Compiler Theory

(A Simple Syntax-Directed Translator)

002

Lecture Outline

- We shall look at a simple programming language and describe the initial phases of compilation.
- We start off by creating a ‘simple’ syntax directed translator that maps infix arithmetic to postfix arithmetic.
- This translator is then extended to cater for more elaborate programs such as (check page 39 Aho)
  - While (true) { x=a[i]; a[i]=a[j]; a[j]=x; }
- Which generates simplified intermediate code (as on pg40 Aho)

Two Main Phases (Analysis and Synthesis)

- ***Analysis Phase*** *:- Breaks up a source program into constituent pieces and produces an internal representation of it called* *intermediate code*.

- ***Synthesis Phase*** *:- translates the intermediate code into the target program.*

- *During this lecture we shall focus on the analysis phase (compiler front end … see figure next slide)*

A model of a compiler front end

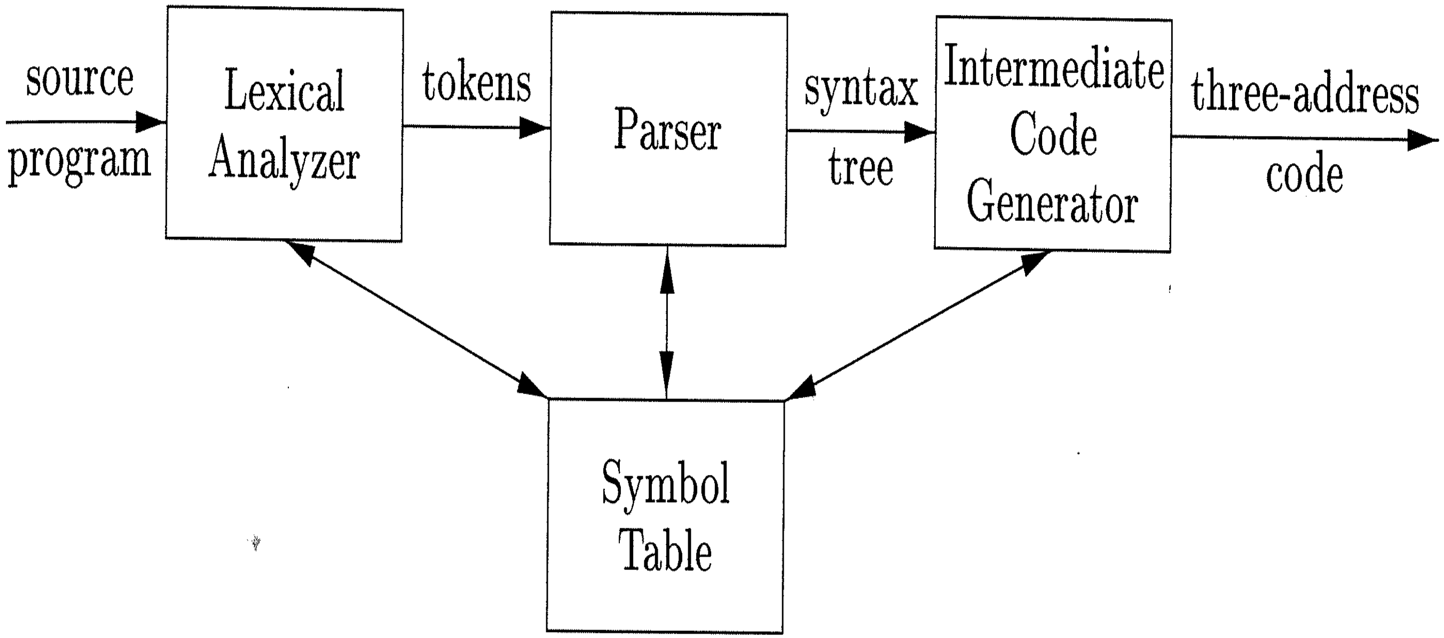


Figure 2.3: A model of a compiler front end

Syntax vs Semantics

- The *syntax* of a programming language describes the proper form of its programs
- The *semantics* of the language defines what its programs mean.
- e.g. fact n = if (n==0) 1 else n*fact (n-1)

A note on Grammars (context-free) !!

- Consider the Maltese grammar. It specifies how correct Maltese sentences should be.
- A formal grammar is used to specify the syntax of a formal language (for example a programming language like C, Java)
- Here grammar describes the structure (usually hierarchical) of programming languages.
- For e.g. in Java an IF statement should fit in
  - **if** (expression) statement **else** statement
- statement -> **if** (expression) statement **else** statement
- Note the recursive nature of statement.

A CFG has four components …

- A set of <u>terminal</u> symbols, sometimes referred to as 'tokens'. The terminals are the elementary symbols of the language defined by the grammar.
- A set of <u>non-terminals</u>, sometimes called 'syntactic variables'. Each non-terminal represents a set of strings of terminals.
- A set of <u>productions</u> (LHS → RHS), where each production consists of a non-terminal (LHS) and a sequence of terminals and/or non-terminals (RHS)
- A designation of one of the non-terminals as the <u>start</u> symbol

A Grammar for 'list of digits separated by + or –'

- list → list + digit
  list → list – digit
  list → digit
  digit → 0 | 1 | … | 9

- Accepts strings such as 9-5+2, 3-1, or 7.

- *list* and *digit* are non-terminals
- 0 | 1 | … | 9, +, - are the terminal symbols

Parsing … and derivations

- Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar,

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a non-terminal by the body of a production,

- If it cannot be derived from the start symbol then reporting syntax errors within the string.

Parse Trees (and their Ambiguities)

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language

- A grammar can have more than one parse tree generating a given string of terminals (thus making it ambiguous);

- If we did not distinguish between digits and lists in the previous grammar then we would end up with ambiguous parse trees; (9-5)+2 and 9-(5+2)

- Check grammar below :
- string → string + string | string – string | 0 … 9

Operator Associativity and Precedence

- To resolve some of the ambiguity with grammars that have operators we use:
  - <u>Operator associativity</u> :- in most programming languages arithmetic operators have left associativity.
    - Eg 9+5-2 = (9+5)-2
    - However = has right associativity, i.e.
      - a=b=c is equivalent to a=(b=c)
  - Operator Precedence :- if an operator has higher precedence then it will bind to it's operands first.
    - eg. * has higher precedence then +, therefore
    - 9+5*2 is equivalent to 9+(5*2)

A grammar for a subset of Java statements

- stmt → id = expression;
  | if (expression) stmt
  | if (expression) stmt else stmt
  | while (expression) stmt
  | do stmt while (expression);
  | { stmts }

  stmts → stmts stmt
  | e

Syntax Directed Translation (Rules)

- Done by attaching rules (or program fragments) to productions in a grammar.
- E.g. With expr -> expr1 + term ,
  - one would apply rules
    - translate expr1, then
    - translate term and finally
    - Handle +
- Syntax Directed translation will be used here to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs.

Postfix Notation (defined for E)

- If E is a variable or constant, then the postfix notation for E is E itself.

- If E is an expression of the form E1 op E2, where op is any binary operator, then the postfix notation for E is E1' E2' op, where E1' and E2' are the postfix notations for E1 and E2, respectively.

- If E is a parenthesized expression of the form (E1), then the postfix notation for E is the same as the postfix notation for E1.

Synthesised Attributes (i)

- Associate attributes with non-terminals and terminals in a grammar.
- Then, attach rules to the productions of the grammar which describe how the attributes are computed.

- Syntax-directed definition associates
  - A set of attributes with each grammar symbol
  - A set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production.

Synthesised Attributes (ii)

- Suppose node N is labelled by grammar symbol X
- X.a denotes the value of attribute a of X at that node.
- expr.t = 95-2+ (attribute value at the root of parse tree for 9-5+2.
- Check parse tree for 9-5+2 (page 54 Aho)

- An attribute is said to be synthesised if its value at a parse-tree node N is determined from attribute values of the children of N and at N itself.
- Therefore , if this is the case for every attribute, we can evaluate a parse tree in a single bottom-up traversal.
- Eventually we shall discuss "inherited" attributes as well.

Semantic Rules for infix to postfix

- The annotated parse tree of 9-5+2 is based on the following syntax directed definition. || represents string concatenation.

PRODUCTION	SEMANTIC RULE
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel$ '+'
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel$ '-'
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t :=$ '0'
$term \rightarrow 1$	$term.t :=$ '1'
. . .	. . .
$term \rightarrow 9$	$term.t :=$ '9'

Fig. 2.5. Syntax-directed definition for infix to postfix translation.

Tree Traversals

- A *traversal* of a tree starts at the root and visits each node of the tree in some order.
- Breadth First
- Depth First
  - Preorder traversal of node N consists of N, followed by the pre-orders of the subtrees of each of its children, if any, from the left.
  - Postorder traversal of node N consists of the postorders of each of the subtrees for the children of N, if any, from the left, followed by N itself.

Actions translating 9-5+2 into 95-2+

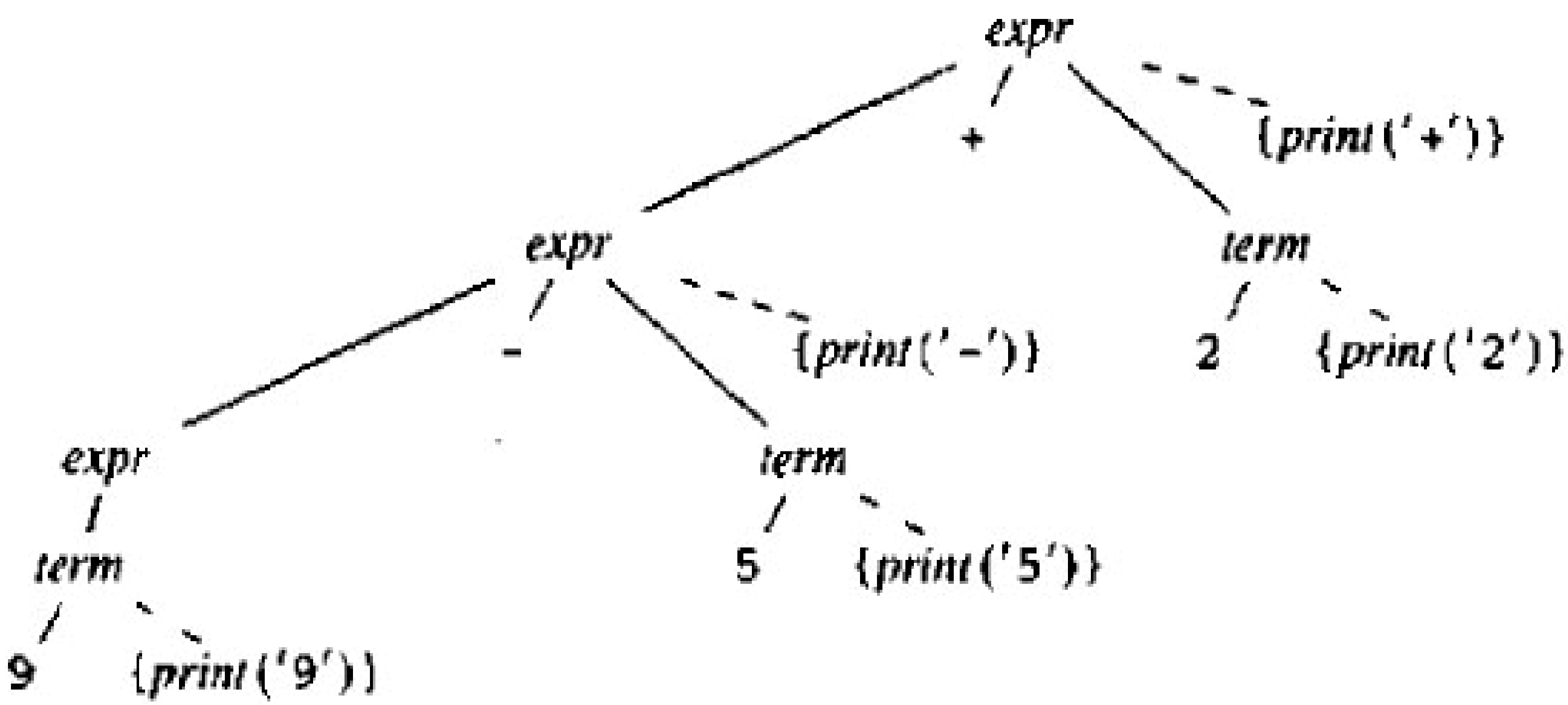


Fig. 2.14. Actions translating 9-5+2 into 95-2+.

Translation Schemes

- Instead of attaching strings as attributes to the nodes we can execute program fragments (and not manipulate strings)
- *Semantic Actions* : program fragments embedded within production bodies
- The position at which an action is to be executed is shown by enclosing it between curly braces.
- e.g. (check pg59 Aho for full grammar)
  - Expr -> expr1 + term {print('+')}
  - Expr -> term
  - Expr -> 1 {print('1')}
- Check next slide for parse tree ... postorder traversal gives us the required postfix translation (95-2+)

Parsing

- Parsing is the process of determining how a string of terminals can be generated by a grammar.
- *Recursive descent parsing* : technique which can be used both to parse and to implement syntax-directed translators.
- Two classes :-
  - Bottom-up, where construction starts at the leaves and proceeds towards the root;
  - Top-down, where construction starts at the root and proceeds towards the leaves.

Top-Down parsing (i)

- Let us first look at a simplified (abstracted) C/Java grammar.
- *stmt* ->
  - **expr;**
  - **if (expr)** *stmt*
  - **for** *(optexpr; optexpr; optexpr) stmt*
  - **other**
- **optexpr ->**
  - **ε**
  - **expr**

Top-Down parsing (ii)

- Construction of the parse tree is carried out by starting from the root (call it node N), labelled with the starting non-terminal *stmt,*
  - At node N, labelled with a non-terminal A, select one of the productions for A and construct children at N for the symbols in the production body,
  - Find the next node at which a sub-tree is to be constructed, typically the leftmost unexpanded non-terminal of the tree and repeat step 1.
- Next slide shows the parse tree for statement
  - for (; expr ; expr) other

Top-down parsing while scanning the input from left to right (Aho pg 63) – Using Lookahead

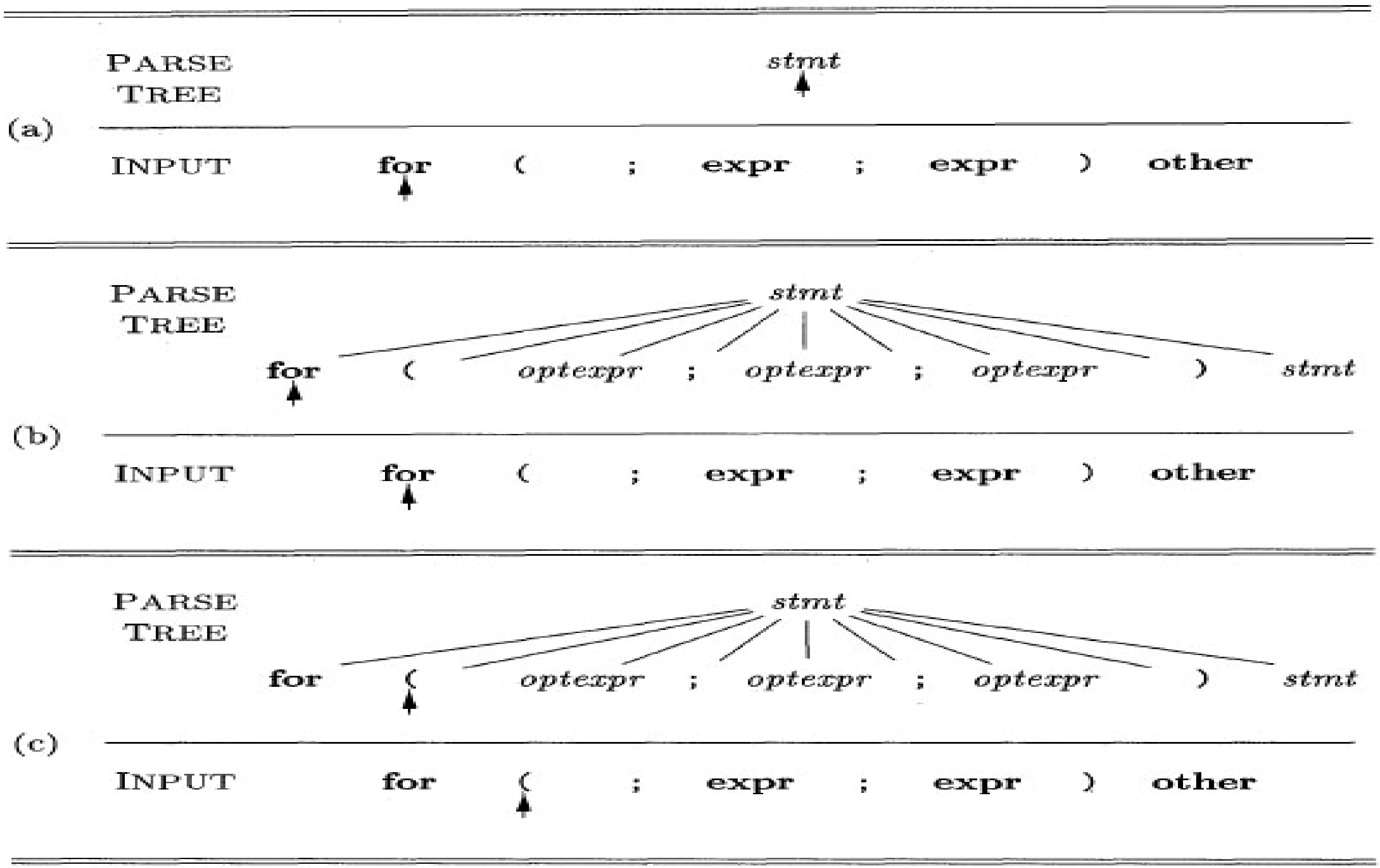


Figure 2.18: Top-down parsing while scanning the input from left to right

Predictive Parsing (top-down)

- In general choosing which production to expand is trial and error where backtracking might be used.
- But not in *predictive parsing* ! (which is a simple form of recursive-descent parsing)
- The lookahead symbol **unambiguously** determines the flow of control through the procedure body of each non-terminal.
- The sequence of procedure calls during the analysis of an input string implicitly defines the parse tree for the input.

Predictive Parser (pseudo code)

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}
```

Predictive Parser (pseudo code)

```
void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Figure 2.19: Pseudocode for a predictive parser

Predictive parsing (iii)

- Let *α* be a string of grammar symbols (terminals and/or non-terminals)
- Let First(*α*) be the set of terminals that appear as the first symbols of one or more strings of terminals generated from *α*. e.g. First(*stmt*) = {**expr**, **if**, **for**, **other**}. First (**expr ;**) = {**expr**}
- Given any two productions in the grammar A->*α* and A->*β*, then a predictive parser **requires** that First(*α*) is disjoint from First(*β*).
- We shall see how First(*α*) is computed later on.
- The lookahead symbol determines which production to expand. Lookahead changes when a terminal is matched.

Predictive parsing (iv)

- When to use **ε** production ??
  - When you've got no other rule to match.

- If we had
  - Optexpr -> expr | **ε**

- If the lookahead symbol is not in First(expr) then the **ε**-production is used !

Left Recursion (i)

- *expr -> expr + term*
- Productions like the above make it possible for a recursive-descent parser to loop forever, since the leftmost symbol of the body is the same as the non-terminal at the head of the production.
- Since the lookahead symbol changes only when a terminal is matched, no change to the input takes place between recursive calls of *expr*.

Left Recursion (and how to avoid it)

- A **->** $A\alpha$ | β
  - (note that Aα may be derived through intermediate productions)
- A new non-terminal *R* is required to remove left recursion ...
  - A -> βR
  - R -> αR | **ε**
- Check out derivation for βααα...αα (pg 68)

Postfix to infix removal of Left Recursion in Translation Scheme

- expr ->
  - expr **+** term { print('+') }
  - expr **–** term { print('-') }
  - Term
- term ->
  - 0 { print('0') } ....
  - 9 { print('9') }

- A -> Aa | Ab | y
- This will always start with a 'y' and end with an 'a' or a 'b'.

- expr -> term rest
- rest ->
  - **+** term { print('+') } rest
  - **–** term { print('-') } rest
  - **ε**
- term ->
  - 0 { print('0') } ....
  - 9 { print('9') }

- A -> yR
- R -> aR | bR | **ε**

New Parse Tree for 95-2+ (pg 71)

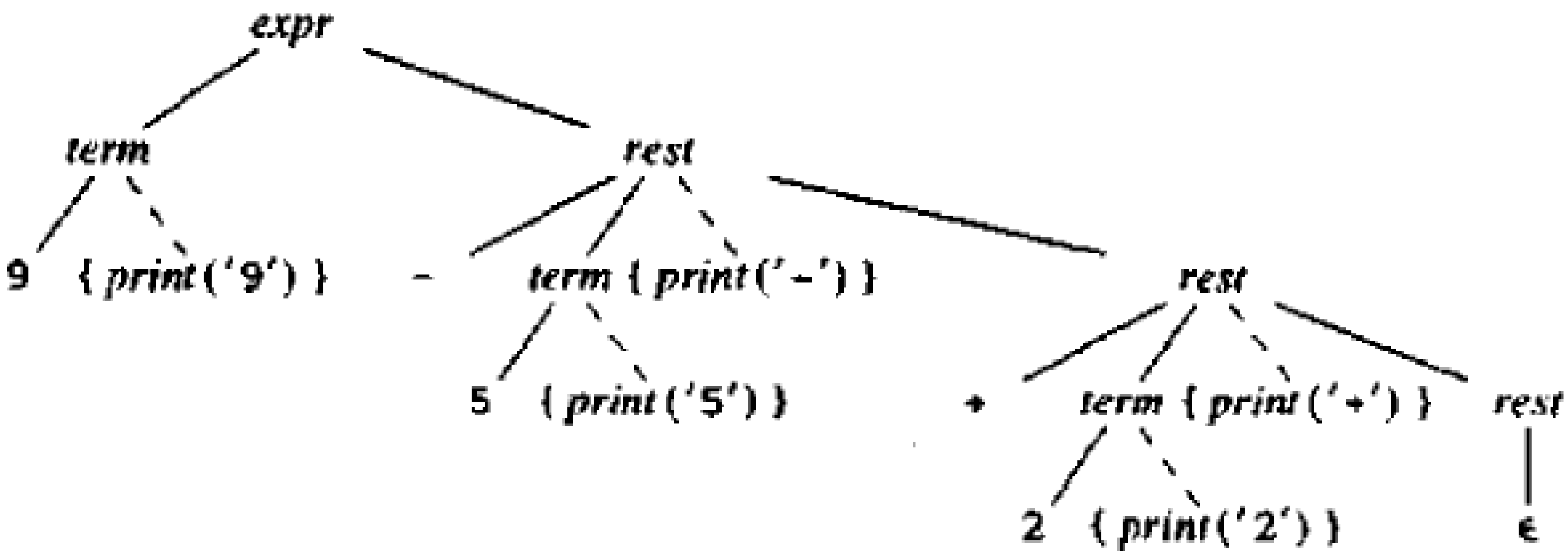


Fig. 2.21. Translation of 9-5+2 into 95-2+.

Abstract and Concrete Syntax Trees

- In an abstract syntax tree, each interior node represents an operator (programming constructs); the children of the node represent the operands of the operator

- In a concrete syntax tree (parse tree) the interior nodes represent non-terminals in the grammar.

- Ideally our parse tree go as close to abstract syntax trees as possible.

Lexical Analysis

- Consider
  - Factor -> (expr) | **num** | **id**

- A lexer will not find terminals **num** and **id** in the input.

- These range over a number of inputs which the lexer must recognise.

- Attribute num.value stores the value of the number
- Attribute id.lexeme stores the string of the id

Reading Ahead – Input Buffer

- Is it '>' or '>=' ? ... The lexer needs to read one character in order to decide what token to return to the parser.
- One-character read ahead usually suffices, so a simple solution is to use a variable, call it *peek*, to hold the next input character.
- If (peek holds a digit) {
  - v = 0;
  - Do {

    v = v * 10 + integer value of digit peek;

    Peek = next input character;
  - } while (peek holds a digit);
  - Return token <num, v>
- Simulate parsing some number .... e.g. 256

Recognising keywords and identifiers

- <id, 'count'> <=> <id, 'count'> <+> <id, 'inc'> <;>
- We can identify between keywords and identifiers by creating a table and initializing it with the keywords and their tokens. When matching the input the lexical analyser return the tokens stored in this table (for keywords) otherwise creates a new one and returns token <id, 'cnt'>

- Dragon book has a Java implementation of a lexer using this technique. (pg 83 and 84)

Symbol Table(s)

- Data structures that are used by compilers to hold information about the source-program constructs.
- Information is collected incrementally throughout the analysis phase and used for the synthesis phase.
- One symbol table per scope (of declaration)...
- { int x; char y; { bool y; x; y; } x; y; }
  - { { x:int; y:bool; } x:int; y:char; }

Intermediate Code Generation

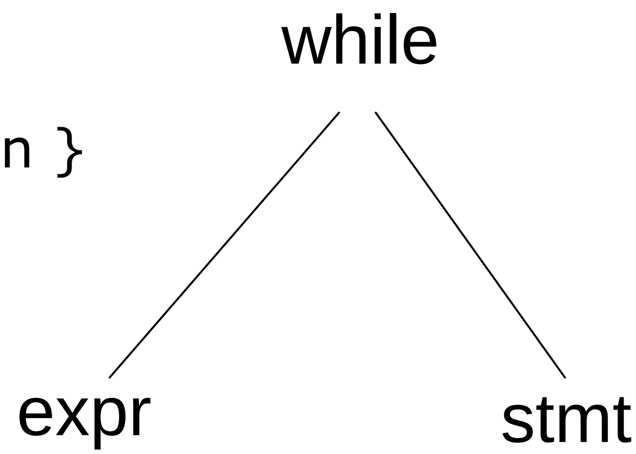
- The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program.

- Let us (just for now) consider only expressions and statements.

- Two main options
  - Trees, including parse trees + (abstract) syntax trees
  - Linear representation, mainly "three-address code"

Syntax Trees

- Pg 94 (Aho) describes a translation scheme that constructs syntax trees. This is then modified to emit three-address code.
- *stmt* **-> while** (expr) stmt
- { stmt.n = new While(expr.n, stmt.n }
- n is a node in the syntax tree
- *stmts* -> stmts_1 stmt
- { stmts.n = new Seq(stmts_1.n, stmt.n); }



Part of a syntax tree

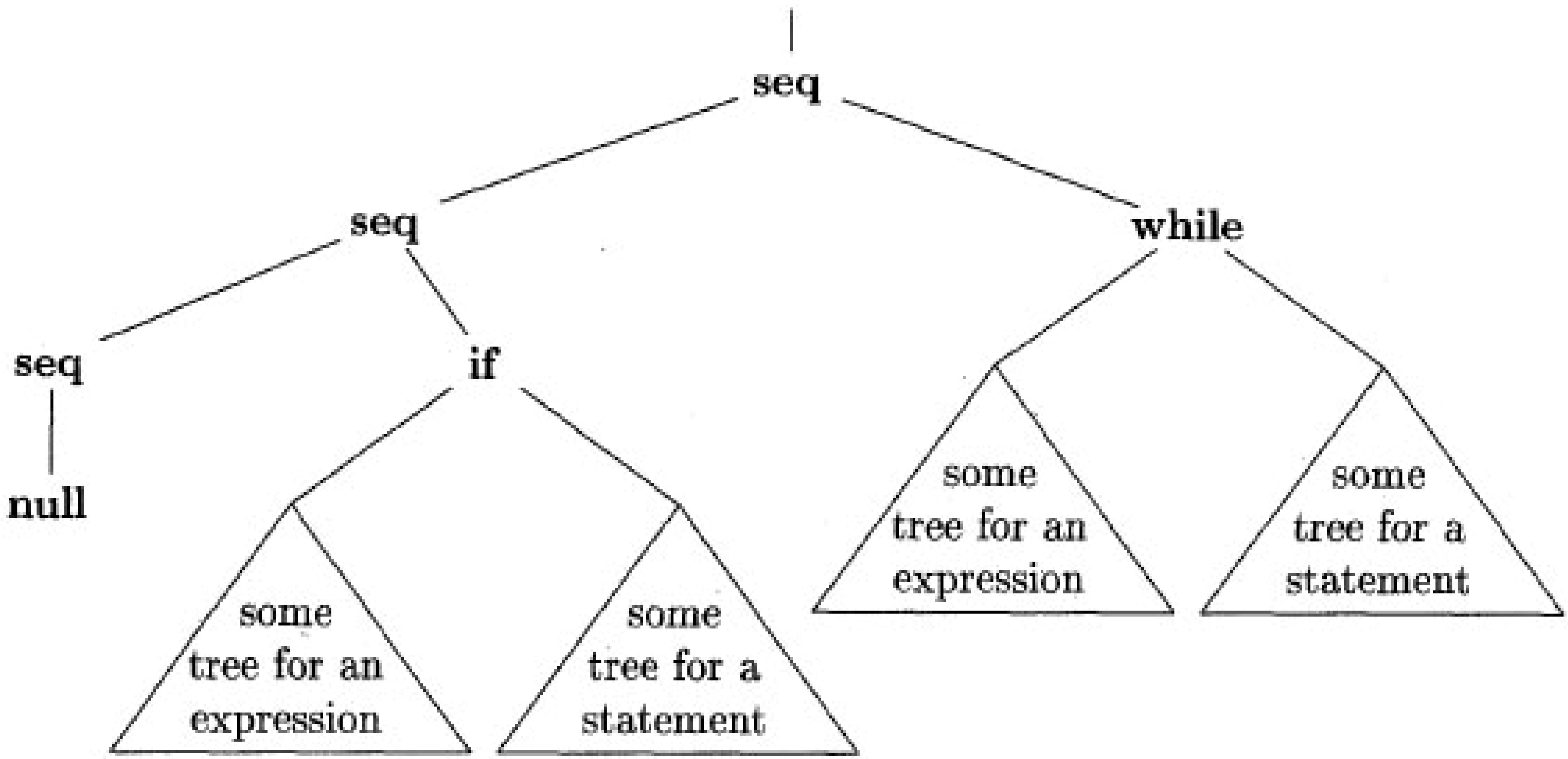


Figure 2.40: Part of a syntax tree for a statement list consisting of an if-statement and a while-statement

Syntax Trees for Expressions

- term -> $term_1$ * factor
  - { term.n = new Op('*', $term_1$.n, factor.n); }
- Class Op can implement operators +, -, *, /, %.
- Note how in the syntax tree we loose information from the parse tree ... as in term, $term_1$, etc.
- The parameter to Op (e.g. '*' identifies the actual operator, in addition to the nodes $term_1$.n and factor.n for the sub-expressions.

Three Address Code

- Now that we have a syntax tree ...
- We can write functions, which process it and as a side-effect, emit the necessary three-address code.
- x = y **op** z (instructions in a three-address code)
- Executed in a numerical sequence unless a jump is encountered. e.g. ifFalse/ifTrue x goto L, goto L
- Arrays
  - x [y] = z
  - x = y [z]
- Copy value
  - x = y

Translation of Statements

- Use jump instructions to implement the flow of control through the statement.
- The statements 'if **expr** then **stmt**' can be represented in 3-address code using,
  - ifFalse x goto after

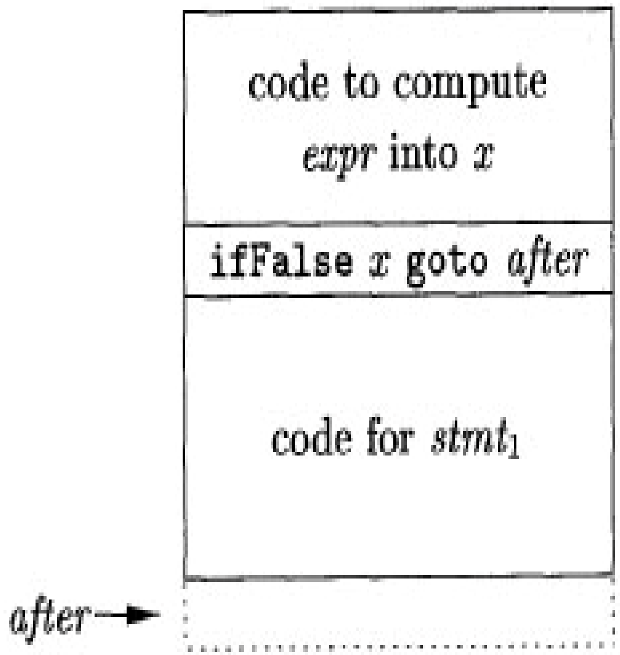


Figure 2.42: Code layout for if-statements

Translation of Expressions

- Expressions contain binary operators, array accesses, assignments, constants and identifiers.
- We can take the simple approach of generating one three-address instruction for each operator node in the syntax tree of an expression.
- Expression: i-j+k translates into
  - t1 = i-j
  - t2 = t1+k
- Expression: 2 * a[i] translates into
  - t1 = a [i]
  - t2 = s * t1

Functions *lvalue(x:Expr)* and *rvalue(x:Expr)*

- In a = a + 1, a is computed differently on the LHS and the RFS of the instruction
- Hence we need a way to distinguish between (L|R)HS
- The simple approach is to use two functions:
  - *Rvalue*, which when applied to a nonleaf node x, generates the instructions to compute x into a temporary var, and returns a new node representing the temporary var.
  - *Lvalue*, which when applied to a nonleaf, generates instructions to compute the subtrees below x, and returns a node representing the **"address"** for x
- R-values is what we usually think of as "values" while L-values are "locations"

lvalue(x:Expr) -> Expr

- x = identifier e.g. a
  - return x
- x = array access e.g. a[i]
  - Return Access(y, rvalue(z)), where
    - y = name of array
    - z = index in array
- Note call to rvalue(z) in order to generate instructions, if needed, to compute the r-value of z
- e.g. If x is a[2*k] then lvalue(x) first generates the instruction “t = 2 * k” which computes the index and then returns a new node x' representing the l-value a[t]

rvalue(x:Expr) -> Expr

- x = constant or identifier
  - return x
- x = y **op** z
  - First compute y' = rvalue(y) and z' = rvalue(z), then generates an instruction t = y' **op** z'. Return new node for temporary t
- x = y[z]
  - Similar to lvalue
- x = y=z
  - First compute z' = rvalue(z), then generate instruction for lvalue(y) = z' (this is like a side-condition) and finally return z'. e.g. a = b = 7

*e.g. a[i] = 2 * a[j-k]*

- rvalue(a[i] = 2* a[j-k])
  - t3 = j – k
  - t2 = a [t3]
  - t1 = 2 * t2
  - a[i] = t1

- Check out pg 104 (and the rvalue pseudo-code) in you have difficulties understanding how the instructions have been generated.

Two possible translations of a statement

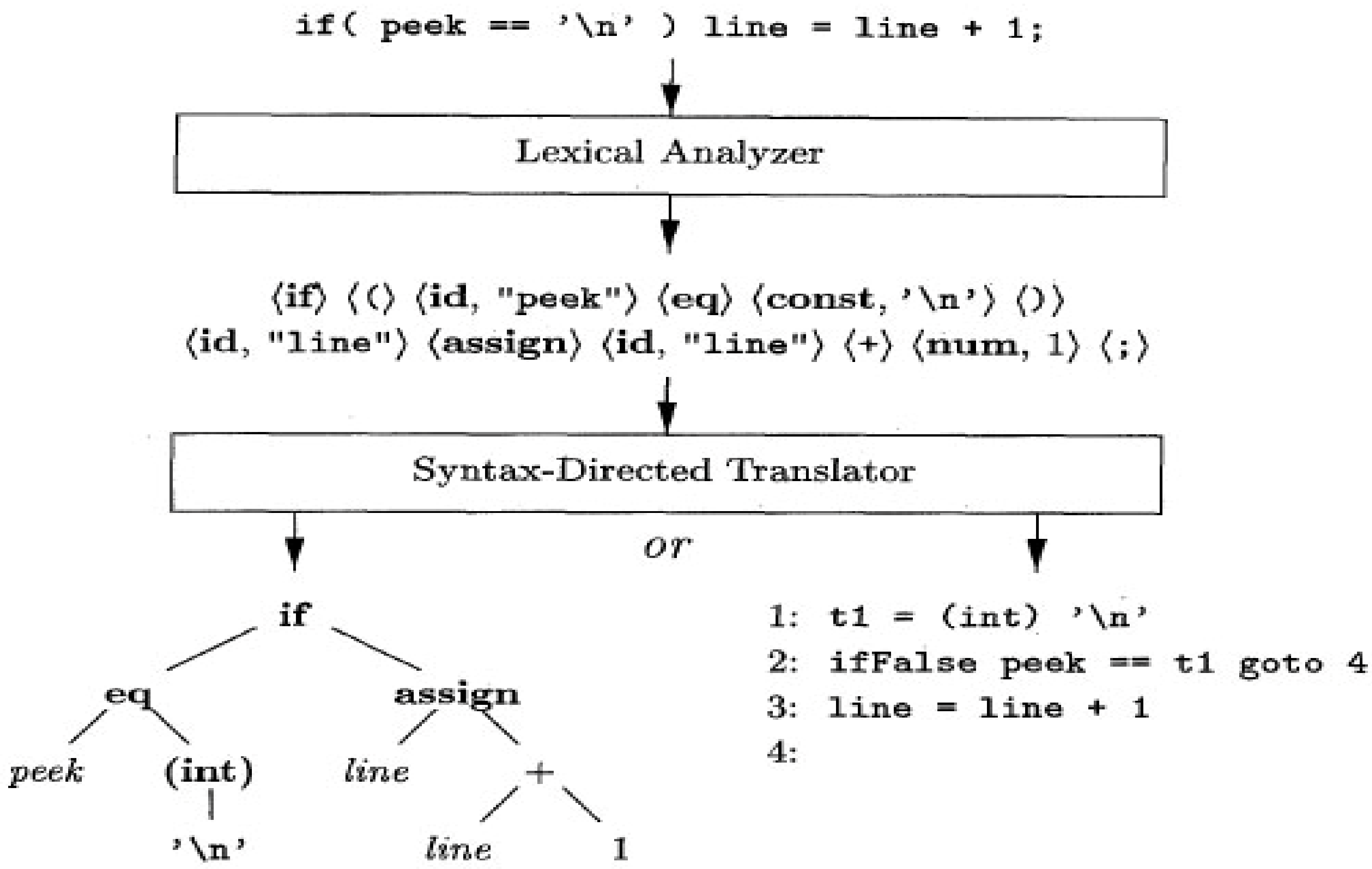


Figure 2.46: Two possible translations of a statement