THE PROGRAMMING LANGUAGE NINC INTERNAL REPRESENTATION PROJECT

DR. GODFREY C. MUGANDA
NORTH CENTRAL COLLEGE

In this project, you will implement the third phase of the NINC programming language interpreter. To do this, you will build on top of your parser to perform type-checking and create an internal representation of the NINC program.

1. Type Checking Rules

You will perform type-checking in the parsing functions for the non-terminals. To this end, we embellish the EBNF grammar to show where the various type-checking operations need to be performed. You should incorporate these type rules as comments in your code.

PROGRAM ::= VARDECS PROGBODY
Generate an **exit** *statement at the end of* PROGBODY.

VARDECS ::= { VARDECLIST }

VARDECLIST ::= TYPE id {, id };
TYPE ::= **int** | **bool**
Record the type of each declared variable.
No variable may be declared more than once.

PROGBODY ::= **void main** ( ) BLOCKSTMT
BLOCKSTMT ::= "{" STMTLIST "}"
STMTLIST ::= {TSTMT}
TSTMT ::= STMT; | BLOCKSTMT | IFSTMT
TSTMT ::= | WHILESTMT | SWITCHSTMT
STMT ::= ASSIGNSTMT | INPUTSTMT | OUTPUTSTMT

STMT ::= **break**
break *can only occur in a* WHILESTMT *or a* SWITCHSTMT.

STMT ::= **continue**
continue *can only occur in a* WHILESTMT.

STMT ::= **exit** | ϵ

IFSTMT ::= **if** (EXPR) TSTMT [**else** TSTMT]
Type of EXPR *in* IFSTMT *must be boolean.*

WHILESTMT ::= **while** (EXPR) TSTMT
Type of EXPR *in* WHILESTMT *must be boolean.*

SWITCHSTMT ::= **switch** (EXPR) SWITCHBODY
Type of EXPR *in* SWITCHSTMT *must be boolean or integer.*

SWITCHBODY ::= "{" {CASESTMT}[**default** : STMTLIST] "}"
CASESTMT ::= **case** CASELABEL : STMTLIST
CASELABEL ::= number | **true** | **false**
Value of case label must be unique within a switch statement.
Type of case label must be same as type of EXPR *in containing* SWITCHSTMT.

ASSIGNSTMT ::= id = EXPR
id *must have been declared.*
If id *is not declared, assume a declared type of integer.*
Type of id *must be same as type of* EXPR.

INPUTSTMT ::= **cin** >> id { >> id }
id *must have been declared.*
Type of id *must be integer.*

OUTPUTSTMT ::= **cout** << OUTPUTEXPR { << OUTPUTEXPR }
OUTPUTEXPTR ::= EXPR | **endl** | string

EXPR ::= SIMPLEEXPR [RELOP SIMPLEEXPR]
If RELOP *is present, type of the two* SIMPLEEXPR*s must be the same.*
If RELOP *is present, type of* EXPR *is boolean.*
If RELOP *is not present, type of* EXPR *becomes type of* SIMPLEEXPR.

SIMPLEEXPR ::= TERM { ADDOP TERM }
If there is no ADDOP, *type of* SIMPLEEXPR *is same as type of* TERM.
If ADDOP *is present, type of the two* TERM*s must be the same.*
If ADDOP *is* ||, *type of the two* TERM*s must be boolean, and*
the type of SIMPEEXPR *also becomes boolean.*
If ADDOP *is not* ||, *type of the two* TERM*s must be integer, and*
the type of SIMPEEXPR *also becomes integer.*

TERM ::= FACTOR {MULTOP FACTOR }
If there is no MULTOP, *type of* TERM *is same as type of* FACTOR.
If MULTOP *is present, type of the two* FACTOR*s must be the same.*
If MULTOP *is* &&, *type of the two* FACTOR*s must be boolean, and*
the type of TERM *also becomes boolean.*
If ADDOP *is not* &&, *type of the two* TERM*s must be integer, and*
the type of TERM *also becomes integer.*

FACTOR ::= -FACTOR | ! FACTOR

If the operator is −, *type* FACTOR *on right must be integer,*
and type of FACTOR on left becomes integer.
If the operator is !, *type* FACTOR *on right must be boolean,*
and type of FACTOR on left becomes boolean.

FACTOR ::= number
Type of FACTOR *is integer.*

FACTOR ::= **false** | **true**
Type of FACTOR *is boolean.*

FACTOR ::= id
id *must have been declared.*
If id *is not declared, assume a declared type of integer.*
Type of FACTOR *is type of* id.

FACTOR ::= (EXPR)
Type of FACTOR *is type of* EXPR.

RELOP ::= == | < | <= | > | >= | !=
MULTOP ::= ∗ | / | % | &&
ADDOP ::= + | - | ||

2. Implementation Hints

The following sections offer a sequence of steps you can follow to implement this project.

Add the following members to the `Parser` class in `Parser.h`.

```
// Needed for the Internal Representation
int typeErrorsCount = 0;  // Number of  static semantics errors

// The top of this stack holds the indices of break statements
// inside the switch or loop statement currently being compiled.
// At the end of the switch or loop statement, the destination of
// all these break statements need to be patched.
// Need to push an empty vector at the beginning of a while or switch,
// and pop the stack at the end.
stack<vector<int> *> breakStmtsStack{};

// This stack holds the statement table indices of the (while) loop
// statement currently being compiled. The index at the top of this stack is
// used as the target of  continue statements that are nested inside
// the  (while) loop.
// Need to push a statement index at the beginning of the (while) loop
// and pop the stack, at the end.
stack<int> continueStmtsStack{};
```

3. Type Checking and Static Semantics Errors

Type errors and static semantics errors are reported to the listing file without terminating the parser. To write these errors to the listing file, use the `Parser` member function

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

Note that this function has been modified to increment the `typeErrorsCount` variable.

4. Build the Identifier Table

Do this by modifying the `vardeclist()` function. At the end of the `vardecs()` function, insert a call to the `dump()` method of the identifier table to dump the representation of the identifier table to the listing file. Test with the appropriate test files, and examine the listing file to verify that the identifier table is being correctly built.

Be sure to load the variable in the runtime variable table as you are entering it into the compile-time identifier table.

5. Implement the Code to Build Representations of Expressions

Modify each of the expression parsers (`factor()`, `term()`, `simpleexpr()`, and `expr()` to take reference parameters for

(1) pointer to `EXPR`.
(2) `NINCType`.

Each expression parsing function sets these parameters to the representation of the expression that was parsed, and the type of the expression, respectively.

Modify all parsers that call `expr()` so that they pass the function the appropriate parameters. You need to do this to get the program to compile and build. For the time being, you are not doing anything with values set by those function calls.

If you are anxious to test the code from this step, you can print the string returned by the `to_string()` member function of `EXPR` in the `expr()`, `factor()`, `term()`, and `simple_expr()`, or you can wait to test until you have done the next implementation step.

While compiling expressions, you will need to translate operators that are enumeration constants in `Token` to enumeration values in `UnaryOp` or `BinaryOp` operators. You can add these member functions to your `Parser` class.

```
UnaryOp Parser::tokenToUnaryOperator(Token t)
{
    static map<Token, UnaryOp> transMap
    {
        {Token::MINUS, UnaryOp::MINUS},
        {Token::NOT, UnaryOp::NOT}
    };

    map<Token, UnaryOp>::iterator it = transMap.find(t);
    if (it == transMap.end())
         throw "Exception: this token is not a unary operator.";
    return it->second;
}

BinaryOp Parser::tokenToBinaryOperator(Token t)
{
    static map<Token, BinaryOp> transMap
    {
        {Token::MINUS, BinaryOp::MINUS},
        {Token::PLUS, BinaryOp::PLUS},
        {Token::MULT, BinaryOp::MULT},
        {Token::DIV, BinaryOp::DIV},
        {Token::MOD, BinaryOp::MOD},
        {Token::OR, BinaryOp::OR},
        {Token::AND, BinaryOp::AND},
        {Token::EQ, BinaryOp::EQ},
        {Token::NE, BinaryOp::NE},
        {Token::GT, BinaryOp::GT},
        {Token::GE, BinaryOp::GE},
        {Token::LT, BinaryOp::LT},
        {Token::LE, BinaryOp::LE}
    };

    map<Token, BinaryOp>::iterator it = transMap.find(t);
    if (it == transMap.end())
         throw "Exception: this token is not a binary operator.";
    return it->second;
}
```

6. Implement Assignment

Modify the parsing function `assignment()` to build a representation for assignment statement and load the representation into the statement table. Be sure to enforce relevant type rules.

Modify the code for `program()` so it calls the `dump()` member function on the statement table just before it returns.

You can now test both `assignment()` and expressions with the appropriate test files provided.

7. Implement the Input Statement

Note that this is similar to the assignment statement, but without the expression. Implement and test with the given test file.

8. Implement Output Expression and Output Statement

Add a `EXPR *` reference parameter to `outputexpr()`, and modify the method so it sets its parameter to the representation of the output expression. Implement the output statement. This will of course involve loading string into the string table.

Modify the `program()` function so it dumps the string table. Test.

9. Implement the While Statement

Implement the while statement as according to the strategy discussed in class. Test.

10. Implement the If Statement

Implementation of an If statement is similar to that of a While statement. Test.

11. Implement the Exit Statement

This is straightforward. Make up your own test file.

12. Implement the Switch Statement

The key is to have the parser for the switch statement pass construct a representation for the header of the switch statement, and then pass a pointer to this representation, as well as the type of the switch expression, to `casestmt()`. This will in turn pass the same parameter to `caselabel()`:

```
void casestmt(SwitchSTMT *p, NINCType t);
void caselabel(SwitchSTMT *p, NINCType t);
```

The `caselabel()` function will check for case label duplicates, update the jump table, and check for type consistency between the case label and the type of the switch expression.

Test.

13. Implement the Break Statements

Two things need to happen for *break* statements.

(1) we need to check that they occur only in switch statements or loops.
(2) we need to patch their targets to point to the end of the switch or loop statement that they are contained in.

To do this, use a stack of vectors of integers. The vector at the top of this stack holds the statement indices of all break statements in the loop statement or switch statement currently being compiled.

It turns out that pushing a vector of integers does not work (this was the bug alluded to in class). Instead, we must use a stack of *pointers* to vectors of integers.

Each switch or loop will maintain a local vector of integers, to hold the locations in the statement table of each break statement that occurs inside that switch or loop statement.

```
 vector<int> breakStmts;
```

At the very beginning of the switch or while statement, you must push the address of this vector onto the class-level break statement stack:

```
breakStmtsStack.push(&breakStmts);
```

Each break statement will add its index to the vector at the top of this stack.

Just before exiting a loop or switch, all the breaks statements in that switch statement or loop will be patched to point to the end of that switch or loop. For example at the end of a switch statement:

```
//  patch break statements
int location = RunTime::stmtTable.getLocationCounter();
RunTime::stmtTable.load(new MarkerSTMT("end switch"));

for (int ix : breakStmts)
{
   GotoSTMT *pGoto = dynamic_cast<GotoSTMT *>(RunTime::stmtTable.get(ix));
   pGoto->parchTarget(location);
}
this->breakStmtsStack.pop();
```

Remember to pop the stack of break statements. Test with the appropriate test files.

14. Implement the Continue Statements

Continue statements can occur only in loops. Keep a stack of statement table indices of loops that are in the process of being compiled. When a *continue* statement is encountered, set its target to the integer at the top of the `continueStmtsStack`.

Test with the appropriate test files.

15. Due Date

Due Tuesday of Week 9.