Syntax

Constructs specification (by Context Free Grammar)

2.1.2, 2.1.3

Introduction

- A designer specifies a language by specifying its syntax and semantics
- To specify syntax, a designer uses
  - Regular expression – tokens
  - Context free grammar – constructs
- The syntax is used
  - by programmers to understand this language
  - by implementers of language to build a compiler

Context Free Grammar

- Motivating example: arithmetic expression, as a nested structure, can not be represented by regular expressions
- An attempt to define arithmetic expression (in English)
  - A number is an *arithmetic expression*
  - A variable (identifier) is an *arithmetic expression*
  - If α and β are arithmetic expressions, $\alpha - \beta$ and $\alpha + \beta$ are *arithmetic expressions*.

- *Context free grammar* consist of
  - A set of *terminals* *T* (each terminal is an identifier or a character)
  - A set of *non-terminals* *N* (a non-terminal is a name inside <>)
  - A *start symbol* *S* (a non-terminal)
  - A set of *production rules* of the form:

    P -> a string of terminals or non-terminals or space or |

    Where *P* is a non-terminal.

- A context free grammar specifies all “valid” sentences of a language.
- Informally, the start symbol represents all sentences “valid” in the language specified by the grammar.

Specifying a language

- To specify a programming language, the syntax is separated into two parts
  - The part for tokens
  - The part for constructs

- Example – the language of artithmetic expressions
  - First part – tokens: id, number,
    - RE for these tokens

      id -> letter letter^   letter -> a|....|z|A...|Z

      number -> (0|…|9) (0|…|9)^

      op -> + | - | * | /

- Second part – constructs: e.g., 4*5-10+5
    - Context free grammar for expressions

      <expr> -> id | number | <expr> op <expr> | -<expr>
- The token names in part one are taken as *terminals* in part two.

Derivations and parse trees

- The question: is a sentence valid in a language?
  - A sentence is valid in a language if it follows its grammar.
  - In other words, the sentence follows the definition of the start non-terminal
  - Example: 10 – 5 – 5 (try to apply the first production on the start non-terminal)

Derivation

- Derivation
  - A derivation is "a series of *replacement* operations that derive a string of terminals from *the start symbol*."
  - *Replacement*: replace a non-terminal N by the right hand side of a production whose left hand side is N.
  - <expr> => <expr> op <expr> => <expr> op number => …
  - <expr> =>* number op number op number

- *Leftmost derivation*: replace the leftmost non-terminal
- *Rightmost derivation*: replace the rightmost non-terminal

Parsing Tree

- A visual form of derivation – *parsing tree*
  - Example: two parsing trees of 10 – 5 – 5
- Ambiguous grammar
  - A grammar is *ambiguous* if an input string has two different parsing trees.
  - Problem: will create semantic problems! A legal sentence could have more than one meaning, which is not desirable in most cases.

Specifying a language (2)

- Given an “intended language”, there are infinite number of ways to write its grammar
- What is a good grammar?
  - No ambiguity
  - Reflect the structure of the language
  - Useful to the rest of the compiler (e.g., sentences can be parsed efficiently)

- Find grammars without ambiguity
- The ambiguity in arithmetic is solved by *associativity* and *precedence*.
- We can write a grammar for arithmetic expressions that captures associativity and precedence

Ambiguity removal

- Remove ambiguity by capturing associativity
  - Assume we have only *, /, id, number
  - 10/x/5 should be “composed” of “inseparables”: 10, x, 5
  - Then it is grouped as [[10/5]/5] by left associatively
  - So, we have an English definition of *expression*
    - A number or id is an *expression*
    - An *expression* / (number or id) is an *expression*

– CFG is

<expression> -> number |
id |
<expression> / id |
<expression> / number

– So, we have context free grammar (we use <term> to replace <expr> earlier)

```
<term> -> number | id | <term> <multOp> number
            | <term> <multOp> id
<multOp> -> * | /
```

– Improvement: define "inseparable" as <atom>

```
<term> -> <atom> | <term> <multOp> <atom>
<atom> -> number | id
<multOp> -> * | /
```

- Remove ambiguity by capturing precedence
  - Now consider: *,/, in addition to +/-, id, number
  - 10 - 5*3 – 15
    - Firt find the “inseparable” (with respect to +,-) unit [10]-[5*3]-[15] (each unit is a term!)
    - Then group them using associativity: [[[10]-[5*3]]-[15]]
    - Note that each “inseparable” (with respect to +,-) is exactly captured by <term>

```
<expr> -> <term> | <expr> <addOp> <term>
<addOp> -> + | -
```

- More challenges
  - How about allowing parenthesis in addition of all tokens discussed before?
    - The key is () introduces new “inseparable” unit with respect to both +, -, *, /
    - We only need to revise the definition of <atom> to accommodate the “inseparable” unit.

```
<atom> -> id | number | (<expr>)
```

- Context free grammar is also called *Backus-Naur Form*
  - BNF [using ::= instead of ->, and non-terminals are within <…>]

    <number> ::= <digit> | <digit><number>

    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Properties of CFG

- When the productions allow parentheses, and Kleene star, the CFG is called extended CFG (BNF).
- Note
  - Extended CFG is NOT more powerful than CFG
  - CFG without | is as powerful as the CFG
- For any context free language (i.e., there exists a CFG for it), there are *infinite* CFG for it.

Summary

- We have introduced *context free grammar* to specify constructs in a programming language
- *Derivation* and *parse tree* of a string with respect to a grammar
- Remove ambiguity in a grammar
  - Capturing associativity
  - Capturing precedence
- Some properties of *CFG*