Data Structures and Algorithms II
Lecture 11:
Introduction to Parsing

- Introduction
- Grammars
- Simple Parsing Methods

Introduction

- Any natural language or programming language has a “vocabulary” of words.

  e.g., + - ; == x for if while ..
  John Mary cat loves hates the ..

- Some sequences of words are legal, some aren’t.

  ’x = for while 1’ NOT LEGAL!
  ’John Mary cat the’ NOT LEGAL!

- Grammars provide a method of specifying which of the sequences are legal, and which are not.
- Parsers do the job of checking whether a sequence is legal.

Example Grammar

Here’s an example grammar for a very tiny subset of the language Pascal:

```
1. program ->    stmts '.'
2. stmts   ->    stmt ';' stmts
3. stmts   ->    stmt
4. stmt    ->    ID ':=' expr
5. expr    ->    NUM
6. expr    ->    expr '+' NUM
```

- Each rule consists of a left-hand-side (LHS) and a right-hand-side (RHS) separated by a `->`. (Other notations may use a ‘:==’ to separate).
- `->` can be read as “can consist of”.
- The LHS consists of a single symbol, whereas the RHS can consist of a sequence of grammar symbols.
- Some symbols are *terminal symbols* corresponding to words in the language, some are *non-terminal* and internal to the grammar.

- In the above example, NUM and ID are special terminal symbols that match a whole set of words (ie, any valid number or variable name). These symbols will be in capitals.

The above grammar allows the following as a legal program:

```
x := 1; y := 1; z := 1;
a := x+y+z.
```

Rewriting

- The rules are often called *rewrite rules*. Each rule states that if we have an occurence of the LHS symbol we can rewrite (replace) it with an occurent of the RHS symbols.
- If we can rewrite a *start symbol* (e.g., program) to our input sequence then that shows that the sequence is a legal one in the language, e.g.,:

```
program ==>     stmts .        by rule 1
        ==>     stmt .         by rule 3
        ==>     ID := expr .   by rule 4
        ==>     ID := NUM .    by rule 5
```

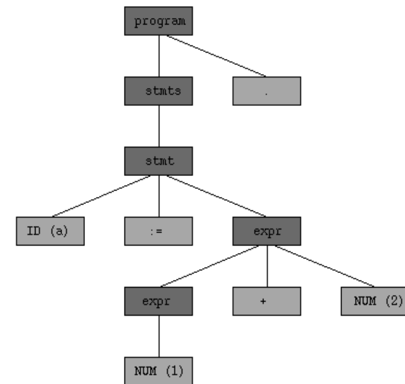
Shows that a sequence such as “x := 2.” is a legal program.

- We say we have *derived* the sentence “x := 2.” from the start symbol. The language defined by the grammar is the set of all such sequences (called *sentences*), which can be derived from the start symbol by applying rewrite rules.

Parse Trees

- Determining whether a sequence is legal is only half the story..
- For almost all applications, you also want to find out the *structure* of the sequence (who elements are grouped). This will make further processing possible (such as compiling program into sequence of primitive instructions).
- This will be a tree structure: the parse tree.
- The grammar reflects the structure of the language (each symbol should correspond to a meaningful unit), and the parse tree can be based directly on the rewrite rules applied.

For the program “a:=1.” we have the following parse tree.

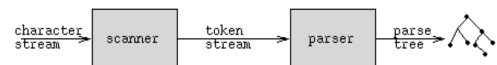


The root of the parse tree correponds to the start symbol. For each step in the derivation of the sequence from the start symbol, when the LHS is replaced by the RHS of a rule, the RHS symbols are added as child nodes in the tree.

When the derivation is complete, the leaf nodes should correspond to the words in the sequence being analysed.

Language Analysis

- Our goal in parsing is to take a sequence of characters and extract a parse tree (at the same time we verify that the sequence is legal).
- This is usually done in two stages:



In practice the two stages are usually interleaved, but are handled by separate functions.

Scanner

- A *scanner* or *tokeniser* takes the sequence of characters and returns a sequence of meaningful *tokens* corresponding to words. For example, it might take the sequence “var:=35” and split it into tokens corresponding to “var” “:=” and “35”.
- The tokeniser should extract both the text (*lexeme*) and the *class* of the item. So, a token for “35” should contain the text (35) and the type NUM.
- So.. a suitable datatype for a token will be a struct or class such as the following:

```
struct token {
  char* text;
  tokentype type;
};
```

where tokentype is an enumerated type specifying possible types of token.

Parser

- The parser accepts a stream of tokens, checks that it is legal according to the grammar, and builds a parse tree.
- Parsers can work *top down* or *bottom up*.
  - Top down parsers start with the start symbol and try and derive the input sequence, applying rules left to right.
  - Bottom up parsers start with the input sequence and try and derive the start symbol applying rules right to left (rewriting symbols corresponding with RHS of rule with symbol on LHS of rule)

Top Down Parsing

- In top down parsing we keep trying to rewrite symbols, while traversing through a sequence of tokens.
- When the first symbol in the sequence is a terminal symbol, we check it againsts the current token. If they match, we get the next token, and remove the terminal symbol from the sequence.

Example:

```
1. program ->    stmts '.'
2. stmts   ->    stmt ';' stmts
3. stmts   ->    stmt
4. stmt    ->    ID ':=' expr
5. expr    ->    NUM
6. expr    ->    expr '+' NUM
```



```
Input sequence: 'a := 1.'
First token = 'a'

program  ==>  stmts '.'
         ==>  stmt '.'
         ==>  ID ':=' expr '.'
  (ID matches 'a', so get next token ':='
  and remove ID from sequence)
         ==>  ':=' expr '.'
  (':=' matches token, so get next token '1'
  and remove ':=' from sequence)
         ==>  expr '.'
         ==>  NUM '.'
  (next token = '.')

         ==>  '.'
         ==>
```

In this form of derivation, the parse is successful when there are no more symbols to rewrite, and we have processed all the tokens on the input stream.

Look Ahead

- For both top down and bottom up parsers, the problem is that there is usually more than one possible rewrite rule that can be applied.
  - Which symbol should be rewritten?
  - Which rule should be used to rewrite it (for example, for top down parsing and our example grammar there are two rules for `stmts` and for `expr`.)
- For top down parsing we can choose to always rewrite the first non-terminal in the current sequence.
- We can usually decide which rule to use by *looking ahead* one token in the input sequence.

Top Down Parsing with one Token Look Ahead

Consider the following grammar, for bracketed arithmetic expressions.

```
1. expr  ->  '(' expr op expr ')'
2. expr  ->  NUM
3. op    ->  '+'
4. op    ->  '-'
```

It allows sequences like the following:

```
(1 + (2 - 4))
((4 - 5) + (6 + 4))
```

Let's see how we could rewrite 'expr' to '(1 + (2-4))' using one token lookahead to decide which rules to use:

```
   first token = '('
   apply rule 1 token matches first symbol
   of RHS.
expr ==> '(' expr op expr ')'
     ==> expr op expr ')'
   get next token '1' and remove '('
     ==> expr op expr ')'
```



```
   apply rule 2 as token matches RHS.
     ==> NUM op expr ')'
   get next token '+'
     ==> op expr ')'
     ==> op expr ')'
   apply rule 3
     ==> '+' expr ')'
etc
```

Each time a choice needs to be made between two rules, this can be done on the basis of the current token and the first symbol of the RHS of the rule.

Left Recursive Rules

- Simple top down parsing schemes will fail if grammars are *left recursive*.
- If the first symbol on the RHS is the same as the symbol on the LHS we say the rule is directly left recursive, e.g.,:

```
expr  ->  expr '+' NUM   left recursive
expr  ->  NUM
```

- Rules can also be *indirectly* left recursive:

```
expr    -> result '+' NUM
result  -> expr ..
```

- It is usually possible to transform such grammars into non-left recursive ones, e.g.,

```
expr     ->  NUM exprr
est
exprrest -> '+' NUM exprrest
```

Summary

- Parsing involves checking that a sequence is legal in a language and obtaining a tree structure of the sequence.
- Grammars specify the legal sentences in a language – if a start symbol can be rewritten to give the sentence, by applying rewrite rules, then that sentence is legal.
- Normally parsing process split into scanner+parser. Scanner returns meaningful items in the character sequence (e.g., if(num==33) has tokens if, (, num, ==, 33,).)
- Parsing can proceed top town or bottom up. Top down starts with start symbol and tries to derive sentence, bottom up does opposite.
- Usually several rewrite rules may apply, and parsing must be controlled by “looking ahead” one token. Choose rule where current token matches first symbol on RHS of rule.