Chapter Fifteen:
Stack Machine Applications

The parse tree (or a simplified version called the abstract syntax tree) is one of the central data structures of almost every compiler or other programming language system. To parse a program is to find a parse tree for it. Every time you compile a program, the compiler must first parse it. Parsing algorithms are fundamentally related to stack machines, as this chapter illustrates.

Outline

- 15.1 Top-Down Parsing
- 15.2 Recursive Descent Parsing
- 15.3 Bottom-Up Parsing
- 15.4 PDAs, DPDAs, and DCFLs

Parsing

- To parse is to find a parse tree in a given grammar for a given string
- An important early task for every compiler
- To compile a program, first find a parse tree
  – That shows the program is syntactically legal
  – And shows the program's structure, which begins to tell us something about its semantics
- Good parsing algorithms are critical
- Given a grammar, build a parser…

CFG to Stack Machine, Review

read	pop	push
ε	X	y
a	a	ε

- Two types of moves:
  1. A move for each production $X \to y$
  2. A move for each terminal $a \in \Sigma$
- The first type lets it do any derivation
- The second matches the derived string and the input
- Their execution is interlaced:
  - type 1 when the top symbol is nonterminal
  - type 2 when the top symbol is terminal

Top Down

- The stack machine so constructed accepts by showing it can find a derivation in the CFG
- If each type-1 move linked the children to the parent, it would construct a parse tree
- The construction would be top-down (that is, starting at root *S*)
- One problem: the stack machine in question is highly nondeterministic
- To implement, this must be removed

Almost Deterministic

$S \rightarrow aSa \mid bSb \mid c$

	read	pop	push
1.	ε	S	aSa
2.	ε	S	bSb
3.	ε	S	c
4.	a	a	ε
5.	b	b	ε
6.	c	c	ε

- Not deterministic, but move is easy to choose
- For example, *abbcbba* has three possible first moves, but only one makes sense:

$$(abbcbba, S) \mapsto_1 (abbcbba, aSa) \mapsto \ldots$$

$$(abbcbba, S) \mapsto_2 (abbcbba, bSb) \mapsto \ldots$$

$$(abbcbba, S) \mapsto_3 (abbcbba, c) \mapsto \ldots$$

Lookahead

$S \rightarrow aSa \mid bSb \mid c$

	read	pop	push
1.	ε	S	aSa
2.	ε	S	bSb
3.	ε	S	c
4.	a	a	ε
5.	b	b	ε
6.	c	c	ε

- To decide among the first three moves:
  - Use move 1 when the top is *S,* next input *a*
  - Use move 2 when the top is *S,* next input *b*
  - Use move 3 when the top is *S*, next input *c*
- Choose next move by peeking at next input symbol
- One symbol of lookahead lets us parse this deterministically

Lookahead Table

	a	b	c	$\$$
S	$S \rightarrow aSa$	$S \rightarrow bSb$	$S \rightarrow c$	

- Those rules can be expressed as a two-dimensional *lookahead table*
- *table*[*A*][*c*] tells what production to use when the top of stack is *A* and the next input symbol is *c*
- Only for nonterminals *A*; when top of stack is terminal, we pop, match, and advance to next input
- The final column, *table*[*A*][$], tells which production to use when the top of stack is *A* and all input has been read
- With a table like that, implementation is easy…

```
void predictiveParse(table, S) {
  initialize a stack containing just S
  while (the stack is not empty) {
    A = the top symbol on stack;
    c = the current symbol in input (or $ at the end)
    if (A is a terminal symbol) {
      if (A != c) the parse fails;
      pop A and advance input to the next symbol;
    }
    else {
      if table[A][c] is empty the parse fails;
      pop A and push the right-hand side of table[A][c];
    }
  }
  if input is not finished the parse fails
}
```

The Catch

- To parse this way requires a parse table
- That is, the choice of productions to use at any point must be uniquely determined by the nonterminal and one symbol of lookahead
- Such tables can be constructed for some grammars, but not all

LL(1) Parsing

- A popular family of top-down parsing techniques
  - Left-to-right scan of the input
  - Following the order of a leftmost derivation
  - Using 1 symbol of lookahead
- A variety of algorithms, including the table-based top-down parser we just saw

LL(1) Grammars And Languages

- LL(1) grammars are those for which LL(1) parsing is possible
- LL(1) languages are those with LL(1) grammars
- There is an algorithm for constructing the LL(1) parse table for a given LL(1) grammar
- LL(1) grammars can be constructed for most programming languages, but they are not always pretty…

Not LL(1)

$S \rightarrow (S) \mid S{+}S \mid S{*}S \mid a \mid b \mid c$

- This grammar for a little language of expressions is not LL(1)
- For one thing, it is ambiguous
- No ambiguous grammar is LL(1)

Still Not LL(1)

$$
\boxed{\begin{array}{l} S \rightarrow S{+}R \mid R \\ R \rightarrow R{*}X \mid X \\ X \rightarrow (S) \mid a \mid b \mid c \end{array}}
$$

- This is an unambiguous grammar for the same language
- But it is still not LL(1)
- It has left-recursive productions like $S \rightarrow S{+}R$
- No left-recursive grammar is LL(1)

LL(1), But Ugly

$S \to AR$
$R \to +AR \mid \varepsilon$
$A \to XB$
$B \to *XB \mid \varepsilon$
$X \to (S) \mid a \mid b \mid c$

	a	*b*	*c*	+	*	(	)	$
S	$S \to AR$	$S \to AR$	$S \to AR$			$S \to AR$		
R				$R \to +AR$			$R \to \varepsilon$	$R \to \varepsilon$
A	$A \to XB$	$A \to XB$	$A \to XB$			$A \to XB$		
B				$B \to \varepsilon$	$B \to *XB$		$B \to \varepsilon$	$B \to \varepsilon$
X	$X \to a$	$X \to b$	$X \to c$			$X \to (S)$		

- Same language, now with an LL(1) grammar
- Parse table is not obvious:
  - When would you use $S \to AR$?
  - When would you use $B \to \varepsilon$?

Outline

- 15.1 Top-Down Parsing
- 15.2 Recursive Descent Parsing
- 15.3 Bottom-Up Parsing
- 15.4 PDAs, DPDAs, and DCFLs

Recursive Descent

- A different implementation of LL(1) parsing
- Same idea as a table-driven predictive parser
- But implemented without an explicit stack
- Instead, a collection of recursive functions: one for parsing each nonterminal in the grammar

$S \to aSa \mid bSb \mid c$

```
void parse_S() {
  c = the current symbol in input (or $ at the end)
  if (c=='a') { // production S → aSa
    match('a'); parse_S(); match('a');
  }
  else if (c=='b') { // production S → bSb
    match('b'); parse_S(); match('b');
  }
  else if (c=='c') { // production S → c
    match('c');
  }
  else the parse fails;
}
```

- Still chooses move using 1 lookahead symbol
- But parse table is incorporated into the code

Recursive Descent Structure

- A function for each nonterminal, with a case for each production:

```
if (c=='a') { // production S → aSa
  match('a');parse_S(); match('a');
}
```

- For each RHS, a call to `match` each terminal, and a recursive call for each nonterminal:

```
void match(x) {
  c = the current symbol in input
  if (c!=x) the parse fails;
  advance input to the next symbol;
}
```

Example:

	a	b	c	$+$	$*$	$($	$)$	$\$$
S	$S \to AR$	$S \to AR$	$S \to AR$			$S \to AR$		
R				$R \to +AR$			$R \to \varepsilon$	$R \to \varepsilon$
A	$A \to XB$	$A \to XB$	$A \to XB$			$A \to XB$		
B				$B \to \varepsilon$	$B \to {*XB}$		$B \to \varepsilon$	$B \to \varepsilon$
X	$X \to a$	$X \to b$	$X \to c$			$X \to (S)$		

```
void parse_S() {
  c = the current symbol in input (or $ at the end)
  if (c=='a' || c=='b' ||
      c=='c' || c=='(') { // production S → AR
    parse_A(); parse_R();
  }
  else the parse fails;
}
```

Example:

	a	b	c	$+$	$*$	(	)	\$
S	$S \to AR$	$S \to AR$	$S \to AR$			$S \to AR$		
R				$R \to +AR$			$R \to \varepsilon$	$R \to \varepsilon$
A	$A \to XB$	$A \to XB$	$A \to XB$			$A \to XB$		
B				$B \to \varepsilon$	$B \to *XB$		$B \to \varepsilon$	$B \to \varepsilon$
X	$X \to a$	$X \to b$	$X \to c$			$X \to (S)$		

```
void parse_R() {
  c = the current symbol in input (or $ at the end)
  if (c=='+') // production R → +AR
    match('+'); parse_A(); parse_R();
  }
  else if (c==')' || c=='$') { // production R → ε
  }
  else the parse fails;
}
```

Where's The Stack?

- Recursive descent vs. our previous table-driven top-down parser:
  - Both are top-down predictive methods
  - Both use one symbol of lookahead
  - Both require an LL(1) grammar
  - Table-driven method uses an explicit parse table; recursive descent uses a separate function for each nonterminal
  - Table-driven method uses an explicit stack; recursive descent uses the call stack
- A recursive-descent parser is a stack machine in disguise

Outline

- 15.1 Top-Down Parsing
- 15.2 Recursive Descent Parsing
- 15.3 Bottom-Up Parsing
- 15.4 PDAs, DPDAs, and DCFLs

Shift-Reduce Parsing

- It is possible to parse bottom up (starting at the leaves and doing the root last)
- An important bottom-up technique, shift-reduce parsing, has two kinds of moves:
  - (shift) Push the current input symbol onto the stack and advance to the next input symbol
  - (reduce) On top of the stack is the string x of some production $A \to x$; pop it and push the A
- The shift move is the reverse of what our LL(1) parser did; it popped terminal symbols off the stack
- The reduce move is also the reverse of what our LL(1) parser did; it popped A and pushed x

$S \rightarrow aSa \mid bSb \mid c$

Input	Stack	Next move
$\underline{a}bbcbba\$$	ε	shift
$a\underline{b}bcbba\$$	a	shift
$ab\underline{b}cbba\$$	$b\,a$	shift
$abb\underline{c}bba\$$	$b\,b\,a$	shift
$abbc\underline{b}ba\$$	$\underline{c}b\,b\,a$	reduce by $S \rightarrow c$
$aaac\underline{b}bb\$$	$S\,b\,b\,a$	shift
$abbcb\underline{b}a\$$	$\underline{bSb}ba$	reduce by $S \rightarrow bSb$
$abbcb\underline{b}a\$$	$S\,b\,a$	shift
$abbcbb\underline{a}\$$	$\underline{bSb}a$	reduce by $S \rightarrow bSb$
$abbcbb\underline{a}\$$	$S\,a$	shift
$abbcbba\underline{\$}$	$\underline{aSa}$	reduce by $S \rightarrow aSa$
$abbcbba\underline{\$}$	S	

- A shift-reduce parse for *abbcbba*
- Root is built in the last move: that's bottom-up
- Shift-reduce is central to many parsing techniques…

LR(1) Parsing

- A popular family of shift-reduce parsing techniques
  - Left-to-right scan of the input
  - Following the order of a rightmost derivation in reverse
  - Using 1 symbol of lookahead
- There are many LR(1) parsing algorithms
- Generally trickier than LL(1) parsing:
  - Choice of shift or reduce move depends on the top-of stack string, not just the top-of-stack symbol
  - One cool trick uses stacked DFA state numbers to avoid expensive string comparisons in the stack

LR(1) Grammars And Languages

- LR(1) grammars are those for which LR(1) parsing is possible
  - Includes all of LL(1), plus many more
  - Making a grammar LR(1) usually does not require as many contortions as making it LL(1)
  - This is the big advantage of LR(1)
- LR(1) languages are those with LR(1) grammars
  - Most programming languages are LR(1)

Parser Generators

- LR parsers are usually too complicated to be written by hand
- They are usually generated automatically, by tools like yacc:
  - Input is a CFG for the language
  - Output is source code for an LR parser for the language

Beyond LR(1)

- LR(1) techniques are efficient
- Like LL(1), linear in the program size
- Beyond LR(1) are many other parsing algorithms
- Cocke-Kasami-Younger (CKY), for example:
  - Deterministic
  - Works on all CFGs
  - Much simpler than LR(1) techniques
  - But cubic in the program size
  - Much to slow for compilers and other programming-language tools

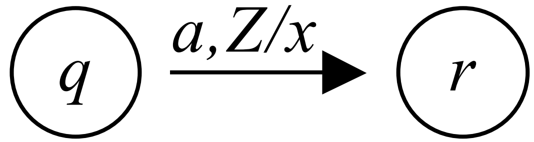
Outline

- 15.1 Top-Down Parsing
- 15.2 Recursive Descent Parsing
- 15.3 Bottom-Up Parsing
- 15.4 PDAs, DPDAs, and DCFLs

PDA

- A widely studied stack-based automaton: the pushdown automaton (PDA)
- A PDA is like an NFA plus a stack machine:
  - States and state transitions, like an NFA
  - Each transition can also manipulate an unbounded stack, like a stack machine

PDA Transitions



- Like an NFA transition: in state *q*, with *a* as the next input, read past it and go to state *r*
- Plus a stack machine transition: reading an *a*, with *Z* as the top of the stack, pop the *Z* and push an *x*
- All together:
  - In state *q*, with *a* as the next input, and with *Z* on top of the stack, read past the *a*, pop the *Z*, push *x*, and go to state *r*

Variations

- Many minor PDA variations have been studied:
  - Accept by empty stack (like stack machine), or by final state (like NFA), or require both to accept
  - Start with a special symbol on stack, or with empty stack
  - Start with special end-of-string symbol on the input, or not
- DFAs and NFAs are comparatively standardized

Why Study PDAs

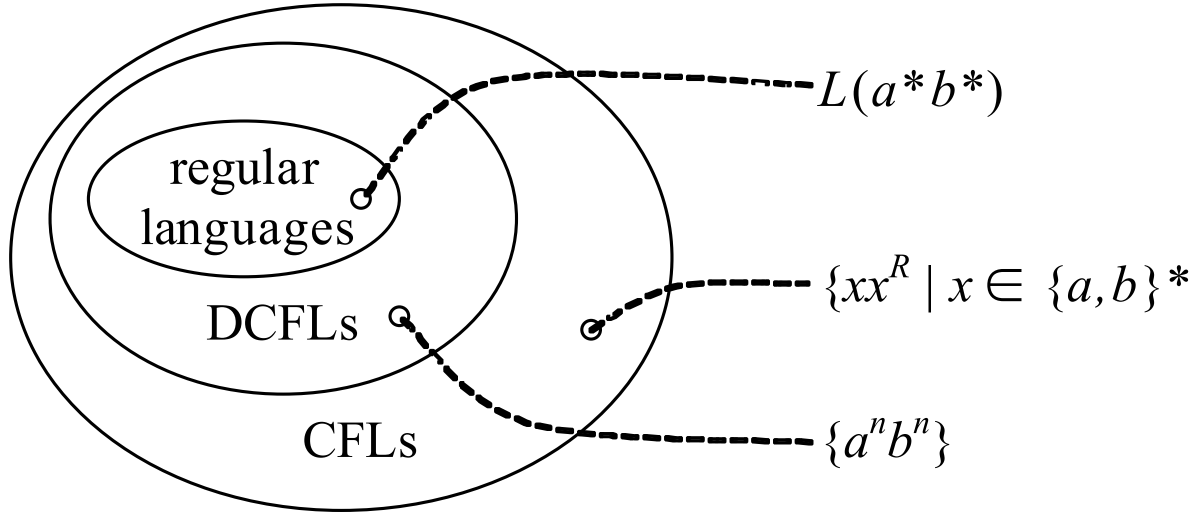
- PDAs are more complicated than stack machines
- The class of languages ends up the same: the CFLs
- So why bother with PDAs?
- Several reasons:
  - They make some proofs simpler: to prove the CFLs closed for intersection with regular languages, for instance, you can do a product construction combining a PDA and an NFA
  - They make a good story: an NFA is bitten by a radioactive spider and develops super powers…
  - They have an interesting deterministic variety: the DPDAs...

Deterministic Restriction

- Finite-state automata
  - NFA has zero or more possible moves from each configuration
  - DFA is restricted to exactly one
  - DFA defines a simple computational procedure for deciding language membership
- Pushdown automata
  - PDA, like a stack machine, has zero or more possible moves from each configuration
  - DPDA is restricted to no more than one
  - DPDA gives a simple computational procedure for deciding language membership

Important Difference

- The deterministic restriction does not seriously weaken NFAs: DFAs can still define exactly the regular languages
- It *does* seriously weaken PDAs: DPDAs are strictly weaker than PDAs
- The class of languages defined by DPDAs is a proper subset of the CFLs: the DCFLs
- A deterministic context-free language (DCFL) is a language that is $L(M)$ for some DPDA M



- DCFLs includes all the regular languages
- But not all CFLs: for instance, those xx^R languages
- Intuitively, that makes sense: no way for a stack machine to decide where the middle of the string is
- On the other hand, $\{xcx^R \mid x \in \{a,b\}^*\}$ is a DCFL

Closure Properties

- DCFLs do not have the same closure properties as CFLs:
  - Not closed for union: the union of two DCFLs is not necessarily a DCFL (though it is a CFL)
  - Closed for complement: the complement of a DCFL is another DCFL
- Can be used to prove that a given CFL is not a DCFL
- Such proofs are difficult; there seems to be no equivalent of the pumping lemma for DCFLs

There It Is Again

- Language classes seem more important when they keep turning up:
  - Regular languages turn up in DFAs, NFAs, regular expressions, right-linear grammars
  - CFLs turn up in CFGs, stack machines, PDAs
- DCFLs also receive this kind of validation:
  - LR(1) languages = DCFLs