بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

کامپایلر

حسین فلسفین

Bottom-Up Parsing

Top-down parsers start with the starting symbol. Bottom-up parsers, in contrast, start with the first token of the input. Their mode of operation may seem very much more intuitive than top-down parsing. They repeatedly match symbols from the input with the strings on the right-hand sides of production rules, replacing the matched strings with the corresponding left-hand sides. This continues until, hopefully, just the starting symbol remains.

Bottom-up parsers discover a derivation by working from the words in the program toward the start symbol, S*. We can think of this process as building the parse tree starting from its leaves and working toward its root.*

Bottom-Up Parsing

In general we will not have this derivation to hand, so the bottom-up parser has to decide itself which substring to reduce at each stage. This substring that is matched with the right-hand side of a production and replaced by the corresponding left-hand side is called the handle and the key problem of bottom-up parsing is the identification of the handle at each stage.

$\mathrm{LR}(k)$

The most prevalent type of bottom-up parser today is based on a concept called $\mathrm{LR}(k)$ parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases $k = 0$ or $k = 1$ are of practical interest, and we shall only consider LR parsers with $k \leq 1$ here. When (k) is omitted, k is assumed to be 1.

$\mathrm{LR}(k)$ ***stands for left-to-right parse, rightmost-derivation, k-token lookahead.***

$\mathrm{LR}(k)$

An $\mathrm{LR}(k)$ parser uses the contents of its stack and the next k tokens of the input to decide which action to take. For $k = 2$, the table has columns for every two-token sequence and so on; in practice, $k > 1$ is not used for compilation. This is partly because the tables would be huge, but more because most reasonable programming languages can be described by $LR(1)$ grammars. $LR(0)$ grammars are those that can be parsed looking only at the stack, making shift/reduce decisions without any lookahead. This class of grammars is too weak to be very useful.

Grammars used in compiling usually fall in the $\mathrm{LR}(1)$ class, with one symbol of lookahead at most.

Parser Generators

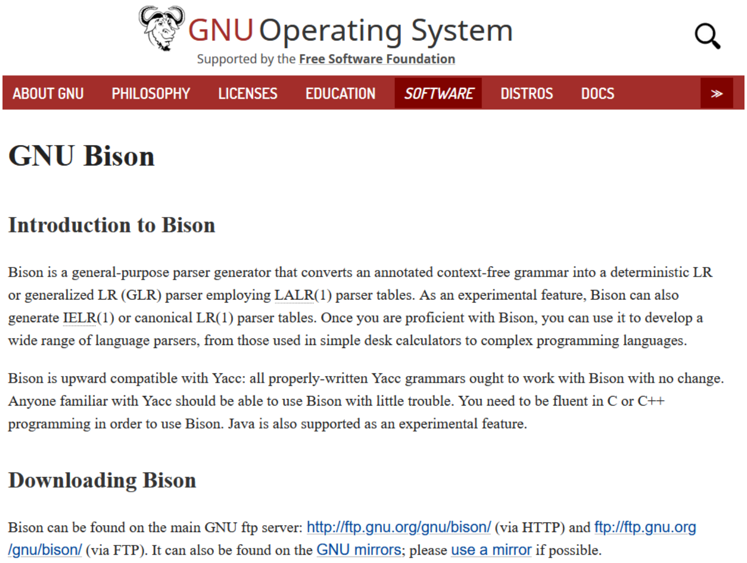
The price paid for the potential power of bottom-up parsing is a significant increase in software complexity, effectively making it advisable to make use of parser generator tools, rather than coding the parser directly in a conventional programming language. Using these specialized tools certainly simplifies the process, but it is important to have some knowledge of what is going on in the operation of a bottom-up parser.

Bison is descended from yacc, a parser generator written between 1975 and 1978 by Stephen C. Johnson at Bell Labs. In fact, Flex and bison are modern replacements for the classic lex and yacc that were both developed at Bell Laboratories in the 1970s.

Parser Generators

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed. Fortunately, many such generators are available. Such a generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

https://www.gnu.org/software/bison/



GNU Bison

Introduction to Bison

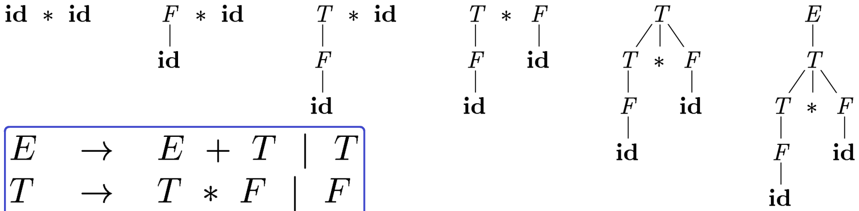
Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C or C++ programming in order to use Bison. Java is also supported as an experimental feature.

Downloading Bison

Bison can be found on the main GNU ftp server: http://ftp.gnu.org/gnu/bison/ (via HTTP) and ftp://ftp.gnu.org/gnu/bison/ (via FTP). It can also be found on the GNU mirrors; please use a mirror if possible.

A bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We introduce a general style of bottom-up parsing known as shift-reduce parsing.



$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Example 4.37 : The snapshots in Fig. 4.25 illustrate a sequence of reductions; the grammar is the expression grammar (4.1). The reductions will be discussed in terms of the sequence of strings

$$\mathbf{id} * \mathbf{id}, \quad F * \mathbf{id}, \quad T * \mathbf{id}, \quad T * F, \quad T, \quad E$$

The strings in this sequence are formed from the roots of all the subtrees in the snapshots. The sequence starts with the input string $\mathbf{id}*\mathbf{id}$. The first reduction produces $F*\mathbf{id}$ by reducing the leftmost **id** to F, using the production $F \rightarrow \mathbf{id}$. The second reduction produces $T * \mathbf{id}$ by reducing F to T.

Now, we have a choice between reducing the string T, which is the body of $E \rightarrow T$, and the string consisting of the second **id**, which is the body of $F \rightarrow \mathbf{id}$. Rather than reduce T to E, the second **id** is reduced to F, resulting in the string $T * F$. This string then reduces to T. The parse completes with the reduction of T to the start symbol E. $\square$

By definition, a reduction is the reverse of a step in a derivation (recall that in a derivation, a nonterminal in a sentential form is replaced by the body of one of its productions). The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following corresponds to the parse in Fig. 4.25:

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

This derivation is in fact a rightmost derivation.

Bottom-up parsing during a left-to-right scan of the input constructs a right-most derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of $\textbf{id}_1 * \textbf{id}_2$ according to the expression grammar (4.1) are as in Fig. 4.26. Although T is the body of the production $E \to T$, the symbol T is not a handle in the sentential form $T * \textbf{id}_2$. If T were indeed replaced by E, we would get the string $E * \textbf{id}_2$, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\textbf{id}_1 * \textbf{id}_2$	$\textbf{id}_1$	$F \to \textbf{id}$
$F * \textbf{id}_2$	F	$T \to F$
$T * \textbf{id}_2$	$\textbf{id}_2$	$F \to \textbf{id}$
$T * F$	$T * F$	$T \to T \; * \; F$
T	T	$E \to T$

Figure 4.26: Handles during a parse of $\textbf{id}_1 * \textbf{id}_2$

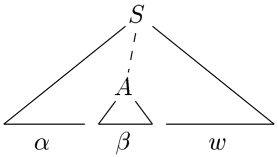


Figure 4.27: A handle $A \to \beta$ in the parse tree for $\alpha\beta w$

***Formally*, if $S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm}^{*} \alpha\beta w$, as in Fig. 4.27, then production $A \to \beta$ in the position following α is a *handle* of $\alpha\beta w$. *Alternatively*, a handle of a right-sentential form γ is a production $A \to \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.**

☞ *Notice that the string w to the right of the handle must contain only terminal symbols.*
☞ *For convenience, we refer to the body β rather than $A \to \beta$ as a handle.*
☞ *Note we say "a handle" rather than "the handle," because the grammar could be ambiguous, with more than one rightmost derivation of $\alpha\beta w$. If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.*

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals w to be parsed. If w is a sentence of the grammar at hand, then let $w = \gamma_n$, where γ_n is the nth right-sentential form of some as yet unknown rightmost derivation

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$$

To reconstruct this derivation in reverse order, we locate the handle β_n in γ_n and replace β_n by the head of the relevant production $A_n \to \beta_n$ to obtain the previous right-sentential form γ_{n-1}. Note that we do not yet know how handles are to be found, but we shall see methods of doing so shortly. We then repeat this process. That is, we locate the handle β_{n-1} in γ_{n-1} and reduce this handle to obtain the right-sentential form γ_{n-2}.

If by continuing this process we produce a right-sentential form consisting only of the start symbol S, then we halt and announce successful completion of parsing. The reverse of the sequence of productions used in the reductions is a rightmost derivation for the input string.

The algorithm used to identify the handle obviously has to be based on the identity of the lexical tokens read from the input. Ideally, the number of tokens needed to identify the handle should be limited in order to achieve an efficient parser. Furthermore, backtracking should not be necessary, again for efficiency reasons. So the identification of the handle should be possible by just considering the tokens of the handle itself together with tokens in the immediate locality of the handle. In other words, we may need to examine some left context, the handle itself and some lookahead.