CS 4300: Compiler Theory

Chapter 2
A Simple Syntax-Directed Translator

Dr. Xuejun Liang

Outline

- This chapter is an introduction to the compiling techniques in Chapters 3 to 6 of the Dragon book
- It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code
- The major topics are
    2. Syntax Definition
    3. Syntax-Directed Translation
    4. Parsing
    5. A Translator for Simple Expressions
    6. Lexical Analysis
    7. Symbol Tables
    8. Intermediate Code Generation

7. Symbol Tables

Given input :

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

The goal is to produce output:

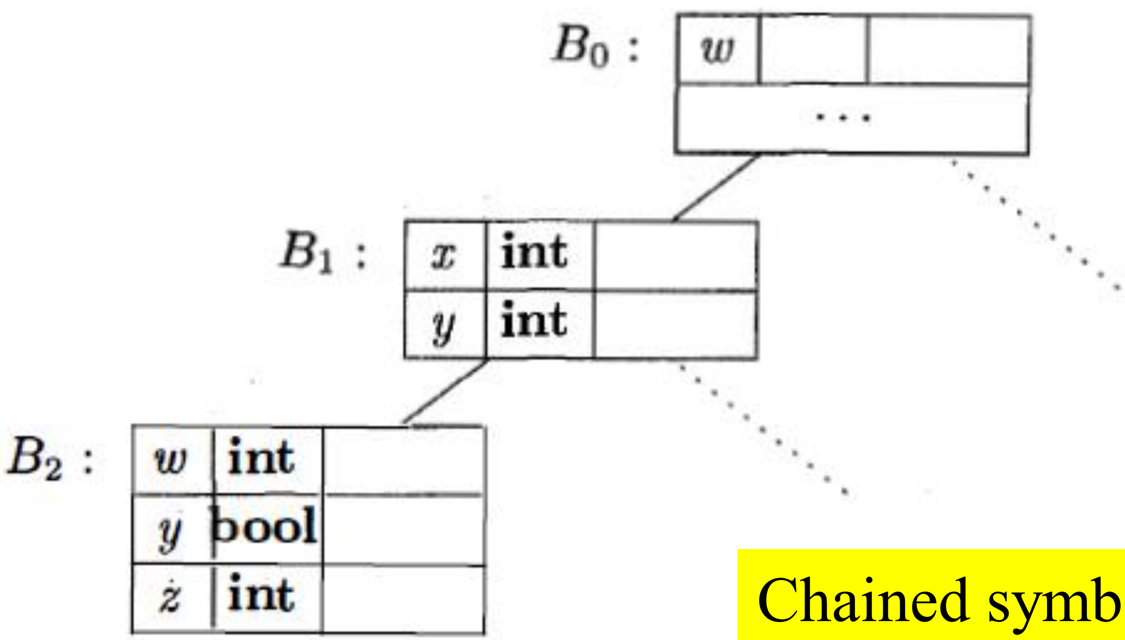
{ { x : int ; y : bool ; } x : int ; y : char ; }

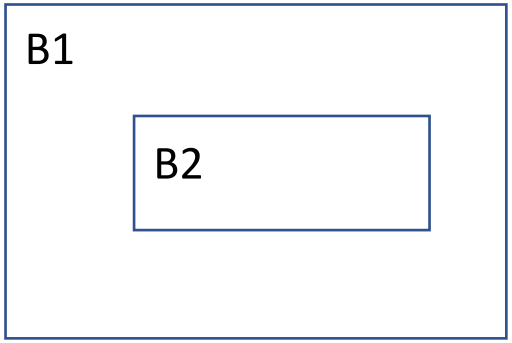
The most- closely nested rule for blocks

an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears

Symbol Table Per Scope

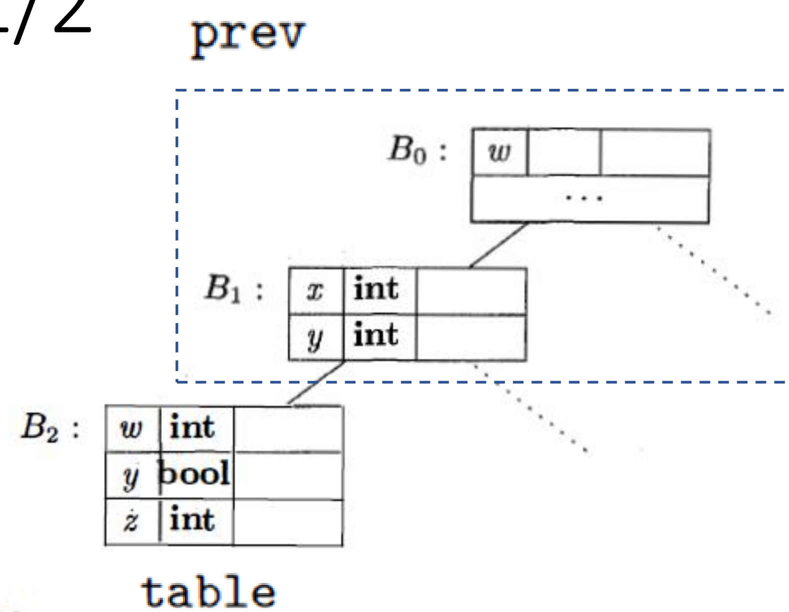
$$
\begin{array}{lll}
\{ & \textbf{int}\ x_1;\ \ \textbf{int}\ y_1; & \\
 & \{ \quad \textbf{int}\ w_2;\ \ \textbf{bool}\ y_2;\ \ \textbf{int}\ z_2; & \\
 & \quad \cdots\ w_2 \cdots;\ \ \cdots\ x_1 \cdots;\ \ \cdots\ y_2 \cdots;\ \ \cdots\ z_2 \cdots; & \\
 & \} & \\
 & \cdots\ w_0 \cdots;\ \ \cdots\ x_1 \cdots;\ \ \cdots\ y_1 \cdots; & \\
\} & &
\end{array}
$$





Chained symbol tables

Class Env implements chained symbol tables 1/2



// File Env.java

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

    public Env(Env p) {
        table = new Hashtable(); prev = p;
    }
```

Class Env implements chained symbol tables 2/2

```
    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;
    }
}
```

Grammar of the input program

Given input :

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

program	→	block
block	→	'{' decls stmts '}'
decls	→	decls decl | ε
decl	→	**type id ;**
stmts	→	stmts stmt | ε
stmt	→	block | factor ;
facor	→	**id**

The goal is to produce output:

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

The Use of Symbol Tables 1/2

The use of symbol tables for translating a language with blocks

$$
\begin{array}{llll}
\mathit{program} & \rightarrow & & \{\ \mathit{top} = \textbf{null};\ \} \\
 & & \mathit{block} & \\
 & & & \\
\mathit{block} & \rightarrow & \text{'\{'} & \{\ \mathit{saved} = \mathit{top}; \\
 & & & \quad \mathit{top} = \textbf{new}\ \mathit{Env}(\mathit{top}); \\
 & & & \quad \text{print("\{ "); \}} \\
 & & \mathit{decls}\ \mathit{stmts}\ \text{'\}'} & \{\ \mathit{top} = \mathit{saved}; \\
 & & & \quad \text{print("\} "); \}} \\
 & & & \\
\mathit{decls} & \rightarrow & \mathit{decls}\ \mathit{decl} & \\
 & | & \epsilon &
\end{array}
$$

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

The Use of Symbol Tables 2/2

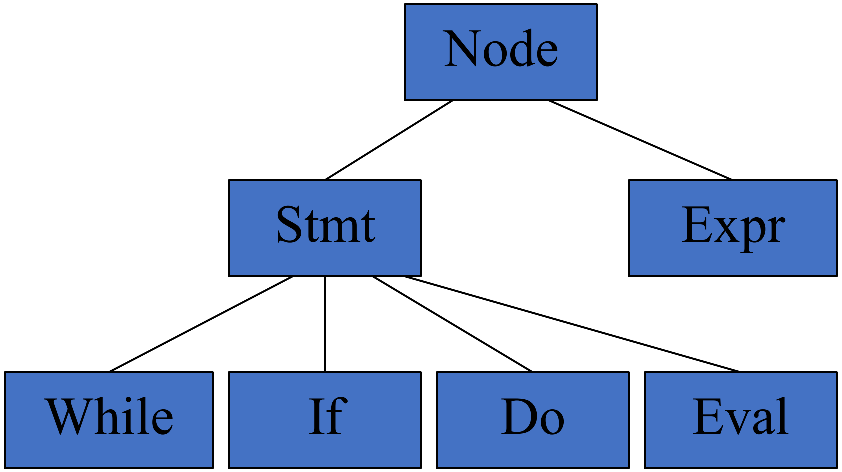
$$
\begin{array}{lcll}
decl & \rightarrow & \textbf{type id ;} & \{\ s = \textbf{new}\ Symbol; \\
 & & & \quad s.type = \textbf{type}.lexeme \\
 & & & \quad top.put(\textbf{id}.lexeme,\ s);\ \} \\
 & & & \\
stmts & \rightarrow & stmts\ stmt & \\
 & | & \epsilon & \\
 & & & \\
stmt & \rightarrow & block & \\
 & | & factor\ ; & \{\ \text{print}("; \ ");\ \} \\
 & & & \\
factor & \rightarrow & \textbf{id} & \{\ s = top.get(\textbf{id}.lexeme); \\
 & & & \quad \text{print}(\textbf{id}.lexeme); \\
 & & & \quad \text{print}(":"); \\
 & & & \quad \text{print}(s.type);\ \}
\end{array}
$$

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

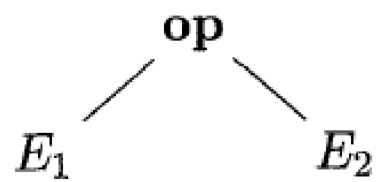
8. Intermediate Code Generation

- Consider intermediate representations for expressions and statements (No declarations)
- Two most important intermediate representations are
  - Trees, including parse trees and (abstract) syntax trees
  - Linear representations, especially "three-address code"
- Construction of Syntax Trees (8.2)
  - Syntax Trees for Statements
  - Representing Blocks in Syntax Trees
  - Syntax Trees for Expressions
- Static Checking (8.3)
- Emit three-address code (8.4)

8.2 Syntax Trees for Statements



One class per statement



AST

$\textbf{while} \ (\ expr \ ) \ stmt$

while statement

$\textbf{new} \ \ While\,(x, y)$

Create a While node

$$stmt \ \rightarrow \ \textbf{if} \ (\ expr \ ) \ stmt_1 \qquad \{ \ stmt.n = \textbf{new} \ If\,(expr.n, stmt_1.n); \ \}$$

Production

Semantic action

Syntax Trees for Statements (Cont.)

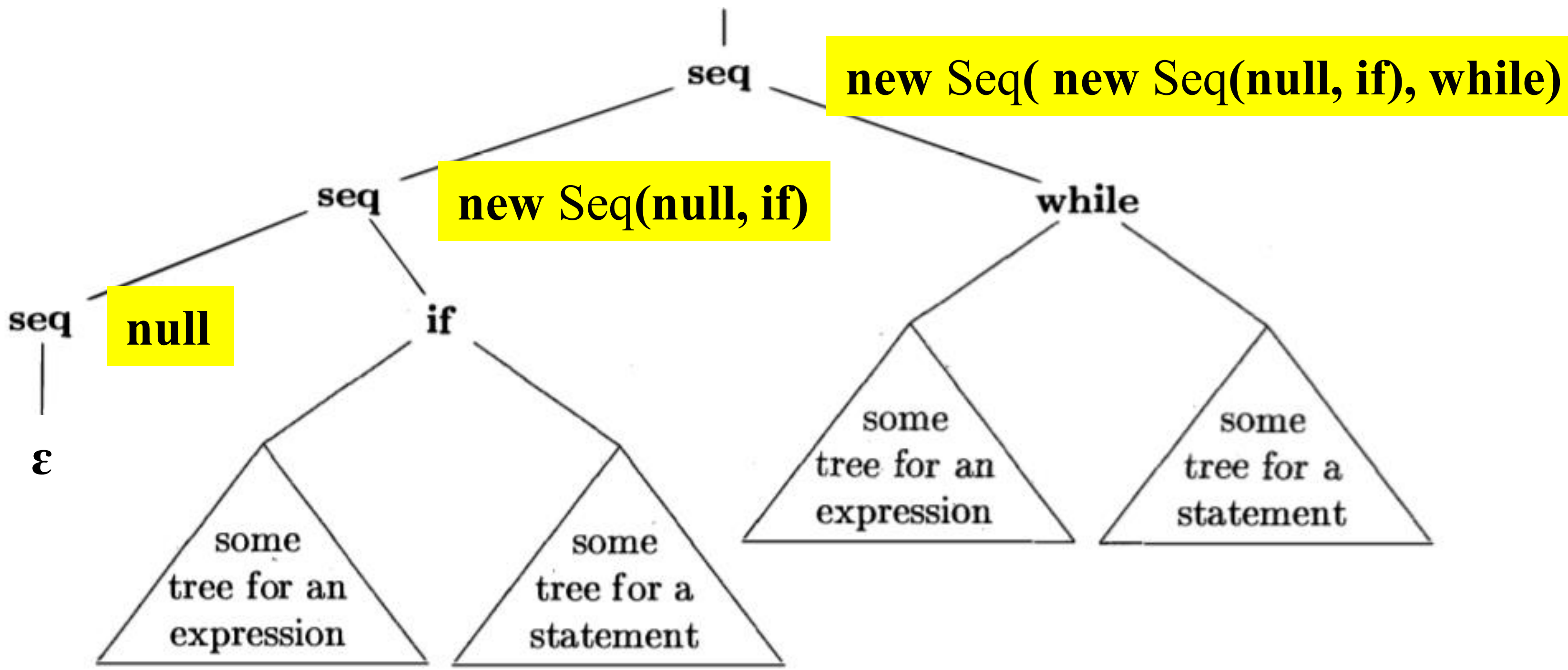
$$
\begin{array}{lcll}
program & \rightarrow & block & \{ \text{ return } block.n; \} \\
block & \rightarrow & '\{' \ stmts \ '\}' & \{ \ block.n = stmts.n; \} \\
\\
stmts & \rightarrow & stmts_1 \ stmt & \{ \ stmts.n = \textbf{new } Seq(stmts_1.n, stmt.n); \} \\
& | & \epsilon & \{ \ stmts.n = \textbf{null}; \} \\
\\
stmt & \rightarrow & expr \ ; & \{ \ stmt.n = \textbf{new } Eval(expr.n); \} \\
& | & \textbf{if} \ (\ expr \ ) \ stmt_1 & \\
& & & \{ \ stmt.n = \textbf{new } If(expr.n, stmt_1.n); \} \\
& | & \textbf{while} \ (\ expr \ ) \ stmt_1 & \\
& & & \{ \ stmt.n = \textbf{new } While(expr.n, stmt_1.n); \} \\
& | & \textbf{do} \ stmt_1 \ \textbf{while} \ (\ expr \ ) \ ; & \\
& & & \{ \ stmt.n = \textbf{new } Do(stmt_1.n, expr.n); \} \\
& | & block & \{ \ stmt.n = block.n; \}
\end{array}
$$

Block

sequence

Example: Part of Syntax Tree

Part of a syntax tree for a statement list: **if** (…) …; **while** (…) …;



$$
\begin{array}{llll}
stmts & \rightarrow & stmts_1 \;\; stmt & \{\; stmts.n = \textbf{new}\; Seq(stmts_1.n, stmt.n); \;\} \\
 & | & \epsilon & \{\; stmts.n = \textbf{null}; \;\}
\end{array}
$$

Syntax Trees for Expressions

- Group "similar" operators to reduce the number of classes of nodes in an implementation of expressions.
- "similar" to mean that the type-checking and code-generation rules for the operators are similar

CONCRETE SYNTAX	ABSTRACT SYNTAX
=	assign
||	cond
&&	cond
== !=	rel
< <= >= >	rel
+ -	op
* / %	op
!	not
$-_{unary}$	minus
[]	access

Syntax Trees for Expressions

$$
\begin{array}{rcll}
expr & \rightarrow & rel = expr_1 & \{\ expr.n = \textbf{new}\ Assign('=', rel.n, expr_1.n);\ \} \\
 & | & rel & \{\ expr.n = rel.n;\ \} \\
 & & & \\
rel & \rightarrow & rel_1 < add & \{\ rel.n = \textbf{new}\ Rel('<', rel_1.n, add.n);\ \} \\
 & | & rel_1 <= add & \{\ rel.n = \textbf{new}\ Rel('\leq', rel_1.n, add.n);\ \} \\
 & | & add & \{\ rel.n = add.n;\ \} \\
 & & & \\
add & \rightarrow & add_1 + term & \{\ add.n = \textbf{new}\ Op('+', add_1.n, term.n);\ \} \\
 & | & term & \{\ add.n = term.n;\ \} \\
 & & & \\
term & \rightarrow & term_1 * factor & \{\ term.n = \textbf{new}\ Op('*', term_1.n, factor.n);\ \} \\
 & | & factor & \{\ term.n = factor.n;\ \} \\
 & & & \\
factor & \rightarrow & (\ expr\ ) & \{\ factor.n = expr.n;\ \} \\
 & | & \textbf{num} & \{\ factor.n = \textbf{new}\ Num(\textbf{num}.value);\ \}
\end{array}
$$

8.3 Static Checking

- Static checks are consistency checks that are done during compilation
    - Syntactic Checking.
        - There is more to syntax than grammars
    - Type Checking
        - Assure that an operator or function is applied to the right number and type of operands
- L-values and R-values
    - *r*-values are what we usually think of as "values," while *l*-values are locations.
- Coercion
    - A coercion occurs if the type of an operand is automatically converted to the type expected by the operator

8.4 Three-Address Code

- Show how to write functions that process the syntax tree and, as a side-effect, emit the necessary three-address code
- Three-Address Instructions

$x = y \ \mathbf{op} \ z$

$x = y$

$x \, [\, y \,] = z$

$x = y \, [\, z \,]$

`ifFalse` x `goto L`

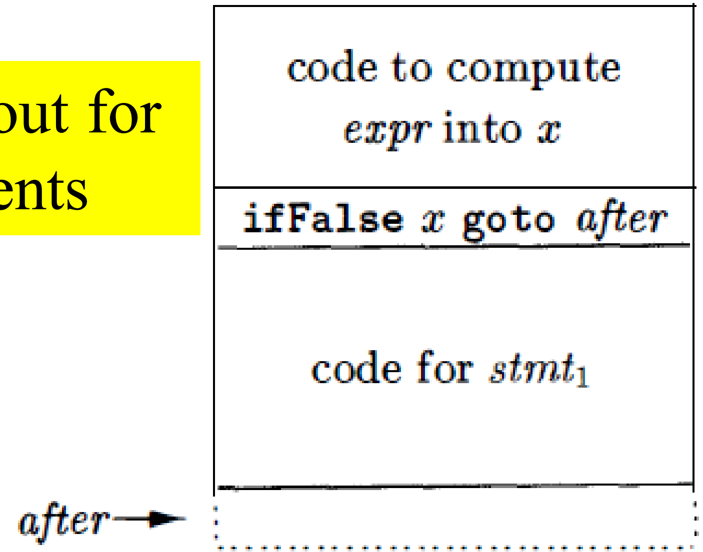
`ifTrue` x `goto L`

`goto L`

- Translation of Statements
  - Example: **if** *expr* **then** $stmt_1$

if *expr* **then** $stmt_1$

Code layout for if-statements



.

Function gen in class If generates three-address code

```
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();
                emit( "ifFalse " + n.toString() + " goto " + after);
                S.gen();
                emit(after + ":");
        }
}
```

Using Translation Scheme

if *expr* **then** $stmt_1$

$stmt \rightarrow$ **if** *expr*

{ *after* = newlabel();
print("`ifFalse goto` *after*:"); }

then $stmt_1$

{ print("`after:` "); }

code for *expr*
`ifFalse goto` *after*
code for $stmt_1$
after:

Translation of Expressions

$x = i - j + k \;\downarrow\; t = i - j,\; x = t + k$

$x = 2*a[i] \;\downarrow\; t = a[i],\; x = 2 * t$

$a[2*k] = x \;\downarrow\; t = 2*k,\; a[t] = x$

$a[i] = 2*a[j-k] \;\downarrow\; t3 = j - k,\; t2 = a[t3],\; t1 = 2 * t2,\; a[i] = t1$

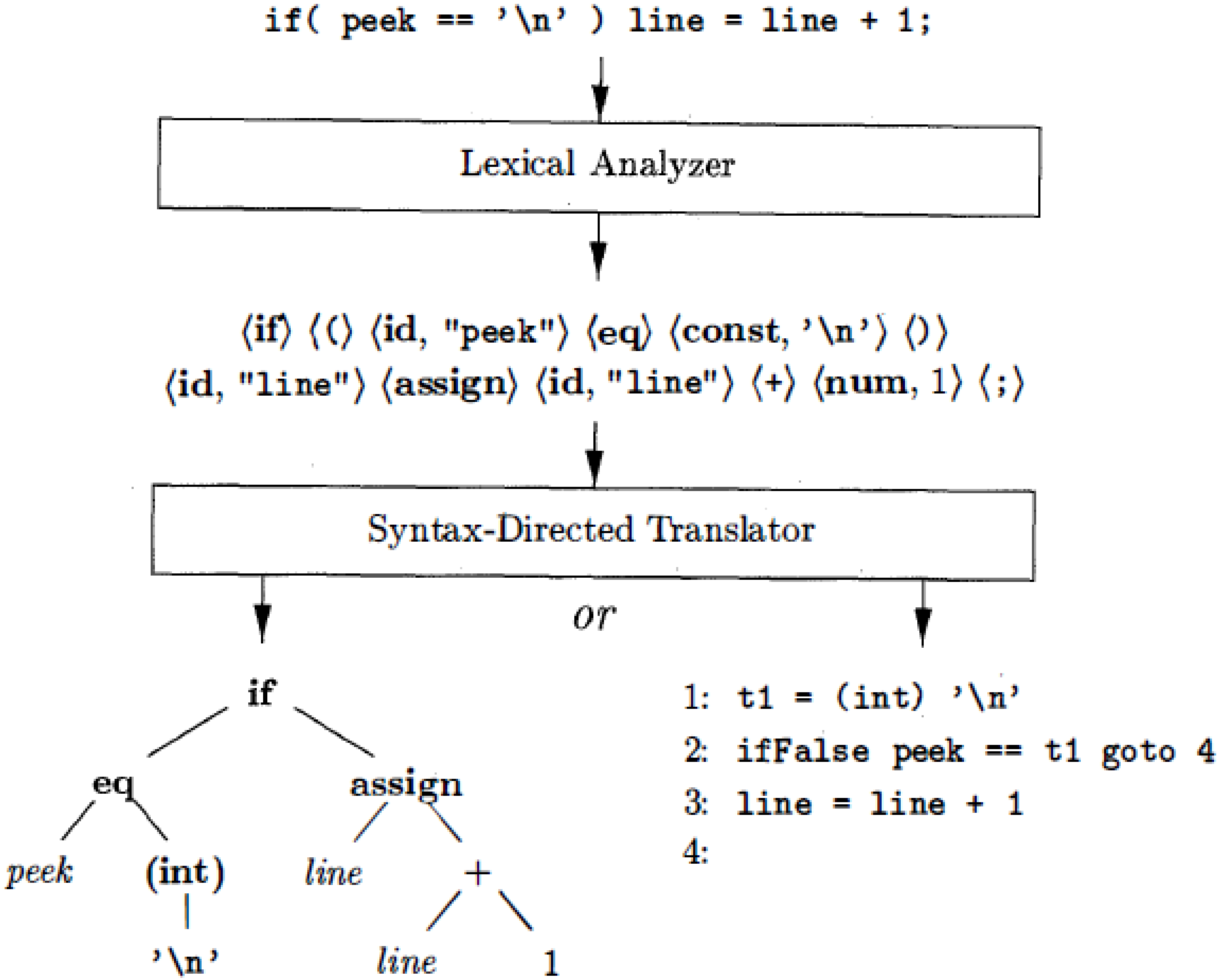
$x = i - j + k$	$x = 2*a[i]$	$a[2*k] = x$	$a[i] = 2*a[j-k]$
↓	↓	↓	↓
$t = i - j$	$t = a[i]$	$t = 2*k$	$t3 = j - k$
$x = t + k$	$x = 2 * t$	$a[t] = x$	$t2 = a[t3]$
			$t1 = 2 * t2$
			$a[i] = t1$

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));
    }
    else error;
}

Pseudocode for function *lvalue*

```
Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op (op, y, z) or a Rel (op, y, z) node ) {
        t = new temporary;
        emit string for t = rvalue(y) op rvalue(z);
        return a new node for t;
    }
    else if ( x is an Access (y, z) node ) {
        t = new temporary;
        call lvalue(x), which returns Access (y, z');
        emit string for t = Access (y, z');
        return a new node for t;
    }
    else if ( x is an Assign (y, z) node ) {
        z' = rvalue(z);
        emit string for lvalue(y) = z';
        return z';
    }
}
```

Pseudocode for function *rvalue*


if(peek == '\n') line = line + 1;
Lexical Analyzer
⟨if⟩ ⟨(⟩ ⟨id, "peek"⟩ ⟨eq⟩ ⟨const, '\n'⟩ ⟨)⟩
⟨id, "line"⟩ ⟨assign⟩ ⟨id, "line"⟩ ⟨+⟩ ⟨num, 1⟩ ⟨;⟩
Syntax-Directed Translator
or
if
eq
assign
peek
(int)
line
+
'\n'
line
1
1: t1 = (int) '\n'
2: ifFalse peek == t1 goto 4
3: line = line + 1
4: