CS4100/5100 COMPILER DESIGN PROJECT CODE GENERATION

SPRING 2022

The fourth step of the compiler project is to modify the Syntax Analyzer 3A to expand it so that it will generate intermediate language 'quad codes' which can be executed using the interpreter function built during Phase 1, the Foundations part of the project. The specifications are as follows:

1) **Syntax Analyzer:** The code for Part 3A of the project is the starting point, and must be modified to parse the basic WHILE and IF/ELSE control structures. Documentation on how to do Code Generation has been provided in three PDF documents specifying the approach and how to handle each of the kinds of statements in the language. Note that the example code in these files provides an excellent road map to implementing both the syntax analysis and the code generation these control features.

2) **Interpreter:** Your program must incorporate the 'Interpreter' function already built in Phase 1. If that is not implemented, working interpreter code will be provided on request, with an associated point deduction on the final project score. It is crucial for students to understand the HOW of the code in any case, whether using their own or the provided code.

3) **Symbol Table, Lexical:** Code generation will use the same Symbol Table from the first assignment for storing the variable identifiers and numeric constants detected by GetNextToken. Variables will not require declaration, but will be automatically initialized to ZERO upon first use.

4) **Quad Table:** The Quad table should store up to 1000 rows of quads, each quad consisting of 4 integer values: an opcode and three operands. The 'AddQuad(opcode, op1, op2, op3)' function should be used to add to the table sequentially. The integer 'NextQuad' function or field, indicating the index where the **next** quad will be added, must be accessible to the parser for branch instruction construction.

5) **Language Subset To Be Implemented:** The parser needs to recognize the subset of the P21 language as specified in the Minimized Language Specification with CFG below.

CS4100/5100 COMPILER DESIGN PROJECT
MINIMIZED LANGUAGE SPECIFICATION
FOR CODE GENERATION - SPRING 2022

The subset of the **PL22** language to be parsed for code generation purpose is described fully below and in the CFG provided.

LEXICAL FEATURES- related to the scanner/lexical analyzer

The Lexical part of the language remains as described in the Lexical Analyzer assignment. The properly-functioning GetNextToken function from that assignment will exactly identify the tokens of PL22.

SYNTAX FEATURES- related to the parser/syntax analyzer

1. **The Program:** Each <program> must have the basic form:

```
<program>       ->      $PROGRAM  <identifier>  $SEMICOLON  <block>
                        $PERIOD
```

See the CFG for further details.

2. **Data types:** There are 2 native literal constant data types: the INTEGER (token code 51), and the STRING (53). All variables are of type INTEGER. Literal STRINGs can only be used for printing, by making them the parameter of the WRITELN statement (see CFG).

3. **Basic statements, only three:**

a) Assignments, as:

```
identifier := <SimpleExpression>
```

where <simple_expression> may contain integer constants, variables, parentheses, unary '+' and '-', binary '+', '-', '*', '/'. Note that precedence is already accounted for in the CFG provided.

b) Outputs, via the reserved function call, PRINTLN, which outputs its parameter, which may be a <string constant> or a <simple expression>, to the console output screen.

c) Inputs, via the reserved function call, READLN, which inputs an integer value to its integer variable from the console keyboard.

4. **Control statements, only two:**

a) **IF** and **IF/ELSE** statements as shown. All conditionals are simple boolean expressions of the form:

<simple expression> <relop> <simple expression>

There are no AND or OR logical operators.

b) **WHILE** pre-test loops with a simple boolean control expression, as with the IF.

PROJECT MINIMIZED CFG

Notation: In the CFG below, the following conventions are used:

1) Anything prefaced by a $ is a terminal token (symbol or reserved word); anything inside of <> pointy brackets is a non-terminal
2) An item enclosed in '[','] ' square braces is optional **unless** a + follows, requiring exactly 1 instance of the item
3) An item enclosed in '{','}' curly braces is repeatable; '*' is '0 or more times', while '+' is '1 or more times'
4) An item enclosed in '(',')' parentheses **requires** exactly one of the optional items listed
5) Vertical bars, '|', are OR connectors; any one of the items they separate may be selected

NOTE: A program, below, may have any identifier for its name, which should not appear as an identifier anywhere else within this program, but no error checking is done for this.

```
<program>       ->      $PROGRAM  <identifier>  $SEMICOLON  <block>
                        $PERIOD
```

NOTE: A block, below, contains a required 'BEGIN', at least one statement, and 'END'. For consistency with the original, unedited CFG, the <block> simply calls <block-body>.

```
<block>         ->      <block-body>

<block-body>    ->      $BEGIN <statement>  {$SCOLN  <statement>}
                                $END
```

Each statement may be of one of the following forms:

```
<statement>->   [
                 <block-body> |

                 <variable>  $ASSIGN  <simple expression> |

                 $IF  <relexpression>  $THEN  <statement>
                        [$ELSE  <statement>] |

                 $WHILE  <relexpression>  $DO  <statement> |


                 $PRINTLN  $LPAR (<simple expression> |
                                        <stringconst>)  $RPAR
                 $READN  $LPAR <variable>  $RPAR
                ]+
```

Note that exactly ONE statement optional item must appear when a <statement> is expected. The

multi-statement <block_body> [a BEGIN-END grouping] is one of these possible options.

```
<variable>      ->      <identifier>

<relexpression> -> <simple expression>  <relop>  <simple expression>

<relop>         ->   $EQ | $LSS | $GTR | $NEQ | $LEQ | $GEQ

<simple expression>->  [<sign>]  <term>  {<addop>  <term>}*

<addop>         ->      $PLUS | $MINUS

<sign>          ->      $PLUS | $MINUS

<term>          ->      <factor> {<mulop>  <factor> }*

<mulop>         ->      $MULT | $DIVIDE

<factor>        ->      <unsigned constant> |
                        <variable> |
                        $LPAR    <simple expression>    $RPAR

<unsigned constant>->   <unsigned number>

<unsigned number>->     $INTTYPE        Token codes 51 only
                           **note: as defined for Lexical

<identifier>    ->      $IDENTIFIER             Token code 50
                           **note: <letter>  {<letter> |<digit> | $ | - }*

<stringconst>   ->      $STRINGTYPE             Token code 53
```

Note that all named elements of the form $SOMETHING are token codes which are defined for this language and returned by the lexical analyzer.