

4. Syntax

Overview

- Tokens and regular expressions
- Syntax and context-free grammars
- Grammar derivations
- Parse trees
- Top-down and bottom-up parsing
- Recursive descent parsing
- Putting theory into practice:
 - Writing a Recursive Descent Parser for Simple Expressions

Note: Study Chapter 2 of the textbook up to and including Section 2.2.3.

Tokens Revisited

- Tokens are the basic building blocks of a programming language: keywords, identifiers, numbers, punctuation
- We saw that the first compiler phase (scanning) splits up a character stream into tokens
- Tokens have a special role with respect to:
 - *Free-format* language: source program is a sequence of tokens and horizontal/vertical position of a token on a page is unimportant (e.g. Pascal)
 - *Fixed-format* language: indentation and/or position of a token on a page is significant (early Basic, Fortran, Haskell)
 - *Case-sensitive* language: upper- and lowercase are distinct (C, C++, Java)
 - *Case-insensitive* language: upper- and lowercase are identical (Ada, Fortran, Pascal)

Describing Tokens by Regular Expressions

- The makeup of a token is described by a *regular expression*
- A regular expression is
 - a character
 - *empty* (denoted *e*)
 - *concatenation*: sequence of regular expressions
 - *alternation*: regular expressions separated by a bar |
 - *repetition*: a regular expression followed by a star * (called Kleene star)

Example regular expressions

```
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
unsigned_integer -> digit digit*
signed_integer -> (+ | - | e) unsigned_integer
```

Note: Java provides a class `StreamTokenizer` with which you can write scanners in Java to convert character streams into token streams

Context-Free Grammars: BNF

- Regular expressions cannot describe nested constructs, but *context-free grammars* can
- Backus-Naur Form (BNF) grammar *productions* are of the form
`<nonterminal> -> sequence of (non)terminals`
- A *terminal* of a grammar is a token e.g. specific programming language keyword, e.g. `return`
- A *<nonterminal>* denotes a syntactic category
- The symbol | denotes *alternative* forms in a production, e.g. different program statements are categorized, e.g.
`<stmt> -> return | break | <id> := <expression>`
- The special symbol *e* denotes *empty*, e.g. used in optional constructs, e.g.
`<optional_static> -> static | e`

Extended BNF

- *Extended* BNF includes an explicit form for *optional* constructs with [and]
For example:
`<stmt> -> for <id> := <expr> to <expr> [step <expr>]
do <stmt>`
- *Extended* BNF includes a *repetition* construct *
For example:
`<decl> -> int <id> (, <id>)*`

Example Grammar for Expressions

Context-free grammar for a simple expression syntax with identifiers, integers, unary minus, parenthesis, and +, -, *, /

Example expression grammar productions

```
<expression> -> identifier  
                unsigned_integer  
                - <expression>  
                ( <expression> )  
                <expression> <operator> <expression>  
  
<operator> -> + | - | * | /
```

Note that `identifier` and `signed_integer` are tokens defined by a regular expression, not by the grammar. They are provided as tokens by the scanner in a compiler.

Derivations

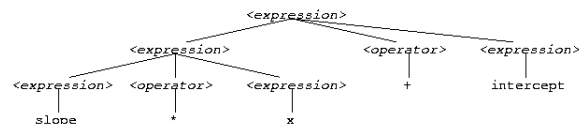
- From a grammar we can derive *strings* (= sequences of tokens/terminals)
- In each *derivation step* a nonterminal is replaced by a right-hand side (part after ->) of a production for that nonterminal
- Each representation after each step is called a *sentential form*
- When the nonterminal on the far right (left) in a sentential form is replaced in each derivation step the derivation is called *right-most* (*left-most*)
- The final form consists of terminals only and is called the *yield* of the derivation
- A context-free grammar is a *generator* of a *context-free language*: the language defined by the grammar is the set of all strings that can be derived

Example derivation (right-most)

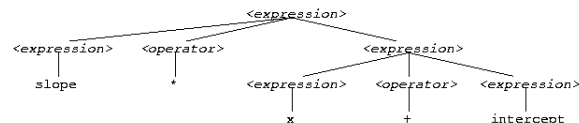
```
<expression>  
=> <expression> <operator> <expression>  
=> <expression> <operator> identifier  
=> <expression> + identifier  
=> <expression> <operator> <expression> + identifier  
=> <expression> <operator> identifier + identifier  
=> <expression> * identifier + identifier  
=> identifier * identifier + identifier
```

Parsing and Parse Trees

- A *parse tree* depicts a derivation as a tree
- The *nodes* are the nonterminals
- The *children* of a node are the symbols (terminals and nonterminals) on a right-hand side of a production
- The *leaves* are the terminals
- For example, given string `slope*x+intercept` a *parser* constructs a *parse tree*:



- An alternative *parse tree* for this string is:



Note: An interactive parser demo demonstrates the parsing of a Pascal example program into a parse tree (see also textbook pp. 20-21)

Ambiguous Grammars

- When more than one distinct derivation of a string exists resulting in distinct parse trees, the grammar is *ambiguous* (as is the case above)
- A programming language construct should have only one parse tree to avoid misinterpretation by a compiler
- For expression grammars, *associativity* and *precedence* of operators need to be included somehow

An unambiguous grammar for simple expressions

```
<expression> -> <term>
                | <expression> <add_op> <term>

<term> -> <factor>
          | <term> <mult_op> <factor>

<factor> -> identifier | unsigned_integer
          | - <factor> | ( <expression> )

<add_op> -> + | -

<mult_op> -> * | /
```

Try this: construct *all* possible left-most derivations of the string $a-b+1$ from the ambiguous simple expression grammar and from the unambiguous grammar. Also construct the parse trees. Answer:

Ambiguous If-Then-Else

- A classical example of an ambiguous grammar are the grammar productions for *if-then-else* in C, C++, and Pascal
- It is possible to write an unambiguous grammar, but the fact that it is not easy indicates a problem in the programming language design

An ambiguous grammar for if-then-else

```
<stmt> -> if <expr> then <stmt>
          | if <expr> then <stmt> else <stmt>
```

- Ada uses `if then [else] end if`

Try this: given the above grammar, find two derivations for the program fragment

```
if C1 then if C2 then S1 else S2
```

(where C_1 and C_2 are some expressions, S_1 and S_2 are some statements)

Answer:

Top-Down and Bottom-Up Parsing

- A parser is a *recognizer* of a context-free language
 - a string can be parsed into a parse tree only if the string is in the language
- For any arbitrary context-free grammar parsing can be done in $O(n^3)$ time, where n is the size of the input
- There are large classes of grammars for which we can construct parsers that run in linear time:
 - Top-down parsers for LL (Left-to-right scanning of input, Left-most derivation) grammars
 - Bottom-up parsers for LR (Left-to-right scanning of input, Right-most derivation) grammars

LL Grammars and Top-Down Parsing

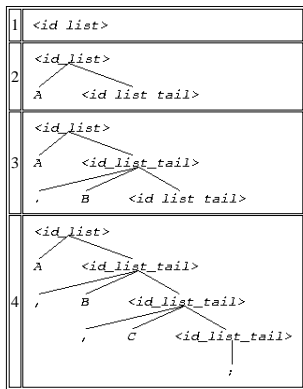
- Top-down parser is a parser for LL class of grammars (which is a subset of the larger LR class of grammars)
- Also called *predictive* parser
- Top-down parser constructs parse tree from the root down
- Easy to implement a predictive parser for an LL grammar by hand
- LL grammars cannot exhibit *left-recursive productions* (but LR can)

Example LL grammar for list of identifiers

```
<id_list> -> identifier <id_list_tail>
<id_list_tail> -> , identifier <id_list_tail>
                | ;
```

Top-Down Parsing Example

Top-down parsing of $A, B, C;$



- Top-down parsing is called *predictive* parsing because it predicts what it is going to see:
 - As root *<id_list>* is predicted
 - After reading A the parser predicts that *<id_list_tail>* must follow
 - After reading , and B the parser predicts that *<id_list_tail>* must follow
 - After reading , and c the parser predicts that *<id_list_tail>* must follow
 - After reading ; the parser stops

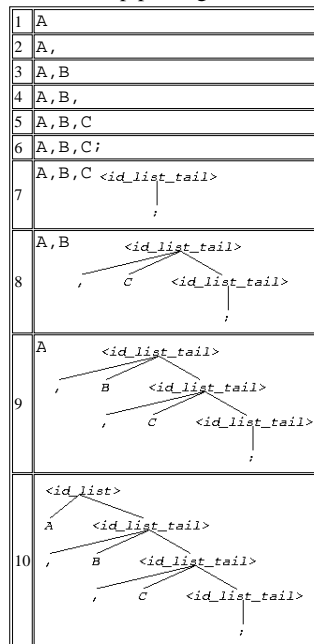
LR Grammars and Bottom-Up Parsing

- Bottom-up parser is a parser for LR class of grammars
- Difficult to implement by hand
- Tools (e.g. bison) exist that generate bottom-up parsers for LR

grammars

- Parsing is based on shifting tokens on a stack until it recognizes a right-hand side of a production which it then reduces to a left-hand side (nonterminal) with a partial parse tree

Bottom-up parsing of A,B,C;



Recursive Descent Parsing

- Predictive parsing method for LL(1) grammar (LL with one token lookahead)
- Based on recursive subroutines
 - Each nonterminal has a subroutine that implements the production(s) for that nonterminal so that calling the subroutine will parse a part of a string described by the nonterminal
 - When more than one alternative production exists for a nonterminal, lookahead token from scanner should decide which production is to be applied

LL(1) for a simple calculator language

```

<expr> -> <term> <term_tail>
<term_tail> -> <add_op> <term> <term_tail> | ε
<term> -> <factor> <factor_tail>
<factor_tail> -> <mult_op> <factor> <factor_tail> | ε
<factor> -> ( <expr> ) | - <factor>
| identifier | unsigned_integer
<add_op> -> + | -
<mult_op> -> * | /

```

A Recursive Descent Parser

Pseudo-code outline of recursive descent parser for the calculator grammar

```

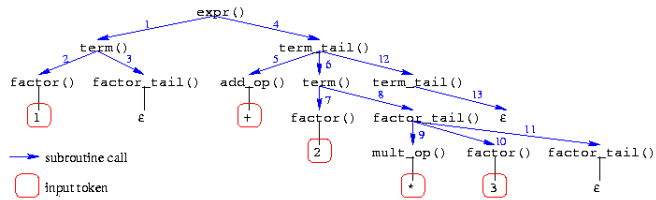
procedure expr()
  term(); term_tail();
procedure term_tail()
  case (input_token())
  of '+' or '-': add_op(); term(); term_tail();
  otherwise: /* skip */
procedure term()
  factor(); factor_tail();
procedure factor_tail()
  case (input_token())
  of '*' or '/': mult_op(); factor(); factor_tail();
  otherwise: /* skip */
procedure factor()
  case (input_token())
  of '(': match('('); expr(); match(')');
  of '-': factor();
  of identifier: match(identifier);
  of number: match(number);
  otherwise: error;
procedure add_op()
  case (input_token())
  of '+': match('+');
  of '-': match('-');
  otherwise: error;
procedure mult_op()
  case (input_token())
  of '*': match('*');
  of '/': match('/');
  otherwise: error;

```

Try this: Write a recursive descent parser in your favorite programming language for the grammar shown above. Answer (Java):

Example Recursive Descent Parsing

- The *dynamic call graph* of a recursive descent parser corresponds exactly to the parse tree of input
- Call graph of input string $1+2*3$



Exercise 1: Write a regular expression to capture the format of floating point constants in C/C++.

Exercise 2: Many IETF (Internet Engineering Task Force) protocols are defined with a grammar.

HTTP/1.1 for example, is defined in RFC2616 [click here]. Read Section 2.1 of RFC2616. Section 3.2.2 defines a HTTP URL:

```
http_URL = "http:" "/" host [ ":" port ] [
  abs_path [ "?" query ] ]
```

Find and write down the definitions of the nonterminals `host`, `port`, `abs_path`, and `query`. Also find the definitions of the nonterminals on which `host`, `port`, `abs_path`, and `query` depend.

Exercise 3: Given the unambiguous grammar for simple expressions shown in the note on "Ambiguous Grammars", construct the parse tree of $1+2*3$