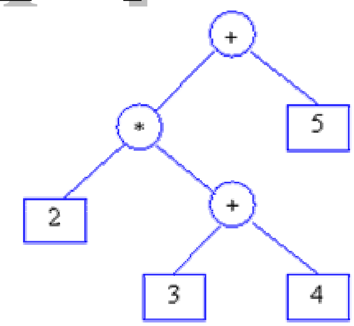
Chapter 4
(c) parsing



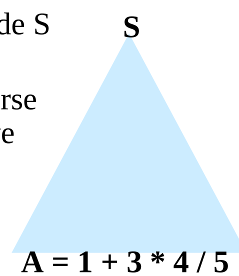
2 * (3 + 4) + 5

Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL
- A *recogniser* simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar
- A *parser* construct a derivation or parse tree for a sentence (if possible)
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A *recursive descent parser* is a way to implement a top-down parser that is particularly simple.

Top down vs. bottom up parsing

- The parsing problem is to connect the root node S with the tree leaves, the input
- **Top-down parsers:** starts constructing the parse tree at the top (root) of the parse tree and move down towards the leaves. Easy to implement by hand, but work with restricted grammars. examples:
  - Predictive parsers (e.g., LL(k))
- **Bottom-up parsers:** build the nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handle a larger class of grammars. examples:
  - shift-reduce parser (or LR(*k*) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).



Top down vs. bottom up parsing

- Both are general techniques that can be made to work for all languages (but not all grammars!).
- Recall that a given language can be described by several grammars.
- Both of these grammars describe the same language

```
E -> E + Num
E -> Num
```

```
E -> Num + E
E -> Num
```

- The first one, with it's left recursion, causes problems for top down parsers.
- For a given parsing technique, we may have to transform the grammar to work with it.

Parsing complexity

- How hard is the parsing task?
- Parsing an arbitrary Context Free Grammar is $O(n^3)$, e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
- If we constrain the grammar somewhat, we can always parse in linear time. This is good!
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy

> - **LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.**
> - **LR(n) : Left to right, Right derivation, look ahead at most n symbols.**

Top Down Parsing Methods

- Simplest method is a full-backup, *recursive descent* parser
- Often used for parsing simple languages
- Write recursive recognizers (subroutines) for each grammar rule
  - If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  - If rule fails, return failure. Caller may try another choice or fail
  - On failure it "backs up"

Top Down Parsing Methods

- Problems
  - When going forward, the parser consumes tokens from the input, so what happens if we have to back up?
  - Algorithms that use backup tend to be, in general, inefficient
  - Grammar rules which are left-recursive lead to non-termination!

Recursive Decent Parsing Example

Example: For the grammar:

```
<term> -> <factor> {(*|/)<factor>}*
```

We could use the following recursive descent parsing subprogram (this one is written in C)

```
void term() {
  factor();     /* parse first factor*/
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();  /* get next token */
    factor();   /* parse next factor */
  }
}
```

Problems

- Some grammars cause problems for top down parsers.
- Top down parsers do not work with left-recursive grammars.
  - E.g., one with a rule like: E -> E + T
  - We can transform a left-recursive grammar into one which is not.
- A top down grammar can limit backtracking if it only has one rule per non-terminal
  - The technique of rule factoring can be used to eliminate multiple rules for a non-terminal.

Left-recursive grammars

- A grammar is left recursive if it has rules like

  X -> X β

  Or if it has indirect left recursion, as in

  X -> A β

  A -> X
- Why is this a problem?
- Consider

  E -> E + Num

  E -> Num
- We can manually or automatically rewrite a grammar to remove left-recursion, making it suitable for a top-down parser.

Elimination of Left Recursion

- Consider the left-recursive grammar

  $S \rightarrow S\ \alpha \mid \beta$
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion

  $S \rightarrow \beta\ S'$

  $S' \rightarrow \alpha\ S' \mid \varepsilon$

More Elimination of Left-Recursion

- In general

  $S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

  $S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$

  $S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar

  $S \rightarrow A\ \alpha \mid \delta$

  $A \rightarrow S\ \beta$

  is also left-recursive because

  $S \rightarrow^{+} S\ \beta\ \alpha$

  where ->+ means "can be rewritten in one or more steps"
- This indirect left-recursion can also be automatically eliminated

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by restricting the grammar, allowing us to successfully *predict* which rule to use.

Predictive Parser

- A **predictive parser** uses information from the ***first terminal symbol*** of each expression to decide which production to use.
- A predictive parser is also known as an **LL(*k*)** parser because it does a ***Left-to-right parse***, a ***Leftmost-derivation***, and ***k-symbol lookahead.***
- A grammar in which it is possible to decide which production to use examining only the first token (as in the previous example) are called **LL(1)**
- LL(1) grammars are widely used in practice.
  - The syntax of a PL can be adjusted to enable it to be described with an LL(1) grammar.

Predictive Parser

Example: consider the grammar

S → **if** *E* **then** *S* **else** *S*
S → **begin** *S L*
S → **print** *E*
L → **end**
L → **;** *S L*
E → num = num

An *S* expression starts either with an IF, BEGIN, or PRINT token, and an *L* expression start with an END or a SEMICOLON token, and an *E* expression has only one production.

LL(k) and LR(k) parsers

- Two important classes of parsers are called LL(k) parsers and LR(k) parsers.
- The name LL(k) means:
  - L - Left-to-right scanning of the input
  - L - Constructing leftmost derivation
  - k – max number of input symbols needed to select a parser action
- The name LR(k) means:
  - L - Left-to-right scanning of the input
  - R - Constructing rightmost derivation in reverse
  - k – max number of input symbols needed to select a parser action
- So, a LL(1) parser never needs to "look ahead" more than one input token to know what parser production to apply.

Predictive Parsing and Left Factoring

- Consider the grammar

  E → T + E | T

  T → int | int * T | (E)
- Hard to predict because
  - For T, two productions start with *int*
  - For E, it is not clear how to predict which rule to use
- A grammar must be <u>left-factored</u> before use for predictive parsing
- Left-factoring involves rewriting the rules so that, if a non-terminal has more than one rule, each begins with a terminal.

Left-Factoring Example

- Consider the grammar

  E → T + E | T

  T → int | int * T | (E)
- Factor out common prefixes of productions

  E → T X

  X → + E | ε

  T → (E) | int Y

  Y → * T | ε

Left Factoring

- Consider a rule of the form

  A -> a B1 | a B2 | a B3 | … a Bn
- A top down parser generated from this grammar is not efficient as it requires backtracking.
- To avoid this problem we left factor the grammar.
  - collect all productions with the same left hand side and begin with the same symbols on the right hand side
  - combine the common strings into a single production and then append a new non-terminal symbol to the end of this new production
  - create new productions using this new non-terminal for each of the suffixes to the common production.
- After left factoring the above grammar is transformed into:

  A –> a A1

  A1 -> B1 | B2 | B3 … Bn

Using Parsing Tables

- LL(1) means that for each non-terminal and token there is only one production
- Can be specified via 2D tables
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production
- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Table Example

- Left-factored grammar

  E → T X　　　　X → + E | ε

  T → (E) | int Y　　　　Y → * T | ε
- The LL(1) parsing table:

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example

- Consider the [E, int] entry
  - "When current non-terminal is E and next input is *int*, use production E → T X
  - This production can generate an *int* in the first place
- Consider the [Y, +] entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation in which Y → ε
- Blank entries indicate error situations
  - Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Bottom-up Parsing

- YACC uses bottom up parsing. There are two important operations that bottom-up parsers use. They are namely shift and reduce.
  - (In abstract terms, we do a simulation of a Push Down Automata as a finite state automata.)
- Input: given string to be parsed and the set of productions.
- Goal: Trace a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol.