CS-1319 - Monsoon 2023 - Assignment 3

Team 9: Saawan Ka Paani

October 28, 2023

1 Introduction

For this assignment, we are building a parser using Bison for nanoC as per the given grammar. We have used Bison version 3.8.2 and we have implemented our program in Linux/Ubuntu 22.04.

- We have the lexer defined in file 9_A3.l which contains the definitions for the language nanoC.
- The 9_A3.y file contains the grammar specifications for nanoC.
- The Makefile is made such that it has a path for the 9_A3.nc file which contains the test code.
- The 9_A3.c is the main file where the parser gets called and returns the appropriate output (either parsing complete or parsing failed).

2 Changes made to Assignment 2:

In the lexer file, that is the .l file instead of printing the tokens (as done before) we are simply returning them. We made a new .y file to specify the tokens for the parses specification for bison. For example: we have specified '(' as OP_PARENTHESES, ']' as CL_SQUARE, return as RET, etc.

3 Phase Structure Grammar of nanoC

Given that the grammar is structured in a hierarchical way with precedents resolved and associativity handled by left or right recursion, the explanation for the expressions, declarations, statements and translation unit is as follows:

Expressions

- **Primary Expression:** Accepts the following:
  - Simple identifier
  - Constant (integer or character constant)
  - String literal
  - Expression enclosed within parentheses: `( expression )`
- **Postfix Expression:** Expressions with postfix operators. Left associativity in C; non-associative here. It can be one of the following:
  - Primary expression
  - Postfix expression followed by an expression enclosed in square brackets: `postfix-expression [ expression ]`

  - Postfix expression followed by a function invocation (optional argument expression list):
    `postfix-expression ( argument-expression-list opt )`
  - Postfix expression followed by pointer and identifier:
    `postfix-expression -> identifier`

- **Argument Expression List:** A list of argument expressions, which can be one of the following:
  - Assignment expression
  - Argument expression list followed by a comma and another assignment expression:
    `argument-expression-list , assignment-expression`

- **Unary Expression:** An expression that can be one of the following:
  - Postfix expression
  - Unary operator followed by another unary expression (Right associativity in C, non-associative here):
    `unary-operator unary-expression`

- **Unary Operator:** One of the following operators (address, de-reference, sign, boolean negation):
  - &
  - *
  - +
  - -
  - !

- **Multiplicative Expression:** Expressions involving left associative operators, which can be one of the following:
  - Unary expression
  - Multiplicative expression multiplied by unary expression:
    `multiplicative-expression * unary-expression`
  - Multiplicative expression divided by unary expression:
    `multiplicative-expression / unary-expression`
  - Multiplicative expression modulo unary expression:
    `multiplicative-expression % unary-expression`

- **Additive Expression:** Expressions involving left associative operators, which can be one of the following:
  - Multiplicative expression
  - Additive expression added to multiplicative expression:
    `additive-expression + multiplicative-expression`
  - Additive expression subtracted by multiplicative expression:
    `additive-expression - multiplicative-expression`

- **Relational Expression:** Expressions involving left associative operators, which can be one of the following:
  - Additive expression
  - Relational expression less than additive expression:
    `relational-expression < additive-expression`
  - Relational expression greater than additive expression:
    `relational-expression > additive-expression`

  - Relational expression less than or equal to additive expression: `relational-expression <= additive-expression`
  - Relational expression greater than or equal to additive expression: `relational-expression >= additive-expression`

- **Equality Expression:** Expressions involving left associative operators, which can be one of the following:
  - Relational expression
  - Equality expression equal to relational expression:
    `equality-expression == relational-expression`
  - Equality expression not equal to relational expression:
    `equality-expression != relational-expression`

- **Logical AND Expression:** Expressions involving left associative operators, which can be one of the following:
  - Equality expression
  - Logical-AND-expression AND equality expression:
    `logical-AND-expression && equality-expression`

- **Logical OR Expression:** Expressions involving left associative operators, which can be one of the following:
  - Logical AND expression
  - Logical OR expression OR logical AND expression:
    `logical-OR-expression || logical-AND-expression`

- **Conditional Expression:** Right associative operator, in the following form:
  - Logical OR expression
  - Logical OR expression followed by a question mark followed by an expression followed by a colon followed by a conditional expression:
    `logical-OR-expression ?  expression :  conditional-expression`

- **Assignment Expression:** Right associative operator, in the following form:
  - Conditional expression
  - Unary expression assigned to assignment expression:
    `unary-expression = assignment-expression`

- **Expression:** A top-level expression that can be one of the following:
  - Assignment expression

Declarations

- **Declaration:** A simple identifier, a 1-D array, or a function declaration of a built-in type, structured as:
  - Type specifier followed by an init-declarator and a semicolon:
    `type-specifier init-declarator ;`
- **Init Declarator:** A declarator or a declarator with an initializer, structured as:
  - Declarator
  - Declarator followed by an equal sign and an initializer:
    `declarator = initializer`
- **Type Specifier:** Built-in types, which can be one of the following:
  - `void`
  - `char`
  - `int`
- **Declarator:** Consists of an optional pointer followed by a direct declarator, structured as:
  - Optional pointer followed by a direct declarator: `pointeropt direct-declarator`
- **Direct Declarator:** A direct declarator can be one of the following:
  - Simple identifier
  - Simple identifier followed by an integer constant enclosed in square brackets, representing a 1-D array or a pointer to it:
    `identifier [ integer-constant ]`
  - Simple identifier followed by a parameter list, representing a function header with parameters of built-in type or pointers to them:
    `identifier ( parameter-list opt )`
- **Pointer:** Denoted by '*', indicating a pointer.
- **Parameter List:** A parameter list consists of one or more parameter declarations. It can be structured as:
  - Single parameter declaration
  - Parameter list followed by a comma and another parameter declaration:
    `parameter-list , parameter-declaration`
- **Parameter Declaration:** A parameter declaration includes a type specifier, an optional pointer, and an identifier. It only allows simple identifiers of built-in type or pointers to them:
  `type-specific pointeropt identifieropt`
- **Initializer:** An initializer can be an assignment expression:
  `assignment-expression`

Statements

- **Statement:** A statement can be one of the following:
  - Compound statement: Consists of multiple statements and/or nested blocks enclosed within curly braces.
  - Expression statement: Represents any expression or a null statement (an empty statement).
  - Selection statement: Represents 'if' statements, possibly including an 'else' branch.
  - Iteration statement: Represents 'for' loops.
  - Jump statement: Represents 'return' statements.
- **Compound Statement:** A compound statement is enclosed within curly braces and contains a list of block items (which is optional). It can be structured as:
  - { `block-item-list opt` }
- **Block Item List:** A block item list contains one or more block items. It can be structured as:
  - Single block item
  - Block item list followed by another block item: `block-item-list block-item`
- **Block Item:** A block item can be one of the following:
  - Declaration: Represents variable or function declarations within the block.
  - Statement: Represents statements within the block.
- **Expression Statement:** An expression statement is an optional expression followed by a semicolon: `expression opt ;`.
- **Selection Statement:** A selection statement can be one of the following:
  - `if` statement followed by an expression enclosed in parentheses and a statement.
  - `if` statement followed by an expression enclosed in parentheses, a statement, and an `else` statement.
- **Iteration Statement:** An iteration statement represents a `for` loop with optional expressions. It contains a for followed by three optional expressions separated by semicolons in parentheses and a statement:
  `for ( expression opt ; expression opt ; expression opt ) statement`
- **Jump Statement:** A jump statement represents a `return` statement with an optional expression:
  `return expression opt`

Translation Unit

- **Translation Unit:** A translation unit represents a single source file containing the ‘main()‘ function and can consist of one or more external declarations. It can be structured as:
  - Single external declaration:
    `external-declaration`
  - Translation unit followed by another external declaration:
    `translation-unit external-declaration`
- **External Declaration:** An external declaration can be one of the following:
  - Declaration: Represents variable or function declarations.
  - Function Definition: Represents a function definition.
- **Function Definition:** A function definition is structured as follows:
  - Type specifier followed by a declarator and a compound statement:
    `type-specifier declarator compound-statement`