CSE443 Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Figure 1.6, page 5 of text
Review
4.6 Sizes

Some of the aspects of C++’s fundamental types, such as the size of an `int`, are implementation-defined (§C.2). I point out these dependencies and often recommend avoiding them or taking steps to minimize their impact. Why should you bother? People who program on a variety of systems or use a variety of compilers care a lot because if they don’t, they are forced to waste time finding and fixing obscure bugs. People who claim they don’t care about portability usually do so because they use only a single system and feel they can afford the attitude that “the language is what my compiler implements.” This is a narrow and shortsighted view. If your program is a success, it is likely to be ported, so someone will have to find and fix problems related to implementation-dependent features. In addition, programs often need to be compiled with other compilers for the same system, and even a future release of your favorite compiler may do some things differently from the current one. It is far easier to know and limit the impact of implementation dependencies...
The reason for providing more than one integer type, more than one unsigned type, and more than one floating-point type is to allow the programmer to take advantage of hardware characteristics. On many machines, there are significant differences in memory requirements, memory access times, and computation speed between the different varieties of fundamental types. If you know a machine, it is usually easy to choose, for example, the appropriate integer type for a particular variable. Writing truly portable low-level code is harder.

Sizes of C++ objects are expressed in terms of multiples of the size of a `char`, so by definition the size of a `char` is 1. The size of an object or type can be obtained using the `sizeof` operator (§6.2). This is what is guaranteed about sizes of fundamental types:

\[
1 \equiv \text{sizeof}(\text{char}) \leq \text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long})
\]

\[
1 \leq \text{sizeof}(\text{bool}) \leq \text{sizeof}(\text{long})
\]

\[
\text{sizeof}(\text{char}) \leq \text{sizeof}(\text{wchar_t}) \leq \text{sizeof}(\text{long})
\]

\[
\text{sizeof}(\text{float}) \leq \text{sizeof}(\text{double}) \leq \text{sizeof}(\text{long double})
\]

\[
\text{sizeof}(N) \equiv \text{sizeof}(\text{signed } N) \equiv \text{sizeof}(\text{unsigned } N)
\]

where \( N \) can be `char`, `short int`, `int`, or `long int`. In addition, it is guaranteed that a `char` has at least 8 bits, a `short` at least 16 bits, and a `long` at least 32 bits. A `char` can hold a character of the machine’s character set.
The precision of these objects depends on the machine at hand; the table below shows some representative values.

<table>
<thead>
<tr>
<th></th>
<th>DEC PDP-11</th>
<th>Honeywell 6000</th>
<th>IBM 370</th>
<th>Interdata 8/32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>char</strong></td>
<td>ASCII 8 bits</td>
<td>ASCII 9 bits</td>
<td>EBCDIC 8 bits</td>
<td>ASCII 8 bits</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>16</td>
<td>36</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td><strong>short</strong></td>
<td>16</td>
<td>36</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td><strong>long</strong></td>
<td>32</td>
<td>36</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td><strong>float</strong></td>
<td>32</td>
<td>36</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>64</td>
<td>72</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

The intent is that **short** and **long** should provide different lengths of integers where practical; **int** will normally reflect the most "natural" size for a particular machine. As you can see, each compiler is free to interpret **short** and **long** as appropriate for its own hardware. About all you should count on is that **short** is no longer than **long**.
TOOLS

Lexical analysis:
LEX/FLEX (regex → lexer)

Syntactic analysis:
YACC/BISON (grammar → parser)
Top-down & bottom-up

- A top-down parser builds a parse tree from root to the leaves
  - easier to construct by hand

- A bottom-up parser builds a parse tree from leaves to root
  - Handles a larger class of grammars
  - tools (yacc/bison) build bottom-up parsers
Our presentation
First top-down, then bottom-up

- Present top-down parsing first.
- Introduce necessary vocabulary and data structures.
- Move on to bottom-up parsing second.
Vocab: look-ahead

- The current symbol being scanned in the input is called the lookahead symbol.
Top-down parsing
Top-down parsing

- Start from grammar's start symbol
- Build parse tree so its yield matches input
- Predictive parsing: a simple form of recursive descent parsing
Basic idea:
try to build a derivation
\[ S \Rightarrow^* \text{input} \]

\[ S \Rightarrow^* \alpha \]
\[ ...?... \]
\[ \Rightarrow^* \text{input} \]
FIRST(\(\alpha\))

- If \(\alpha \in (NUT)^*\) then FIRST(\(\alpha\)) is "the set of terminals that appear as the first symbols of one or more strings of terminals generated from \(\alpha\)." [p. 64]

- Ex: If \(A \rightarrow a \beta\) then FIRST(A) = \{a\}

- Ex. If \(A \rightarrow a \beta | B\) then FIRST(A) = \{a\} \cup FIRST(B)
FIRST(\(\alpha\))

- First sets are considered when there are two (or more) productions to expand \(A \in N: A \rightarrow \alpha \mid \beta\)

- Predictive parsing requires that \(\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset\)
$\varepsilon$ productions

- If lookahead symbol does not match first set, use $\varepsilon$ production not to advance lookahead symbol but instead "discard" non-terminal:

  - $\text{optexpt} \rightarrow \text{expr} \mid \varepsilon$

- "While parsing optexpr, if the lookahead symbol is not in FIRST(expr), then the $\varepsilon$ production is used" [p. 66]
Left recursion

Grammars with left recursion are problematic for top-down parsers, as they lead to infinite regress.
Left recursion example

Grammar:

\[ \text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term} \]

\[ \text{term} \rightarrow \text{id} \]

\textit{FIRST} sets for rule alternatives are not disjoint:

- \textit{FIRST}(\text{expr}) = \text{id}
- \textit{FIRST}(\text{term}) = \text{id}
Left recursion example

Grammar:

\[
expr \rightarrow expr + term \mid term
\]

\[
term \rightarrow id
\]

FIRST sets for rule alternatives are not disjoint:

\[
FIRST(expr) = id
\]

\[
FIRST(term) = id
\]
Rewriting grammar to remove left recursion

- Expr rule is of form \( A \rightarrow A \alpha | \beta \)

- Rewrite as two rules
  - \( A \rightarrow \beta R \)
  - \( R \rightarrow \alpha R | \epsilon \)
Grammar is re-written as

- $\text{expr} \rightarrow \text{term} \ R$
- $R \rightarrow + \text{term} \ R \mid \varepsilon$
A grammar $G$ is ambiguous if $\exists \sigma \in L(G)$ that has two or more distinct parse trees.

Example - dangling 'else':

if <expr> then if <expr> then <stmt> else <stmt>

if <expr> then {
    if <expr> then <stmt>
} else <stmt>

if <expr> then {
    if <expr> then <stmt> else <stmt>
}
dangling else resolution

- usually resolved so else matches closest if-then
- we can re-write grammar to force this interpretation (ms = matched statement, os = open statement)
  
  <stmt> -> <ms> | <os>
  
  <ms> -> if <expr> then <ms> else <ms> | ...
  
  <os> -> if <expr> then <stmt> | if <expr> then <ms> else <os>
Left factoring

- If two (or more) rules share a prefix then their FIRST sets do not distinguish between rule alternatives.

- If there is a choice point later in the rule, rewrite rule by factoring common prefix

- Example: rewrite

\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \]

as

\[ A \rightarrow \alpha A' \]

\[ A' \rightarrow \beta_1 \mid \beta_2 \]
Predictive parsing: a special case of recursive-descent parsing that does not require backtracking

Each non-terminal $A \in N$ has an associated procedure:

```c
void A() {
    choose an A-production $A \rightarrow X_1 X_2 \ldots X_k$
    for (i = 1 to k) {
        if ($x_i \in N$) {
            call $x_i()$
        } else if ($x_i = \text{current input symbol}$) {
            advance input to next symbol
        } else error
    }
```


Predictive parsing: a special case of recursive-descent parsing that does not require backtracking.

Each non-terminal $A \in N$ has an associated procedure:

```c
void A() {
    choose an A-production $A \rightarrow X_1 X_2 \ldots X_k$
    for (i = 1 to k) {
        if ($x_i \in N$) {
            call $x_i()$
        }
        else if ($x_i =$ current input symbol) {
            advance input to next symbol
        }
        else error
    }
}
```

There is non-determinism in choice of production. If "wrong" choice is made the parser will need to revisit its choice by backtracking.

A predictive parser can always make the correct choice here.
\textbf{FIRST}(X)

(text p. 221)

- if $X \in T$ then $\text{FIRST}(X) = \{ X \}$

- if $X \in N$ and $X \rightarrow Y_1 \ Y_2 \ldots \ Y_k \in P$ for $k \geq 1$, then
  
  add $a \in T$ to $\text{FIRST}(X)$ if $\exists i$ s.t. $a \in \text{FIRST}(Y_i)$ and $\varepsilon \in \text{FIRST}(Y_j) \ \forall \ j < i$ (i.e. $Y_1 \ Y_2 \ldots \ Y_i-1 \Rightarrow^* \varepsilon$)

- if $\varepsilon \in \text{FIRST}(Y_j) \ \forall \ j \leq k$ add $\varepsilon$ to $\text{FIRST}(X)$

- if $X \rightarrow \varepsilon \in P$, add $\varepsilon$ to $\text{FIRST}(X)$
Place $ in FOLLOW(S), where $ is the start symbol
($ is an end marker)

if A → αBβ ∈ P, then FIRST(β) - {ε} is in FOLLOW(B)

if A → αB ∈ P or A → αBβ ∈ P where ε ∈ FIRST(β),
then everything in FOLLOW(A) is in FOLLOW(B)
Table-driven predictive parsing
Algorithm 4.32 (p. 224)

- **INPUT:** Grammar $G = (N,T,P,S)$
- **OUTPUT:** Parsing table $M$

For each production $A \rightarrow \alpha$ of $G$:

1. For each terminal $a \in \text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$

2. If $\varepsilon \in \text{FIRST}(\alpha)$, then for each terminal $b$ in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,b]$

3. If $\varepsilon \in \text{FIRST}(\alpha)$ and $\$ \in \text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A,\$]$