***term*** **|** ***expr*** **+** ***term*** **|** ***term***
term **→ 0 | 1 | 2 | 3 | … | 9**

- Semantic Rules for *expr* define *t* as a "synthesized attribute" i.e., the various copies of *t* obtain their values from "children *t*'s"

For example, E → E + T { E.value = E.value + T.value }

Syntax-Directed Definition (2)

- Each Production Rule of the CFG Has a Semantic Rule

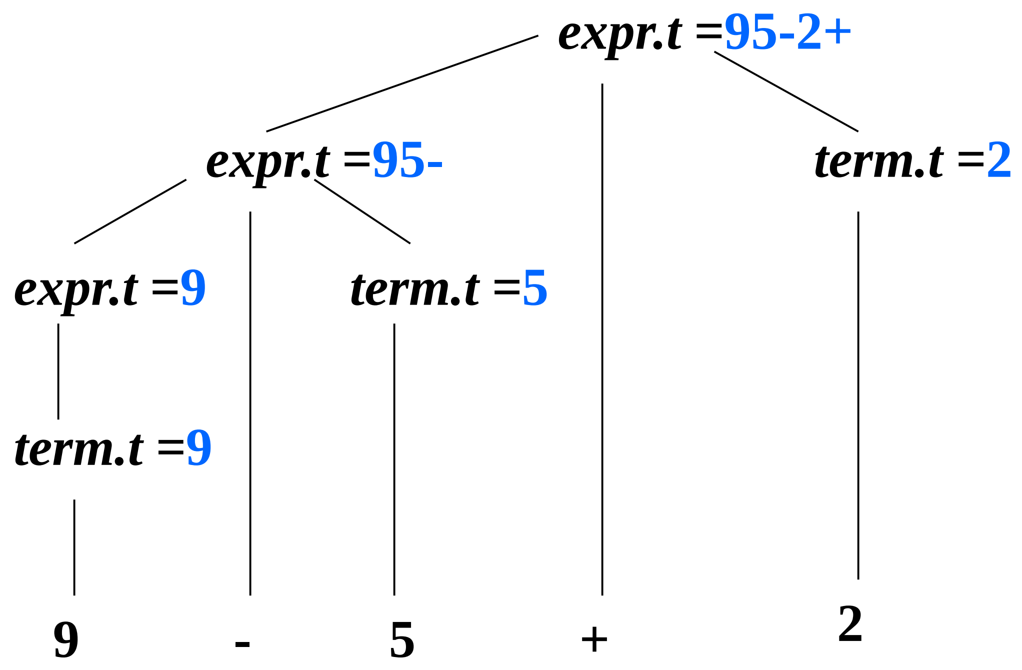
Production	Semantic Rule
expr → *expr* + *term*	*expr.t* := *expr.t* || *term.t* || '+'
expr → *expr* – *term*	*expr.t* := *expr.t* || *term.t* || '-'
expr → *term*	*expr.t* := *term.t*
term → 0	*term.t* := '0'
term → 1	*term.t* := '1'
….	….
term → 9	*term.t* := '9'

Semantic rules for postfix notation

- Semantic rules are then embedded in the parse tree for the process of translation.
- A parse tree showing all the attribute values at each node is called annotated parse tree.

Syntax-Directed Translation

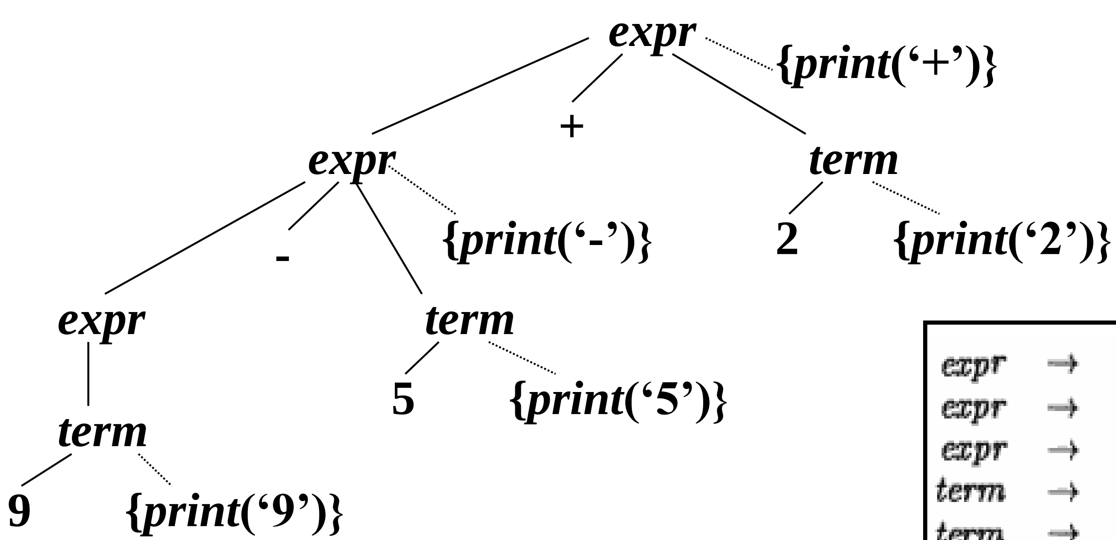
- The translation process starts at the root and recursively visits the children of each node in left-to-right order.
- The semantic rules at a given node are evaluated once all descendants of that node have been visited.



Translation of 9 – 5 + 2 to 95-2+

Translation Schemes

- Translation scheme contains embedded Semantic Actions into the right sides of the productions.
- A translation scheme is like a syntax-directed definition except the order of evaluation of the semantic rules is explicitly shown.



expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ - *term*	{print('-')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		...	
term	→	9	{print('9')}

Intermediate Code Generation

- Intermediate code is an abstract (machine independent) code.
- It is generated from annotated parse tree or abstract syntax tree, AST.
- It is very useful because of its simplicity and portability; since it is machine independent and enables common optimizations.
- It has many forms such as:
  - Stack machine code.
  - Three address code.

Generating Abstract Stack Machine Code

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.

One popular form of intermediate representation is code for an ***abstract stack machine***.

I will show you how code will be generated for it.

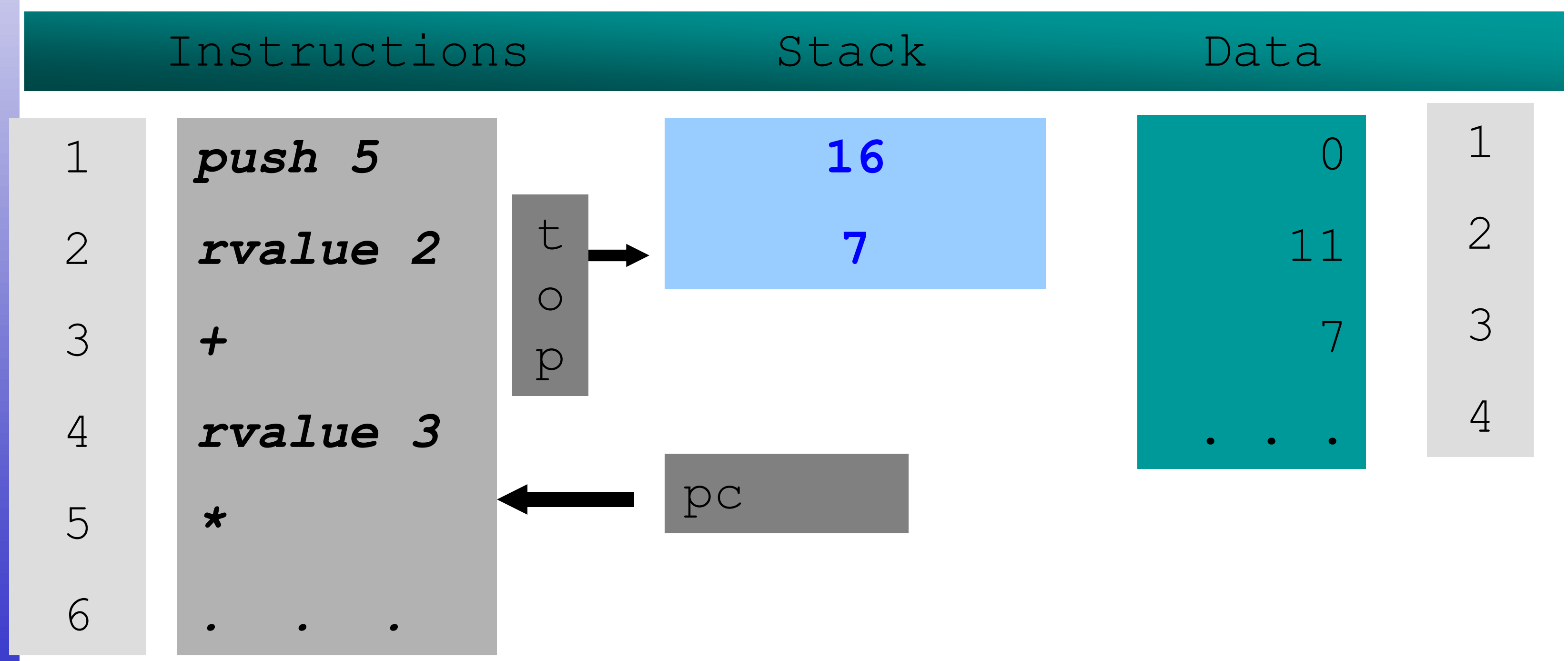
The properties of the machine

1. **Instruction memory**
2. **Data memory**
3. **All arithmetic operations are performed on values on a stack**

Abstract Stack Machine Code: Instructions

Instructions fall into three classes.

1. Integer arithmetic
2. Stack manipulation
3. Control flow



Abstract Stack Machine Code: **L-value and R-value**

What is the difference between left and right side identifier?

L-value Vs. R-value of an identifier

I : = 5 ; L - Location

I : = I + 1 ; R – Contents

The right side specifies an integer value, while left side specifies where the value is to be stored.

Usually,

r-values are what we think as values

l-values are locations.

Abstract Stack Machine Code: **Stack manipulation**

push v	**push v onto the stack**
rvalue l	**push contents on data location l**
lvalue l	**push address of data location l**
pop	**throw away value on top of the stack**
:=	**the r-value on top is placed in the l-value below it and both are popped**
copy	**push a copy of the top on the stack**

Abstract Stack Machine Code: Translation of Expressions

```
Day = (1461*y) mod 4 + (153*m +2 ) mod 5 + d
```

```
lvalue day
push 1461
rvalue y
*
push 4
mod
push 153
rvalue m
*
```

```
push  2
+
push 5
mod
+
rvalue d
+
:=
```

0
1
2
-3
. . .

1	
2	day
3	y
4	m
5	d

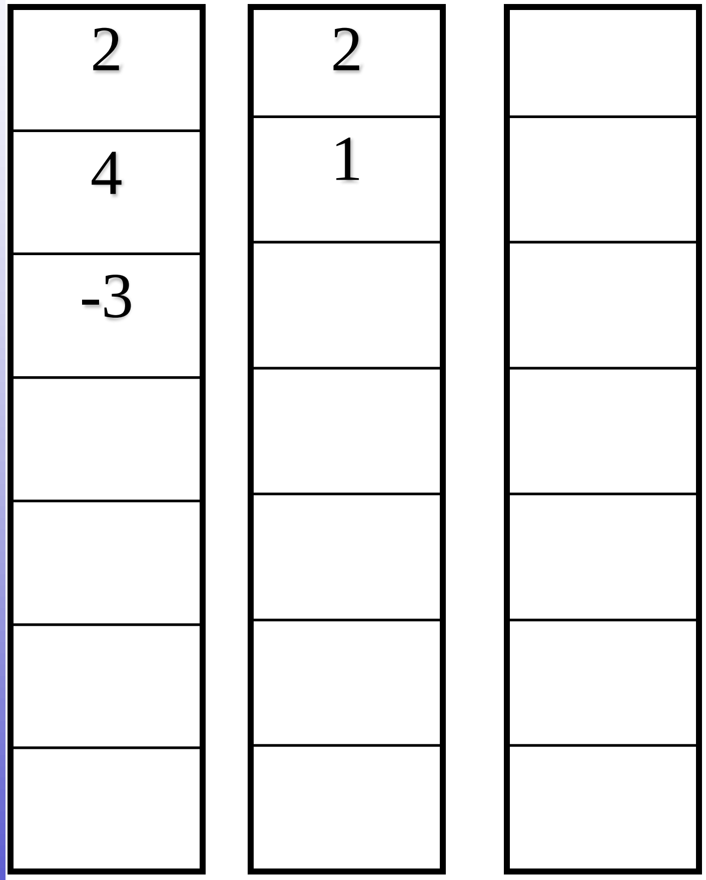
Translation of Expressions (2)

2	2	2	2	2	2	2
	1461	1461	1461	1461	1	1
		1		4		153

Translation of Expressions (3)

2	2	2	2	2	2	2
1	1	1	1	1	1	4
153	306	306	308	308	3	
2		2		5		

Translation of Expressions (4)



0
1
2
-3
. . .

1	
2	day
3	y
4	m
5	d

Intermediate Code Generation: Three-address Code

- Intermediate code generator receives input from, semantic analyzer, in the form of an annotated syntax tree.

- That syntax tree then can be converted into a linear representation.

- For example: a = b + c * d; the intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

  r1 = c * d;   r2 = b + r1;   a = r2;

- A three-address code has at most three address locations to calculate the expression.

Intermediate Code Generation: Three-address Code

Concerning the code segment given in chapter 2 :

COST = RATE * (START – FINISH) + 2 * RATE * (START – FINISH) ;

, The BNF :

Statement → *id = expr ;*

$id \rightarrow l(l|d)^{*}$, $literal \rightarrow d^{+}$, $ter\_sym \rightarrow \; =|\,*\,|\,(\,|\,)\,|-|+|$

and the grammar productions:

expr → ***expr*** **+** ***term*** **|** ***expr*** **–** ***term*** **|** ***term***

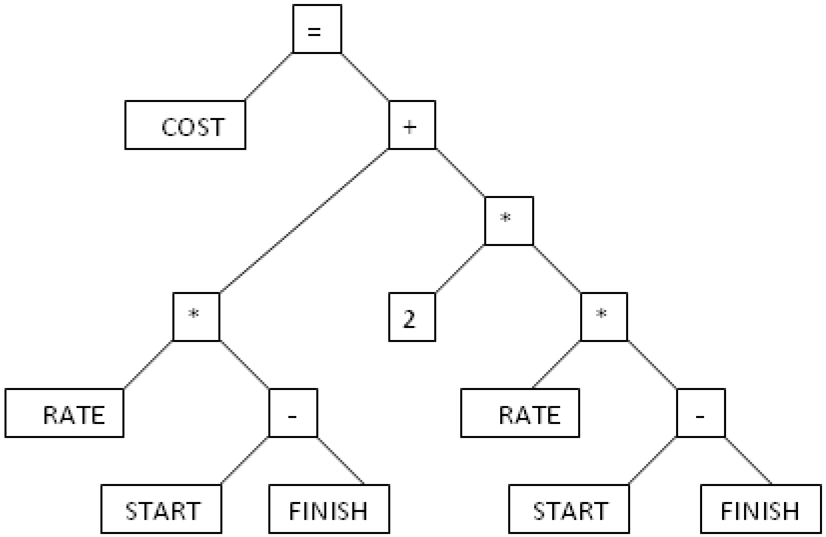
term → ***term*** ***** ***factor*** **|** ***term*** **/** ***factor*** **|** ***factor***

factor → **digit |** ***id*** **|** ***(expr)***

digit → 0 | 1 | 2 | 3 | … | 9

Intermediate Code Generation: Three-address Code

According to the operator precedence rules provided by the given grammar the abstract syntax tree, AST, will be as follows:



Intermediate Code Generation: Three-address Code

The three-address code and the optimized code are given below:

M#	Operation	OP1	Op2
1	-	START	FINISH
2	*	RATE	M1
3	-	START	FINISH
4	*	RATE	M3
5	*	2	M4
6	+	M2	M5
7	=	COST	M6

3-address code matrix

M#	Operation	OP1	Op2
1	-	START	FINISH
2	*	RATE	M1
3			
4			
5	*	2	M2
6	+	M2	M5
7	=	COST	M6

Optimized 3-address code matrix

The End