A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

Introduction
Overview

- Tasks for Compiling a Programming Language:
  - How to define the Syntax & Semantics of a P.L.
  - How to translate the source syntax/semantics into another form

- Syntax: what a program looks like
  - CFG or BNF (Backus-Naur Form)

- Semantics: what the elements of a program mean
  - Distinguish different elements (tokens, sub-structures) by attributes
  - Apply certain constraints that cannot be explicitly specified by syntax

- Translation: how translation/compilation could be done
  - Syntax directed translation: syntax specified by CFG/BNF & attributes associated with the syntax can be used to guide the translation process
Example: A code fragment & intermediate code

```plaintext
```int i; int j; float[100] a; float v; float x;
```while (true) {
  do i = i + 1 ; while ( a[i] < v );
  do j = j -1 ; while ( a[j] > v );
  If ( i >= j ) break;
  x = a[ i ]; a[ i ] = a[ j ]; a[ j ] = x;
} // while
```

<table>
<thead>
<tr>
<th>High-Level Source Code</th>
<th>3-address code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: i = i + 1</td>
<td>3-address code</td>
</tr>
<tr>
<td>2: t1 = a [ i ]</td>
<td></td>
</tr>
<tr>
<td>3: if t1 &lt; v goto 1</td>
<td></td>
</tr>
<tr>
<td>4: j = j - 1</td>
<td></td>
</tr>
<tr>
<td>5: t2 = a [ j ]</td>
<td></td>
</tr>
<tr>
<td>6: if t2 &gt; v goto 4</td>
<td></td>
</tr>
<tr>
<td>7: ifFalse i &gt;= j goto 9</td>
<td></td>
</tr>
<tr>
<td>8: goto 14</td>
<td></td>
</tr>
<tr>
<td>9: x = a [ i ]</td>
<td></td>
</tr>
<tr>
<td>10: t3 = a [ j ]</td>
<td></td>
</tr>
<tr>
<td>11: a [ i ] = t3</td>
<td></td>
</tr>
<tr>
<td>12: a [ j ] = x</td>
<td></td>
</tr>
<tr>
<td>13: goto 1</td>
<td></td>
</tr>
<tr>
<td>14: // end of program</td>
<td></td>
</tr>
</tbody>
</table>
Syntax-Directed Translation (SDT): From source to AST to Intermediate Code

```
1: i = i + 1
2: t1 = a[i]
3: if t1 < v goto 1
4: j = j - 1
5: t2 = a[j]
6: if t2 > v goto 4
7: ifFalse i >= j goto 9
8: goto 14
9: x = a[i]
10: t3 = a[j]
11: a[i] = t3
12: a[j] = x
13: goto 1
14: // end of program
```
Example of Java Bytecode:
Stack-Based Intermediate Code

```java
class ExprTest {
    int test(int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;

        if (f > 2) {
            f = f - c;
        }

        return f;
    }
}
```

![Java Program](adapted from “joeq compiler system”, Benjamin Livshits, http://suif.stanford.edu/~courses/cs243/joeq/)
Register-Based Virtual Machine for Android Phone – Dalvik VM

- **Java VM (JVM) – Stack-based Instruction Set**
  - Normally less efficient than RISC or CISC instructions
    - Limited memory organization
    - Requires too many swap and copy operations

- **Dalvik VM (for Android OS) – Register-based Instruction Set**
  - Smaller size
  - Better memory efficiency
  - Good for phone and other embedded systems

- **Generation and Execution of Dalvik byte codes**
  - Compiled/Translated from Java byte code into a new byte code
  - app.java (Java source)
  - javac (Java Compiler) |=> app.class (executable by JVM)
  - dx (in Android SDK tool) |=> app.dex (Dalvik Executable)
  - compression |=> apps.apk (Android Application Package)
  - Dalvik VM |=> (execution)
A Simple Compiler: Goals

- An Example for infix-to-postfix translation
  - Infix: operators within/between operands: \((a+b\times c)\)
  - Postfix: operators after operands: \(abc^+\), \(ab+c^*\)

- Emphasizing the implementation of the front end part
  - Lexical analysis, parsing, intermediate code generation
  - Extensive use of analysis and translation techniques in the front end of a compiler

- The task is extensible to the compilation of more general programming constructs
Example Task: Infix to Postfix

- Infix: 9-5+2, 9-5*2 (ambiguous!!)
  - Unambiguous expression: ((9-5)+2), (9-(5*2))
  - Simplification: replacing parentheses with operators
    - Postfix or prefix
    - Postfix Programming Language: FORTH (for stack machines)

- Postfix: 95-2+, 952*-
  - operators are sitting at the positions of right-hand-side parentheses
  - can be evaluated using a stack
    - Left-to-right scan
    - Push upon operand
    - Pop upon operator, apply operation, then push result
      - Stack top always keeps the result of the latest operation
  - no parentheses are required

- Prefix: +9-52, -9*52
  - Ditto (but processing in right-to-left direction)
The Structure of a Syntax-Directed Compiler
Structure of Compiler Front End

Character stream → Lexical analyzer → Token stream → Syntax directed translation → Intermediate representation
Homework

- Given the following expression, write down the key-pressing sequences that will be able to get its result using a simple calculator that has only the ‘+’, ‘-’, ‘*’, ‘/’, ‘=’, ‘M+’, ‘M-’, ‘MC’ and ‘MR’ keys.
  - M+/M-: add/subtract temp. result in memory by the current result
  - MR: memory recall (recall memorized result as current result)
- Expression: \((1461 \times 2) \div 4 - (5 + 3 \times 2) + (153 \times 6 + 2) \div 5 + 3\)
- Goal: to know why structure analysis is important in determining the execution sequences of instructions in a real machine that has very simple ALU.
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

- Specification of Syntax (Grammar)
- Specification of Semantic Actions
- Parsing (based on Grammar)
- Translation (based on Semantic Rules)
Syntax Analysis (Parsing): match input tokens against a grammar of the language

- To ensure that the input tokens form a legal sentence (statement)
- To build the structure representation of the input tokens
- So the structure can be used for translation (or code generation)

Knowledge source:
- Grammar in CFG (Context-Free Grammar) form
- Additional semantic rules for semantic checks and translation (in later phases)

```
id_1 := id_2 + id_3 * 60
```

**Parse Tree** (Concrete syntax tree)

- s
  - id_1 := e
    - id_2 + t
      - id_3 * 60
Specifications of the One-Pass Compiler: Syntax Definition

- Specification of syntax of a language: CFG (Context-Free Grammar)
- CFG: describes hierarchical structure with production rules
- production rules: relationship between a grammatical construct (parent) and its constituents (children)
  - Example: if ( expression ) statement else statement
  - CFG production: \( stmt \rightarrow if( expr ) stmt else stmt \)
- tokens: lexical elements in a production
  - Keywords (“if”, “else”), parenthesis (“(, )”),
- non-terminals: variables like \( expr \) and \( stmt \)
Specifications of the One-Pass Compiler: Syntax Definition

- Four components in a CFG
  - a set of tokens: *terminal* symbols
  - a set of *nonterminals*
  - a set of *productions*:
    - LHS (left-hand side): nonterminal
    - RHS (right-hand side): a sequence of terminals or nonterminals
  - a *start symbol* (from nonterminals) specifying the topmost construct of the program
CFG: Components

- CFG: formal specification of parse trees
  - $G = \{\Sigma, N, P, S\}$
  - $\Sigma$: terminal symbols
  - $N$: non-terminal symbols
  - $P$: production rules
  - $S$: start symbol

- $\Sigma$: terminal symbols
  - the input symbols of the language
    - programming language: tokens (reserved words, variables, operators, …)
    - natural languages: words or parts of speech
  - pre-terminal: parts of speech (when words are regarded as terminals)

- $N$: non-terminal symbols
  - groups of terminals and/or other non-terminals

- $S$: start symbol: the top-most constituent of a grammar
  - @ root of each parse tree

- $P$: production (re-writing) rules
  - form: $\alpha \rightarrow \beta$ ($\alpha$: non-terminal, $\beta$: string of terminals and non-terminals)
  - meaning: $\alpha$ re-writes to (“consists of”, “derived into”) $\beta$, or $\beta$ reduced to $\alpha$
  - start with “S-productions” ($S \rightarrow \beta$)
Syntax Definition: Example

- Grammar for “lists of digits separated by plus/minus signs”
  - Example: 1+2-3+4-5
  - list → list + digit
  - list → list - digit
  - list → digit
  - digit → 0 | 1 | 2 | ... | 9

- Short hands
  - list → list + digit | list - digit | digit
  - digit → 0 | 1 | 2 | ... | 9

- Terminal tokens: ‘+’, ‘-’, ‘0’, ‘1’, …, ‘9’

- Non-terminals: {list, digit}

- Start symbol: list

- Language:
  - L(G-digits) = {9-5+2, 2+3+4-5, …}
 CFG: a Simple English Grammar

- Grammar Rules
  - S $\rightarrow$ NP VP
  - NP $\rightarrow$ Pron $|$ Proper-Noun $|$ Det Norm
  - Norm $\rightarrow$ Noun Norm $|$ Noun
  - VP $\rightarrow$ Verb $|$ Verb NP $|$ Verb NP PP $|$ Verb PP
  - PP $\rightarrow$ Prep NP

- Lexicon (Lexical Rules in CFG form)
  - Noun $\rightarrow$ girl $|$ park $|$ desk
  - Verb $\rightarrow$ like $|$ want $|$ is $|$ saw $|$ walk
  - Prep $\rightarrow$ by $|$ in $|$ with $|$ for
  - Det $\rightarrow$ the $|$ a $|$ this $|$ these
  - Pron $\rightarrow$ I $|$ you $|$ he $|$ she $|$ him
  - Proper-Noun $\rightarrow$ IBM $|$ Microsoft $|$ Berkeley

S: sentence, NP: noun phrase, VP: verb phrase
Pron: pronoun
Det: determiner, Norm: Noun, PP: prepositional phrase, Prep: preposition
Syntax Definition: Terminology

- “A production for a nonterminal”:
  - nonterminal at LHS of a production
  - $\alpha \rightarrow \beta$ : “a production for $\alpha$”

- N-production:
  - the set of production rules for the nonterminal symbol N
  - E.g., S-production for the start symbol

- “a string of tokens”
  - A sequence of zero or more tokens
  - E.g., “The girl in the park …”
  - Empty string ( $\varepsilon$ ): the string containing zero tokens
Syntax Definition: \( \varepsilon \) -production (optional element)

- Grammar with \( \varepsilon \) -production:
  "lists of optional parameters separated by commas"

  - call \( \rightarrow \) id ( optparams )
  - optparams \( \rightarrow \) params | \( \varepsilon \)
  - params \( \rightarrow \) params , param | param

- Example
  - Weekday(2012, 03, 29)
  - Weekday() \( // \) today, if omitted

\[ \leftrightarrow \] "lists of digits"
  - list \( \rightarrow \) list + digit
  - list \( \rightarrow \) list - digit
  - list \( \rightarrow \) digit
  - digit \( \rightarrow \) 0 | 1 | 2 | ... | 9

(G-function-call)
Syntax Definition: Another Example

- Grammar with $\varepsilon$ -production:
  "lists of statements separated by semicolons"

  - block $\rightarrow$ begin opt_stmts end
  - opt_stmts $\rightarrow$ stmt_list | $\varepsilon$
  - stmt_list $\rightarrow$ stmt_list ; stmt | stmt
  - stmt $\rightarrow$ if-stmt | assign-stmt |

- Non-terminals: {block, opt_stmts, stmt_list, stmt}
- Terminals: {“begin”, “end”, other variables … }
- Start symbol: block
- Comparison with “list” (G-digit)
  - Empty list may be found between “begin”-”end”
  - ‘;’ $\iff$ ‘+’, ‘-’
  - stmt $\iff$ ‘0’, ‘1’, …, ‘9’

- $\iff$ “lists of digits”
  - list $\rightarrow$ list + digit
  - list $\rightarrow$ list - digit
  - list $\rightarrow$ digit
  - digit $\rightarrow$ 0 | 1 | 2 | … | 9
Syntax Definition: Language Generation

- **Language Generation**: A grammar (G) derives strings by beginning with the start symbol and repeatedly replacing a nonterminal with the RHS of a production for that nonterminal, and end up with a string of terminals
  - **Verb**: Derive == generate == re-write
  - **Noun**: Derivation == generation == re-writing
  - Derivation in one step: \( \alpha \Rightarrow \beta \)
  - Derivation in zero or more steps: \( \alpha \Rightarrow^* \beta \)
  - Derivation in one or more steps: \( \alpha \Rightarrow^+ \beta \)

- **Language**: the token strings that can be derived from the start symbol
  - \( L(G) = \{ \text{token strings derived from Start symbol} \} = \{ s | S =^* \Rightarrow s \} \)

- **Empty string**: the string containing zero tokens, written as \( \varepsilon \) (or \( e, \Phi \))

- **Grammar \Leftrightarrow Language** (set of sentences)
  - generation: \( G \rightarrow \text{sentence} \) (by derivation)
  - analysis: \( G \leftarrow \text{sentence} \) (by derivation or reduction)
Syntax Verification: Parsing (Syntax Analysis)

- **Parsing**: an inverse process of derivation
  - get a derivation sequence that produces an input token string
  - find a parse tree for a string of tokens
  - E.g., 9-5+2 =?accept_as=> list

- **Parse Tree (concrete syntax tree)**
  - root: start symbol
  - leaf: terminal
  - internal nodes: nonterminals
  - production
  - “yield”: leave nodes
    read from left to right

```
  list
  /|
list + digit
/ |
list - digit
/  |
digit
/   |
9    5
```

- Parse Tree: graphical representation of structure
  - root node (S): a sentencial level structure
  - internal nodes: constituents of the sentence
  - arcs: relationship between parent nodes and their children (constituents)
  - terminal nodes: surface forms of the input symbols (e.g., words)
  - alternative representation: bracketed notation:
    - e.g., [I saw [the [girl [in [the park]]]]]

- Example:

```
   NP
  /   \
NP   PP
   /   \\
NP   NP
  girl  in  the  park
```
Parse Tree: “I saw the girl in the park”
Syntax Definition: Ambiguity

- Ambiguity: A grammar may have more than one parse tree generating a given string of tokens.
  - Everyone can have his/her own grammar for a language
  - Not every grammar precisely specifies correct syntax

- Ambiguous grammar: at least one input sentence in the language has more than one parse according to that grammar.
  - Ambiguous “grammar” vs inherently ambiguous “language”

- Grammar writing for programming languages
  - Write an unambiguous grammar, or
  - Use an ambiguous one + disambiguation rules
Ambiguous Grammar: 1 input with more than 1 parse

E → E “+” E | E “-” E | D
D → “1” | “2” | … | “9”

(9 - 5) + 2

9 - (5 + 2)
Two Unambiguous Grammars

E → E “+” D | E “-” D | D
D → “1” | “2” | … | “9”

LA: (9 - 5) + 2
RA: 9 - (5 + 2)
Sources of Ambiguity for Math. Expressions

- **Associativity of Operators (結合性)**
  - **Left Association (LA; 左結合):** \(9-5+2 \Leftrightarrow (9-5)+2\)
    - LA: \(\text{cout} << \text{"Hello"} << \text{"World"} << \text{endl} ;\)
      - ⇔ (((cout << "Hello") << "World") << endl) ;  (Y)
      - ⇔ (cout << ("Hello" << ("World" << endl))) ;  (N)
    - ⇔ cout << "Hello"; cout << "World"; cout << endl;
  - LA: \(\text{cin} >> \text{year} >> \text{month} >> \text{day} ;\)
    - ⇔ cin >> year; cin >> month; cin >> day;
  - **Right Association (RA; 右結合):** \(a=b=c \Leftrightarrow a=(b=c)\)
    - right → letter = right | letter
    - letter → a | b | ...| z
    - RA: \(a += b += c \Leftrightarrow a += (b += c)\)

- **Precedence of Operators (優先順序)**
  - **Multiplication/Division-First Addition/Subtraction-Last (by mathematical convention)**
    - \(9+5*2 \Leftrightarrow 9+(5*2)\)
    - **NOT:** \((9+5) * 2\)
Ambiguity Resolution

- Making Language un-ambiguous or Writing un-ambiguous grammar:
  - Use keywords to identify block structures (“begin-end”) or use parentheses (“(, ”) to delimit blocks of statements
  - Enforcing syntactic constraints on languages
  - Artificial language, NOT natural language
  - Write an unambiguous grammar that reflects association and precedence
    - Change grammar without changing the language

- Resolution for Associativity of Operators
  - LA: Left-branching productions
  - RA: right-branching productions

- Resolution for Precedence of Operators
  - Define high precedence expressions (including atomic units) first
  - Low precedence operators can operate on high precedence expressions
Ambiguity Resolution: RA vs. LA Grammars

- Resolution for Associativity of Operators
  - LA: Left-branching productions
  - RA: right-branching productions

- Example: (RA) [assignment operations]
  - $R \rightarrow L = R \mid L$
  - $L \rightarrow a \mid b \mid \ldots \mid z$

- Example: (LA) [mathemetic expressions]
  - $L \rightarrow L + D \mid L - D \mid D$
  - $D \rightarrow 1 \mid 2 \mid \ldots \mid 9$
Ambiguity Resolution:
High vs. Low Precedence Operators

- **Resolution for Precedence of Operators**
  - Define high precedence expressions (including atomic units), which use only high precedence operators, first
  - Define low precedence expressions where low precedence operators operate on higher precedence expressions
  - Define “lists of high precedence expressions separated by low precedence operators”

- **Example: Mathematic Expression (expr)**
  - **factor**: basic units ↔ highest precedence to be evaluated first
    - factor → digit | ( expr )
  - **term**: units operated by higher precedence operators (in LA form)
    - term → term * factor | term / factor | factor
  - **expr**: units operated by lower precedence operators (in LA form)
    - expr → expr + term | expr - term | term

Start symbol
Syntax of Expression without Ambiguity

- Example: Mathematic Expression
  - “lists of high precedence expressions separated by low precedence operators”

  - $\text{expr} \rightarrow \text{expr} + \text{term} | \text{expr} - \text{term} | \text{term}$
    - $\text{Expr} = \{\text{List of terms separated by ‘+’ or ‘-’ operators}\}$
    - $=\{\text{List of mul-or-div-sub-expressions separated by ‘+’ or ‘-’ operators}\}$

  - $\text{term} \rightarrow \text{term} * \text{factor} | \text{term} / \text{factor} | \text{factor}$
    - $\text{Term} = \{\text{List of factors separated by ‘*’ or ‘/’ operators}\}$
    - $= \{\text{mul-or-div-sub-expressions}\}$
    - $=\{\text{List of primitive operators (including parentheses-enclosed-Expr) separated by ‘*’ or ‘/’ operators}\}$

  - $\text{factor} \rightarrow \text{digit} | (\text{expr})$
    - $\text{Factor} = \{\text{primitive operands or/including parentheses-enclosed-Expr}\}$

- “Why NOT?”
  - $\text{expr} \rightarrow \text{term} + \text{expr}$
  - $\text{expr} \rightarrow \text{term} - \text{expr}$
  - $\text{expr} \rightarrow \text{term}$
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

- Specification of Syntax (Grammar)
- Specification of Semantic Actions
- Parsing (based on Grammar)
- Translation (based on Semantic Rules)
Syntax Directed Translation: Infix-to-Postfix
-- some translation can be based on local syntax

\[ E.t = "95-2+" \]

\[ \begin{array}{c}
E.t = "95-" \\
\quad + \\
\quad T.t = "2"
\end{array} \]

\[ \begin{array}{c}
E.t = "9" \\
\quad - \\
\quad T.t = "5"
\end{array} \]

\[ \begin{array}{c}
T.t = "9" \\
\quad 5 \\
\quad 9
\end{array} \]
Syntax Directed Translation: Infix-to-Postfix
-- some translation can be based on local syntax

Production:
expr → expr1 + term
expr → expr1 - term
expr → term
term → 0
term → 1

Semantic Rules:
expr.t := expr1.t || term.t || ‘+’
expr.t := expr1.t || term.t || ‘-’
expr.t := term.t
term.t := ‘0’
term.t := ‘1’
Syntax-Directed Translation (SDT)

- SDT Question: Given a syntax analysis result (e.g., a parse tree) how to translate it into an intermediate representation (or anything associated with the syntax)?
  - Grammar => Syntax => Translation (??)

- How to specify “translation rules”?  
  - i.e., “Input => output” mapping

- Example: infix-to-postfix “translation rules” (postfix definition)
  - E.p = id if E → id is a variable or constant
  - E.p = E1.p E2.p op if E → E1 op E2
  - E.p = E1.p if E → (E1) [enclosed by parentheses]

- Translate from local sub-expression, then propagate to parents, recursively
Syntax-Directed Translation (SDT)

- What has been done in SDT?
  - A “syntax directed” approach: do translation based on local syntax structure
  - Associate each local structure with a set of rules or translation actions
  - Keep some variables (i.e., attributes) for each LHS symbol
    - Including partial or full translation results
  - Specify the translation actions in terms of attribute evaluation actions

- Attributes associated with syntactic constructs
  - keep various information required for translation (or semantic checking)
  - e.g., type, string, memory location, or whatever

Hide translation results in attributes.
Evaluate the attributes based on local syntax.
Syntax-Directed Translation (SDT)

- Two Ways to Specify the Syntax-Directed Translation Process
  - SDD: syntax directed definition
  - TS: Translation Scheme

- Syntax directed definition (SDD): formal specification of translation
  - specify the translation of a construct in terms of attributes associated with syntactic components

- Translation Scheme (TS): procedural notation for specifying translations
  - specify translation actions in terms of program segments …
Syntax-Directed Definition for SDT

- SDD (Syntax Directed Definition): productions + semantic rules
  - SDT Input: parse tree + annotated CFG
  - SDT Output: annotated parse tree (with attribute annotation), where –

- Annotated CFG (aka Attribute Grammar):
  - CFG: specify syntax & Semantic Rules/Actions: how to evaluate attributes
  - each grammar symbol ↦ annotated with a set of attributes
    - For saving local translation results of sub-tree/sub-syntax structure
    - Or auxiliary attributes
  - each production ↦ annotated with a set of semantic rules
    - for computing attribute values of grammar symbols in that production
    - (in terms of attributes of parents, siblings or children)

- Translation process for input x based on SDD:
  - 1. construct parse tree of x
  - 2. X.a (attribute of X) at node n is evaluated using semantic rules for attribute a associated with X-production
SDD for Infix-to-Postfix Translation

Production:
expressions → expression₁ + term
expressions → expression₁ - term
expressions → term
terms → 0
terms → 1

[Note]
- The index ‘1’ in expression₁ is not really written in the productions

Semantic Rules:
expression₁.t := expression₁.t || term.t || ‘+’
expression₁.t := expression₁.t || term.t || ‘-’
term.t := 0
term.t := 1

[Note]
- expression₁ refers to the 1st expression in the RHS (to avoid confusion with other symbols with the same name)
SDD for Infix-to-Postfix Translation

E.t = "95-2+

E.t = "95-" + T.t = "2"

E.t = "9" - T.t = "5"

T.t = "9" 5

9
SDD for Robot’s Position

Input: begin west south …

Seq.x=-1
Seq.y=-1

Instr.dx=0
Instr.dy=-1

south

Aux. Attributes (dx, dy)

Attributes for main result (x,y)
SDD for Robot’s Position

Production:
seq → begin
seq → seq₁ instr

instr → east
instr → north
instr → west
instr → south
...

Semantic Rules:
seq.x := 0 ; seq.y := 0
seq.x := seq₁.x + instr.dx
seq.y := seq₁.y + instr.dy
instr.dx := 1 ; instr.dy := 0
instr.dx := 0 ; instr.dy := 1
instr.dx := -1 ; instr.dy := 0
instr.dx := 0 ; instr.dy := -1
...

...
Attributes

- **Synthesized Attributes:**
  - attribute value is defined in terms of attribute values of children (& itself)
  - can be evaluated during a single bottom-up traversal of parse tree

- **Inherited Attribute:**
  - attribute value is defined in terms of attribute values of parent and/or siblings (& the node itself)
**Production** | **Semantic Rules**
--- | ---
L → E ‘\n’ | L.val := E.val
E → E₁ ‘+’ T | E.val := E₁.val + T.val
E → T | E.val := T.val
T → T₁ ‘*’ F | T.val := T₁.val * F.val
T → F | T.val := F.val
F → ‘(’ E ‘)’ | F.val := E.val
F → digit | F.val := digit.val

Fig. 5.2, 1\(^{st}\) Ed; Fig. 5.1, 2\(^{nd}\) Ed
Synthesized Attributes

$L.val = 19$

$E.val = 19$

$F.val = 5$

$T.val = 4$

$T.val = 3$

$F.val = 3$

$\text{digit}.val = 3$

$3 \times 5 + 4$

$E.val = 15$

$E.val = 19$

$F.val = 4$

$F.val = 4$

$\text{digit}.val = 4$

$\text{digit}.val = 5$

$\text{digit}.val = 3$
Inherited Attributes

/* declaration: */ float id₁, id₂, id₃

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in := T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type := integer</td>
</tr>
<tr>
<td>T → float</td>
<td>T.type := float</td>
</tr>
<tr>
<td>L → L₁ ‘,’ id</td>
<td>L₁.in := L.in  Addtype(id.entry, L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>Addtype(id.entry, L.in)</td>
</tr>
</tbody>
</table>

Fig. 5.4, 1ˢᵗ Ed; Fig. 5.8, 2ⁿᵈ Ed
Inherited Attributes

/* declaration: */ float id₁, id₂, id₃
Attribute Evaluation: Order of Evaluation

- Any order that correctly evaluates the attributes will do
  - Following the dependency order of attributes
  - Any Topological Sorting Sequences
    - If the grammar is well designed a depth-first traversal of the parse tree might be equivalent to a possible topological sorting sequence

- Depth-first traversal (DFT) of a tree (or sub-tree)
  - // Recursive procedure
  - Procedure Visit(node N) {
    - For Each (child C of N, from left to right) {
      - Visit(C);
    }
    - // after all children are visited & evaluated
    - Eval(N); // evaluate semantic rules at node N
    - // probably using semantic attributes of children
    - // (which are evaluated while visiting children, recursively)
  }
Attribute Evaluation: Order of Evaluation

Exercises:
- Write a pseudo code to evaluate the above synthesized attributes of the L-expression (list of math expression)
- Write a pseudo code to evaluate the above inherited attributes of the D expression (Type declaration)
- Write a grammar for a list of output stream insertion operators
  - (cout << obj1 << obj2 <<…)
  - and define the attributes required to convert the sequence with multiple << operators into an equivalent sequence of single stream operation.
  - I.e., ⇔ Cout << obj1; cout << obj2; cout << …
  - And Write a pseudo code, based on the associated semantic rules
Translation Schemes (TS):
Specify translations via some procedural actions

- A procedural definition for translation
  - attach program segments to production rules
  - to explicitly indicate how and when the translation could be done in the middle of parsing
- Semantic actions are embedded in RHS of productions
- Order of evaluation of semantic rules is explicitly shown
  - E.g., rest → + term {print(‘+’)\} rest
- Execute the actions in the order they appear during a depth-first traversal of the parse tree
TS for Infix-to-Postfix Translation

- **Production with Semantic Actions**
  - \( \text{expr} \to \text{expr} + \text{term} \) \{print(‘+’)}
  - \( \text{expr} \to \text{expr} - \text{term} \) \{print(‘-’)}
  - \( \text{expr} \to \text{term} \)
  - \( \text{term} \to 0 \) \{print(‘0’)}
  - \( \text{term} \to 1 \) \{print(‘1’)}
  - \( \text{term} \to 9 \) \{print(‘9’)}

- **Tools:** YACC

[Warning!!] Semantic actions are NOT always located at the end of each production rule.

  e.g., \( \text{rest} \to + \text{term} \) \{print(‘+’)} \text{rest}
SDD for Infix-to-Postfix Translation

E.t = “95-2+”
E.t = “95-”
E.t = “9”
T.t = “9”

T.t = “5”
T.t = “2”

9

5

2
TS for Infix-to-Postfix Translation

```
9 - 5 + 2
```

```
E
```

```
+ T
```

```
print “+”
```

```
E
```

```
- T
```

```
print “-”
```

```
E
```

```
T
```

```
print “2”
```

```
5
```

```
print “5”
```

```
9
```

```
print “9”
```
SDD vs TS: Attribute Evaluation & Semantic Actions

- (General) SDD (Syntax-Directed Definition)
  - Does not specify order of evaluation explicitly
  - Can normally be evaluated in depth-first (if it is a legal topological sorting sequence)
  - BUT, Any order that correctly evaluates the attributes will do
    - Following the dependency order of attributes
  - After parsing

- TS (Translation Scheme):
  - Explicit order of semantic actions (attribute evaluation) in augmented production rules
  - Depth-First Traversal
  - Incrementally emitting partial translation (on-the-fly, without really constructing the parse tree)
  - During parsing

- “Simple” SDD
  - translation string of LHS = concatenation of (translated strings of RHS non-terminals in the same order + additional interleaved strings)
  - .i.e., LHS.t = … || RHS1.t || … || RHS2.t || … || … || RHSn.t | …

- SDD vs. TS:
  - Simple SDD can be implemented with TS
  - Semantic actions print additional strings in the same order as in semantic rules
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

- Specification of Syntax (Grammar)
- Specification of Semantic Actions
- Parsing (based on Grammar)
- Translation (based on Semantic Rules)
Parsing

- Top-Down (TD) vs. Bottom Up (BU) Parsing
- (TD) Recursive Descent Parser
Parsing & Parsers

■ Parsing (Syntax Analysis)
  ◆ Analyzing structures according to grammar
  ◆ Determine if a string of tokens can be generated by a grammar

■ Complexity
  ◆ Any CFG can be parsed in $O(n^{**3})$
  ◆ Most PL can be parsed in linear time, $O(n)$, looking at one lookahead
Parsing Methods

- **Top-down** (whether we can generate a sentence like the input)
  - Construct parse tree nodes from root down to leaves
    - Matching target strings: from start symbol to its constituents (recursively)
  - Easy to constructed by hands
  - Ex. Recursive descent, predictive recursive descent (a kind of LL Parsers)

- **Bottom-up** (whether the input can reduce to a sentence)
  - Construct parent nodes from children up to root node
  - Can handle a larger class of grammars and translation schemes
    - E.g., left-recursive productions cannot be parsed top-down left-to-right, but they may be parsed by a bottom-up parser
  - Efficient parsers are mostly bottom-up
  - Automatic parser generation tools tend to produce bottom-up parsers
  - Ex. LR parsers
Top-Down Parsing (Example in 1st Ed.)

- Root/Start Symbol → Expand nonterminal by productions in the Grammar
- Two Operations:
  - Match lookahead (current token being scanned) against terminal symbols in productions
  - Expand nonterminals with production(s) whose 1st terminal matches the lookahead

- Example: G(2.8)/Fig 2.15 [1st Ed]
  - type → simple
  - | ^ id
  - | array [ simple ] of type
  - simple → integer
  - | char
  - | num dotdot num  // 2 .. 5

- Parsing: “array [ num dotdot num ] of integer” (Fig 2.15, 2.16)
Top-Down Parsing (Example in 2nd Ed.)

- Root/Start Symbol → Expand nonterminal by productions in the Grammar
- Two Operations:
  - **Match** lookahead (current token being scanned) against terminal symbols in productions
  - **Expand** nonterminals with production(s) whose 1st terminal matches the lookahead

- Example: G in Fig 2.16 [2nd Ed]
  - stmt → expr ;
  - | if ( expr ) stmt
  - | for ( optexpr ; optexpr ; optexpr ) stmt
  - | other
  - optexpr → ε
  - | expr

- Parsing: “for ( ; expr ; expr ) other” (Fig 2.17, 2.18)
Recursive Descent Parsing

- **Recursive-descent Parsing**: a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input tokens, starting with the start symbol.

- Recursive procedures:
  - match input and conduct associated actions
## Recursive Descent Parsing (cont.)

- Each nonterminal has an associated *parsing procedure* that can recognize any sequence of tokens generated by that nonterminal.
- To match a nonterminal $A$, we call the parsing procedure corresponding to $A$, **say $A()$**.
  - Task of $A()$:
    - matches a substring of input tokens, from current word index, according to a particular $A$-production
    - creates nodes and links between $A$ and its children
    - and advances the word index (pointer) to the next word immediately after the last matched word
  - These calls may be recursive, thus it is called *recursive decent parsing*
- To match a terminal symbol $t$, we call a procedure **match($t$)**.
  - **match()** calls the scanner to get the next token.
    - Advance the word index (pointer) if lookeahed matches the target ‘$t$’
    - Failed if current input token does not match the target terminal symbol ‘$t$’
Recursive Descent Parsing (cont.)

- The sequence of procedures called in procedure A() corresponds to the RHS symbols of an A-production.

- Example:
  - Grammar (2.8) => Figure 2.17 [1st Ed]
  - Grammar (Fig. 2.16) => Figure 2.19 [2nd Ed]

- G in Fig 2.16 [2nd Ed]
  - stmt → expr ;
  - | if ( expr ) stmt
  - | for ( optexpr ; optexpr ; optexpr ) stmt
  - | other
  - optexpr → ε
  - | expr
Top-Down Parsing Procedures

/* stmt : a recursive procedure that matches remaining input tokens against stmt structure */ // Fig 2.19, 2 Ed
stmt() {
    switch (lookahead) {
    case expr: // stmt → expr ;
        match(expr); match (';'); break;
    case if: // stmt → if(expr) stmt
        match(if); match ('('); match(expr); match (')');
        stmt(); break;
    case for: // stmt → for (optexpr; optexpr; optexpr) stmt
        match(for); match ('('); optexpr(); match (')');
        optexpr(); match (';'); optexpr();
        match (')'); stmt(); break;
    case other: // stmt → other ;
        match(other); break;
    default:
        report("syntax error");
    }
}

/* optexpr : procedure that matches input against optexpr structure */
optexpr() {
    if ( lookahead == expr ) match(expr); // optexpr → expr
    else { /* do nothing */ } // optexpr → ε
Top-Down Parsing Procedures

/* aux function: match terminal lookahead token against target symbol, and advance input pointer if success */
match(TOKEN t) {  // t: target terminal symbol
   if (lookahead == t)  // lookahead: current input symbol
      lookahead = next_token();  // call lexical analyzer
   else
      error();
}
Top-Down Parsing Procedures

main()
{
    lookahead = next_token(); // get 1st token
    stmt();                   // match start symbol
}

next_token()
{
    /* a lexical analyzer that return next input token */
}
Problems with Top-Down Parsing: Prediction

- Which Production to Expand If More than One Choices?
  - (1) Backtracking (without prediction)
  - (2) Predictive Parsing

- Backtracking
  - Select arbitrary un-tested production (or select productions sequentially)
  - Push return configuration before trial
    - Push current starting index & next useable production if fail
  - Pop/Back to previous configuration if production expansion fails, else repeat

- Predictive Parsing (e.g., Fig. 2.17/1st Ed; Fig. 2.19/2nd Ed)
  - Select production based on the FIRST set of the RHS symbols
    - Left-corner parsing
  - FIRST(α) = {a | α =*=> a β} // set of possible 1st/beginning terminals of α
  - If multiple choices for A-production is possible, the production is selected unambiguously by the lookahead symbol based on the FIRST sets of the RHS’s of the A-production.
  - Non-backtracking is possible if the FIRST sets of RHS’s are disjoint.
Problems with Top-Down Parsing: $\varepsilon$ - Production

- How to match “Nothing”?  
  - $\varepsilon$ - Production: $X \rightarrow \varepsilon$

- Solution:  
  - Use $\varepsilon$ - Production as the default production when all other productions produce no match, that is  
  - Assumed that $X \rightarrow \varepsilon$ is applied when all other $X$-productions are tried but failed to match input  
  - WILL NOT advance the input pointer  
    - Lookahead remains the same current input symbol
Problems with Top-Down Parsing: $\varepsilon$ - Production

- How to match "Nothing"?
  - $\varepsilon$ - Production: $X \rightarrow \varepsilon$

```
S \xrightarrow{*} w \cdot X \cdot Y

\{ w \cdot \alpha \}
(2)  w \cdot Y
X \rightarrow \alpha

\Rightarrow \varepsilon
```

```
\text{input}
```
Problems with Top-Down Parsing: Left Recursion

- Left Recursion: unable to deliver a terminal symbol for matching
  - list → list + digit
  - list → digit
- Solution: Re-write left-recursion rules into right-recursion rules

- Example:
  - A → A α | β, // β α*: β followed by list of zero or more α’s

  Rewrite into …
  - A → β R // β : β followed by remaining (right-recursive) part
  - R → α R | ε // R: remaining part is a list of zero or more α’s

- But, Warning: Re-writing the grammar also changes the parse tree… (See more on later slides)
  - May NOT be an appropriate representation of input
    - May not be good for translation
  - May need to convert it into an appropriate syntax tree (for translation)
Problems with Top-Down Parsing: Left Recursion

- Example Left-Recursive Rules:
  - $A \rightarrow A\alpha \mid \beta$, // $A \Rightarrow \beta\alpha^*$: $\beta$ followed by list of zero or more $\alpha$’s

- Rewrite into Right-Recursive Form:
  - $A \rightarrow \beta R$ // $\beta$: $\beta$ followed by remaining (right-recursive) part
  - $R \rightarrow \alpha R \mid \varepsilon$ // $R$: remaining part is a list of zero or more $\alpha$’s
Problems with Top-Down Parsing: Translation - Parse Tree Associated with Right Recursion Rules

- **Warning:** Re-writing the grammar also changes the parse tree
  - May NOT be good for translation

- **Parse Tree:** Left-branching => Right-branching parse tree
  - All right to verify if an input statement is legal according to the grammar
  - May not be appropriate for representing the input and conducting translation
  - The parse tree drawn according to the new (right-recursive) grammar may not well represent the intended semantic or hierarchical relationships among symbols, although it helps to match legal input strings
  - May need to construct a syntax tree during parsing to help translation
    - Section 5.5, Fig. 5.29, p. 306, 1st Ed // SDT for constructing Syntax Tree
    - Section 5.3.1, Fig. 5.13, p. 321, 2nd Ed

- **Remember: Grammar & Parsing:**
  - (1) ensure legal input
    - matching G & build parse tree (a mirror of G)
  - (2) construct appropriate structure (for translation and/or other actions)
    - construct syntax tree if parse tree is not appropriate for translation
    - Example: convert right-recursive parse tree into left-recursive (LA) syntax tree
Parse Tree vs. Syntax Tree

- Parse Tree: (depend on grammar)
  - Input: T + T + T
  - G1: T ((+ T) (+ T) …)
    - E → T R'
    - R' → + T R'
    - R' → <null>
  - G2: ((T) + T) + T …
    - E → E + T
    - E → T

- Syntax Tree:
  - Abstract representation for syntax defined by G1/G2
  - Use operation as parent nodes and operands as children nodes
    - Operation-operand relationship: Easy for instruction selection in code generation (e.g., ADD R1, R2)
SDD for Infix-to-Postfix Translation (Left, Synthesized)

E.t = “95-2+”

E.t = “95-”

- T.t = “5”

T.t = “9”

5

+ T.t = “2”

E.t = “9”

9

T.t = “9”

2
TS for Infix-to-Postfix Translation (Left-Recursive)

```
9 - 5 + 2
 |
T
 |
9
```

```
print “9”
 |

print “-”
 |
5

print “5”
 |

print “+”
 |

print “2”
```
Translation of 9-5+2 to 95-2+
Problems with Top-Down Parsing: Translation Using Right Recursion Rules

- Right Recursion: difficult to evaluate / translate expressions that involves Left Associative operators

- Special Solution: Regard a semantic action which correspond to *synthesized* attributes as a terminal node when re-writing into left-recursion rules
  - Example: infix-to-postfix (next few slides..)
  - It works sometimes… Always check whether the semantic actions in the new productions really do the intended translation

- General Solution: construct an abstract left-branching syntax tree independent of the right-branching parse tree (i.e., concrete syntax tree) and evaluate semantic rules based on abstract syntax tree
  - More on “Top-Down Translation – Eliminating Left Recursion from a Translation Scheme” (5.5, 1st Ed.) (or 5.4/5.4.4, 2nd Ed.)
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

- Specification of Syntax (Grammar)
- Specification of Semantic Actions
- Parsing (based on Grammar)
- Translation (based on Semantic Rules)
A Translator for Simple Expression

A Syntax-Directed Translator for Infix-to-Postfix Translation
Initial TS for Infix-to-Postfix Translation

Production with \{Semantic Actions\}:

- `expr \rightarrow expr_1 + term` {print(‘+’)}
- `expr \rightarrow expr_1 - term` {print(‘-’)}
- `expr \rightarrow term`
- `term \rightarrow 0` {print(‘0’)}
- `term \rightarrow 1` {print(‘1’)}
- `\ldots`
- `term \rightarrow 9` {print(‘9’)}
Adapting the Translation Scheme to Right Recursive Form

- Re-write left-recursion rules into right-recursion rules
  - \( expr \rightarrow expr + term \{ \text{print(‘+’)} \} \)
  - \( expr \rightarrow expr - term \{ \text{print(‘-’)} \} \)

- Technique:
  - \( A \rightarrow A \alpha \mid A \beta \mid \gamma \), rewrite into
    - \( A \rightarrow \gamma R \)
    - \( R \rightarrow \alpha R \mid \beta R \mid \varepsilon \)

- Extending the transformation to include semantic actions (for synthesized attributes): carry the semantic action along in the transformation
  - \( A= expr \land \land \alpha = + term \{ \text{print(‘+’)} \} \)
  - \( \land \land \beta = - term \{ \text{print(‘-’)} \} \)
TS for Infix-to-Postfix Translation in Right-Recursive Form

Production with \{\text{Semantic Actions}\}:

\begin{align*}
\text{expr} & \rightarrow \text{term} \ \text{rest} \\
\text{rest} & \rightarrow \ + \ \text{term} \ \{\text{print}(\text{‘+’})\} \ \text{rest} \\
\text{rest} & \rightarrow \ - \ \text{term} \ \{\text{print}(\text{‘-’})\} \ \text{rest} \\
\text{rest} & \rightarrow \ \varepsilon \\
\text{term} & \rightarrow \ 0 \ \{\text{print}(\text{‘0’})\} \\
\text{term} & \rightarrow \ 1 \ \{\text{print}(\text{‘1’})\} \\
\vdotswithin{\rightarrow} \\
\text{term} & \rightarrow \ 9 \ \{\text{print}(\text{‘9’})\}
\end{align*}
Translation of 9-5+2 to 95-2+
TS for Infix-to-Postfix Translation (Right-Recursive)

Translation of
9-5+2 to 95-2+

Fig. 2.21 1st Ed, Fig. 2.24 2nd Ed
Top-Down Parsing Procedures

/* initialization */
Init_Recursive_Descent_Parsing() {
    lookahead = first_token(); // == next_token()
}

/* aux function: match terminal token and advance input pointer */
m_match (TOKEN t) {
    if (lookahead == t)
        lookahead = next_token();
    else
        error();
}

next_token()
{
    /* a lexical analyzer that return next input token */
}
Top-Down Parsing Procedures

main()
{
    Init_Recursive_Descent_Parsing();
    expr();               // match start symbol
    putchar(’\n’);
}

expr() { // start symbol
    /* <expr> ::= <term> <rest> */
    term(); rest();
}
Top-Down Parsing Procedures

rest() {
    if (lookahead == `+`) {
        /* <rest> ::= + <term> {print(`+`)} <rest> */
        match(`+`); term(); putchar(`+`); rest();
    } else if (lookahead == `-`) {
        /* <rest> ::= - <term> {print(`-`)} <rest> */
        match(`-`); term(); putchar(`-`); rest();
    } else {
        /* <rest> ::= ε */
        /* do nothing to simulate matching ε-production, when no other production was matched */
    }
}
}
Top-Down Parsing Procedures

term() {
    if (isdigit(lookahead)) {
        /* <term> ::= 0 {print(‘0’)
         |1{}|2{}|…|9{} */
        putchar(lookahead); match(lookahead);
        // NOT: match(lookahead); putchar(lookahead);
        /* or
         * digit=lookahead;
         * match(digit); putchar(digit);
        */
    } else {
        /* <term> ::= ε is NOT a production*/
        error();
    }
}

Switch (lookahead) {
    Case ‘0’:
        match(‘0’);
        putchar(‘0’);
        Break;
    Case ‘1’: // …
}
Source Code Optimization

- Tail Recursion: Call oneself at the end of a statement.
  - Optimization: replace procedure call with a GOTO statement

- General source code optimization:
  - Convert recursive functions into non-recursive versions
Source Code Optimization: Tail Recursion

```c
rest() {
    L: if (lookahead == '+') {
        /* <rest> ::= + <term> {print('+')} <rest> */
        match('+'); term(); putchar('+'); goto L /*rest()*/ ;
    } else if (lookahead == '-') {
        /* <rest> ::= - <term> {print('-')} <rest> */
        rest();
        match('-'); term(); putchar('-'); goto L /*rest()*/ ;
    } else { /* <rest> ::= ε */
        /* do nothing to simulate e-production, when no other production was matched */
    }
}
```
Source Code Optimization: Tail Recursion

rest()
{
    while( true ) { /* rest() call ⇔ loop */
        /* to match a list of "+/− term()" */
        if (lookahead == ’+’) {
            match(’+’); term(); putchar(’+’); //goto L
        } else if (lookahead == ’−’) {
            match(’−’); term(); putchar(’−’); //goto L
        } else {
            break;    // return from rest()
        }
    }
}

Source Code Optimization: Tail Recursion

```c
expr()
{
    /* <expr> ::= <term> <rest> */
    term();
    while(1) { /* rest() call ⇔ loop */
        /* to match a list of "+/− term()" */
        if (lookahead == '+') {
            match('+'); term(); putchar('+');
        } else if (lookahead == '-') {
            match('-'); term(); putchar('-');
        } else {
            break;
        }
    }
}
```
Program to translate an infix expression into postfix form

- Fig. 2.24, 1st Ed: C
- Fig. 2.27, 2nd Ed: Java
A Translator for Simple Expression

Lexical Analysis
Lexical Analysis

Interface between Parser and Lexical Analyzer (aka Scanner):

- Characters
- Lexical Analyzer
- Tokens & Attributes
- Parser

```
c=Getchar()
UnGetchar(c, stdin)
```

"\(\leq\)" vs. "\(<\)"

Call Lex()
Lexical Analysis

- **Task:** Convert input lexemes to stream of tokens
  - Lexeme: a sequence of characters (e.g., “hello”) that comprises a single token

- **Typical Functions:**
  - **Removal of white space and comments**
    - instead of writing productions that include spaces and comments
  - **Constants:** Digits into a single Token: `<Token ID + Token Value/Attributes>`
    - instead of writing productions for integer constants
    - “31+28+59” ⟷ `<num, 31> <+>, > <num, 28><+, > <num, 59>`
  - **Recognizing Identifiers and Keywords**
    - Identifiers: `count = count + increment` ⟷ `id = id + id (<id,"count"> <== <id,"…">…)`
    - Keywords: `begin, end, if, else` ⟷ `begin, end, if, else`
  - **Recognizing Operators/punctuations:** ‘>’, ‘<=>’, ‘<>’
    - Reading Ahead (in cases of multiple-character operators)
    - ‘>’ ⟷ ‘>=‘ or ‘>?‘ (? =/= ‘=‘)
    - Use a buffer from which the lexical analyzer can read and push back characters
    - One-character read-ahead usually suffices
Lexical Analysis: Typical Flow

- Lexical analyzer: also known as scanner

- Token scan() {
  - Skip white space;
  - Handle numbers; // return NUM, if identified
  - Handle reserved words and identifiers; // possibly return keyword or ID
  - // else … a single character operator at ‘peek’ (next input character)
  - Token t = new Token(peek); // return single character operator
  - Peek = blank
    - // convention: init. as blank on return if not pointing to next input char
    - // white spaces will be skipped at next scan() call
  - Return t;
}
Lexical Analysis: Example Codes

- Removal of white space and comments
  - Fig. 2.29, 2nd Ed.
- Constants: Grouping digits into integers
  - Fig. 2.30, 2nd Ed
- Recognizing Identifiers and Keywords
  - Fig. 2.31, 2nd Ed

- Program: A lexical analyzer
  - Fig. 2.28, 1st Ed. (in C), and Fig. 2.30 (in Pascal, with Keyword/ID recognition)
  - Fig. 2.34 + Fig. 2.35, 2nd Ed. (in Java)
Example: Representation of “Tokens” in C

- **Token:**
  - A unique Numerical/Symbolic Representation of Input String
  - `typedef enum token_types {
      BEGIN, END, IF, THEN, ELSE, ID,
      NUM, LPAREN, RPAREN,
      SEMICOLON, COMMA, ASSIGNOP, PLUSOP,
      MINUSOP, ENDSCAN
  } token;` // or ...
  - `#define IF 220 // reserved word`
  - `#define NUM 256 // numbers`

- **Lexical Analyzers (or Scanners):**
  - Return a “token”, instead of a string, upon acceptance of an input unit
  - `extern token scanner(void);` // typical prototype
    - `12 + 23 => NUM PLUSOP NUM (&other attributes)`
“Token” Class and subclasses in Java: Common Attribute

- Package lexer;
- Public class Token {
  - Public final int tag; // type of token; common token attribute
  - Public Token (int t) { tag = t; }
}

- Creating token object:
  - new Token(‘+’) // use ASCII code as tag value for operators

- Public class Tag { // constant token ID values
  - Public final static int
  - NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
  - … // single character operators: below 256 (e.g., as ASCII)
  - … // other types of tokens: use numbers above 256 as tag value
}

- Equivalent C #define:
  - #define NUM 256
“Token” Class and subclasses in Java: Special Attributes

- Package lexer;

- Public class Num extends Token {
  - // Public final int tag; // common attribute in super class Token
  - Public final int value; // special attribute of Num tokens
  - Public Num (int v) { super(Tag.NUM); value = v; }

- } // Identifiers & reserved words
  - // if ( Hello == 3 ) ⇔ < tag = WORD, lexeme = “Hello” >
Recognizing Identifiers and Integer Literals

While ((in_char = getchar()) != EOF) {
    if (isspace(in_char)) continue;  // Remove spaces
    else if (is_alpha(in_char)) {  // ID: starts with [a-zA-Z]
        for (c=getchar(); isalnum(c) || c=='_'; c=getchar())
            ;  // ID is a letter followed by [a-z,A-Z,0-9,_] *
        ungetc(c, stdin);  // step back
        return ID;
    }
    else if (isdigit(in_char)) {  // INT: starts with [0-9]
        while (isdigit((c = getchar())))
            ;
        ungetc(c, stdin);  // step back
        return NUM;
    }
    else
        lexical_error(in_char);
}
Recognizing Operators, Comments and Delimiters

While ((in_char = getchar()) != EOF) {
    if (isspace(in_char)  continue; // Remove spaces
    else if (isalpha(in_char){...//get ID([a-zA-Z][a-zA-Z0-9]*)})
    else if (isdigit(in_char){...//get NUM ([0-9]+))
    else if (in_char == ‘(‘) return LPAREN;
    else if (in_char == ‘)’) return RPAREN;
    else if (in_char == ‘;’) return SEMICOLON;
    else if (in_char == ‘,’) return COMMA;
    else if (in_char == ‘+’) return PLUSOP;
    else if (in_char == ‘-’) return MINUSOP;
    else if (in_char == ‘:’) return get_assignment_op(); //`:=`
    else lexical_error(in_char);
}
Recognizing Operators, Comments and Delimiters

get_assignment_op() {
    c = getchar();
    if (c == '=' ) return ASSIGNOP;  // "\:="
    else {
        ungetc(c, stdin);
        lexical_error(in_char);
    }
}
Symbol Table Management

- Symbol table:
  - Store information about source constructs
  - Collected incrementally by analysis phases & used by synthesis phases
  - Need to support multiple declarations of the same identifier

- Lexemes + Attributes == Token
  - Integer valued ID + token value or string pointer
  - Type of identifier, usage (label, variable, procedure), memory address

- Operations:
  - Insert (Lexeme_String, TOKEN)
  - Lookup (Lexeme_String)
  - Example: (Fig. 2.29, 1st Ed.)
Lexical Analyzer

- Fig. 2.28: eliminating white space & collecting numbers
  - If (t == ‘\n’) lineno ++
  - If (isdigit(t))
  - { tokenval = t - ‘0’; /* 1st digit */
  - … tokenval = tokenval * 10 + t - ‘0’; /* succ digits */ }

- Fig. 2.30: symbol table management
  - Lookup(lexbuf) for existence & insert(lexbuf, ID) if is new

- P. 74: lexan() lexical analyzer for infix-to-postfix translation
A Simple Compiler for Infix-to-Postfix Translation
(Sec. 2.9, [Aho et. al, 86])

- Specification (SDTS= CFG + Sem. Act.) for translation: Fig. 2.35
- SDTS after eliminating left-recursion: Fig. 2.38
- Parser + Scanner + STM: Sec. 2.9, pp. 73 [Aho 86], based on Fig. 2.38
  - Modules: Fig. 2.36
  - Token definitions
  - Lexan(): lexical analyzer
  - Parse(): parser (with translation codes for translation scheme)
    - Expr()
    - Term()
    - Factor()
  - Match(): interface between lexical and syntax analysis
    - call lexan() from Parse() and substructures
  - Emit(): codes for implementing translation scheme
  - Symbol Table Management (STM):
    - Lookup()
    - Insert()
    - Init(): insert keyword table
  - Main: init() symbol table & call Parse() to start (which calls lexan() whenever necessary)
Symbol Table Management
(more in 2nd Ed.)

- Symbol Table Per Scope
  - Need to support multiple declarations of the same identifier
    - Subclass can redeclare a method to overwrite a method in a superclass
  - Implement scopes: a separated symbol table for each scope

- Example:
  - identify/print correct type for each use of an identifier
  - Input:
    ```java
    { int x; char y; { bool y; x; y; } x; y; }
    ```
  - Output: // strip declarations, and use of x, y with their correct type
    ```java
    { { x:int; y:bool; } x: int; y:char; }
    ```

- Chained Symbol Tables
  - Most-closely nested rule:
    - An identifier x is in the scope of the most-closely nested declaration of x
    - Examining blocks inside-out, starting with the block in which x appears
  - (Example 2.15, Fig. 2.36, 2nd Ed)
  - Implementation (in Java): Fig 2.37, 2nd Ed
Symbol Table Management
(more in 2nd Ed.)

- Use of Symbol Tables for translating a language with blocks
  - Fig. 2.38, 2nd Ed.

- Example:
  - identify/print correct type for each use of an identifier
  - Input: 
    ```
    decl. use
    { int x; char y; { bool y; x; } x; y; }
    ```
  - Output: // strip declarations, and use of x, y with their correct type
    ```
    { { x:int; y:bool; } x: int; y:char; }
    ```
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

Code Generation for a virtual machine
or a virtual stack machine
A Simple One-Pass Compiler
(A Simple Syntax-Directed Translator)

Translation: IR => codes
- IR:
  - IR:
    - tree: parse tree or (abstract) syntax tree
    - linear representation:
      - 3-address codes
      - stack machine codes
        - e.g., java bytecode
  - Translation:
    - semantic rules & semantic actions
    - semantic attribute evaluation
Code Generation for an Abstract Machine

- Intermediate representation: (IR)
  - Generated by front end, and used by back end for generating target machine program

- Two Kinds of Intermediate Representations:
  - Tree (Hierarchical Representations):
    - Parse Tree (Concrete Syntax Tree), or
    - Syntax Tree (Abstract Syntax Tree)
  - Linear Representations: (intermediate codes)
    - Code for an abstract stack machine (or stack-based language, like FORTH)
      - Stack machine codes: Push, pop, operator operand-1 operand-2
        - (operators: +, -, *, /, Boolean operators) …
      - Easy for evaluating simple expressions
    - Code for non-stack machine that can easily be mapped to real machines:
      - 3-address codes: (t1 = a +b)
      - Easily mapped to x86-like instructions, for instance

- Translation Scheme for Code Generation: Two Sub-Tasks --
  - Construct syntax tree by constructing nodes for each construct
    - Optional/Additional static checking
  - Generate codes by traversing the syntax tree, starting from the root node
Constructing Syntax Trees (& Static Checking)
Constructing Syntax Trees: Basic Ideas

- Math expressions/Math operators: Generate a node for each operators and connect its operands as children
  - \( a + b \) \( \Leftrightarrow \) ‘+’: parent; ‘a’, ‘b’: children \( \Leftrightarrow \) logical form: +(a, b) or ADD(a, b)
  - \( E_1 \) \( \text{op} \) \( E_2 \) \( \Leftrightarrow \) op: parent; \( E_1, E_2 \): children \( \Leftrightarrow \) logical form: \( \text{op}(E_1, E_2) \)

- Control statements/Generalized operators: for any construct --
  - Each construct is represented by a parent node, with semantically meaningful components of the construct as its children
    - \( \Leftrightarrow \) regarding labels of constructs as Generalized operators, like the ‘IF’ operator
  - E.g., \( \text{while} (expr) \) \( \text{stmt} \) \( \Leftrightarrow \) parent: \text{WHILE}; children: \( expr \) and \( stmt \)
    - \( \Leftrightarrow \) logical form: \( \text{WHILE}(expr, stmt) \)
  - E.g., \( \text{IF} \ (E) \ A \) \( \Leftrightarrow \) parent: \text{IF}; children: \( E \) and \( A \)
    - \( \Leftrightarrow \) logical form: \( \text{IF}(E, A) \)
  - E.g., \( \text{IF} \ (E) \) then \( A \) else \( B \) \( \Leftrightarrow \) parent: \text{IF-ELSE}; children: \( E \) and \( A \) and \( B \)
    - \( \Leftrightarrow \) logical form: \( \text{IF-ELSE}(E, A, B) \)
Constructing Syntax Trees

- Operators: Math + Generalized (Fig. 2.41, 2\textsuperscript{nd} Ed, and more...)
  - Assignment: `=`
  - Mathematical: `+`, `-`, `*`, `/`, `%`
  - Bit operations: `&`, `|`, `!`
  - Boolean/Conditional: \textbf{and}, \textbf{or}, \textbf{not} ( `&&`, `||`, `!` )
  - Relational: `==`, `!=`, `<`, `<=`, `>=`, `>`, `<>
  - Unary minus: `-` (e.g., `-3.14`)
  - Array access: `[ ]` (e.g., `a[i+j*k] = b[p+q*r]`)
  - Flow control operators: \textbf{if} (without “else”), \textbf{ifelse} (with “else”), \textbf{while}, \textbf{do}, \textbf{for}, ...
    - Operators: keywords as operators
    - Operands: conditional expression and body of executable statements
Constructing Syntax Trees: Example SDT

- Construction of syntax tree for program constructs [Fig 2.39]

  - **program** → **block** {return block.n; }
  - **block** → ‘{ decls stmts ’} {block.n = stmts.n; }
  - **stmts** → stmts, stmt {stmts.n = new Seq(stmts.n, stmt.n); }
  - **stmt** → expr ; {stmts.n = new Eval(expr.n); }
  - **stmt** → if ( expr ) stmt1 {stmts.n = new If(expr.n, stmt1.n); }
  - **stmt** → while ( expr ) stmt1 {stmts.n = new While(expr.n, stmt1.n); }
  - **stmt** → do stmt1 while ( expr ); {stmts.n = new Do(stmt1.n, expr.n); }
  - **stmt** → block {stmts.n = block.n; }
  - **expr** → rel = expr1 {expr.n = new Assign(‘=’, rel.n, expr1.n);}
  - **expr** → rel {expr.n = rel.n;}
  - **rel** → rel1 < add {rel.n = new Rel(‘<’, rel1.n, add1.n);}
  - **rel** → rel1 <= add {rel.n = new Rel(‘<=’, rel1.n, add1.n);}
  - **rel** → add {add.n = add1.n;}
  - **add** → add1 + term {add.n = new Op(‘+’, add1.n, term.n);}
  - **add** → term {add.n = term.n;}
  - **term** → term1 * factor {term.n = new Op(‘*’, term1.n, factor.n);}
  - **term** → factor {term.n = factor.n;}
  - **factor** → ( expr ) {factor.n = expr.n;}
  - **factor** → num {factor.n = new Num(num.value);}

```
```
a+b>c = p*q<r
```
allowed ?
```
```
  Do(expr.n,stmt1.n); ok??
```
Constructing Syntax Trees: Statements

- Construction of syntax tree for program constructs
  - Main Constructs: **statements**, blocks, expressions
    - SDD: Fig. 2.39, 2nd Ed.
    - Example syntax tree: Fig. 2.40, 2nd Ed.

- Syntax Trees for Statements
  - **if** (without “else”), **ifelse** (with “else”), **while**, **do**, **for**, …
    - Define a Constructor for each generalized operator (e.g., **if** ↔ If(expr,stmt))
    - Use constructor parameters for linking operands nodes in abstract syntax tree

- Translation **Scheme**:
  - **stmt** → if ( expr ) **stmt**₁
    - { **stmt**₁.n = new If(expr.n, **stmt**₁.n); }
      - Create a new node labeled IF with the nodes expr.n and **stmt**₁.n as its children
  - **stmt** → expr ;
    - { **stmt**.n = new Eval(expr.n); }
      - Express statements that do not begin with a keyword (e.g., “a = b + c * d”, “a > b”)
        - Involving assign, cond, rel, math-op, … operators
        - Represented by a hierarchical sub-tree, rooted at an expr node
    - Define a new operator eval and class Eval to represent expressions that are statements, and
    - Create nodes that depend on the math./Boolean operators involved in the expression. (See more on “Syntax Trees for Expressions”)

Tree structure depends on expression

Fixed tree structure
Constructing Syntax Trees: Blocks

- Construction of syntax tree for program constructs
  - Main Constructs: statements, blocks, expressions
    - SDD: Fig. 2.39, 2nd Ed.
    - Example syntax tree: Fig. 2.40, 2nd Ed.

- Representing Blocks in Syntax Tree
  - Block: sequence of statements enclosed by ‘{’ and ‘}’
    - block → ‘{ stmts }’
      \{ block.n = stmts.n; \}
    - stmt → block
      \{ stmt.n = block.n; \} // block can be nested
  - Stmts: represented as a leaf null (empty statement) and a seq (sequence of statements)
    - stmts → stmts, stmt
      \{ stmts.n = new Seq(stmts.n, stmt.n); \}
    - | \epsilon
      \{ stmts.n = null; \}

- Note: Declarations in Blocks: block → ‘{ decls stmts }’
  - Normally, NO nodes are created and attached to syntax trees
  - Therefore, no node-creation attribute is defined in the above SDD
  - Information is incorporated into symbol table (Fig. 2.38, 2nd Ed.)
Constructing Syntax Trees: Expressions

- Construction of syntax tree for program constructs
  - Main Constructs: statements, blocks, expressions
  - SDD: Fig. 2.39, 2nd Ed.
  - Example syntax tree: Fig. 2.40, 2nd Ed.

- Syntax Trees for Expressions
  - Operators: assign, cond, rel, op, not, minus, access ([ ]⇒a[i]) operators
  - Operators nodes are created by an Assign/Rel/Op class, with operands as their children

- Translation Scheme (e.g., for ‘=’, ‘<’, ‘+’, ‘*’)
  - $expr \rightarrow rel = expr_1$
    
  - $rel \rightarrow rel < add$
  
  - $add \rightarrow add_1 + term$
  
  - $term \rightarrow term_1 * factor$

  ```
  {expr.n = new Assign(‘=’, rel.n, expr_1.n);} 
  {rel.n = new Rel(‘<’, rel_1.n, add.n);} 
  {add.n = new Op(‘+’, add_1.n, term.n);} 
  {term.n = new Op(‘*’, term_1.n, factor.n);} 
  ```
Checking Syntax Tree: Static Checking

- Consistency checks at compilation time (catching error earlier)
  - Checking semantics while constructing syntax tree, including
    - Syntactic checking: meaningful syntactic components?
      - There is more to syntax than grammar.
    - Type checking: compatible data type?
  - Syntactic Checking (e.g., “x = a + b” vs “2 = a + b”)
    - i = i + 1; copy value of ‘i’ (+1) to memory location for ‘i’
    - L-value: must be memory locations/addresses
      - 2 = a + b is bad; 2 has no l-value
      - a + b > c = p * q < r (allowed by G @ Fig 2.39??)
    - R-value: must be values of various data types (integer/character/string)
  - Type Checking
    - Ensure compatible types between operands
    - Type is checked when a node is constructed (using a .type attribute)
      - E → E₁, rel E₂ {if ( E₁.type == E₂.type ) E.type = boolean else error; }
    - Type conversion (“Coercion”)
      - 2 * 3.14 => 2.0 * 3.14; rate * 60 => rate * 60.0; // float rate;
    - Overloading: operator with different meanings depending on its context (determined by checking types of operands and result)
      - Addition: 2 + 3, vs
      - String concatenation: “Hello” + “World !”
      - z = x + y: (concatenation if x, y, z: string; addition: if x, y, z: numerical)
Generating Three-Address Codes
Generating Three-Address Codes

- Code generation as SDT: Syntax Tree => 3-address Codes
  - traverse syntax tree, execute semantic actions to generate 3-addr codes
- Three-address instructions: (assumed typical instruction set)
  - Binary ops:
    - \( x = y \text{ op } z \)
      - \( x, y, z \): names, constants, compiler-generated temporaries
      - \text{op: (binary) operator (e.g., '+', '-', '>', '&&')}
  - Array accesses instructions:
    - \( x [y] = z \) // NO complicated array access instructions like \( x[i+j*k] = z \)
    - \( x = y[z] \) // NO complicated instructions like \( x = y[p+q*r] \)
  - Flow control instructions:
    - ifFalse x goto L // conditional, x: is a \text{simple} variable or temp or constant
    - ifTrue x goto L // conditional // NO support to complicated x like ifTrue a>b go...
    - goto L // unconditional
  - Copy value:
    - \( x = y \)
- Two major constructs in code generation
  - Statements: start with reserved words (if, for…) for flow control
  - Expressions: mathematical/Boolean expressions, array access, assignment
Generating Three-Address Codes

- Code generation after tree construction: Two major types --
  - **Statements**: start with reserved words (*if, for* …) for **flow control**
    - **Code Gen. Task**: arrange execution flow of embedded conditional statements and math/Boolean expressions
      - By inserting some conditional test instructions around embedded statements/expressions
  - **Expressions**: mathematical/Boolean **expressions**, **array access**, **assignment**
    - **Code Gen. Task-1**: generate codes that evaluate **values** (*R-values*) of expressions
      - To the right of an ‘*='* operator: math/Boolean expressions
      - By calling an **E.value()** function, for example
    - **Code Gen. Task-2**: generate codes that evaluate **addresses** (*L-values*) of memory locations
      - To the left of an ‘*='* operator: simple variables and array elements
      - By calling an **E.value()** function
    - **Copy R-values to L-values**, if necessary
  - **Blocks**: DO NOT generate any codes
    - Only symbol table management for declarations
Generating Three-Address Codes: Statements

Translation of Control “Statements”

- HowTo: Using jumps to implement flow control through the statements

- Example: if \((\text{expr})\) then \(\text{stmt}_1\)  
  - Code to compute \(\text{expr}\) into \(x\)
  - ifFalse \(x\) goto after
  - Code for \(\text{stmt}_1\)
  - after: …

- General actions: TS for each X-Stmt: (X: IF, IF-ELSE, WHILE, FOR…)
  - Constructor: \(X::X()\) – Node generation (for syntax tree construction)
    - connect current node (parent) and children nodes, and
    - create unique new labels (e.g., “after:"’) required for GOTO instructions
  - Gen() function: \(X::\text{gen}()\) – Code generation
    - generate codes for current statement

- \(\Leftrightarrow\) If-Stmt, Fig. 2.43 (2nd Ed.), (cont. at next page)
Generating Three-Address Codes: IF

Translation of “IF-Statements”

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 + ":");
    }
}

Methods of Nodes ⇔ Fig. 2.43 (2nd Ed.)
- Constructor: connect current node (parent) and children nodes, and create unique new labels (to be used by Gen())
- Gen() function: generate codes for current statement

Structure of Nodes:
- Children nodes in syntax tree: E, S: point to left, right children
- Local nodes for code generation: E.rvalue(), E.lvalue(), …: to emit codes for expression, and get new temp nodes
Generating Three-Address Codes: IF

Example: If \(( y < z )\) stmt ;

\[
x := \text{Code}(\text{expr}) \\
\text{ifFalse} x \text{ goto after} \\
\text{Code}(\text{stmt1}) \\
\text{after:}
\]

\[
x := \text{Code}(E) \\
\text{ifFalse} x \text{ goto after} \\
\text{Code}(S) \\
\text{after:}
\]

\[
x, t: \text{temp variables used by code generation actions} \\
\text{(in various .gen(), .rvalue(), .lvalue() … methods)}
\]
Generating Three-Address Codes: Labels & S.gen()

- $x := \text{Code(expr)}$
- ifFalse $x$ goto after
- Code(stmt1)
- after:

- Code(stmt1) $\Leftrightarrow$ if(e2)s2
- $x2 := \text{Code(e2)}$
- ifFalse $x2$ goto after2
- Code(s2)
- after2:

S.gen(): generate codes for the S part (which can be another if-statement, recursively)

- after = newlabel();
- // generate a unique label on each call
Generating Three-Address Codes: Expressions

- Translation of Math/Boolean “Expressions”
  - HowTo: Generate one 3-address instruction for each operator node in the syntax tree for the expression

- Types of Operators in Expression
  - Binary op
    - generate codes to compute its value (R-value)
  - Array access op
    - generate codes to compute its address (L-value) or value (R-value), depending on whether it is located at the LHS/RHS of an ‘=’ op.
    - Including index computation: generate codes for the R-value of the index sub-expression
  - Assignment op
    - L-value(LHS) = R-value(RHS)
      - Generate codes to compute address of LHS & value of RHS
      - And an additional instruction to assign tmp for value to the address
  - Constants & Identifiers: no code is generated
    - Their values or addresses are represented by themselves
Generating Three-Address Codes: Expressions

- Translation of “Expressions” needs two code generation functions: Rvalue(), Lvalue()
  - Rvalue(x): generates codes to compute x into a temporary, and returns a new node representing the temporary.
  - Lvalue(x): generates codes to compute the subtrees below x, and returns a new node representing the address for x
    - Only ID & ARRAY ACCESS need both functions
    - ID: The lvalue and rvalue of ID (or rvalue of constant) is itself

- Example:
  - Rvalue( a[i+4] ) ⇔ t1= i+4 ; t2=a[t1]; return(“t2”); // the value
    - So that...(x=) a[i+4] * 2 ⇔ t1= i+4 ; t2=a[t1]; t3 = t2 * 2; (x=t3)
  - Lvalue( a[i+4] ) ⇔ t1= i+4 ; return(“a[t1]”); // the address
    - So that... a[i+4] = 2 ⇔ t1= i+4 ; a[t1] = 2;

- Implementation Note: (1st. Ed)
  - Rvalue(x) can call Lvalue(x) and add t2=Lvalue(); then return(“t2”) since their codes are almost identical.
Generating Three-Address Codes

- Math Expression: Binary Ops
  - For a node $x$ of type $Expr$ with a binary op, an instruction is emitted to compute the value at node $x$ into a compiler generate temporary name
    $\Leftrightarrow$ Code to evaluate $Expr$; temp := value_evaluated

- Example: $i - j + k \Leftrightarrow$ syntax tree: $[[ i - j ] + k ]$
  - $'-' \Leftrightarrow t1 = i - j \Leftrightarrow t1 = \text{op}('-', i, j)$
  - $'+' \Leftrightarrow t2 = t1 + k \Leftrightarrow t2 = \text{op}('+', t1, k)$
Translation of Expression --
An Example: i-j+k

Codes(i-j+k):
“t1= i-j”
“t2 = t1+k”

New t1:
i: rvalue(‘i’)
j: rvalue(‘j’)
code: “t1= i-j”

New t2:
t1: rvalue(‘i-j’)
k: rvalue(‘k’)
code: “t2 = t1+k”
Generating Three-Address Codes: Array & “=“

- Array Access / Assignment Ops
  - Need to distinguish between l-values & r-values
  - May need two code generating functions for l-value & r-value

- Example (r-value): (…) = a[i+4] * 2  // as operand of operator
  - ⊸ t1 = i+4
  - ⊸ t2 = a[t1]  // compute r-value of a[i+4] into a temp, t2
  - ⊸ t3 = t2 * 2

- Example (l-value): a[i + 4] = 2  // as destination of assignment
  - ⊸ t1 = i + 4
  - ⊸ a[t1] = 2  // computing l-value (address) of a[i+4] as destination
  - NOT simple variable substitution:
    - ⊸ t1 = i+4;
    - ⊸ t2 = a[t1]; [WRONG!!]  // value to tmp
    - ⊸ t2 = 2;  [WRONG!!]  // then assign to tmp
      - Rvalue(t2)=Rvalue(a[t1]) BUT Lvalue(t2) != Lvalue(a[t1])
Translation of Array Access – (w/o index calculation)
An Example: a[i] * 2

Codes(a[i]*2):
“t1= a[i]”
“t2 = t1*2”

New t2;
t1: rvalue(‘a[i]’)
2: rvalue(‘2’)
code: “t2 = t1*2”

New t1;
i: rvalue(‘i’)
code: “t1= a[i]”
Translation of Array Access – (w/o index calculation)
An Example: \( a[i] = 2 \)

Codes(\( a[i] = 2 \)):
“\( a[i] = 2 \)”

New \( a[i] \);
\( i \): rvalue(‘i’)
lvalue: “\( a[i] \)”

New \( t2 \);
\( a[i] \): lvalue(‘a[i]’)
2: rvalue(‘2’)
code: “\( a[i] = 2 \)”
Translation of Array Access – (w/ index calculation)
An Example: $a[i+4] \times 2$

Codes($a[i+4] \times 2$):
“$t1 = i + 4$”
“$t2 = a[t1]$”
“$t3 = t2 \times 2$”

New $t3$;
t2: rvalue(‘$a[i+4]$’)
t3: “$t3 = t2 \times 2$”

New $t2$;
t1: rvalue(‘$i+4$’)
(code: “$t1 = i + 4$”)
t2: “$t2 = a[t1]$”
Translation of Array Access – (w/ index calculation)
An Example: \(a[i+4] = 2\)

Codes(\(a[i+4]=2\)):
“\(t1= i + 4\)”
“\(a[t1]=2\)”

New \(t2\);
\(a[t1]\): lvalue(‘a[i+4]’) 2: rvalue(‘2’) code: “a[t1]=2”

E1 .lvalue()
Generating Three-Address Codes: Expressions

- Translation of “Expressions” -- Summary
- Lvalue node: code to compute address
  - Id: a (= ...) ⇔ a (= ...)
  - Access: y[z] (= ...) ⇔ z′=rvalue(z); y[z′] (= ...)
- Rvalue node: code to compute value
  - Id/const: (a = ...) y ⇔ (a = ...) y
  - Op/Rel: (a = ...) y op z (...) ⇔ “t = rvalue(y) op rvalue(z)”
  - Access: (a = ...) y[z] (...) ⇔ z′ = rvalue(z); ‘t = y[z]’
  - Assign: (a = ...) y = z (...) ⇔ z′ = rvalue(z) ; “lvalue(y) = z“
- Example: a[i] = 2 * a[j – k]

\[
\begin{cases}
  t3 = j – k \\
  t2 = a[t3] \\
  t1 = 2 * t2 \\
  a[i] = t1
\end{cases}
\]

\[
\begin{array}{c}
\text{lvalue(x): generates codes to compute the subtrees below x, and returns a new node representing the address for x} \\
\text{rvalue(x): generates codes to compute x into a temporary, and returns a new node representing the temp.}
\end{array}
\]

See details on next slide.
Generating Three-Address Codes

Example: \(a[i] = 2 \times a[j - k]\)

- \(t3 = j - k\)
- \(t2 = a[t3]\)
- \(t1 = 2 \times t2\)
- \(a[i] = t1\)

\(rvalue(x)\): generates codes to compute \(x\) into a temporary, and returns a new node representing the temp.

\(lvalue(x)\): generates codes to compute the subtrees below \(x\), and returns a new node representing the address for \(x\)

\(rvalue(‘2*a[j-k]’)\)

Assign.rvalue()
Better Code for Expressions

- Task: Improve function `rvalue()` to generate fewer 3-address instructions.
- Target: Reduce number of copy instructions in a subsequent optimization phrase.
  - E.g., `t = i + 1; i = t` => `i = i + 1`
- HowTo: Taking context into account: generate tmps only when necessary:
  - `x = a + b + c + d` ⇔ `x = (((a + b) + c) + d)` ⇔ the last `+' operation before doing assignment do NOT need a tmp for intermediate result, since the result can be directly assigned to `x`
    - `t1 = a + b; t2 = t1 + c; t3 = t2 + d; x = t3`
    - Better code ⇔ `t1 = a + b; t2 = t1 + c; x = t2 + d`
  - Use a null variable and be replaced by a tmp or a variable
    - Replaced by tmp: current op IS NOT `='
    - Replaced by variable: current op IS `=' (assigned to that variable)
Simple Code for Expressions

\[ x = a + b + c + d \]
Better Code for Expressions

- $x = a + b + c + d$

```
x = a + b + c + d

= 
  
  x +
  
  +
  
  d

  t1 = a + b
  t2 = t1 + c
  x = t2 + d

null => x

+ c

  t1 = a + b
  null = t1 + c
  null = t2 + d

null => t2

a b

  null = a + b

null => t1
```
Code Generation for an Abstract Stack Machine
Code Generation for an Abstract Stack Machine

- Stack machine configuration: (Fig. 2.31)
  - [1] Instruction memory: store program consisting of instructions
    - Instruction Set: arithmetic, stack operations, control flow, ...
    - Arithmetic: simulate postfix evaluation
  - [2] Stack: for evaluating arithmetic operations on their values
    - Push operands when encountered
    - Pop top-K values for applying K-ary operations
    - Push result to Top of Stack (TOS):
      - Arithmetic: Numerical Value at stack top
      - Boolean expression: True/False at stack top
  - [3] Data memory:
    - save variable values (r-values) at particular addresses (l-values)

- Code generation process:
  - Statement => postfix-like expression => stack machine code (.EXE)

- Execution of the stack machine codes:
  - Read codes & Evaluate in postfix manner using stack operations
  - Stack top keeps the result of the most recent instruction (numerical of Boolean)
    - Generated codes must be evaluated to leave such a result at stack top
Mathematic Expression: L-value vs. R-value

- L-value vs. R-value -- Example: A := A + 1
  - Left A: address of A & Right A: value of A
  - L-value: address & R-value: value

- Instructions for Stack manipulation:
  - Push V: push V onto stack
  - Rvalue L: push content of data location L
  - Lvalue L: push address of data location L
  - Pop: throw top of stack
  - ":=": stack[top-1] (L-value) := stack[top] (R-value)
  - Copy: push stack[top] (duplicate a copy of stack top at top)
  - Operators: ‘+’, ‘-’, ‘*’, ‘/’, ...

- Translation: Into stack instructions so as to be evaluated using a stack machine
  - E.g., a+b ⇔ a b + ⇔ rvalue a, rvalue b, +
  - E.g., c := d ⇔ c d := ⇔ lvalue c, rvalue d, :=
  - E.g., c := a + b ⇔ c a b + := ⇔ lvalue c, rvalue a, rvalue b, + , :=

- Evaluation/Execution: (by abstract machine)
  - Push addr(c), Push val(a), Push val(b), Add(stack[top-1], stack[top], &result), top -=2, Push val(result), assign(stack[top-1], stack[top]), top -=2
Translation of Expressions

- Translation of Simple Math./Boolean Expression:
  - Code to evaluate $E + F$ == code to evaluate $E$ || code to evaluate $F$ ||
    instruction to add their results
  - Code$(E+F)$ == Code$(E)$ || Code$(F)$ || ‘+’
  - Similar for ‘-‘, ‘*‘, ‘/‘, div, mod, Boolean AND, OR
  - Example: $a + b$ ↔ $(a + b)$ codes:
    - Rvalue $a$
    - Rvalue $b$
    - +

- Translation of Simple Assignment Operation
  - Code $(L := R$-expr$)$ == $L$-value$(L)$ || Code$(R$-expr$)$ || ‘:=’

- Translation of Complex expression: Parent → Child-1 op Child-2

- Example: $day := (1461*y) \text{ div } 4 + (153*m+2) \text{ div } 5 + d$
Translation of Complex Expressions

- Complex expression: Parent → Child-1 op Child-2

- Example: day := (1461*y) div 4 + (153*m+2) div 5 + d

Postfix: &day 1461 y * 4 div 153 m * 2 + 5 div + d + :=

- Lvalue day
- Push 1461
- Rvalue y
- *
- Push 4
- Div
- Push 153
- Rvalue m
- *
- Push 2
- +
- Push 5
- Div
- +
- Rvalue d
- +
- :=
Translation of Control Flow

- Stack machine instructions for flow control: conditional or not
  - Conditional Jump:
    - jump or not depending on the Boolean value on the stack top (which would be produced by previous operations)

- Instructions:
  - Label L: label for instruction (generated automatically, say by newlabel())
  - Goto L: next instruction is taken from statement with label L
  - GoFalse L: pop stack[top] & jump if it is zero (FALSE)
    - If (stack[top] == 0) Goto L
  - GoTrue L: pop stack[top] & jump if non-zero (TRUE)
    - If (stack[top] != 0) Goto L
  - Halt: stop execution
  - GoFalse/GoTrue: unary operator that consumes a Boolean value at stack[top]
Translation of Control Flow

Translation of IF-THEN statements

- SDD: Stmt → if expr then stmt₁
- SDD: { lab := newlabel(); stmt.t := expr.t || ‘gofalse’ lab || stmt₁.t || ‘label’ lab}

Fig. 2.33 (Left)

TS: … (p. 67 & Algorithm Fig. 2.34)

Stmt → if

expr { out:= newlabel; emit(‘gofalse’, out); }
then
stmt₁ { emit(‘label’, out); }

Code(expr) GoFalse Out Code(stmt1) Label Out
Translation of Control Flow

- Translation of WHILE statements
  - While (expr) stmt == T: if (!expr) goto END else {stmt ; goto T} END:
  - Fig. 2.33 (Right)

```
Label test
Code(expr)
GoFalse Out
Code(stmt1)
Goto test
Label Out
```
Translation of Control Flow

- Translation of FOR-LOOP statements
  - Exercise 2.14

- Translation of DO-WHILE-LOOP statements
  - Exercise
Translation of Control Flow

- Translation of Boolean Constant:
  - Leave a true(T) or false(F) value at stack[top] as outcome upon completion

- Simple Translation: (like other binary operators do)
  - Code(Expr1 OR Expr2) == Code(Expr1) || Code(Expr2) || ‘OR’

- Efficient Translation of logical OR statements
  - Expr1 OR Expr2 == if (expr1) then return(true) else expr2
    | == return(expr1 ? True : expr2 )
  - Code(Expr1) || ‘copy’ || gotrue lab || ‘pop’ || Code(expr2) || ‘label’ lab
    - Translation for AND ?? (your exercise)
Translation of Function Calls and Function Definitions (...more in Chapters 7 & 8)

- **Function/Macro Definition**
  - Return_type myFunction(a, b, c) { // formal parameters
    dcl_local_vars; // allocation of variable storage, scope
    /* statements */
    Return (return_expression)
  }

- **Function Call (in another function or main program)**
  - P := ...; q := ...; r := ...;
  - X := myFunction(P, q, r) // call with actual/real parameters

- **Code Generation: Function call**
  - Push P, Push q, Push r // pass real parameters to stack
  - Call myFunction
  - Pop X // get return value from stack

- **Code Generation: Function definition (myFunction)**
  - { Pop c, pop b, pop a // copy real parameters to local/formal variables
    // Code(myFunction)
    Push return_value // pass return value via stack
    RET }

Translation of Function Calls and Function Definitions (...more in Chapters 7 & 8)

- Translation of Function Calls & Definitions:
  - Insert codes around function call and within function definition to implement parameter passing

- Parameter Passing:
  - Call-by-Value vs. Call-by-Reference

- By Value:
  - Allocate temporary storages (normally in stack) for return value, formal parameters & local variables (known as “activation record”)
  - Make a copy of r-value of real parameters to the storage
  - Run/Jump to the function body to change the local/formal/return variables
    - The variables are relative to the stack top upon entering the function
    - Macro: inline expanded to statements prior to run (most likely by a pre-processor)
  - Free temporary storage (except return value)
  - Manipulate return value (e.g., assign to a destination) and free it

- By Reference
  - Make a copy of l-value of real parameters
Program Image (Run Time Organization)

- Code (aka text)
- Initialized data
- Uninitialized data
- Dynamically allocated memory
- Stack

(dynamic)
Managing Variable Declarations
(…more in Chapter 8)

- Form: (extern) (static) (const) Type var1, var2[], …, varN[][];
  - Simple: char, int, float, double
  - Array: index, offset, and base calculation
  - Structure/Record/Class
  - Storage class: static/dynamic – reserve memory at compilation/execution time
  - Const: read only, cannot be modified
  - Extern: not defined in current module & determine address at link time

- Task1: Allocation of memory storage
  - Address (relative offset) evaluation for the various data types

- Task2: Registration of information into symbol table
  - For extracting l-value & r-value
  - For maintaining misc. type information: int, char, Boolean, array of, class
  - For memory address
  - For maintaining Storage class information: static, automatic
  - For enforcing semantic constraints (e.g., const ⇔ read only, not modifiable)
Error Detection & Recovery

- Friendly compilers always try to
  - Detect syntax/semantic errors as early as possible
    - Provide useful indication on locations of error and possible types of error
  - Do NOT stop upon errors: Skip detected errors and start parsing at the next possible beginning of legal statements
    - Known as panic mode error recovery
    - (See more details in LL/LR Parsers chapters)
A Simple One-Pass Compiler: Appendix

Context Free Grammars

- Parse Tree: graphical representation of structure
  - root node (S): a sentencial level structure
  - internal nodes: constituents of the sentence
  - arcs: relationship between parent nodes and their children (constituents)
  - terminal nodes: surface forms of the input symbols (e.g., words)
  - alternative representation: bracketed notation:
    - e.g., [I saw [the [girl [in [the park]]]]]

- Example:
Parse Tree: “I saw the girl in the park”
CFG: Components

- CFG: formal specification of parse trees
  - G = {Σ, N, P, S}
  - Σ: terminal symbols
  - N: non-terminal symbols
  - P: production rules
  - S: start symbol

- Σ: terminal symbols
  - the input symbols of the language
    - programming language: tokens (reserved words, variables, operators, …)
    - natural languages: words or parts of speech
  - pre-terminal: parts of speech (when words are regarded as terminals)

- N: non-terminal symbols
  - groups of terminals and/or other non-terminals

- S: start symbol: the largest constituent of a parse tree

- P: production (re-writing) rules
  - form: α → β (α: non-terminal, β: string of terminals and non-terminals)
  - meaning: α re-writes to (“consists of”, “derived into”) β, or β reduced to α
  - start with “S-productions” (S → β)
CFG: Example Grammar

- Grammar Rules
  
  - \( S \rightarrow \text{NP } \text{VP} \)
  
  - \( \text{NP} \rightarrow \text{Pron } | \text{Proper-Noun } | \text{Det } \text{Norm} \)
  
  - \( \text{Norm} \rightarrow \text{Noun } \text{Norm } | \text{Noun} \)
  
  - \( \text{VP} \rightarrow \text{Verb } | \text{Verb } \text{NP } | \text{Verb } \text{NP PP } | \text{Verb } \text{PP} \)
  
  - \( \text{PP} \rightarrow \text{Prep } \text{NP} \)

  - S: sentence, NP: noun phrase, VP: verb phrase
  - Pron: pronoun
  - Det: determiner, Norm: Norminal
  - PP: prepositional phrase, Prep: preposition

- Lexicon (in CFG form)
  
  - \( \text{Noun} \rightarrow \text{girl } | \text{park } | \text{desk} \)
  
  - \( \text{Verb} \rightarrow \text{like } | \text{want } | \text{is } | \text{saw } | \text{walk} \)
  
  - \( \text{Prep} \rightarrow \text{by } | \text{in } | \text{with } | \text{for} \)
  
  - \( \text{Det} \rightarrow \text{the } | \text{a } | \text{this } | \text{these} \)
  
  - \( \text{Pron} \rightarrow \text{I } | \text{you } | \text{he } | \text{she } | \text{him} \)
  
  - \( \text{Proper-Noun} \rightarrow \text{IBM } | \text{Microsoft } | \text{Berkeley} \)
CFG: Accepted Languages

- **CFG Operations**
  - derivation: applying a production rule to re-write the LHS non-terminal into its constituents
  - rightmost derivation: a sequence of derivations in which the rightmost non-terminal is always re-write first
  - leftmost derivation: leftmost non-terminal first

- **Context-Free Language**
  - Language accepted by a CFG
    - \( L(G) = \{ w \mid S \Rightarrow^* w \} \) (strings of terminals that can be derived from start symbol)
CFG: Expressive Power

- **CFG vs. Regular Expression (R.E.)**
  - every R.E. can be recognized by a FSA
  - every FSA can be represented by a CFG with production rules of the form: $A \rightarrow aB \mid \varepsilon$
  - therefore, $L(RE) < L(CFG)$

- **Writing a CFG for a FSA (RE)**
  - define a non-terminal $N_i$ for a state with state number $i$
  - start symbol $S = N_0$ (assuming that state 0 is the initial state)
  - for each transition $\delta(i,a)=j$ (from state $i$ to state $j$ on input alphabet $a$), add a new production $N_i \rightarrow aN_j$ to $P$
  - for each final state $i$, add a new production $N_i \rightarrow \varepsilon$ to $P$
CFG: Expressive Power (cont.)

- Writing a CFG for a FSA (RE)
  - define a non-terminal $N_i$ for a state with state number $i$
  - start symbol $S = N_0$ (assuming that state 0 is the initial state)
  - for each transition $\delta(i,a)=j$ (from state $i$ to state $j$ on input alphabet $a$), add a new production $N_i \rightarrow a N_j$ to $P$
  - for each final state $i$, add a new production $N_i \rightarrow \epsilon$ to $P$
  - For example: RE: $(a|b)^* a b b$

```
\begin{align*}
S & \rightarrow a S \mid b S \mid a N_1 \\
N_1 & \rightarrow b N_2 \\
N_2 & \rightarrow b N_3 \\
N_3 & \rightarrow \epsilon \\
\end{align*}
```

![Diagram of a FSA with states 0, 1, 2, and 3, transitions a and b, and production rules for S and N1 to N3]
CFG: Expressive Power (cont.)

- **Chomsky Hierarchy:**
  - R.E.: regular set (FSA)
  - CFG: context-free (pushdown automata)
  - CSG: context-sensitive (linear bounded automata)
  - unrestricted: recursively enumerable (Turing Machine)
CFG: Equivalence

- Chomsky Normal Form (CNF) (Chomsky, 1963):
  - $\varepsilon$-free, and
  - Every production rule is in either of the following form:
    - $A \rightarrow A_1 A_2$
    - $A \rightarrow a$ (A1, A2: non-terminal, a: terminal)
    - two non-terminals or one terminal at the RHS
  - generate binary tree
  - good simplification for some algorithms (e.g., grammar training with the inside-outside algorithm (Baker 1979))

- Every CFG can be converted into a weakly equivalent CNF
  - equivalence: $L(G_1) = L(G_2)$
    - strong equivalent: assign the same phrase structure to each sentence (except for renaming non-terminals)
    - weak equivalent: do not assign the same phrase structure to each sentence
  - e.g., $A \rightarrow B C D \equiv \{ A \rightarrow B X, X \rightarrow CD \}$
CFG vs. Finite-State Machine

- Inappropriateness of FAS
  - Constituents
  - Recursion

- RTN (Recursive Transition Network)
  - FSA with augmentation of recursion
  - arc: terminal or non-terminal
  - if arc is non-terminal: call to a sub-transition network & return upon traversal