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
  - satisfy certain constraints that cannot be explicitly specified by syntax

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

```c
{  
  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
}
```

```
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
```
Example: Intermediate Code for Program fragment

1: \( i = i + 1 \)  
2: \( t_1 = a[i] \)  
3: if \( t_1 < v \) goto 1  
4: \( j = j - 1 \)  
5: \( t_2 = a[j] \)  
6: if \( t_2 > v \) goto 4  
7: ifFalse \( i \geq j \) goto 9  
8: goto 14  
9: \( x = a[i] \)  
10: \( t_3 = a[j] \)  
11: \( a[i] = t_3 \)  
12: \( a[j] = x \)  
13: goto 1

\[
\text{do } i = i + 1; \text{ while } (a[i] < v); \]

(abstract) syntax tree

\[
\begin{align*}
\text{do-while} & \rightarrow \\
\text{body} & \rightarrow \\
\text{assign} & \rightarrow \\
[ ] & \rightarrow \text{v} \\
i + a_i & \rightarrow \text{i 1} \\
a_i & \rightarrow \text{v} \\
i & \rightarrow \text{i}
\end{align*}
\]
A Simple Compiler: Goals

- An Example for infix-to-postfix translation
  - Infix: operators within/between operands: (a+b*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
  - no parentheses are required

- Prefix: +952, -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
A Simple One-Pass Compiler

Specification
Specifications of the One-Pass Compiler: Syntax Definition

- Specification of syntax of a language: CFG
- CFG: describes hierarchical structure with production rules
  - e.g., if ( expression ) statement else statement
  - CFG: stmt → 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 largest constituent of a 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”
  - 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’

- nonterminals: {list, digit}

- Start symbol: list

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

Grammar Rules

- S → NP VP
- NP → Pron | Proper-Noun | Det Norm
- Norm → Noun Norm | Noun
- VP → Verb | Verb NP | Verb NP PP | Verb PP
- PP → Prep NP

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

Lexicon (if in CFG form)

- Noun → girl | park | desk
- Verb → like | want | is | saw | walk
- Prep → by | in | with | for
- Det → the | a | this | these
- Pron → I | you | he | she | him
- Proper-Noun → IBM | Microsoft | Berkeley
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

- 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
  - max(x, y)

- $\leftrightarrow$ “list”
  - list $\rightarrow$ list + digit
  - list $\rightarrow$ list - digit
  - list $\rightarrow$ digit
  - digit $\rightarrow$ 0 $|$ 1 $|$ 2 $|$ …$|$ 9
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 | ...

- nonterminals: {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”
  - ‘;’ $\Leftrightarrow$ ‘+’, ‘-’
  - stmt $\Leftrightarrow$ ‘0’, ‘1’, ..., ‘9’

$\Leftrightarrow$ “list”
  - list $\rightarrow$ list + digit
  - list $\rightarrow$ list - digit
  - list $\rightarrow$ digit
  - digit $\rightarrow$ 0 | 1 | 2 | ... | 9
Syntax Definition: Language

- 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
  - Derive == generate == re-write
  - Derivation == generation == re-writing

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

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

- **Parsing**: an inverse process of derivation
  - get a derivation sequence that produces an input token string
  - finding a parse tree for a string of tokens
  - $9-5+2 =?accept_as=>\text{list}$

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

- 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]]]]]

- For example:

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

- A grammar can have more than one parse tree generating a given string of tokens
  - Everyone can have his/her own grammar for a language
  - Ambiguous “grammar” vs inherently ambiguous “language”

- Grammar writing for programming languages
  - Write an unambiguous one, or
  - Use an ambiguous one + disambiguation rules
Ambiguous Grammars

\[ E \rightarrow E \text{ "+" } E \mid E \text{ "-" } E \mid D \]
\[ D \rightarrow \text{ "1" } \mid \text{ "2" } \mid \ldots \mid \text{ "9"} \]

\[ (9 - 5) + 2 \]

\[ 9 - (5 + 2) \]
Sources of Ambiguity

- **Associativity of Operators**
  - **Left Association (LA):** 9-5+2 ⇔ (9-5)+2
    - LA: cout << “Hello” << “World” << endl;
    - ⇔ cout << “Hello” ; cout << “World”; cout << endl;
    - LA: cin >> year >> month >> day ;
    - ⇔ cin >> year; cin >> month; cin >> day;
  - **Right Association (RA):** a=b=c ⇔ a=(b=c)
    - right → letter = right | letter
    - letter → a | b | ... | z
    - RA: a += b += c ⇔ a += (b += c)

- **Precedence of Operators**
  - 9+5*2 ⇔ 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 syntax 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 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)
  - $R \rightarrow L = R \mid L$
  - $L \rightarrow a \mid b \mid \ldots \mid z$

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

Resolution for Precedence of Operators
- Define high precedence expressions (including atomic units) first
- Low precedence operators operate on high precedence expressions

Example: Mathematic Expression

- factor: basic units
  - 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
  - expr → expr + term | expr - term | term
Syntax of Expression without Ambiguity

Example: Mathematic Expression

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

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

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

- Question: Given analysis results (a parse tree) how to translate them into intermediate representation?

- How to specify “translation rules”?
  - “Input => output” mapping

- Example: infix-to-postfix “translation rules” (also, its definition)
  - E.p = E if E 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

- What has been done?
  - A “syntax directed” approach
  - Associate each local structure with a set of rules or translation actions
  - Keep some variables (attributes) for each LHS symbol
Syntax-Directed Translation

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

- 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: procedural notation for specifying translations
Syntax-Directed Definition

- Syntax directed definition (SDD):
  - Input: annotated CFG, where --
  - CFG: specify syntax
  - 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)
  - Output: annotated parse tree (with attribute annotation)

- 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:**

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’
...

Jing-Shin Chang
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

Jing-Shin Chang
SDD for Robot’s Position

Input: begin west south ...

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

Instr.dx=0
Instr.dy=-1

Attributes for main result (x,y)

Aux. Attributes (dx, dy)

Jing-Shin Chang
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
...

Jing-Shin Chang
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)
Synthesized Attributes

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>L → E ‘\n’</td>
<td>L.val := E.val</td>
</tr>
<tr>
<td>E → E₁ ‘+’ T</td>
<td>E.val := E₁.val + T.val</td>
</tr>
<tr>
<td>E → T</td>
<td>E.val := T.val</td>
</tr>
<tr>
<td>T → T₁ ‘*’ F</td>
<td>T.val := T₁.val * F.val</td>
</tr>
<tr>
<td>T → F</td>
<td>T.val := F.val</td>
</tr>
<tr>
<td>F → ‘(’ E ‘)’</td>
<td>F.val := E.val</td>
</tr>
<tr>
<td>F → digit</td>
<td>F.val := digit.val</td>
</tr>
</tbody>
</table>

Fig. 5.2, 1st Ed; Fig. 5.1, 2nd Ed
Synthesized Attributes

\[3 \times 5 + 4\]

\[
\begin{array}{c}
\text{L.val} = 19 \\
\text{E.val} = 19 \\
\text{E.val} = 15 \\
\text{T.val} = 4 \\
\text{T.val} = 15 \\
\text{T.val} = 3 \\
\text{F.val} = 5 \\
\text{digit.val} = 4 \\
\text{digit.val} = 5 \\
\text{digit.val} = 3
\end{array}
\]

‘\n’

‘+’

F.val = 4

digit.val = 4
Inherited Attributes

<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, 1\textsuperscript{st} Ed; Fig. 5.8, 2\textsuperscript{nd} Ed
Inherited Attributes

\[
\begin{align*}
T.\text{type} &= \text{float} \\
\text{id}_1, \text{id}_2, \text{id}_3
\end{align*}
\]
Attribute Evaluation

- Any order that correctly evaluates the attributes will do
  - Following the dependency order of attributes
  - Any Topological Sorting Sequences

- 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
    - $(\text{Cout} \ll \text{obj1} \ll \text{obj2} \ll \ldots)$
    - and define the attributes required to convert the sequence with multiple $\ll$ operators into an equivalent sequence of single stream operation.
    - I.e., $\Leftrightarrow \text{Cout} \ll \text{obj1}; \text{cout} \ll \text{obj2}; \text{cout} \ll \ldots$
    - And Write a pseudo code, based on the associated semantic rules
Translation Schemes

- A procedural definition for translation
  - attach program segments to production rules
  - to explicitly indicate how the translation could be done
- 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

```
rest

+  term  {print(‘+’)  }  rest
```
TS for Infix-to-Postfix Translation

- Production with {Semantic Actions}:

  expr → expr₁ + term {print(‘+’)}
  expr → expr₁ – term {print(‘-’)}
  expr → term
  term → 0 {print(‘0’)}
  term → 1 {print(‘1’)}

- Tools: Yacc
SDD for Infix-to-Postfix Translation

E.t = “95-2+”

E.t = “95-”

E.t = “9”

T.t = “5”

T.t = “9”

T.t = “2”

9

5

2
TS for Infix-to-Postfix Translation

```
print "9"

print "5"

print "2"

print "+"

print "-"
```
Summary: Attribute Evaluation

- Any order that correctly evaluates the attributes will do
  - Following the dependency order of attributes

- SDD
  - Does not specify order of evaluation
  - Normally evaluated in depth-first

- “Simple” SDD
  - Translation string of LHS = concatenation(translated strings of RHS non-terminals in the same order + additional interleaved strings)

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

- SDD vs. TS:
  - Simple SDD can be implemented with TS
  - Semantic actions print additional strings in the same order as in semantic rules
Parsing

- Parsing
  - Analyzing structures according to grammar
  - Determine if a string of tokens can be generated by a grammar
  - Any CFG can be parsed in $O(n^{**3})$
  - Most PL can be parsed in linear time, looking at one lookahead
Parsing Methods

- **Top-down**
  - Construct root nodes down to leaves
  - Easily constructed by hands
  - Ex. Recursive descent, predictive recursive descent (LL Parser)

- **Bottom-up**
  - Construct parent nodes from children
  - Can handle a larger class of grammars and translation schemes
    > E.g., left-recursive productions cannot be parsed top-down left-to-right
  - Automatic tools tends to produce bottom-up parsers
  - Ex. LR parsers
Top-Down Parsing (1st Ed.)

- Root/Start Symbol $\rightarrow$ Expand nonterminal by productions in the Grammar
- Match lookahead (current token being scanned)
  - expand production with matching lookahead

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

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

- Root/Start Symbol $\rightarrow$ Expand nonterminal by productions in the Grammar
- Match lookahead (current token being scanned)
  - expand production with matching lookahead

- Example: G in Fig 2.16 [2nd Ed]
  - stmt $\rightarrow$ expr ;
  - | if ( expr ) stmt
  - | for ( optexpr ; optexpr ; optexpr ) stmt
  - | other
  - optexpr $\rightarrow$ 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 (match, and conduct associated actions against) the input tokens, starting with the start symbol.
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 to the next word immediately after the last matched word
  - These calls may be recursive, thus it is also called *recursive decent parsing*
- To match a terminal symbol $t$, we call a procedure $\text{match}(t)$
  - $\text{match}()$ calls the scanner to get the next token.
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]
Problems with Top-Down Parsing: Prediction

- Which Production to Expand?
  - (1) Backtracking
  - (2) Predictive Parsing

- Backtracking
  - Select arbitrary production
  - Push return configuration (current starting index & next useable production)
  - back to previous configuration if fails

- 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
    - \( \text{FIRST}(\alpha) = \{a \mid \alpha =*=> a \beta \} \)
  - If multiple choice 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 are disjoint.
Problems with Top-Down Parsing: \( \varepsilon \) - Production

- How to match “Nothing”?  
  - \( \varepsilon \) - Production

- Solution:  
  - Use \( \varepsilon \) - Production as the default production when other productions produce no match, that is  
  - Assumed that \( X \rightarrow \varepsilon \) is applied when all other \( X \)-productions are tried and failed to match input
Problems with Top-Down Parsing: Left Recursion

- Left Recursion: unable to deliver a terminal symbol for matching
  - list → 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
Problems with Top-Down Parsing: Translation using Right Recursion Rules

- Right Recursion: difficult to evaluate values for Left Associative operators
- Solution: Regard semantic actions which correspond to synthesized attributes as a terminal node when re-writing into left-recursion 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 → expr₁ + term  {print(‘+’)}
expr → expr₁ − term  {print(‘-’)}
expr → term

term → 0           {print(‘0’)}
term → 1           {print(‘1’)}
...
term → 9           {print(‘9’)}
Adapting the Translation Scheme

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

- Technique:
  - $A \rightarrow A \alpha | A \beta | \gamma$, rewrite into
  - $A \rightarrow \gamma R$
  - $R \rightarrow \alpha R | \beta R | \varepsilon$

- Extending the transformation to include semantic actions (for synthesized attributes): carry the semantic action along in the transformation
  - $A= expr \&\& \alpha = + term \{\text{print}(\text{'+'})\}$
  - $\&\& \beta = - term \{\text{print}(\text{'-'})\}$
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{rest} \\
\text{rest} & \rightarrow \ - \ \text{term} \ \{\text{print(`-`)\}} \ \text{rest} \\
\text{rest} & \rightarrow \ \varepsilon \\
\text{term} & \rightarrow \ 0 \ \{\text{print(`0`)\}} \\
\text{term} & \rightarrow \ 1 \ \{\text{print(`1`)\}} \\
\ldots \\
\text{term} & \rightarrow \ 9 \ \{\text{print(`9`)\}}
\end{align*}
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 */
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

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

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

- Tail Recursion: call oneself at the end of a statement.
  - Optimization: replace procedure call to a GOTO statement
- General: 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> */
        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(1) { /* rest() call ⇔ loop */
        /* to match a list of "+/- term()" */
        if (lookahead == '+') {
            match('+'); term(); putchar('+');
        } else if (lookahead == '-') {
            match('-'); term(); putchar('-');
        } else {
            break;
        }
    }
}
Source Code Optimization: Tail Recursion

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:

c=Getchar()

Characters

UnGetchar (c, stdin)

"<=" vs. "<"

Tokens & Attributes

Parser

Call Lex()
Lexical Analysis

- Convert input lexemes to stream of tokens
  - Lexeme: a sequence of characters that comprises a single token

- Typical Functions:
  - Removal of white space and comments
    - instead of writing productions that include spaces and comments
  - Reading Ahead
    - ‘>’ ⇔ ‘>=‘ or ‘>?‘ (? =/= ‘=‘)
    - Use a buffer from which the lexical analyzer can read and push back characters
    - One-character read-ahead usually suffices
  - Constants: Digits into <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
    - Keywords: begin, end, if, else ⇔ begin, end, if, else

Jing-Shin Chang
Lexical Analysis

- 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

- 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)
Lexical Analysis

- Token scan() {
  - Skip white space;
  - Handle numbers; // return NUM, if identified
  - Handle reserved words and identifiers; // possibly return ID
  - // else ... a single character operator at 'peek' (next input character)
  - Token t = new Token(peek);
  - Peek = blank
    - // convention: init. as blank on return if not pointing to next input char
  - Return t;
- }
Examples of “Tokens”

■ Token:
  - a Numerical/Symbolic Representation of Input String
  - `typedef enum token_types {
    BEGIN, END, IF, THEN, ELSE, ID,
    NUM, LPAREN, RPAREN,
    SEMICOLON, COMMA, ASSIGNOP, PLUSOP,
    MINUSOP, ENDSAN
  } token;` // or ...
  ```
  #define NUM 256
  ```

■ Lexical Analyzers (or Scanners) return a “token”, instead of a string, upon acceptance of an input unit:
  - `extern token scanner(void);`
“Token” Class and subclasses in Java

- Package lexer;
- Public class Token {
  - Public final int tag;
  - Public Token (int t) { tag = t; }
}

  - Creating token object:
    `new Token('+')`

- 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)
}

Jing-Shin Chang
“Token” Class and subclasses in Java

- Package lexer;
- Public class Num extends Token {
  - // Public final int tag; // in super class Token
  - Public final int value;
  - Public Num (int v) { super(Tag.NUM); value = v; }
  }

- Public class Word extends Token {
  - // Public final int tag; // in super class Token
  - Public final String lexeme;
  - Public Word (int t, String s) { super(t); lexeme = new String(s); }
  }

Jing-Shin Chang
Recognizing Identifiers and Integer Literals

```c
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);
}
```

Identifiers: [a-zA-Z][a-zA-Z0-9]*
Integer Constants: [0-9]+
Recognizing Operators, Comments, and Delimiters

While ((in_char = getchar()) != EOF) {
    if (isspace(in_char)  continue;
    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 phase & used by synthesis phases
- Need to support multiple declarations of the same identifier

Lexemes + attributes
- 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)
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: pseudo-code for a lexical analyzer
  ◆ Lookup(lexbuf) for existence & insert(lexbuf, ID) if is new 

■ P. 74: lexan() for infix-to-postfix translation

Jing-Shin Chang 84
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
- Listing: 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:
    - 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:
    - `{ int x; char y; { bool y; x; y; } x; y; }
  - Output: // strip declarations, and use of x, y with their correct type
    - `{ { 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:
    ```plaintext
    { int x; char y; { bool y; x; y; } x; y; }
    ```
  - Output: // strip declarations, and use of x, y with their correct type
    ```plaintext
    { { x:int; y:bool; } x: int; y:char; }
    ```
A Simple One-Pass Compiler

Code Generation for a virtual stack machine
A Simple One-Pass Compiler

Translation
- IR:
  - tree: parse tree or (abstract) syntax tree
  - linear representation:
    - 3-address codes
    - stack machine codes
  - semantic rules & semantic actions
  - semantic attribute evaluation
Code Generation for an Abstract Machine

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

- Two Kinds of Intermediate Representations:
  - Tree (Hierarchical Representations):
    - Parse Tree (Concrete Syntax Tree)
    - Abstract Syntax Tree
  - Linear Representations:
    - Code for an abstract stack machine
      - Push, pop, 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:
  - Construct syntax tree by constructing nodes for each construct
  - Generate codes by traversing the syntax tree, starting from the root node
Constructing Syntax Tree
Constructing Syntax Tree

- Generate a node for each operators and connect its operands as children
  - $a + b \Leftrightarrow \text{`+': parent; `a', `b': children}$
  - $E_1 \text{ op } E_2 \Leftrightarrow \text{op: parent; } E_1, E_2: \text{children}$
- Any construct:
  - Each construct is represented by a node, with children for the semantically meaningful components of the construct ($\Leftrightarrow$ generalized operators, like the `IF’ operator)
  - E.g., While $(expr) \text{ stmt} \Leftrightarrow \text{parent: while; children: expr and stmt}$
Constructing Syntax Tree

- Operators: (Fig. 2.41, 2\textsuperscript{nd} Ed, and more...)
  - Assignment: ‘=’
  - Mathematical: ‘+’, ‘-’, ‘*', ‘-’, ‘%’
  - Bit operations: ‘&’, ‘|’, ‘!’
  - Boolean/Conditional: and, or, not ( ‘&&’, ‘||’, ‘!’)
  - Unary minus: ‘-’
  - Array access: ‘[]’
  - Flow control operators: \texttt{if} (without “\texttt{else}”), \texttt{ifelse} (with “\texttt{else}”), \texttt{while}, \texttt{do}, \texttt{for}, ...
    - Operators: keywords as operators
    - Operands: conditional expression and body of executable statements
Constructing Syntax Trees

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

■ Syntax Trees for Statements
  ◆ if (without “else”), ifelse (with “else”), while, do, for, ...
    F A Constructor for each operator (e.g., if ⇔ If(expr,stmt))
    F Constructor parameters for operands in abstract syntax tree
  ◆ Translation Scheme:
    F \( stmt \rightarrow \text{if ( expr ) stmt}_1 \) \hspace{1cm} \{ stmt.n = \text{new If(expr.n, stmt}_1.n); \}
    • Create a new node labeled IF with the nodes expr.n and \( \text{stmt}_1.n \) as children
    F \( stmt \rightarrow \text{expr} \) \hspace{1cm} \{ stmt.n = \text{new Eval(expr.n);} \}
Constructing Syntax Trees

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

- Representing Blocks in Syntax Tree
  - Block: sequence of statements enclosed by ‘{’ and ‘}’
    - \( stmt \rightarrow block \) \( \{ stmt.n = block.n; \} \)
    - \( block \rightarrow \{ '{' stmts '}' \} \) \( \{ block.n = stmts.n; \} \)
  - Stmts: represented as a leaf \textbf{null} (empty statement) and a \textbf{seq} (sequence of statements)
    - \( stmts \rightarrow stmts, stmt \) \( \{ stmts.n = new \text{Seq}(stmts_.n, stmt.n); \} \)
    - \( | \varepsilon \) \( \{ stmts.n = \text{null}; \} \)
  - Declarations:
    - Normally, no nodes are created and attached to syntax trees
    - Information is incorporated into symbol table (Fig. 2.38, 2nd Ed.)
Constructing Syntax Trees

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

- Syntax Trees for Expressions
  - Operators: assign, cond, rel, op, not, minus, access ([i] a[i]) operators
  - Translation scheme (e.g., ‘*’)
    - $term \rightarrow term_1 \ast factor$
      \begin{verbatim}
      {term.n = new Op(‘*’, term_1.n, factor.n); }
      \end{verbatim}
Constructing Syntax Trees

- Static checking (at compilation time, catching error earlier)
  - Checking semantics while constructing syntax tree
  - Syntactic checking
    - $i \leftarrow i + 1$
    - L-value: must be memory locations
    - R-value: must be integer/character/string values

- Type checking
  - Ensure compatible types between operators
  - Check is checked when a node is constructed
    - $E \rightarrow E_1 \text{ rel } E_2$\{if $(E_1\text{.type} == E_2\text{.type}) E\text{.type} = \text{boolean} \text{ else error; }\}$
  - Type conversion ("Coercion")
  - Overloading: operator with different meanings depending on its context
    (determined by checking types of operands and result)
Generating Three-Address Codes
Generating Three-Address Codes

- Three-address instructions: (typical instruction set)
  - Binary ops:
    - $x = y \text{ op } z$
      - $X, y, z$: names, constants, compiler-generated temporaries
      - Op: (binary) operator
  - Array accesses instructions:
    - $x [y] = z$
    - $x = y[z]$
  - Flow control instructions:
    - $\text{IfFalse } x \text{ goto } L$
    - $\text{ifTrue } x \text{ goto } L$
    - goto $L$
  - Copy value:
    - $x = y$

- Code generation after tree construction:
  - Statements
  - Expressions
Generating Three-Address Codes

Translation of “Statements”

◆ Using jumps to implement flow control through the statements

◆ **if ( expr ) then stmt₁ ⇔**
  - Code to compute expr into x
  - ifFalse x goto after
  - Code for stmt₁
  - after: …

◆ ⇔ Fig. 2.43 (2nd Ed.)
  - Constructor: connect current node (parent) and children nodes, create unique new labels
  - Gen() function: generate codes for current statement

```plaintext
x := Code(expr)  
ifFalse x goto after  
Code(stmt1)  
after: ...
```
Generating Three-Address Codes

- Translation of “Statements”
  - Calss 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, create unique new labels
  - 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

\[ x := \text{Code(expr)} \]
\[ \text{ifFalse x goto after} \]
\[ \text{Code(stmt1)} \]
\[ \text{after:} \]
Generating Three-Address Codes

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

\[ x = y \text{ op } z \]
\[ \text{ifFalse } x \text{ goto after} \]

\[ x := \text{Code}(E) \]

\[ \text{S} \]

\[ x, t: \text{temp variables generated by code generation actions} \]
\[ \text{(in } \text{.gen()}, \text{.rvalue()}, \text{.lvalue()} \ldots \text{ methods)} \]

Jing-Shin Chang
Generating Three-Address Codes

- Translation of “Expressions”
  - Generate one 3-address instruction for each operator node in the syntax tree for the expression
    - Binary ops
    - Array access
    - Assignment
  - Constants & identifiers: no code is generated
Generating Three-Address Codes

■ Expression: Node x of type Expr with op
  ∨ Code to evaluate Expr; temp := value_evaluated
  ∨ Example: \( i - j + k \)
    \( \Leftrightarrow \ t1 = i - j \)
    \( \Leftrightarrow \ t2 = t1 + k \)

■ Array access/assignment
  ∨ Need to distinguish between l-values & r-values
  ∨ Example (r-value): \( 2 \times a[i] \)
    \( \Leftrightarrow \ t1 = a[i] \)
    \( \Leftrightarrow \ t2 = 2 \times t1 \)
  ∨ Example (l-value): \( a[i + 4] = 2 \)
    \( \Leftrightarrow \ t1 = i + 4 \)
    \( \Leftrightarrow \ a[t1] = 2 \)
Translation of Expression

Codes(i-j+k):
“t1 = i-j”
“t2 = t1+k”

New t2;
t1: rvalue(‘i-j’)
k: rvalue(‘k’)
code: “t2 = t1+k”

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

Jing-Shin Chang
Translation of Array Access

Codes\(a[i]*2\): "t1 = a[i]"
"t2 = t1*2"

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

New t1;
a: lvalue\(\text{‘}a\text{’}\)
i: rvalue\(\text{‘}i\text{’}\)
code: “t1 = a[i]”

Jing-Shin Chang
Generating Three-Address Codes

■ Translation of “Expressions”

■ lvalue: address
  - \(a = \ldots\) \(\Leftrightarrow\) \(a\) \(= \ldots\)
  - \(y[z] = \ldots\) \(\Leftrightarrow\) \(\text{code.rvalue}(z)\) \(; y[z’] = \ldots\)

■ rvalue: value
  - \((a = \ldots)\) \(y\) \(\Leftrightarrow\) \((a = \ldots)\) \(y\)
  - \((a = \ldots)\) \(y\) \(\text{op}\) \(z\) \(\ldots\) \(\Leftrightarrow\) \(t = \text{rvalue}(y)\) \(\text{op}\) \(\text{rvalue}(z)\)
  - \((a = \ldots)\) \(y[z]\) \(\ldots\) \(\Leftrightarrow\) \(z’ = \text{rvalue}(z)\) \(t = y[z’]\)
  - \((a = \ldots)\) \(y = z\) \(\ldots\) \(\Leftrightarrow\) \(z’ = \text{rvalue}(z)\) \(; \text{lvalue}(y) = z’\)

■ Example: \(a[i] = 2 * a[j - k]\)
  - \(t3 = j - k\)
  - \(t2 = a[t3]\)
  - \(t1 = 2 * t2\)
  - \(a[i] = t1\)

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

\(\text{lvalue}(x)\): generates codes to compute the subtrees below \(x\), and returns a new node representing the address for \(x\)
Generating Three-Address Codes

- Example: \( a[i] = 2 * a[j - k] \)
  - \( t3 = j - k \)
  - \( t2 = a[t3] \)
  - \( t1 = 2 * t2 \)
  - \( a[i] = t1 \)

\[ \text{rvalue}(x), \text{lvalue}(x): \text{generate codes & temporary variables (nodes)} \]

Assign.rvalue()
Better Code for Expressions

- Improve function rvalue() to generate fewer 3-address instructions:
- Reduce number of copy instructions in a subsequent optimization phrase.
  - E.g., t = i + 1; i = t => i = i + 1
- Taking context into account: generate tmps only when necessary:
  - x = a + b + c + d ⇔ x = (((a + b) + c) + d) ⇔ the last ‘+’ operation 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
Better Code for Expressions

- \( x = a + b + c + d \)
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 stack top:
      - 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:
  - Statement => postfix-like expression => stack machine code (.EXE)

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

- 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) div 4 + (153*m+2) 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
  - 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}

  ![Algorithm Diagram]

  - Fig. 2.33 (Left)
  - TS: ... (p. 67 & Algorithm Fig. 2.34)
  - Stmt → if
  - expr { out:= newlabel; emit('gofalse', out); }
  - then
  - stmt₁ { emit('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

- **Efficient Translation of logical OR statements**
  - Expr1 or Expr2 == if (expr1) then true else expr2
    - == return(expr1 ? True : expr2 )
  - Code(Expr1) || 'copy' || gotrue lab || pop || Code(expr2) || 'label' lab

- **Translation for AND statements:** (exercise)
  - Code(expr1)
  - Copy
  - GoTrue Lab
  - Pop
  - Code(expr2)
  - Label Lab
  - Return a T/F at top
  - If fail to pass, remove the original F, relying on Code(expr2) to return T/F

- **Duplicate a T/F for GoTrue Test, and leave original T as returned value if pass the test**
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

- 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 statements 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

Jing-Shin Chang
Managing Variable Declarations
(…more in Chapter 8)

- (extern) (static) (const) Type var1, var2, …, varN;
  - Simple: char, int, float, double
  - Array: index, offset, and base calculation
  - Structure/Record
  - class

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

- 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)
A Simple One-Pass Compiler: Appendix

Context Free Grammars

- Parse Tree: graphical representation of structure
  - root node (S): a sentential 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]]]]]

- For example:
Parse Tree: “I saw the girl in the park”
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 largest constituent of a 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” ($S \rightarrow \beta$)
CFG: Example 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

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

Lexicon (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
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: $A \rightarrow aB | \varepsilon$
  - 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 \( N_i \rightarrow \varepsilon \) to \( P \)
  - For example: RE: \((a|b)^* a \ b \ b\)

\[
\begin{align*}
S & \rightarrow a \ S \mid b \ S \mid a \ N_1 \\
N_1 & \rightarrow b \ N_2 \\
N_2 & \rightarrow b \ N_3 \\
N_3 & \rightarrow \varepsilon
\end{align*}
\]
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 (Tuning 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