

Chapter 2
A Simple Syntax-Directed Translator



Outline

- Introduction to the compiler front end
- Syntax definition
- Syntax-directed translation
- Parsing
- A translator for simple expressions
- Lexical analysis
- Symbol tables
- Intermediate code generation



Introduction to the Compiler Front End

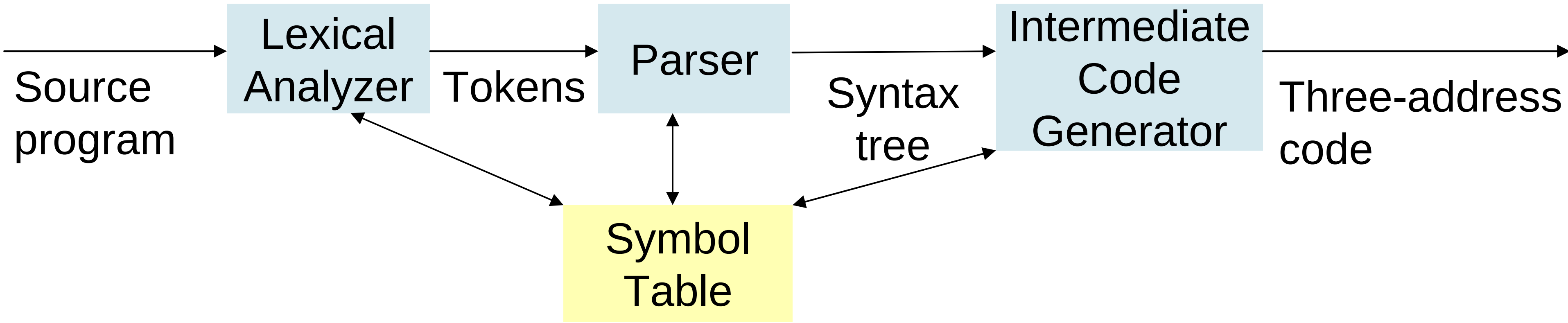


Introduction

- This chapter emphasizes on the front end of a compiler with a working Java program.
  - A simple example to introduce lexical analysis, parsing, and intermediate code generation
  - A simple syntax-directed translator is created
    - To map infix arithmetic expressions to postfix expressions.
    - To map code fragments into three-address code.
  - The syntax specification used in this simple translator is the context-free grammar or BNF (Backus-Naur Form)
    - Context free means parentheses of different types should be nested (and should not overlap).

Introduction (Cont.)

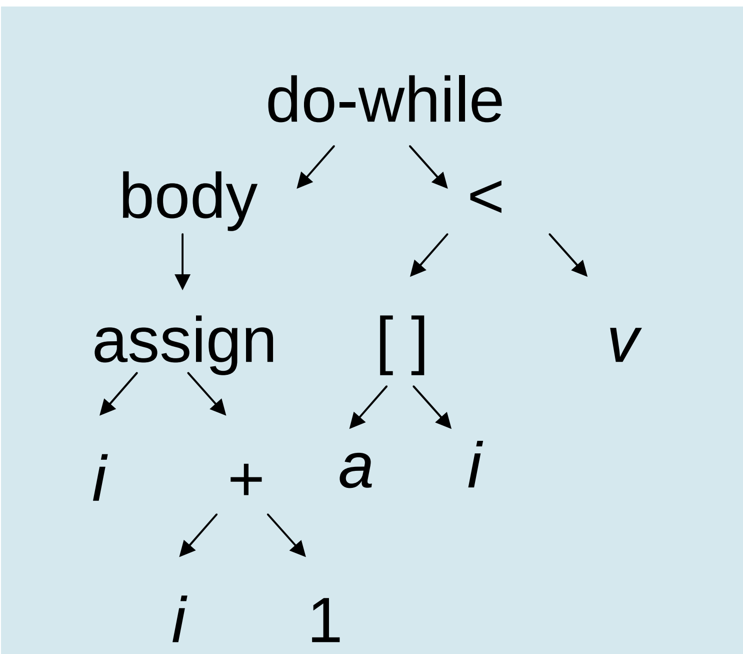
- In a programming language
  - The syntax describes the proper form of its programs.
  - The semantics defines what its programs mean (i.e., what each program does when it executes.)



Note: The semantic analysis is skipped in this figure.

Introduction (Cont.)

- Two forms of intermediate code:
  - E.g., "do i = i + 1; while (a [i] < v);"



(Abstract) syntax tree

```
1:  i= i + 1
2:  t1 = a [ i ]
3:  if  t1 < v  goto 1
```

Three-address code



Syntax Definition

Context-Free Grammar

- Components
  - Terminal (also called tokens)
    - The elementary symbols of the language defined by the grammar.
  - Nonterminal (also called syntactic variables)
    - Each nonterminal represents a set of strings of terminals.
  - Production
    - Each production consists of a nonterminal (called the head or left side of the production), an arrow, and a sequence of terminals/nonterminals (called the body or right side).
  - Start symbol
    - A designation of one of the nonterminals as the start symbol
- Productions for the start symbol is listed first.



Context-Free Grammar (Cont.)

- An example:

An if-else statement

if (expression) statement **else** statement

Context-free grammar

stmt → **if** (expr) stmt **else** stmt → (A production)

Can have the form

- Variables like *expr* and *stmt* are nonterminals (i.e., sequences of terminals).
- Keywads ("if" and "else") and parentheses are called terminals.



Tokens vs. Terminals

- A token consists of a token name and an attribute value.
  - A token name is a terminal that is an abstract symbol for syntax analysis
  - An attribute value is a pointer to the symbol table containing additional information about the token. (not part of the grammar)



Simple Example of Productions

- A string consists of digits (single digit), plus, and minus signs. E.g., 9-5+2
  - 13 productions
  - 2 nonterminals: list, digit
  - 12 terminals: + - 0 1 2 3 4 5 6 7 8 9

list → list + digit | list – digit | digit

Start symbol

list → list + digit	(2.1)
list → list – digit	(2.2)
list → digit	(2.3)
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	(2.4)

Note: a production is *for* a nonterminal if the nonterminal is the head of the production.



Derivations

list $\rightarrow$ list + digit (2.1)
list $\rightarrow$ list – digit (2.2)
list $\rightarrow$ digit (2.3)
digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 (2.4)

- Derivations(推導):
  - A grammar derives strings by
    - beginning with the start symbol and
    - repeatedly replacing a nonterminal by the body of a production for that nonterminal.
  - The terminal strings that can be derived from the start symbol form the language defined by the grammar.
- E.g., 9-5+2
  - 9 is a *list* by production (2.3) since 9 is a *digit*
  - 9-5 is a *list* by production (2.2) since 9 is a *list* and 5 is a *digit*.
  - 9-5+2 is a *list* by production (2.1) since 9-5 is a *list* and 2 is a *digit*.

A Grammar for Empty List of Parameters

- A function call might consist of an empty list of parameters.
  - E.g., a function call *max()*
- An example of the grammar for empty list of parameters:

Optional parameter list

Empty list (epsilon)

call → id (*optparams*)
optparams → *params* | ε
params →*params*, *param* | *param*

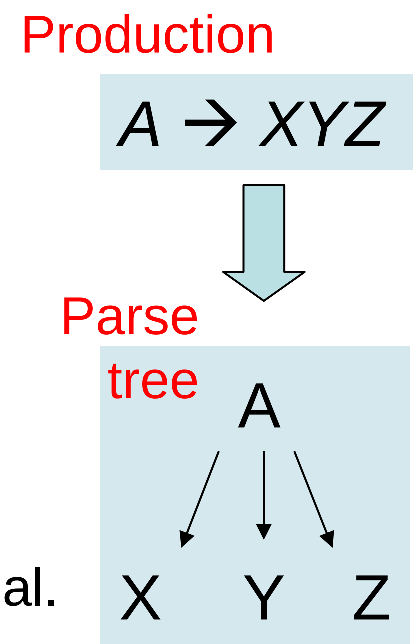


Parsing

- Parsing is the problem of
  - Taking a string of terminals
  - Figuring out how to derive it from the start symbol of the grammar
  - Reporting syntax errors within the string if it can’t be derived.
- Parsing is one of the most fundamental problems in all of compiling.

Parse Tree

- A parse tree pictorially shows how the start symbol of grammar derives a string in the language.
  - Given a context-free grammar (or grammar), a parse tree according to the grammar is a tree.
  - Parse tree properties:
    - The root is labeled by the start symbol.
    - Each leaf is labeled by a terminal or by ε.
    - Each interior node is labeled by a nonterminal.
    - If A is an interior node and $X_1, X_2, \ldots, X_n$ are the children of that node from left to right, there must be a production $A \rightarrow X_1 X_2 \ldots X_n$, where each X_i stands for a terminal or nonterminal.

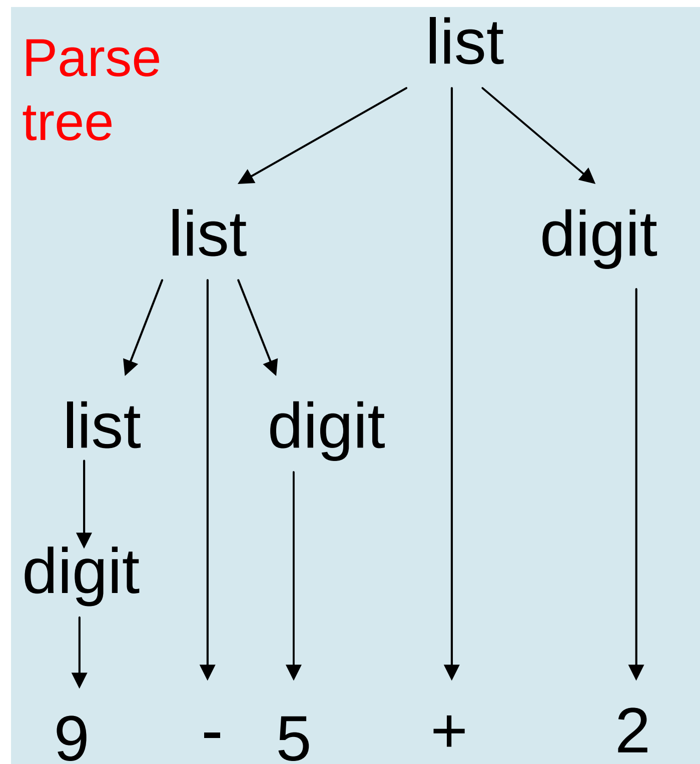


An Example of the Parse Tree

- The parse tree of 9-5+2
  - Each node is labeled with a grammar symbol.
  - An interior node and its children correspond to a production.
    - Interior node: head of the production
    - Children: body of the production
- Parsing a tree is to find a parse tree for a given string of terminals.

Productions

list → list + digit	(2.1)
list → list – digit	(2.2)
list → digit	(2.3)
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	(2.4)



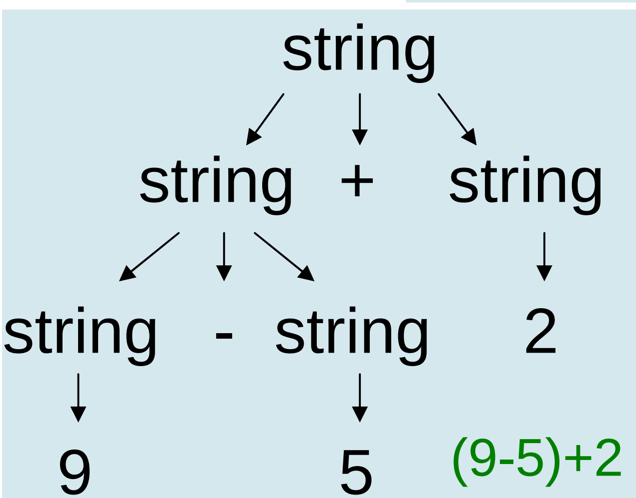
Ambiguity

- A grammar is ambiguous if it can have more than one parse tree generating a given string of terminals.
  - A string with more than one parse tree usually has more than one meaning.

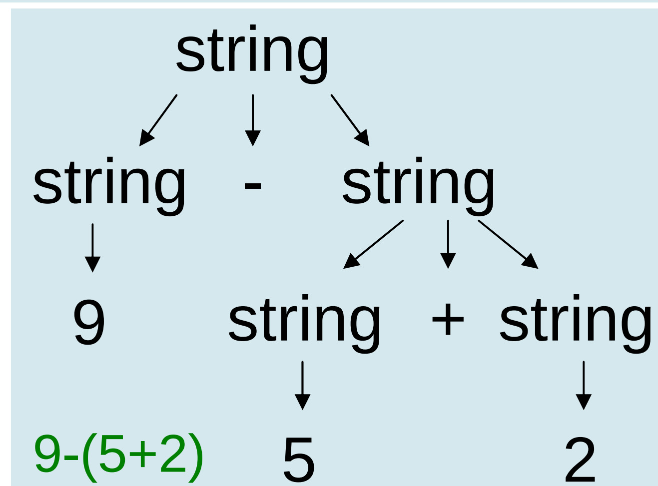
Productions

- E.g., 9-5+2

string → string + string | string – string | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Two parse tree

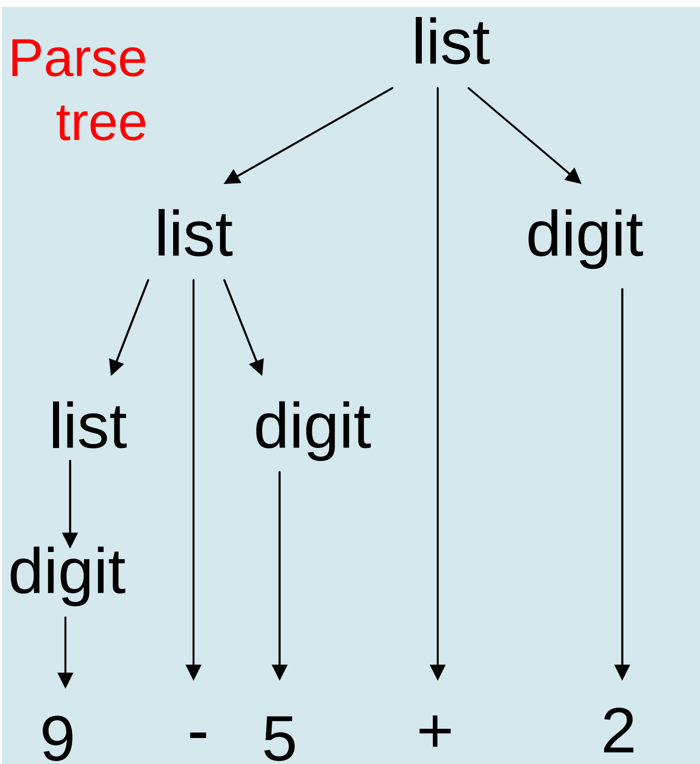


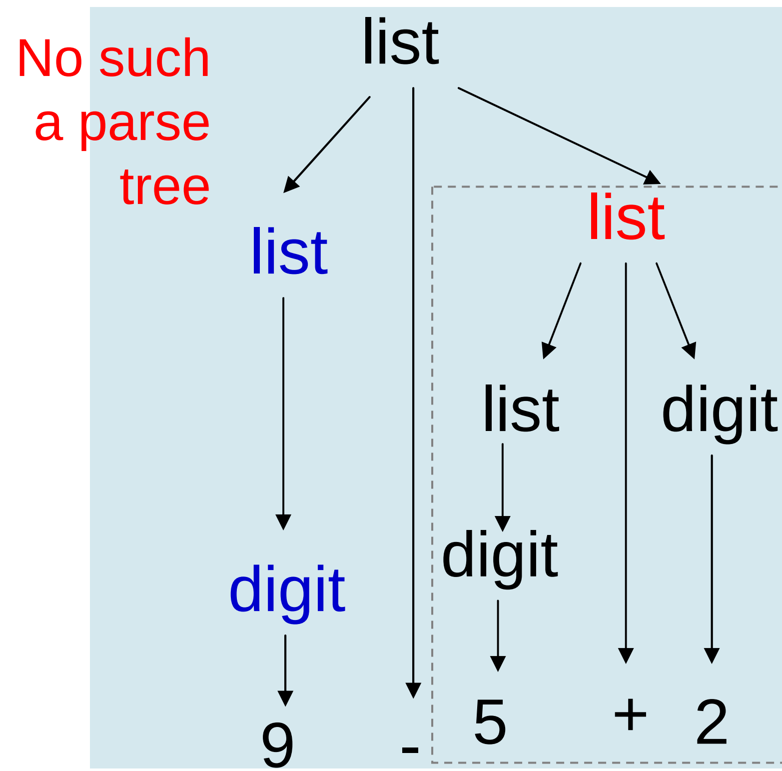
Ambiguity (Cont.)

- E.g., 9-5+2

Productions

list → list + digit	(2.1)
list → list – digit	(2.2)
list → digit	(2.3)
digit →0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	(2.4)







Associativity of Operators

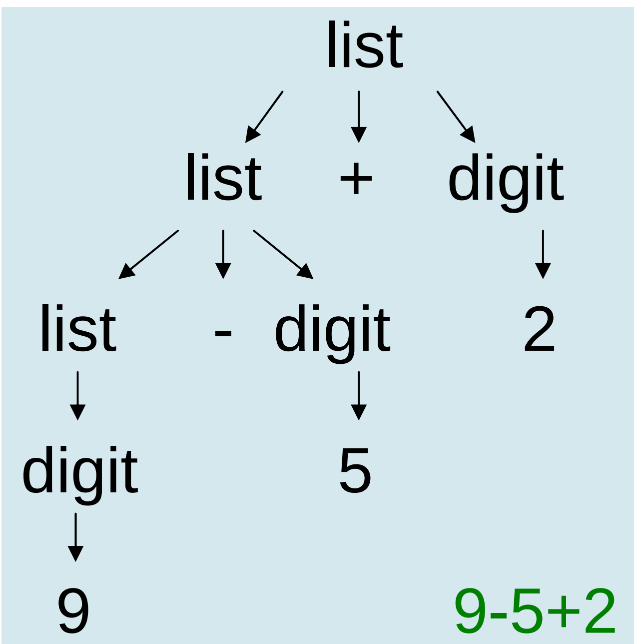
- Left associativity:
  - Operators of the same precedence are processed from left to right.
  - E.g., 9+5+2 = (9+5)+2
- Right associativity:
  - Operators of the same precedence are processed from right to left.
  - E.g., a=b=c equals to a=(b=c)

right → letter = right | letter
letter → a | b | … | z

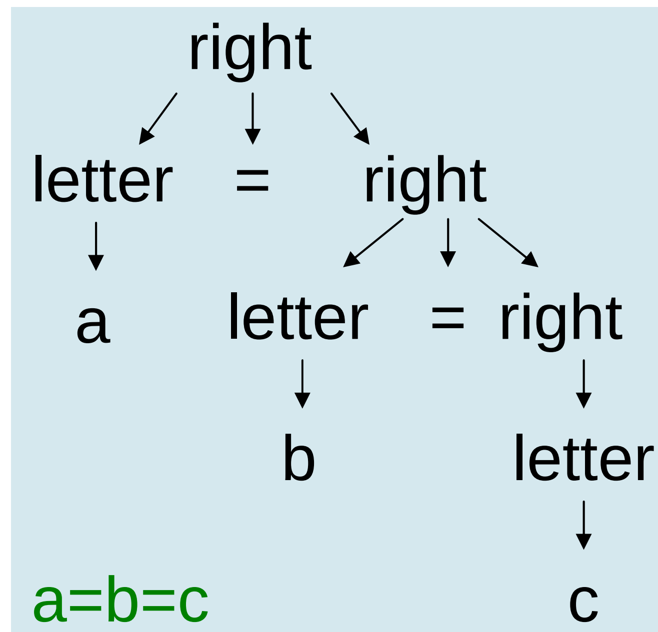
Associativity of Operators (Cont.)

list → list + digit
list → list – digit
list → digit
digit →0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

right → letter = right | letter
letter → a | b | … | z



Left-associative



Right-associative



Precedence of Operators

- A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators.
  - E.g.,
    Left-associative: + - (lower precedence)
    Left-associative: * / (higher precedence)
  - E.g., 9+5*2 = 9+(5*2), 9*5+2 = (9*5)+2



Grammar with Precedence (+ - * /)

- Define nonterminals:
  - factor: for generating basic units in expressions
  - term: for the precedence level of * and /
  - expr: for the precedence level of + and –
- Guidance:
  - n precedence levels need (n+1) nonterminals
- Grammar

Start symbol

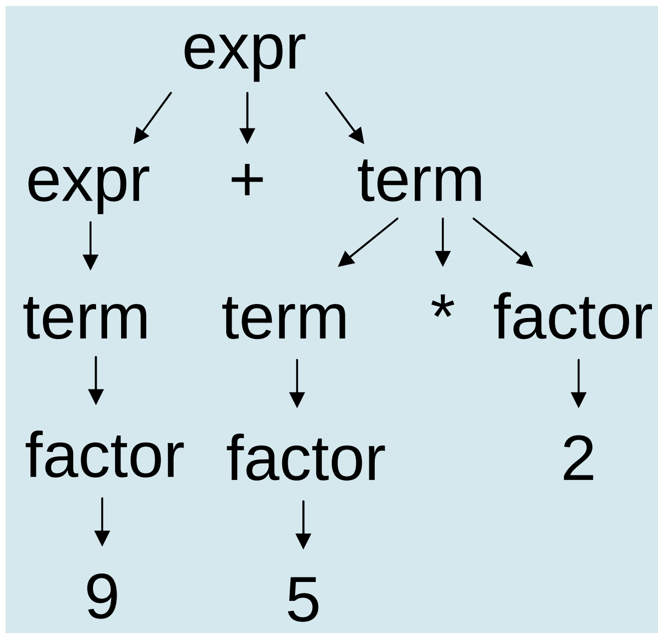
expr → expr + term | expr – term | term
term → term * factor | term / factor | factor
factor → digit | (expr)



Grammar with Precedence (+ - * /) (Cont.)

- E.g., 9+5*2

expr → expr + term | expr – term | term
term → term * factor | term / factor | factor
factor → digit | (expr)



A Grammar for a Subset of Java Statements

The semicolon come from other expressions

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

stmts → *stmts stmt*
  | ε



Syntax-Directed Translation

Syntax-Directed Translation

- Syntax-directed translation is done by attaching rules or programs to productions in a grammar.
  - E.g.,

$expr \rightarrow expr_1 + term$

translation ↓

translate $expr_1$;
translate $term$;
handle +;

The subscript in $expr_1$ is only used to distinguish the instance of $expr$.

Pseudo-code

Concepts Related to Syntax-Related Translation

- Two main concepts:
  - Attributes:
    - An attribute is any quantity associated with a programming construct (程式結構).
    - E.g.,
      - Data types of expressions
      - The number of instructions in the generated code
      - The location of the first instruction in the generated code for a construct.
  - Translation schemes:
    - A translation scheme is a notation for attaching program fragments to the productions of a grammar.
      - The program fragments are executed when the production is used during syntax analysis.



Synthesized Attributes

- Attribute synthesis:
  - Attach associate attributes with nonterminals and terminals.
  - Then attach (semantic) rules to the productions of the grammar.
    - These rules describe how the attributes are computed at nodes of the parse tree.
    - A production is used to relate a node to its children.
- Attribute evaluation:
  - For a given input string *x*,
    - Construct a parse tree for *x*.
    - Then apply the semantic rules to evaluate attributes at each nodes in the parse tree.
- An attribute is synthesized if its value at a parse-tree node N is determined from attributes values at the node N and the children of the node N.
- Synthesized attributes can be evaluated during a single bottom-up traversal of a parse tree.

Postfix Notation

- Postfix notation is easier to generate the three-address code.
- No parentheses are needed in postfix notation.
- Definition of postfix notation:
  - Rule 1: E is a variable or constant → **E**
  - Rule 2: E is an expression of the form $\mathbf{E_1}$ **op** $\mathbf{E_2}$ where op is a binary operator, → $\mathbf{E_1}$ $\mathbf{E_2}$ **op**
  - Rule 3: E is a parenthesized expression of the form $\mathbf{(E_1)}$ → $\mathbf{E_1}$
- E.g.,

Infix → Postfix
(9-5)+2 → 95-2+

Postfix Notation (Cont.)

- The steps to solve the postfix expression:
  1. Scan the postfix string from the left until encountering an operator.
  2. Look to the left for the proper number of operands.
  3. Evaluate the operator on the operands, and replace them by the result.
- E.g., 9 5 2 + - 3 *

9 **5 2 +** - 3 *
→ 9 7 – 3 *
→ **9 7 –** 3 *
→ 2 3 *
→ **2 3 ***
→ 6

Annotated Parse Tree

- Annotated parse tree is a parse tree showing the attribute values at each node.
  - E.g., 9-5+2

Productions

expr → expr + term | expr – term | term
term → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
          expr
       /   |   \
    expr   +   term
   /  |  \        |
expr  -  term     2
 |         |
term       5
 |
 9                9-5+2
```

Parse tree

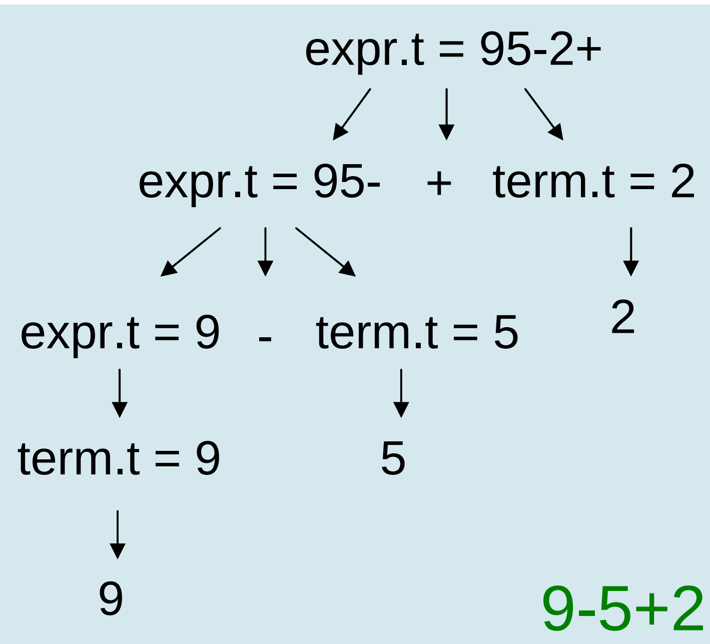
```
                    expr.t = 95-2+
                  /       |       \
        expr.t = 95-      +      term.t = 2
       /      |      \               |
expr.t = 9    -    term.t = 5        2
    |                  |
term.t = 9             5
    |
    9                                  9-5+2
```

t is the attribute of *expr* and *term*.

Annotated parse tree: postfix

Syntax-Directed Definition for Infix to Postfix Translation

Formalization of the definition of postfix expression



Annotated parse tree

Production	Semantic Rules
expr → $expr_1$ + term	expr.t = $expr_1$.t || term.t || "+"
expr → $expr_1$ - term	expr.t = $expr_1$.t || term.t || "-"
expr → term	expr.t = term.t
term → 0	term.t = '0'
term → 1	term.t = '1'
…	…
term → 9	term.t = '9'

|| : String concatenation
→ Attach strings as attributes

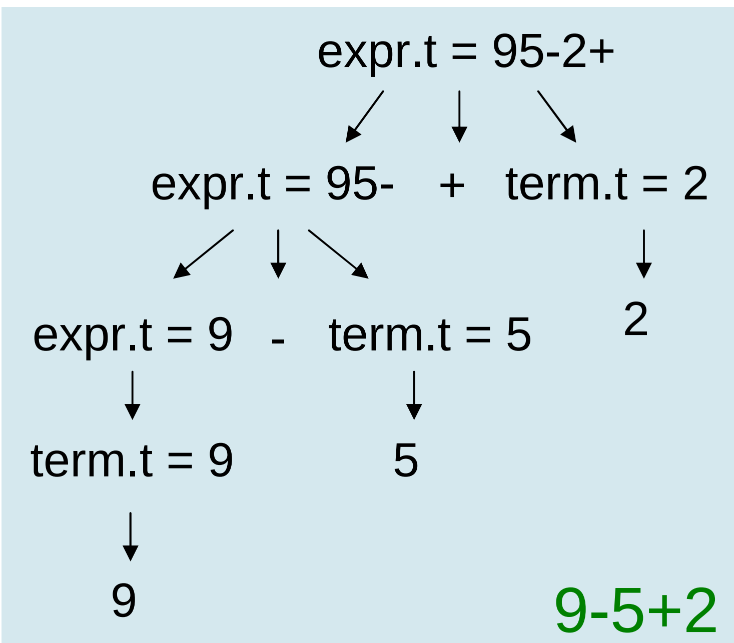


Tree Traversals

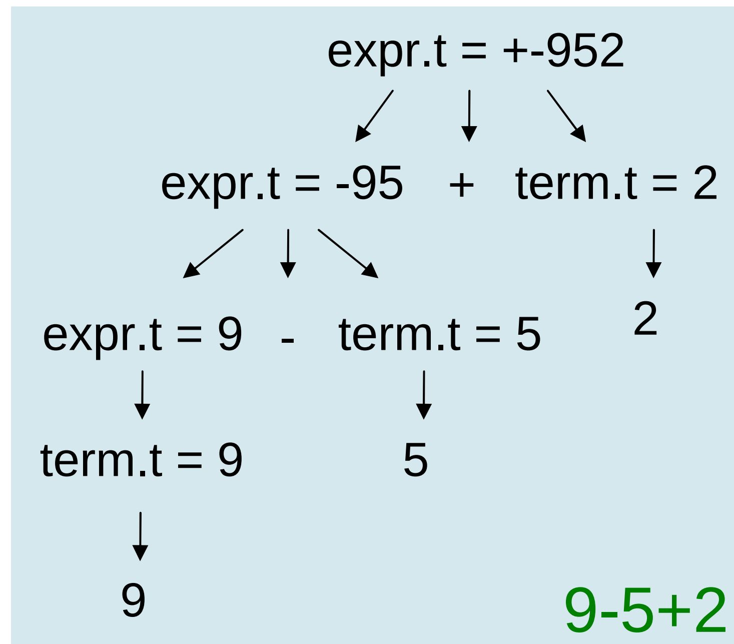
- Tree traversals are used
  - for describing attribute evaluation and
  - for specifying the execution of code fragments in a translation scheme.
- A tree traversal starts at the root and visits each node of the tree in the same order.
  - A depth-first traversal starts at the root and recursively visits the children of each node in any order (not necessary from left to right).
  - Synthesized attributes can be evaluated during any bottom-up traversal.
    - i.e., attributes of a node can only be evaluated after the attributes of its children are evaluated.

Postorder and Preorder Traversal

- If we traverse a tree by visiting the children of each node of a tree from left to right,
  - Postorder: the action of the node is done when we leave the node.
  - Preorder: the action of the node is done when we first visit the node.



Annotated parse tree: postorder

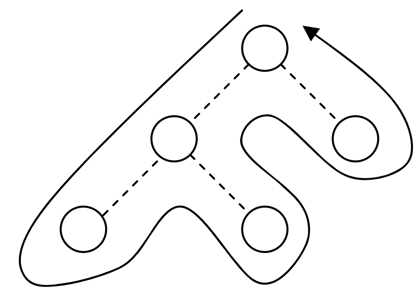


Annotated parse tree: preorder

Tree Traversals (Cont.)

- An example of a depth-first traversal

```
Procedure visit(node N) {
    for (each child C of N, from left to right) {
        visit(C);
    }
    evaluate semantic rules at node N;
}
```



Postorder traversal

```
Procedure visit(node N) {
    evaluate semantic rules at node N;
    for (each child C of N, from left to right) {
        visit(C);
    }
}
```

Preorder traversal

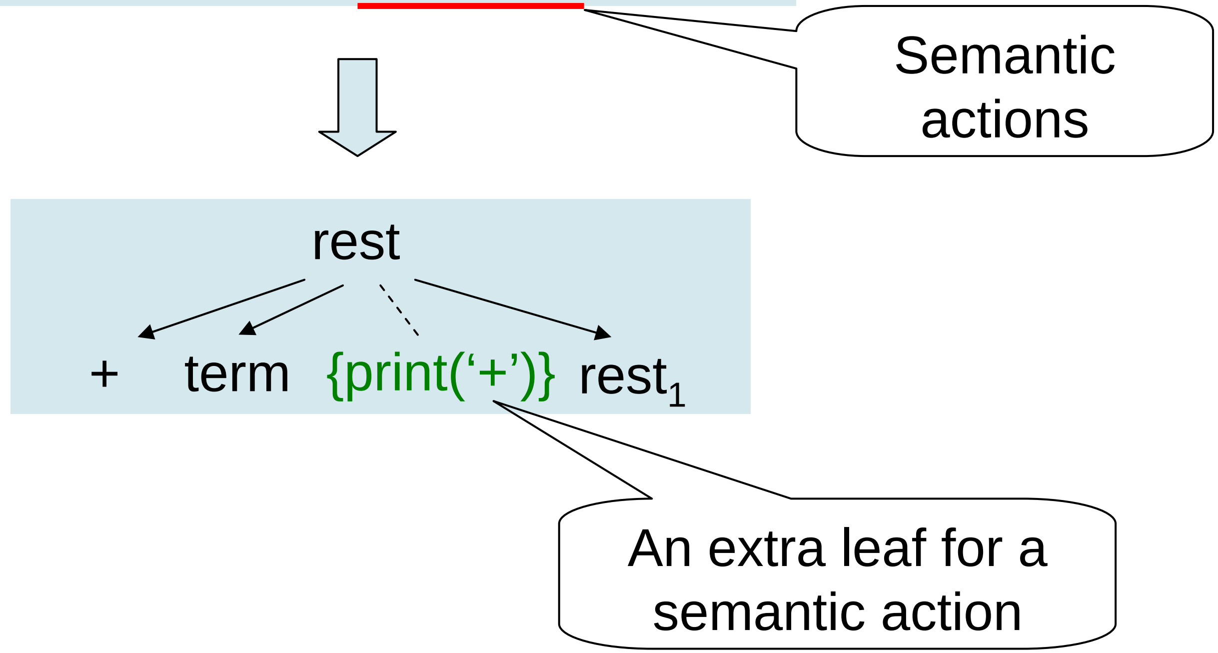


Translation Schemes

- A syntax-directed translation scheme is to attach program fragments to productions in a grammar.
  - Similar to a syntax-directed definition (syntax definition), except that the order of evaluation of the semantic rules is explicitly specified.
- A syntax-directed translation scheme often serves as the specification for a translator.

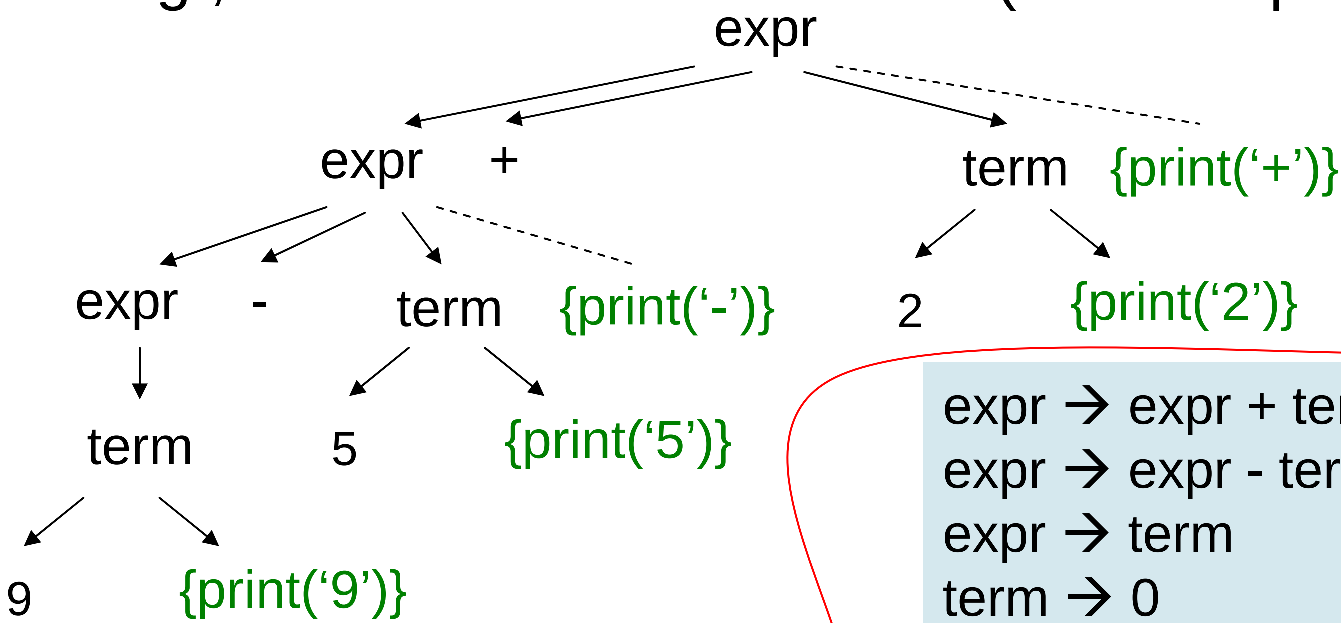
Semantic Actions

- Semantic actions are program fragments embedded within production bodies. (encoded in { })
  - E.g., Rest → + term {print('+')} $rest_1$



Semantic Actions (Cont.)

- E.g., translate 9-5+2 into 95-2+ (infix into postfix)



Parse tree with semantic actions in a postorder traversal.

expr → expr + term	{printf('+')}
expr → expr - term	{printf('-')}
expr → term	
term → 0	{printf('0')}
term → 1	{printf('1')}
…	…
term → 9	{printf('9')}

Translation scheme with semantic actions into postfix notation
→ print the translation incrementally



Semantic Actions (Cont.)

- Semantic actions (the implementation of a translation scheme)
  - Should be performed in the order they would appear during tree traversal.
  - Need not actually construct a parse tree.
  - Need not any storage for the translation of subpexressions.



Parsing



Parsing

- Parsing is the process of determining how a string of terminals can be generated by a grammar.
  - A parser doesn't need to construct a parse tree, but should be able to construct a parse tree so as to guarantee the correctness of the translation.
  - Parsers almost make a single left-to-right scan over the input, looking ahead one terminal at a time (to construct the parse tree).
- Time complexity
  - For any context-free grammar, there is a parser that takes at most $O(n^3)$ to parse a string of n terminals.
  - In general, linear time algorithms suffice to parse essentially all languages in practice.

Parsing Methods

- Two parsing classes:
  - Top-down method (by hand-designed parsers):
    - Constructions start at the root and proceeds towards the leaves.
    - Efficient parsers can be constructed more easily.
  - Bottom-up method (preferred by software generated parsers):
    - Constructions start at the leaves toward the root.
    - A larger classes of grammars and translation schemes can be handled with software tools.

Top-Down Parsing

- Start with the root, and repeatedly perform the following two steps:
  1. At node N (labeled with nonterminal A),
     1. Select one of the productions for A and
     2. Construct children at N for the symbols in the production body.
  2. Find the next node at which a subtree is to be constructed.
- The current terminal being scanned in the input is referred to as the lookahead symbol.

An Example of Top-Down Parsing (Cont.)

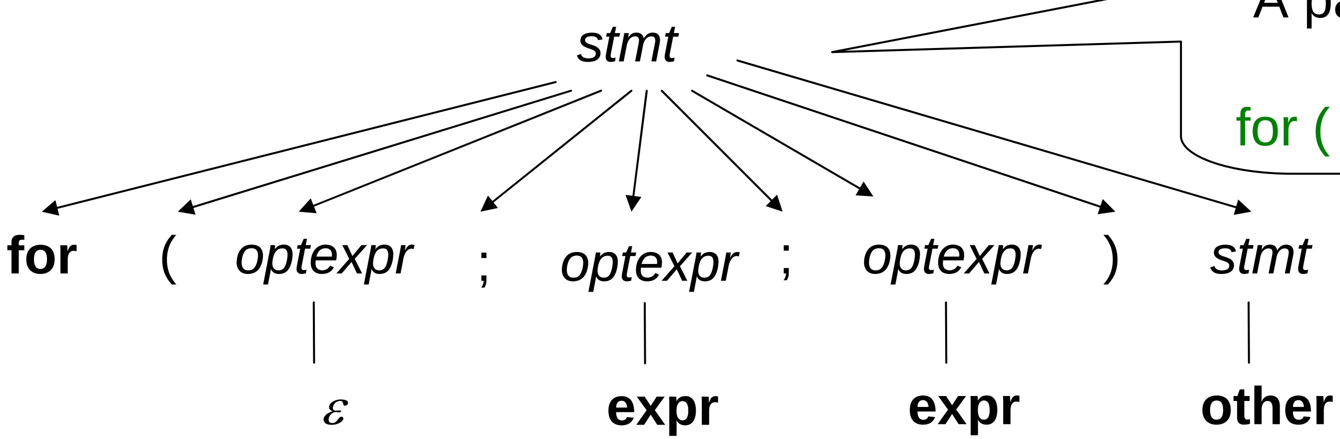
stmt → **expr** ;
  | **if** (**expr**) *stmt*
  | **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*
  | **other**

optexpr → ε
  | expr

A grammar for some statements in C and Java

ε → epsilon (e in Greek)
→ empty (or null)

A parse tree of the **for** statement:
for (; expr ; expr) other



An Example of Top-Down Parsing (Cont.)

(a)

PARSE TREE: stmt

INPUT: **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*

(b)

PARSE TREE: *stmt* → **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*

INPUT: **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*

Match **for**: expend the parse tree

(c)

PARSE TREE: *stmt* → **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*

INPUT: **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*

Advance to the next child

Predictive Parsing

- The problems of top-down parsing:
  - The selection of a production for a nonterminal may involve trial-and-error (heuristic method).
  - Backtracking is needed if a selected production is unsuitable.
- Predictive parsing
  - Is a recursive-descent parsing (a top-down method), in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.
  - Relies on information about the first symbols that can be generated by a production body.
  - Consists of a procedure for every nonterminal.



Pseudocode for a Predictive Parser

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

Global variable

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

nonterminal

FIRST(*stmt*) = {**expr**, **if**, **for**, **other**}
FIRST(**expr;**) = {**expr**}

terminal

Define FIRST(α) to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from α.

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

stmt → **expr** ;
  | **if** (**expr**) *stmt*
  | **for** (*optexpr* ; *optexpr* ; *optexpr*) *stmt*
  | **other**
optexpr → ε
  | expr

Grammar

E.g., **for** (; **expr** ; **expr**) **other**



FIRST(α)

- Define FIRST(α) to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from α.
  - If α begins with a terminal, the terminal is the only symbol in FIRST(α).
    - E.g., FIRST(**expr ;**) = {**expr**}
  - If α begins with a nonterminal, the first terminal in each body of its productions is in FIRST(α).
    - E.g., FIRST(*stmt*) = {**expr**, **if**, **for**, **other**}
  - If α is ε or can generate ε, then ε is also in FIRST(α).

Predictive Parser Design

- The procedure of a predictive parser for a nonterminal *A* does two things:
  - First decide which A-production to use by examining the lookahead symbol.
    - The production with body α is used if the lookahead symbol is in FIRST(α).
    - If the lookahead symbol is not in the FIRST set for any production body for A, the ε-production (for A) is used.
  - Then mimic the body of the chosen production.
    - A nonterminal is executed by a call to the procedure for that nonterminal.
      - A terminal matching the lookahead symbol is executed by reading the next input symbol.
      - If the terminal in the body of the matched production doesn't match the lookahead symbol, a syntax error is reported.



Left Recursion

- A recursive-descent parser might loop forever due to the "left-recursive" productions.
  - E.g., the leftmost symbol is the same as the nonterminal at the head of the production.

$$expr \rightarrow expr + term$$

  - The lookahead symbol changes only when a terminal in the body is matched, so that the call to *expr* might loop forever.
- Left recursive productions lead the tree growing down the left.

Left Recursion (Cont.)

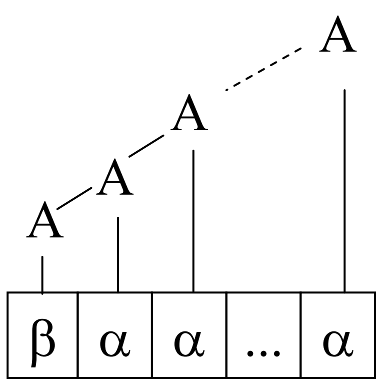
- The way to prevent loop-forever in left recursion:
  - Consider a nonterminal A with two productions:

$$A \rightarrow A\alpha \mid \beta$$

    - If A = *expr*, string α = + *term*, and string β = *term*, then

$$expr \rightarrow expr + term \mid term$$

    - When A is finally replaced by β, we have a β followed by a sequence of zero or more α's.



Right Recursion

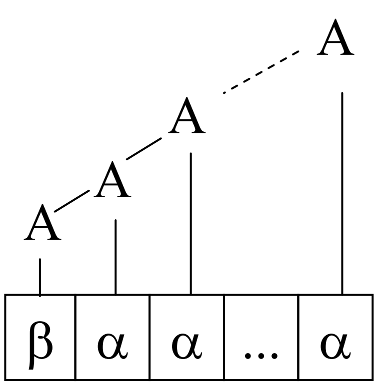
- Right recursive productions lead the tree growing down the right.

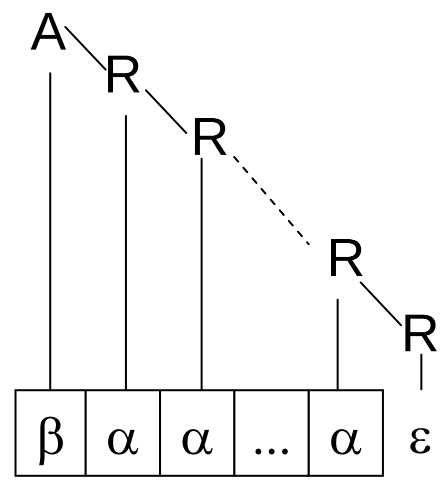
$A \rightarrow A\alpha \mid \beta$

Left recursion to right recursion

Left-recursion elimination

$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \varepsilon$





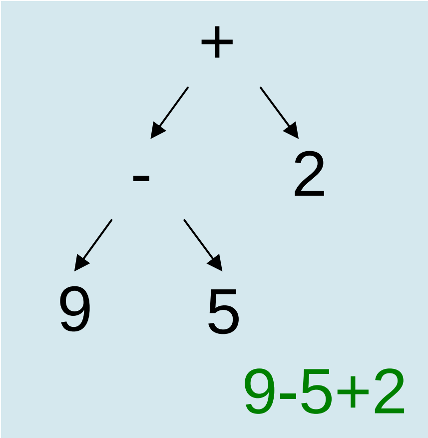


A Translator for Simple Expressions



Abstract and Concrete Syntax Trees

- Abstract syntax tree (Syntax tree)
  - Each interior node represents an operation.
  - Children of the node represent the operands of the operator.
  - No helper nodes (e.g., *factor*, *term*) for single productions are needed.
- Concrete syntax tree (Parse tree)
  - Each interior node represent a nonterminal.
  - Many nonterminals represent programming construct, but others are “helpers.”
  - The underlying grammar for the parse tree is called a concrete syntax.



Syntax tree

Single production: a production whose body consists of a single nonterminal. (e.g., “*expr* → *term*” is a single production)

Left Recursion Elimination

A = expr, α = + term, β = - term, γ = term

$A \rightarrow A\alpha \mid A\beta \mid \gamma$

Left recursion

Left recursion elimination

$A \rightarrow \gamma R$

$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$

right recursion

```
expr → expr + term     {printf('+')}
     | expr - term     {printf('-')}
     | term
term → 0               {printf('0')}
     | 1               {printf('1')}
       …               …
     | 9               {printf('9')}
```

Actions to translate into postfix notation

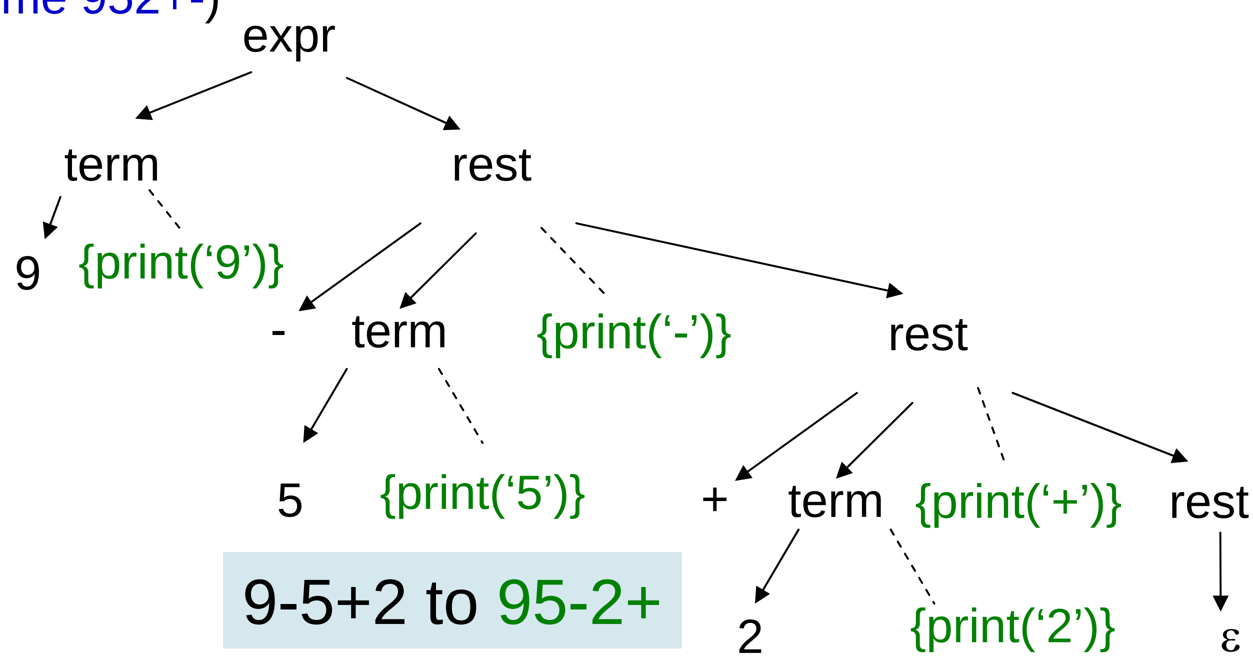
```
expr → term rest
rest → + term {print('+')} rest
     | - term {print('-')} rest
     | ε
term → 0 {print('0')}
     | 1 {print('1')}
       …
     | 9 {print('9')}
```

A = expr
R = rest
α = + term {print('+')}
β = - term {print('-')}
γ = term



Left Recursion Elimination (Cont.)

- Left-recursion elimination must be done carefully to ensure the order of semantic actions.
  - E.g., actions {print('+')} and {print('-')} in the middle of a production body
    - If the print actions are moved to the end, the translation would be incorrect. (9-5+2 would become 952+-)



9-5+2 to 95-2+

expr → term rest
rest → + term {print('+')} rest
  | - term {print('-')} rest
  | ε
term → 0 {print('0')}
  | 1 {print('1')}
  …
  | 9 {print('9')}

Translation scheme

Procedure for the Nonterminals

```
void expr () {
    term(); rest();
}
```

```
void rest () {
    if (lookahead == '+') {
        match('+'); term(); print('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); print('-'); rest();
    }
    else { } // do nothing with the input
}
```

Tail recursive

Tail recursive

expr → term rest
rest → + term {print('+')} rest
    | - term {print('-')} rest
    | ε
term → 0 {print('0')}
    | 1 {print('1')}
    …
    | 9 {print('9')}

Translation scheme

```
void term () {
    if (lookahead is a digit) {
        t = lookahead; match(lookahead); print('t');
    }
    else report("syntax error");
}
```

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

Procedure for the nonterminals

Translation Simplification

- When expressions with multiple levels of precedence are translated, simplifications could reduce the number of needed procedures.
  - Tail recursion can be replaced by iterations.
    - Tail recursion is when the last statement executed in a procedure body is a recursive call to the same procedure.

Tail recursive

```
void rest () {
    if (lookahead == '+') {
        match('+'); term(); print('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); print('-'); rest();
    }
    else { } // do nothing with the input
}
```

Tail recursion elimination

```
void rest () {
    while (true) {
        if (lookahead == '+') {
            match('+'); term(); print('+'); continue;
        }
        else if (lookahead == '-') {
            match('-'); term(); print('-'); continue;
        }
        break;  // break out of the while loop
    }
}
```



Translation Simplification (Cont.)

```
void expr () {
    term(); rest();
}
```

```
void rest () {
    if (lookahead == '+') {
        match('+'); term(); print('+'); rest();
    }
    else if (lookahead == '-') {
        match('-'); term(); print('-'); rest();
    }
    else { } // do nothing with the input
}
```

Tail recursive

Tail recursion elimination

```
void rest () {
    while (true) {
        if (lookahead == '+') {
            match('+'); term(); print('+'); continue;
        }
        else if (lookahead == '-') {
            match('-'); term(); print('-'); continue;
        }
        break;  // break out of the while loop
    }
}
```

Merge *expr()* and *rest()*

```
void expr () {
    term();
    while (true) {
        if (lookahead == '+') {
            match('+'); term(); print('+'); continue;
        }
        else if (lookahead == '-') {
            match('-'); term(); print('-'); continue;
        }
        break;  // break out of the while loop
    }
}
```

An Infix-to-Postfix Translator (in Java)

```
Import java.io.*; // include the IO package
Class Parser { // in file Parser.java
    static int lookahead;

    public Parser() throws IOException{ //constructor
        lookahead = System.in.read(); //read first char
    }

    void expr() throws IOException {
        term();
        while (true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if ( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }
```

```
public class Postfix { // in file Postfix.java
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
}
```

Entry function

```
    void term() throws IOException {
        if ( Character.isDigit( (char)lookahead) ) {
            System.out.write ( (char)lookahead );
            match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match (int t) throws IOException {
        if (lookahead == t) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}
```

Exception occurs when no input to be read.

Read next char / symbol



Lexical Analysis



Lexical Analyzer

- Lexical analyzer read characters from the input and groups them into "token objects."
  - A token object is a terminal symbol (for parsing decision) with additional information in the form of attribute values.
  - A sequence of input characters that comprises a single token is called a lexeme.
- Assumption
  - The lexical analyzer allows numbers, identifiers, and "white space."
  - Attribute
    - **num**.value: integer value
    - **id**.lexeme: string for its name

```
expr → expr + term      { print('+') }
     |  expr – term     { print('-') }
     |  term
term → term * factor    { print('*') }
     |  term / factor   { print('/') }
     |  factor
term → (expr)
     |  num             { print(num.value) }
     |  id              { print(id.lexeme) }
```



Removal of White Space and Comments

- Most languages
  - Allow arbitrary amounts of white space
    - While space includes blank, tab, newline.
  - Ignore comments during parsing
  - Show line numbers and context within error messages.

```
for ( ; ; peek = next input character) {
    if (peek is a blank or a tab) do nothing;
    else if ( peek is a newline) line = line + 1;
    else break;
}
```



Reading Ahead

- Lexical analyzers might need to read ahead some characters before deciding a token.
  - E.g., when the character > is seen:
    - The lexeme for the token might be >= or >.
  - One-character read-ahead usually suffices (but not always).
    - Suppose that the read-ahead character is stored in variable *peek* that is blank if the read-ahead character (e.g., *) is not necessary.
- Input buffer
  - A general approach is to maintain an input buffer for the lexical analyzer to read and push back characters.
  - It is usually more efficient to fetch a block of characters instead of reading a character at a time.



Constants

- Arbitrary integer constants
  - When a sequence of digits appears in the input stream, the lexical analyzer passes a token to the parser.
    - The token consists of the terminal **num** along with an integer-valued attribute computed from the digits.
    - E.g., The input 31 + 28 + 59 is transformed into <num, 31> <+> <num, 28> <+> <num, 59>

```
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>;
}
```



Recognizing Keywords and Identifiers

- Difference between keywords and identifiers:
  - Keywords:
    - Are character strings to identify programming constructs.
    - E.g., for, do, if
  - Identifiers:
    - Are character strings to name variables, arrays, function, and the like.
    - Treated as terminals to simplify the parser.
- A mechanism is needed for deciding whether a lexeme forms a keyword or an identifier.



Recognizing Keywords and Identifiers (Cont.)

- E.g.,
  - The input:
    - count = count + increment;
  - The parser considers the input as:
    - id = id + id;
  - The token for id has an attribute that holds the lexeme. Write tokens as tuples:
    - <id, “count”> <=> <id, “count”> <+> <id, “increment”> <;>



Recognizing Keywords and Identifiers (Cont.)

- One solution to recognize keywords and identifiers is to maintain a table to hold character strings. It solves two problems:
  - Single representation:
    - A string table can insulate the rest of the compiler from the representation of strings.
    - The compiler can work with references or pointers to the strings in the string table because references can be manipulated more efficiently.
  - Reserved words:
    - Reserved words can be implemented by initializing the string table with the reserved strings and their tokens.
    - When the lexical analyzer reads a string or lexeme, it checks whether the lexeme is in the string table. If so, it returns the token; otherwise, it returns a token with terminal **id**.



Recognizing Keywords and Identifiers (Cont.)

- An example with Java:
  - Create a hash table as the string table

```
Hashtable words = new Hashtable();
```

  - Distinguish keywords and identifiers (pseudocode)

```
if ( peek holds a letter) {
    Collect letters or digits into a buffer b;  // collect a string beginning with a letter
    s = string formed from the characters in b; // put the collected string to s as a lexeme
    w = token returned by words.get(s); // check the string table
    if (w is not null) return w; // the token for lexeme s exists
    else {
        Enter key-value pair (s, <id, s>) into words; // put the s (as the key) to the table as a new token
        return token <id, s>; // return the newly created token for lexeme s.
    }
}
```



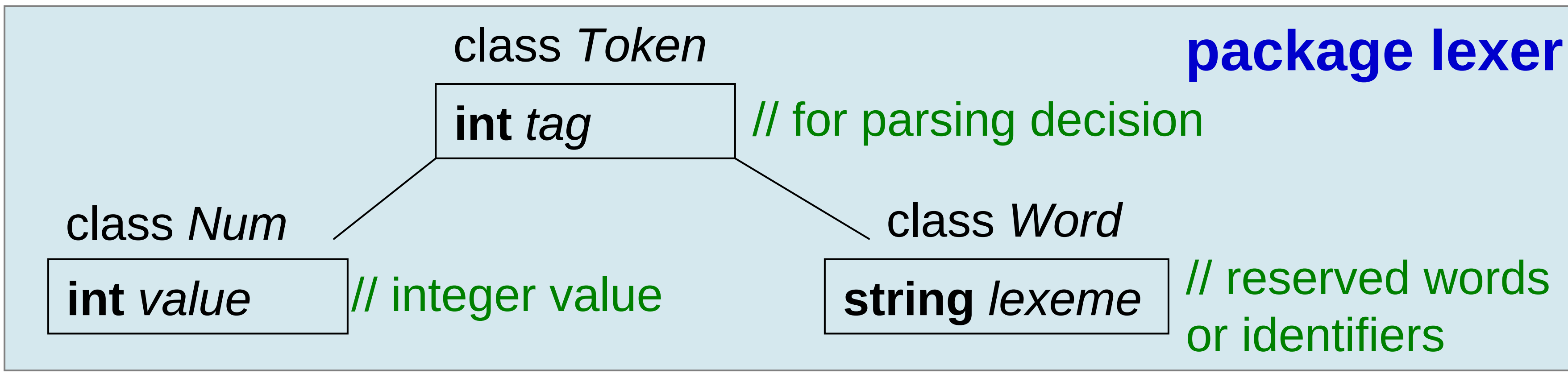
Token Scanner

- An example of the token scanner is as follows (pseudocode):

```
Token scan() {
    Skip white space;
    Handle numbers;
    Handle reserved words and identifiers;
    // if we get here, treat read-ahead character peek as a token
    Token t = new Token(peek); // might be an operator or others
    peek = blank; // initialization
    return t;
}
```



Token Scanner in Java



Token Scanner in Java (Cont.)

Identify package

“final” can’t be changed once it is set.

```
package lexer; // file Token.java
public class Token {
    public final int tag;
    public Token (int t) { tag = t; }
}                    // t = ‘+’;
```

```
package lexer;  // file Tag.java
public class Tag {
    public final static int
        NUM = 256, ID = 257;
        TRUE = 258, FALSE = 259;
}
```

Constructor:
e.g., **new Token(‘+’);**

Constants. Equal to #define NUM 256 in C
0~255 are reserved for ASCII (e.g., operator *, +)

```
package lexer; // file Num.java
public class Num extends Token {
    public final int value;
    public Num (int v) {
        super(Tag.NUM);
        value = v; // 30
    }
}
```

Calls the constructor of its parent

e.g., **new Num(30);**

```
package lexer; // file Word.java
public class Word extends Token {
    public final String lexeme;
    public Word (int t, String s) {
        super(t); // setup tag value, t = 258
        lexeme = new String(s); //s = “true”
    }
}
```

For keywords and identifiers
e.g., **new Word(Tag.TRUE, “true”);**

Token Scanner in Java (Cont.)

```
package lexer; // file Token.java
Import java.io,*, import java.util.*;
public class Lexer {
    public int line = 1; // initialize line counts
    private char peek = ' '; // initialize peek
    private Hashtable word = new Hashtable();
    void reserve(Word t) { words.put(t.lexeme, t); }
    public Lexer() {
        reserve( new Word(Tage.TRUE, "true");
        reserve( new Word(Tage.FALSE, "false");
    }
    public Token scan() throws IOException {
        for ( ; ; peek = (char)System.in.read() ) {
            if (peek == ' ' || peek == '\t') continue;
            else if ( peek == '\n' ) line = line + 1;
            else break;
        }
}
```

```
if( Character.isDigit(peek) ) {
    int v = 0;
    do {
        v = 10*v + Character.digit(peek. 10);
        peek = (char)System.in.read();
    } while (Character.isDigit(peek) );
    return new Num(v);
}
if ( Character.isLetter(peek) ) {
    StringBuffer b = new StringBuffer();
    do {
        b.append(peek);
        peek = (char)System.in.read();
    } while (Character.isLetterOrDigit(peek) );
    String s = b.toString();
    Word w = (Word)words.get(s);
    if (w != null) return w;
    w = new Word(Tag.ID, s); //add new word
    words.put(s, w);
    return w;
}
Token t = new Token(peek);
peek = ' '; // Read-ahead is not necessary
return t;
```

Handle numbers

Handle reserved words and identifiers

Look up the string table

Count line number

For reserved words

Skip white space

not reserved word, identifiers, white space, or numbers



Symbol Tables



Symbol Tables

- Symbol tables are data structures to hold information about source-program constructs.
  - Collected incrementally by the analysis phase
  - Used by the synthesis phases to generate the target code.
- Symbol tables typically need to
  - Support multiple declarations of the same identifier.
  - Separate a table for each scope. E.g.,
    - A program block with declarations has its own symbol table with an entry for each declaration in the block.
    - E.g., A class would have its own table with any entry for each field and method.
- Entries in symbol tables
  - Contain information about an identifier, e.g., its lexeme, type, position in storage, and any other relevant information.

Sample Program

- E.g., { int x; char y; { bool y; x; y; } x; y; }

A use of the identifier (simplified version)

A definition of the identifier

The goal is to remove the declarations, and to show each statement with an identifier followed by a colon and its type.

{ { x:int; y: bool; } x:int; y:char; }

Reference outer x

Reference inner y

Reference outer x

Reference outer y



Symbol Table Per Scope

- Scopes are important.
  - The same identifier can be declared multiple times.
  - Common names like i and x often have multiple uses.
  - Subclasses can redeclare a method name to override a method in a superclass.
- E.g., block → '{' *decls stmts* '}'
  - If *stmts* can generate a block, then nested blocks can be created and an identifier could be redeclared.



Most-Closely Nested Rule

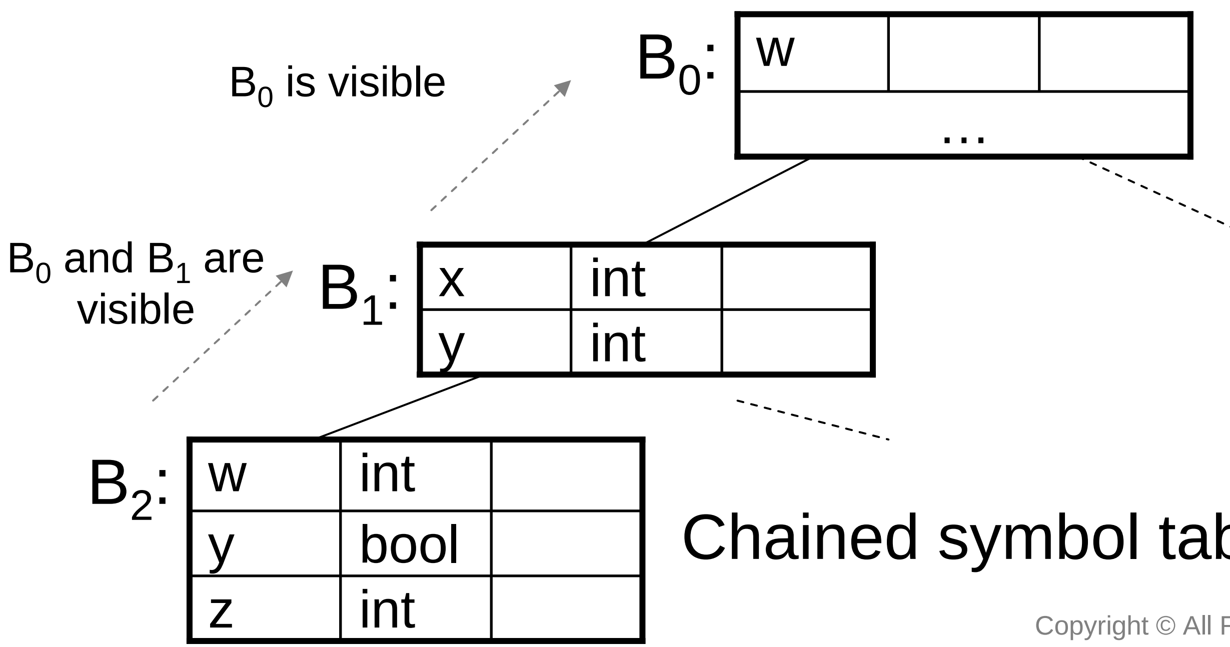
- The most-closely nested rule:
  - An identifier x is in the scope of the most-closely nested declaration of x.
    - i.e., the declaration of *x* found by examining blocks inside-out, starting with the block where *x* appears.
  - This rule can be implemented by chaining symbol tables.
    - That is, the table for a nested block points to the table for its enclosing block.

1) { int x_1; int y_1;
2) { int w_2; bool y_2; int z_2;
3) … w_2 …; … x_1 …; … y_2 …; … z_2 …;
4) }
5) … w_0 …; … x_1 …; … y_1 …;
6) }

The subscript is the line number of the declaration.

Most-Closely Nested Rule (Cont.)

1) { int x_1; int y_1;
2) { int w_2; bool y_2; int z_2;
3) … w_2 …; … x_1 …; … y_2 …; … z_2 …;
4) }
5) … w_0 …; … x_1 …; … y_1 …;
6) }



Chained symbol tables (form a tree)



An Example of Chained Symbol Tables in Java

```
package symbols;
import java.util.*;
Public class Env {
    private Hashtable table;
    protected Env prev;

    public Env (Env p) { // constructor
        table = new Hashtable(); // create a new symbol table
        prev = p;  // point to the previous (above) Env object
    }
    public void put (String s, Symbol sym) {
        table.put(s, sym);
    }
    public Symbol get (String s) {
        for (Env e = this; e != null; e = e.prev) {
            Symbol found = (Symbol) (e.table.get(s));
            if (found != null) return found;
        }
        return null;
    }
}
```

Constructor: reate a hash table with an parameter pointing to the previous Env object

Put a symbol to the symbol table

s: key
sym: value

Search the chained tables for the entry of an identifier



The Use of Symbol Tables

- The role of a symbol table is to pass information from declarations to uses.
  – A semantic action "puts" information about identifier *x* into the symbol table when the declaration of *x* is analyzed.
  – Then, a semantic action associated with a production such as factor → id "gets" information about the identifier from the symbol table.



The Use of Symbol Tables (Cont.)

Grammar		Semantic Action
program	→ block	{ top = null;}
block	→ '{'	{ saved = top; top = new Env(top); **print("{");** }
	decls stmts '}'	{ top = saved; **print("} ");** }
decls	→ decls decl | ε	
decl	→ type id;	{ s = **new** Symbol; s.type = **type**.lexeme; top.put(**id**.lexeme, s); }
stmts	→ stmts stmt | ε	
stmt	→ block	
	| factor;	{ **print("; ");** }
factor	→ id	{ s = top.get(**id**.lexeme); **print(id.lexeme); print(":"); print(s.type);** }

Top table

{ int x; char y; { bool y; x; y; } x; y; }

{ { x:int; y: bool; } x:int; y:char; }

Save a reference to the current table with the local variable ***saved***

Create a new table, and set the variable ***top*** to the newly created and chained table

Restore top (i.e., pup up the top table)

Put a new declaration (identifier) with its type into the table

Use the chained symbol tables to get the entry for the identifier

The translation scheme creates and discards symbol tables upon block entry and exit, respectively.



Intermediate Code Generation



Intermediate Representations

- Two kinds of intermediate representations
  - Trees, including parse trees and (abstract) syntax trees.
    - Syntax-tree nodes are created to represent significant programming constructs.
    - As analysis proceeds, information is added to the nodes in the form of attributes.
      - The choice of attributes depends on the translation to be performed.
  - Linear representations, especially "three-address code."
    - Three-address code
      - Is a sequence of elementary program steps without hierarchical structure.
      - Is helpful for significant code optimization.
    - The sequence of three-address statements forms a program into "basic blocks".
      - Statements in a basic block are executed one-after-the-other without branching.

Construction of Syntax Trees

- Syntax trees can be created for any construct.
  - Each construct is represented by a node with children for the semantically meaningful components of the construct.
  - E.g., Syntax tree construction with Java
    - Each node is implemented as objects of class ***Node***.
    - Class ***Node*** has two immediate subclasses:
      - ***Expr*** for all kinds of expressions.
      - ***Stmt*** for all kinds of statements.
        - » Each type of statement has a corresponding subclass of *Stmt*.
        - » E.g., operator while corresponds to subclass ***While***, where *While* is a subclass of *Stmt*.

while (*expr*) *stmt*

$\Updownarrow$

while
$\swarrow \quad \searrow$
expr *stmt*

op
$\swarrow \quad \searrow$
E_1 E_2

new *While* (*x*, *y*) → The constructor corresponds to the operator *While*.
→ The parameters x and y corresponds to the operands.



Syntax Trees for Statements

- For each statement construct, we define an operator in the abstract syntax.
  - For constructs beginning with a keyword, we should use the keyword for the operator.
    - An operator **while** for while statements
    - An operator **do** for do-while statements
    - Operators **ifelse** and **if** for if-statements with and without an else part, respectively.
  - Each statement operator has a corresponding class of the same name.
    - E.g., class ***If*** corresponds to **if**.
      class ***Seq*** represents a sequence of statements.



Syntax Trees for Statements (Cont.)

- An example of the construction of syntax tree nodes

  stmt → if (*expr*) *stmt*$_1$ { *stmt*.*n* = **new** If(*expr*.*n*, *stmt*$_1$.*n*); }

  Each nonterminal in this translation scheme has an *attribute* ***n***.

  – The semantic action
    - Defines the node *stmt*.*n* as a new object of subclass *If*.
    - Creates a new node labeled **if** with the nodes *expr*.*n* and *stmt*$_1$.*n* as children.

  – Expression statements do not begin with a keyword.
    - An operator **eval** and class *Eval* (a subclass of *Stmt*) to represent expressions that are statements.
    - E.g., *stmt* → *expr* ; { *stmt*.*n* = **new** *Eval*(*expr*.*n*); }

Representing Blocks in Syntax Trees

- An example of blocks in syntax trees:

stmt → *block* ;	{ *stmt.n* = *block.n*; }
block → '{' *stmts* '}'	{ *block.n* = *stmts.n*; }

The syntax tree for nonterminal *block* is simply the syntax tree for the sequence of statements in the block.

When a statement is a block, it has the same syntax tree as the block.

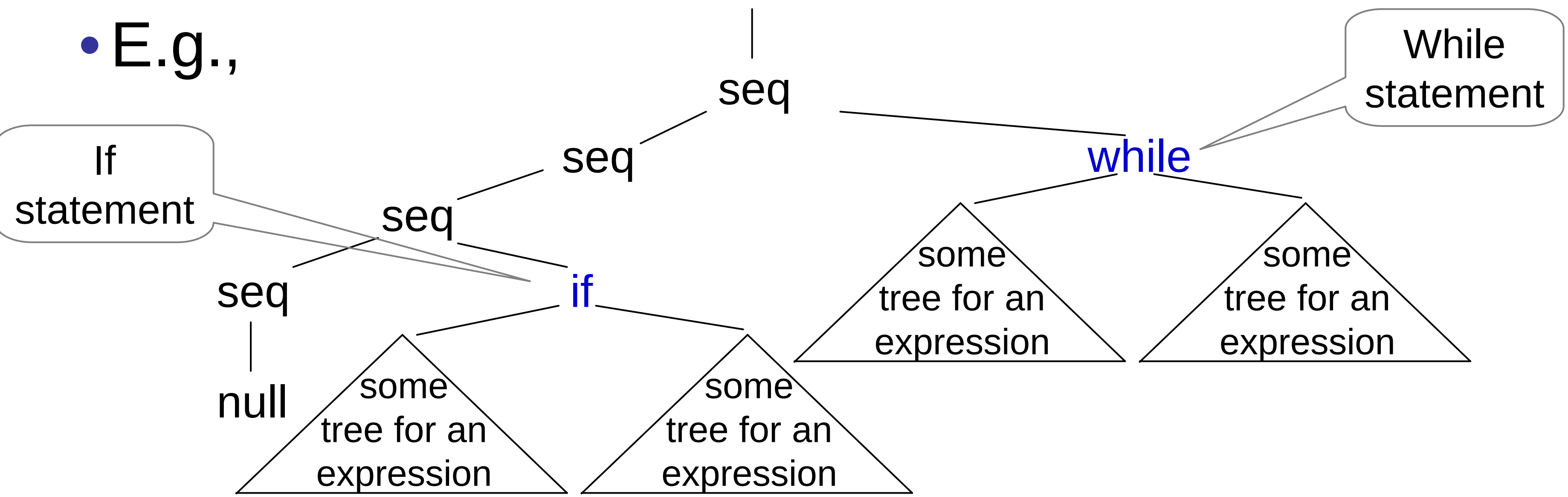
  - Information from declarations is incorporated into the symbol table, so that declarations are not in the syntax tree.
  - Blocks, w/wo declarations, appear to be just another statement construct in intermediate code.

Sequence of Statements

- A sequence of statements is represented by using
  - A leaf **null** for an empty statement
  - An operator **seq** for a sequence of statements

$stmts \rightarrow stmts_1\ stmt \quad \{\ stmts.n = \mathbf{new}\ Seq(stmts_1.n,\ stmt.n);\ \}$

- E.g.,



Syntax Trees for Expressions

- Grouping of operators
  - To reduce the number of cases and subclasses of nodes in implementation

Concrete Syntax	Abstract Syntax
=	assign
||	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
-(unary)	minus
[ ]	access

Increasing precedence

  - E.g., $term \rightarrow term_1 * factor$ { $term.n$ = **new** Op('*', $term_1.n$, $factor.n$); }

    → Create a node of class **Op** that implements the operators grouped under op.



Translation Scheme for Construction of Syntax Trees

program	→ block	{ return block.n;}
block	'{' stmts '}'	{ block.n = stmts.n; }
stmts	→ stmts$_1$ stmt	{ stmts.n = **new** Seq (stmts$_1$.n, stmt.n); }
	| ε	{ stmts.n = **null**; }
stmt	→ expr ;	{ stmt.n = **new** Eval (expr.n); }
	| **if** (expr) stmt$_1$	{ stmt.n = **new** If (expr.n, stmt$_1$.n); }
	| **while** (expr) stmt$_1$	{ stmt.n = **new** While (expr.n, stmt$_1$.n); }
	| **do** stmt$_1$ **while** (expr) ;	{ stmt.n = **new** Do (stmt$_1$.n, expr.n); }
	| block	{ stmt.n = block.n; }
expr	→ rel = expr$_1$	{ expr.n = **new** Assign ('=', rel.n, expr$_1$.n); }
	| rel	{ expr.n = rel.n; }
rel	→ rel$_1$ < add	{ rel.n = **new** Rel ('<', rel$_1$.n, add.n); }
	| rel$_1$ <= add	{ rel.n = **new** Rel ('<=', rel1.n, add.n); }
	| add	{ rel.n = add.n; }
add	→ add$_1$ + term	{ add.n = **new** Op ('+', add$_1$.n, term.n); }
	| term	{ add.n = term.n; }
term	→ term$_1$ * factor	{ term.n = **new** Op ('*', term$_1$.n, factor.n); }
	| factor	{ term.n = factor.n; }
factor	→ (expr)	{ factor.n = expr.n; }
	| **num**	{ factor.n = **new** Num (**num**.value); }



Static Checking

- Static checks are consistency checks and includes:
  - Syntactic checking:
    - Check syntactic constraints that are not part of grammar, e.g.,
      - An identifier can be declared at most once in a scope.
      - A break statement must have an enclosing loop or switch statement.
  - Type checking:
    - Ensure that an operator or function is applied to the right number and type of operands, e.g.,
      - When an integer is added to a float, the type-checker can insert an operator in the syntax tree to represent the type conversion (**coercion**).
- Complex static checks may need to be done by first constructing an intermediate representation.

L-values and R-values

- Differences
  - L-value refers to location that are appropriate on the left side of an assignment.
  - R-value refers to values that are appropriate on the right side of an assignment.

```
i = 5;
i = i + 1;
```

L-value: where to be stored

R-value: what's the value



Type Checking

- Type checking assures that the type of a construct matches the expected type.
  - E.g., **if** (*expr*) *stmt* (*expr* is expected to have type boolean.)
- Type checking rules follow the operator / operand structure.
  - E.g., the operator **rel** represents relational operators, such as <=.
    - The type rule for the relational operator is to have two operands with the same type and to have the result with type boolean.

if (E_1.type == E_2. type) E.type = boolean;
else error;



Type Checking (Cont.)

- Type matching continues to apply even in the following situations:
  - Coercions:
    - The type of an operand is automatically converted.
      - E.g., 2 * 3.14 → the integer 2 is converted into 2.0
    - The language definition specifies the allowable coercions.
  - Overloading:
    - A symbol is *overloaded* if it has different meanings depending on its context.
      - E.g., a = “b” + “c”; // string concatenation
      - a = 2 + 3; // integer addition

Three-Address Instruction

- Once syntax trees are constructed, the tree-address code could be generated by walking the syntax trees.
  - Three-address instructions

    x = y **op** z

    - x, y, and z are names, constants, or compiler-generated temporaries.
    - **op** stands for an operator.
    - E.g., x[y] = z → put the value of z in the location x[y].

      x = y[z] → put the value of y[z] in the location x.
    - Flow control of the three-address instructions

      ifFalse x goto L → If x is false, next execute the instruction labeled L.

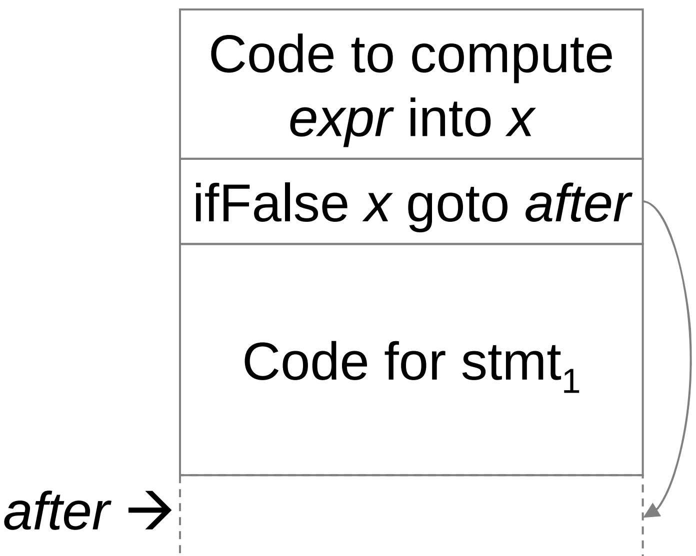
      ifTrue x goto L → If x is true, next execute the instruction labeled L.

      goto L → next execute the instruction labeled L

      A label L can be attached to any instruction by prepending a prefix L:
    - Copy a value: x = y

Translation of Statements

- Statements are translated into three-address code by using jump instructions to control the flow.



One the entire syntax tree is constructed, the function gen() is called at the root of the syntax tree.

```
Class If extends Stmt {
    Expr E; Stmt S;
    public If (Expr x, Stmt y) {
        E = x; S = y; after = newlabel();
    }
    public void gen() {
        Expr n = E.rvalue (); // the boolean result
        emit ("ifFalse" + n.toString() + "goto" + after );
        S.gen(); // call gen() of class Stmt
        emit (after + ":");
    }
}
```

All statement classes contain a function gen()

Function *gen*() in class *If*



Translation of Expressions

- Simple approach
  - Generate one three-address instruction for each operator node in the syntax tree for an expression.
  - Don’t generate code for identifiers or constants since they can appear as addresses in instructions.
  - E.g., if a node *x* of class *Expr* has operator **op**, then an instruction is emitted to compute the value at node *x* into a compiler generated “temporary” name.

i – j + k → Translated to →
t1 = i – j
t2 = t1 + k

2*a[i] → Translated to →
t1 = a [i]
t2 = 2 * t1

If a[i] appears on the left side, we can’t simply use a temporary in place of a[i].



Translation of Expressions (Cont.)

- Functions *lvalue* and *rvalue* of the simple approach
  - When function *rvalue* is applied to a nonleaf node *x*, it
    - Generates instructions to compute *x* into a temporary and
    - Returns a node representing the temporary.
  - When function *lvalue* is applied to a nonleaf node *x*, it
    - Generates instructions to compute the subtrees below *x*, and
    - Returns a node representing the “address” of *x*.

Function lvalue

- Cases of function *lvalue*
  - Function *lvalue* simply returns x if x is the node for an identifier.
  - When node *x* represents an array access (e.g, a[i]), x will have the form *Access(y,z)*, where
    - class Access is a subclass of Expr,
    - y is the name of the accessed array, and
    - z is the offset (index) of the chosen element in that array.

Expr lvalue (*x* : *Expr*) {
  if (*x* is an *Id* node) **return** *x*;
  else if (*x* is an *Access(y,z)* node and *y* is an *Id* node) {
    return new *Access* (y, rvalue (z)); // compute the rvalue
  }
  else error;
}

e.g., a [2 * k]:
y = a
z = 2 * k

a [2*k]

⇩

a [t]
(t = 2 * k)

→ New node x' represents the l-value a[t].
→ New node z' represents the temporary name t.

Function rvalue

- Function rvalue generates instructions and returns a possible new node.

```
Expr rvalue (x : Expr) {
    if (x is an Id node or a Constant node) return x; //return itself
    else if ( x is an Op(op, y, z) or a Rel(op, y, z) node) {
        t = new temporary; // 2. t1, 4. t3
        emit string for t = rvalue(y) op rvalue(z); // 4. t3 = j-k, 2. t1=2*t2
        return a new node for t; // 4. return t3, 2. return t1
    }
    else if ( x is an Access(y, z) node) {
        t = new temporary; // 3. t2
        call lvalue(x), which returns Access(y,z'); // z' = t3
        emit string for t = Access(y,z'); // 3. t2=a[t3]
        return a new node for t; // return t2
    }
    else if ( x is an Assign(y, z) node) {
        z' = rvalue(z); // z' = t1
        emit string for lvalue(y) = z'; //1. a[i]=t1
        return z';
    }
}
```

4. j-k

rvalue(j-k)

2

3. a[j-k]

2. 2*a[j-k]

a[j-k]

1. a[i] = 2*a[j-k]

a [i] = 2*a[j-k]

⇩

t3 = j – k
t2 = a [t3]
t1 = 2 * t2
a [i] = t1



Better Code for Expressions

- We can improve the function *rvalue*:
  - Reduce the number of copy instructions.
    - E.g., t = i + 1 and i = t → i = i + 1
  - Generate fewer instructions by taking context into account.
    - E.g.,
      - If the left side of a three-address assignment is an array access a[t], then the right side must be a name, a constant, or a temporary (that needs just one address).
      - If the left side is a name x, the right side can be an operation *y* **op** *z* that uses two addresses.

```
t1 = j + k
i = t1
```

```
null = j + k
```

↓

```
i = j + k
```

The null result address is later replaced by either an identifier or a temporary.