THE PROGRAMMING LANGUAGE NINC
PARSER PROJECT

DR. GODFREY C. MUGANDA
NORTH CENTRAL COLLEGE

In this project, you will write a parser for the NINC programming language.

1. NINC Overview

NINC contains three types: integer, boolean, and strings. The use of strings is severely curtailed: they can only be printed. Thus for all practical purposes, NINC supports only two types. NINC supports variables and a full complement of operators on the integer and boolean types, as well as procedures and functions capable of accessing both global variables, local variables, and parameters, although only value parameters are supported. Structures and arrays are not supported.

The syntax of the language is described using the following EBNF form, a variant of context free grammars. The non-terminal symbols are given in uppercase, whereas the terminal symbols (tokens) are given in lower case. The tokens written in bold face are reserved words. The empty string can appear as part of some syntactic constructs: it is denoted by the Greek letter ϵ.

2. MetaLanguage and Meta-symbols

The EBNF grammar used here uses {, }, [,], |, and ::= as meta-symbols. We employ double quotes to indicate that what would otherwise be a meta-symbol is actually an object symbol in the object language. Thus "{" and "}" are brace tokens in NINC, while { and } are EBNF meta-symbols.

3. The EBNF Grammar for NINC

PROGRAM ::= VARDECS PROGBODY
VARDECS ::= { VARDECLIST }
VARDECLIST ::= TYPE id {, id };
TYPE ::= **int** | **bool**
PROGBODY ::= **void main** ( ) BLOCKSTMT
BLOCKSTMT ::= "{" STMTLIST "}"
STMTLIST ::= {TSTMT}
TSTMT ::= STMT; | BLOCKSTMT | IFSTMT
TSTMT ::= | WHILESTMT | SWITCHSTMT
STMT ::= ASSIGNSTMT | INPUTSTMT | OUTPUTSTMT
STMT ::= **break** | **continue** | **exit** | ϵ
IFSTMT ::= **if** (EXPR) TSTMT [**else** TSTMT]

WHILESTMT ::= **while** (EXPR) TSTMT
SWITCHSTMT ::= **switch** (EXPR) SWITCHBODY
SWITCHBODY ::= "{" {CASESTMT}[**default** : STMTLIST] "}"
CASESTMT ::= **case** CASELABEL : STMTLIST
ASSIGNSTMT ::= id = EXPR
INPUTSTMT ::= **cin** >> id { >> id }
OUTPUTSTMT ::= **cout** << OUTPUTEXPR { << OUTPUTEXPR }
OUTPUTEXPTR ::= EXPR | **endl** | string
CASELABEL ::= number | **true** | **false**
EXPR ::= SIMPLEEXPR [RELOP SIMPLEEXPR]
SIMPLEEXPR ::= TERM { ADDOP TERM }
TERM ::= FACTOR {MULTOP FACTOR }
FACTOR ::= -FACTOR | ! FACTOR
FACTOR ::= number | **false** | **true**
FACTOR ::= id | (EXPR)
RELOP ::= == | < | <= | > | >= | !=
MULTOP ::= $*$ | / | % | &&
ADDOP ::= + | - | ||

4. Start and Follow Symbols of NINC Nonterminals

The concept of start and follow symbols is very useful in the construction of parsers. Table 1 gives the start and follow symbols for NINC non-terminal symbols.

Notice that *follow* symbols of a non-terminal X are only useful for parsing when there is a production

$$X ::= \alpha_1\alpha_2\cdots\alpha_k$$

where $\alpha_k \Longrightarrow^* \epsilon$ is possible (some suffix of the right hand side of the production for X can produce the empty string). In this case, the end of the string derived from X becomes uncertain, and a follow symbol of X in the input signals the end of the X-string.

To make parsing decisions, you will use a look-ahead token, together with the concept of *start symbols*, also called *first symbols* or *begin symbols*, and also the concept of *follow symbols*.

5. Organizing Your Code

Make a copy of your lexical analyzer project and rename the project to `ParserProject` or something equivalent. Add a header file `parser.h` that contains the declaration of a `Parser` class, and add a `parser.cpp` file that contains the implementation of the parser.

Rename the main file from the lexical analyzer to `lexdriver.cpp`, and rename the `main()` function `lexmain()`. Finally, add a new `main.cpp` file that will contain the driver code for the parser. Details follow.

Non-terminal Symbol	Start Symbols	Follow Symbols
PROGRAM	VOID, INT, BOOL	EOS
VARDECS	INT, BOOL	VOID
VARDECLIST	INT, BOOL	VOID
TYPE	INT, BOOL	N/A
PROGBODY	VOID	EOS
BLOCKSTMT	LBRACE	N/A
STMT	ID, CIN, COUT, BREAK, CONTINUE, EXIT	SEMICOLON
TSTMT	ID, CIN, COUT, BREAK, CONTINUE, EXIT, LBRACE, IF, WHILE, SWITCH, SEMICOLON	N/A
STMTLIST	ID, CIN, COUT, BREAK, CONTINUE, EXIT, LBRACE, IF, WHILE, SWITCH, SEMICOLON	RBRACE, CASE, DEFAULT
IFSTMT	IF	N/A
WHILESTMT	WHILE	N/A
SWITCHSTMT	SWITCH	N/A
ASSIGNMENT	ID	N/A
INPUTSTMT	CIN	N/A
OUTPUTSTMT	COUT	N/A
CASESTMT	CASE, DEFAULT	CASE, DEFAULT, RBRACE
CASELABEL	NUM, TRUE, FALSE	N/A
OUTPUTEXPR	ENDL, MINUS, STR, NOT, NUM, FALSE, TRUE, ID, LPAREN	N/A
EXPR	MINUS, NOT, NUM, FALSE, TRUE, ID, LPAREN	N/A
SIMPLEEXPR	MINUS, NOT, NUM, FALSE, TRUE, ID, LPAREN	N/A
TERM	MINUS, NOT, NUM, FALSE, TRUE, ID, LPAREN	N/A
FACTOR	MINUS, NOT, NUM, FALSE, TRUE, ID, LPAREN	N/A

Table 1. Start and Follow Symbols of NINC non-terminal symbols

6. The Parser header File

Here are the contents of the `parser.h` file. You should be able to use this as is, or with very minor modifications.

```
#ifndef PARSER_H
#define PARSER_H
#include "Lexer.h"
#include <set>

class Parser
{
public:
    Parser(istream &source, ostream &listing);
    void program();
private:
    // input and listing files
    istream& source;
    ostream& listing;

    Lexer lex;     // lexical analyzer object
    // Convenience functions to access lexical analyzer member functions
    Token getToken() { return lex.getToken();}
    string getLexeme() { return lex.getLexeme();}
    string stringify(Token t) { return lex.stringify(t);}

    Token token; // lookahead

    // Error reporting functions
    void errorMessage(string error);
    void parseError(string errorMessage);
    void parseError(Token expected, Token found);
    void parseError(const set<Token>& expected, Token found);

    // Parsing Functions
    void vardecs();  void vardeclist(); void progbody(); void stmtlist();
    void blockstmt(); void tstmt(); void stmt(); void ifstmt();
    void whilestmt(); void switchstmt(); void switchbody(); void casestmt();
    void assignment(); void inputstmt(); void outputstmt(); void outputexpr();
    void expr(); void simpleexpr(); void term(); void factor();
    void caselabel();

    // verify and skip an expected token
    void accept(Token t);

    // Misc sets of tokens
    set<Token> relops
    {
        Token::LE, Token::LT, Token::GE,
```

```
        Token::GT, Token::NE, Token::EQ
    };
    set<Token> multops
    {
        Token::DIV, Token::MOD, Token::MULT,Token::AND
    };
    set<Token> addops
    {
       Token::PLUS, Token::MINUS, Token::OR
    };
    set<Token> stmtBeginSymbols
    {
       Token::ID, Token::CIN, Token::COUT,
       Token::BREAK, Token::CONTINUE, Token::EXIT
    };

    set<Token> tstmtBeginSymbols
    {
       Token::ID, Token::CIN, Token::COUT,
       Token::BREAK, Token::CONTINUE, Token::EXIT,
       Token::LBRACE, Token::IF, Token::WHILE,
       Token::SWITCH,  Token::SEMICOLON,
    };
    set<Token> outputExprBeginSymbols
    {
        Token::ENDL, Token::MINUS, Token::STR,
        Token::NOT, Token::NUM, Token::FALSE,
        Token::TRUE, Token::ID, Token::LPAREN
    };
    set<Token> exprBeginSymbols
    {
        Token::MINUS, Token::NOT, Token::NUM,
        Token::FALSE, Token::TRUE, Token::ID,
        Token::LPAREN
    };
};
#endif /* PARSER_H */
```

The parser class contains several members:

(1) For each nonterminal, there is a member function function that parses strings derived from that nonterminal. There is a public member function `program()` that corresponds to the start symbol, as well as parsing functions for the other non-terminals.

(2) There is an `accept(Token t)` method for recognizing specific tokens.

(3) There is a lexical analyzer object that is initialized by the `Parser` constructor.

(4) There are several convenience methods for easy access to member functions defined by the lexical analyzer object.

(5) There are source and listing stream objects that are initialized by the `Parser` constructor.

(6) There is a member variable `Token token` that represents the look-ahead token.

(7) There are several sets of tokens that represent significant sets of tokens, such as begin symbols of some of the not terminals.

(8) There are several error reporting functions, as explained below.

   (a) `void errorMessage(string error);`
   Writes an error message to the listing file. This may be useful during debugging: you can place debugging messages in the listing file.

   (b) `void parseError(string errorMessage);`
   Writes an error message to the listing file and exits the program. This is used to report a custom syntax error.

   (c) `void parseError(Token expected, Token found);`
   This is called by `accept(t)` when the look-ahead token does not match the expected token. It adds an error message to the listing file and terminates the program. This is called when the parser is expecting a single specific token, and the look-ahead turns out to be a different token.

   (d) `void parseError(const set<Token>& expected, Token found);`
   Use this to report a syntax error when the parser is expecting one of a certain set of tokens, say, the begin symbols of a statement or an expression, and the look-ahead turns out not to belong to that set.

7. The Parser Implementation File

A skeleton parser implementation file is given below.

```
#include "Parser.h"
#include "TRACER.h"
#include <sstream>

// Returns true if and only if  tokenset contains the token t
bool contains(const set<Token>& tokenset, Token t )
{
   return tokenset.find(t)  != tokenset.end();
}

// Accept is the parser for a single token
void Parser::accept(Token expected)
{
    if (token == expected)
        token = getToken();
```

```
    else
    {
        errorMessage("From Accept" );
        parseError(expected, token);
    }
}

// Error reporting functions
// Reports error, but does not exit.
void Parser::errorMessage(string error)
{
    listing << "^^^^" << error<< endl;
}
void Parser::parseError(string errorMessage)
{
    listing << "^^^^" << errorMessage << endl;
    exit(1);
}

void Parser::parseError(Token expected, Token found)
{
    ostringstream ostr;
    ostr << "I was expecting " <<  stringify(expected) <<  " ";
    ostr << "but I found " << stringify(found) << ".";
    parseError(ostr.str());
}

void Parser::parseError(const set<Token>& expected, Token found)
{
    ostringstream ostr;
    ostr << "I was expecting one of [";
    for (Token t : expected )
    {
        ostr << stringify(t) << "," ;
    }
    ostr << "] but I found " <<  stringify(found) << ".";
    parseError(ostr.str());
}

// Parser constructor initializes the source and
// listing members, the lexical analyzer object,
// and gets the first look-ahead token.
Parser::Parser(istream& source, ostream& listing)
    :source(source), listing(listing), lex(source, listing)
{
    token = getToken();
}
```

```
//PROGRAM ::= VARDECS  PROGBODY
void Parser::program()
{
    vardecs(); progbody();
}

//VARDECS ::=  { VARDECLIST }
void Parser::vardecs()
{

}

//VARDECLIST ::= TYPE id {, id };
//TYPE ::=  int | boolean
void Parser::vardeclist()
{

}

//PROGBODY ::=   void main ( ) BLOCKSTMT
void Parser::progbody()
{

}

// STMTLIST ::=   {TSTMT}
void Parser::stmtlist()
{

}

//BLOCKSTMT ::=  "{" STMTLIST "}"
void Parser::blockstmt()
{

}

//TSTMT ::= STMT; | BLOCKSTMT | WHILESTMT | SWITCHSTMT | IFSTMT
void Parser::tstmt()
{

}


//STMT ::= ASSIGNSTMT
//STMT ::= INPUTSTMT | OUTPUTSTMT
//STMT ::= break | continue | exit | empty
void Parser::stmt()
```

```
{

}

//IFSTMT ::=  if (EXPR)  TSTMT  [  else TSTMT ]
void Parser::ifstmt()
{

}

//WHILESTMT ::=  while (EXPR) TSTMT
void Parser::whilestmt()
{

}

//SWITCHSTMT ::=  switch (EXPR) SWITCHBODY
//SWITCHBODY ::= "{" {CASESTMT}[ default : STMTLIST] "}"
void Parser::switchstmt()
{

}

//CASESTMT ::= case CASELABEL : STMTLIST
void Parser::casestmt()
{

}

//CASELABEL ::= number | true | false
void Parser::caselabel()
{

}

//ASSIGNSTMT ::= id = EXPR
void Parser::assignment()
{

}

//INPUTSTMT ::=  cin  >> id  { >> id }
void Parser::inputstmt()
{

}

//OUTPUTSTMT ::=  cout << OUTPUTEXPR { << OUTPUTEXPR }
```

```
void Parser::outputstmt()
{

}

//OUTPUTEXPTR ::= EXPR  |  endl | string
void Parser::outputexpr()
{

}

//EXPR ::= SIMPLEEXPR [ RELOP  SIMPLEEXPR ]
void Parser::expr()
{

}

//SIMPLEEXPR ::= TERM { ADDOP  TERM }
void Parser::simpleexpr()
{

}

//TERM ::= FACTOR {MULTOP FACTOR }
void Parser::term()
{

}

//FACTOR ::=   -FACTOR  | ! FACTOR
//FACTOR ::= number |  false  |  true
//FACTOR ::= id  | (EXPR)

void Parser::factor()
{

}
```

8. The Parser Driver file

The parser driver file is given below. It processes the command line arguments as before, creates a parser object to process the source file and generate the listing, and calls the parser's `program()` member funtion.

```
#include <cstdlib>
#include "Lexer.h"
#include "Parser.h"
#include <sstream>
#include <fstream>
```

```
#include <iostream>
#include <string>
using namespace std;

void processCmdLine(int, char **, ifstream&, ofstream&);

int main(int argc, char** argv)
{

    ifstream sourceFile;
    ofstream listingFile;

    processCmdLine(argc, argv, sourceFile, listingFile);
    Parser parser(sourceFile, listingFile);

    try
    {
        parser.program();
        cout << "There were no parse errors" << endl;
        return 0;
    }    catch (char const * exception)
    {
        cout << exception << endl;
    }
}


// process the command line args and set the input and listing streams
void processCmdLine(int argc, char **argv,
                    ifstream &sourceFile, ofstream &listingFile)
{
    if (argc != 3)
    {
        cout << "You need to provide a source and listing file"
                " on the command line" << endl;
        cout << argv[0] << " source_file listing_file" << endl;
        exit(1);
    }
    // Open the source file
    sourceFile.open(argv[1]);
    if (!sourceFile)
    {
        cout << "Cannot open the file " << argv[1] << endl;
        exit(2);
    }
    // Open the listing file
    listingFile.open(argv[2]);
    if (!listingFile)
```

```
    {
        cout << "Cannot open the file " << argv[1] << endl;
        exit(3);
    }
}
```

You will run the program on the command line. Once the program terminates, examine the listing file. The listing file should be a copy of the source program up to the first syntax error. If there was a syntax error, the error message will be at the end of a the listing file.

9. Sample Test Files

Sample test files will be posted at the course website. All test files are of syntactically correct CINC programs. You should create additional test files yourself. In particular, make sure you test your parser with some files that contain syntax errors, and make sure your program can detect and report such errors.

10. Due Date

Tuesday of Week 6.