CPSC 4660 PL Scanner

Steven Deutekom
Ricky Bueckert
University of Lethbridge

April 1, 2020

1 Purpose of Software

The software we are building is a PL compiler that will compile files written in the PL language so that they can be interpreted by a virtual stack machine. In this final phase code generation was added to the parser. The running of the parser now produces an intermediate code file with some pseudo instructions and other information. The compiler has also had an assembler added that will take this intermediate code file and convert it to code that can be run on the virtual stack machine. The vertual stack machine is included as an interpreter to run the compiled programs. Most of the work in this phase involved using examples and hints given by the professor to properly add instructions for generating code.

2 Design

This section decribes each class, its function, and how it connects to other classes. More detailed documentation of the specific functions and members of each class is available in the **ReferenceDoc.pdf**. The **New Additions** start in the Parser section.

2.1 Scanner

The Scanner class is the main class for this part of the project. It is responsible for converting a string of characters into Tokens. It collects groups of characters in the text based on the first character of the group and returns a token using the longest valid lexeme that can be made from the group. The groups are Identifiers and keywords, Integers, And various symbols, as defined by the PL language.

Once a group of characters is classified a token is created and it is returned to the process that called `Scanner::getToken()`. If the lexeme is a name it is added to the Symbol table and the token is updated to discover if the lexeme is a keyword or an Identifier.

If the scanner discovers a character that is not in the alphabet an bad character error token is returned. If the lexeme is a number and it is larger than a valid integer then a bad number error token is returned. If the symbol table is full then the scanner returns a full table error. All of these errors do not stop the scanning process or change the way subsequent lexemes are scanned.

2.2 Token

The token class holds information about a lexeme. In our implementation we have given it 3 fields. First a Symbol to identify what kind of token it is. Second there is the lexeme that the token was created from. Finally we have an integer value that stores the value of an integer token or the position of an identifier or keyword in the symbol table. The token class is only responsible for holding data and has public methods to access and mutate its data members. For debugging purposes it also has a method to return a string representation of itself that can be printed.

2.3 Symbol

Symbol is an enum that collects all the possible token types. These are used to classify and identify different tokens. In the same file there is also a table that maps token types to a string identifer so symbols can be printed in a readable way.

2.4 Administration

The Administration class is responsible for connecting all the other classes. It holds information for input and output files. It also is responsible for calling the scanner to get tokens and keeping track of line numbers. The class outputs each intermediate code instruction to a file while it runs. It also keeps track of the number of errors per line and makes sure that only one error is printed per line. It also keeps track of the total number of lines with errors. After 10 lines have encountered errors the parsing is suspended.

2.5 Parser

The paser class is responsible for ensuring the syntax of the input program is correct. There is only one public function in the parser `parse()`. Calling this function begins the recursive descent parsing of the input file. Through the administration class the next token is obtained from the scanner. One lookahead token is maintained at all times. This token is used to decide what grammar rules to pursue while parsing is taking place.

Each function in the parser follows a rule in the grammar of pl. These functions either call other functions representing more rules as well as match tokens. If the lookahead token does not match up with the next expected terminal symbol then an error is thrown. The parser calls the administration class to print the error contents. Then in order to recover from the error the parser looks for the next token that will put the parser back in a stable state so it can continue parsing. In order to do this, the parser must pass a set of tokens that it expects to see to the recursive functions. This way it knows when to stop advancing after an error. Theses sets are explained in more detail in the grammar section.

If the parser is free of errors there is currently no output. Debugging information that displays the parser functions that are called and what tokens are matched. When errors are encountered it is possible to see what rule was being parsed.

The parser now has rules for record types and parameters with procedures. Each of these required a few new rules to be added to the parser. Records need their own parsing to allow variables to be declared as a record type, as well as having the records fields specified. For record fields to be accessed we had to also make some changes to the way variables are accessed. Now if a variable is followed by a dot then it means the nest identifier is a field of that record. With scope and type checking this adds the need to ensure that a field is actually a member of the record being accessed. If it is not an error is thrown. There is more about this aspect of the parser below.

Procedures now have parameters. They are able to take a list of variables of a given type and work a lot like variables in C/C++. If a parameter definition is preceded by "var" it means that the argument is passed by reference, otherwise it will be passed by copy. Parameter lists now need to be kept with the procedure name in some way for scope and type checking. This will ensure that the inside of the procedure can access the parameters and that when the procedure is called we can check the number and type of arguments passed.

The scope and type checking phase adds the necessary code to ensure that variable types and scope are used properly. We implemented this with a data structure called a BlockTable. The BlockTable is a stack of maps that contain TableEntry objects. A Table entry object holds type information and symbol table ids for an identifier. Each level of the stack represents a level of scope. So when entering a new scope we push a new level and when leaving a scope we pop the top level. We have methods that search for an identifier id in the top level only. This will help us to see if a new variable is a redeclaration in the current scope. We have a method that searches the whole block table at every level from top to bottom to find the most

recent defenition of a variable if it does not exist in the current scope. If the identifier is not found it was not declared and cannot be used.

For type checking we ensure that no variable can be assigned a type that does not match it. So a float cannot take a boolean or integer value. In expressions types cannot be mixed and matched. For example trying to add a float and an integer. This type of testing can also be done in if and do statements to make sure that the expression given in the guard evaluates to a boolean. We do not convert numeric types into Booleans like in C/C++. Array access must be done with an integer. Parameter calls must have the appropriate number of arguments and each positional argument must be the correct. We also need to make sure that if an parameter is pass by reference that the given value is not a constant or a literal. However, with all the rule updates this phase was a bit more work that expected and we did not manage to implement this check. One of our error tests has a call to produce this error and we can add the implementation and verify its correctness at the beginning of the next phase.

Type and scope errors do not need to be recovered from in the same way as syntax errors. If these errors are encountered they are identified to the user. Because there is only one error per line if there are multiple mistakes on a line only one will be identified. Also, often if a syntax error is present at the same point in the code it is usually the error printed.

NEW In order to generate the intermediate code for the assembler the parser now does code generation. This involved adding a few peices to the program. Many of the rules in the parser now have emit statements in them. These statements output instructions and addressing information to a file. The addressing information is used by the assembler to properly set up addresses for the interpreter. The psuedo instructions are later replaced by the assembler with opCodes used by the interpreter. However, it is useful to have the intermediate code use the pseudo instructions for debugging purposes. They make the intermediate code more readable. Because the interpreter is a stack machine all of the instructions are output in postfix notation. This works nicely, but it gave us a little trouble when adding the emit instructions. We put some things in the wrong places and our instructions were not output in the correct order. However, having some example programs and their correct assembly output made it possible to find these errors and debug them without too much trouble.

It is important to note that because of the suspension of school we did not get to learn as much about how the code generation works. There are addressing instructions used by the assembler that we do not fully understand. We know that they output information about the order of defined variables and offsets in arrays, but exactly how they do this is not completely clear. We needed to add some extra information to our block table entries to keep some of this addressing information and variable sizes. Without the examples and hints provided this would have been very difficult to finish properly.

The other thing that was not able to be added to the finished compiler was the new features we added in the previous phase. The assemlber and interpreter do not deal with function parameters or record types or float numbers. The parser will still parse them, but no code is generated for them. This does not affect the running of the compiler as long as these types are not included. Our sample programs all stick to the code as it was before these additions.

2.6 Compiler, Assembler, and Interpreter

NEW We were provided with code for an assembler and an interpreter to finalize the compiler. The assembler code was integrated into our compiler. We were given a driver for the assembler that was designed to include our parser, but we elected to add the assembler to our program instead. Now running our compiler will first produce the intermediate code in a file called `pl.asm` and then the assembler will be run on this file. When the assembler is done it's two passes to replace addresses and opCodes it will produce a file called `pl.out` or it will use a name for this file given by the user. This final file is then ready to be run with the interpreter. Building our project will compile the interpreter and produce a binary that can run the programs produced by our compiler. The process is not that different from other interpreted languages that compile to byte code before being run on a virtual machine, such as Java.

2.7 Grammar

In order to properly perform error recovery the parser must know what terminal symbols are expected to put a the parser back in a stable state. The set of expected tokens is called a stopset. These stopsets are built by the first sets that were built from the pl grammar.

In the grammar header we explicitly build all the necessary first sets and store them in a map. We maintain an enum with non-terminal symbols to index this map and access the first sets. This map allows us build stopsets easily by unioning these sets together and passing them around during the parsing. Along with this map there are some helper functions that allow easy union a set and easily check a set for membership. The membership function is not stricly necessary, and is not being used yet, but we hope it will improve the readability when checking to see if a symbol is in a stopset.

2.7.1 BNF

```
# Names for rules are abbreviated #

PROGRAM     := BLOCK .
BLOCK       := begin DEF_PART STMT_PART end
DEF_PART    := { DEF ; }
DEF         := CONST_DEF |
               VAR_DEF |
               PROC_DEF
CONST_DEF   := const CONST_NAME = CONST
VAR_DEF     := TYPE_SYM VPRIME |
        record ID FIELD_LIST end
VPRIME      := VAR_LIST |
               array VAR_LIST '[' CONST ']'
FIELD_LIST  := REC_SEC { ; REC_SEC }
REC_SEC     := TYPE_SYM FIELD_NAME { , FIELD_NAME }
TYPE_SYM    := integer |
               Boolean |
               float
VAR_LIST    := VAR_NAME { , VAR_NAME }
PROC_DEF    := proc PROC_NAME PROC_BLOCK
PROC_BLOCK  := [ ( FORM_PLIST ) ] BLOCK
FORM_PLIST  := PARAM_DEF { ; PARAM_DEF }
PARAM_DEF   := [ var ] TYPE_SYM VAR_LIST
STMT_PART   := { STMT ; }
STMT        := EMPTY_STMT |
               READ_STMT |
               WRITE_STMT |
               ASSGN_STMT |
               PROC_STMT |
               IF_STMT |
               DO_STMT |
EMPTY_STMT  := skip
READ_STMT   := read VACS_LIST
VACS_LIST   := VACS { , VACS }
WRITE_STMT  := write EXPR_LIST
EXPR_LIST   := EXPR { , EXPR }
ASGN_STMT   := VACS_LIST ':=' EXPR_LIST
PROC_STMT   := call PROC_NAME [ ( ACT_PLIST ) ]
```

```
ACT_PLIST   := ACT_PARAM { , ACT_PARAM }
ACT_PARAM   := EXPR_LIST | VACS_LIST
IF_STMT     := if GRCOM_LIST fi
DO_STMT     := do GRCOM_LIST od
GRCOM_LIST  := GRCOM { '[]' GRCOM }
GRCOM       := EXPR -> STMT_PART
EXPR        := PRIM_EXPR { PRIM_OP PRIM_EXPR }
PRIM_OP     := & | '|'
PRIM_EXPR   := SIMP_EXPR [ REL_OP SIMP_EXPR ]
REL_OP      := < | = | >
SIMP_EXPR   := [ - ] TERM { ADD_OP TERM }
ADD_OP      := + | -
TERM        := FACTOR { MULT_OP FACTOR }
MULT_OP     := * | / | \
FACTOR      := NUM | BOOL_SYM |
               VACS |
               ( EXPR ) |
               ~ FACTOR
VACS        := VAR_NAME SELECT
SELECT      := IDX_SEL | FIELD_SEL
IDX_SEL     := '[' EXPR ']'
FIELD_SEL   := . FIELD_NAME
CONST       := NUM CPRIME | BOOL_SYM | CONST_NAME
CPRIME      := . NUM | epsilon
NUM         := DIGIT { DIGIT }
BOOL_SYM    := false | true
NAME        := LETTER { LETTER | DIGIT | _ }   // ID
LETTER      := a - z | A - Z
DIGIT       := 0 - 9
```

2.7.2 First Sets

```
NAME(ID)    [name]
BOOL_SYM    [false, true]
NUM         [num]
CONST       [num, false, true, name]
FIELD_SEL   [.]
IDX_SEL     [ '[' ]
SELECT      [ '[', .]
VACS        [name]
FACTOR      [name, false, true, num, '(', '~']
MULT_OP     ['*', '/', '\']
TERM        [num, false, true, name, '(', '~']
ADD_OP      ['+', '-']
SIMP_EXP    [num, name, false, true, '-', '(', '~']
REL_OP      ['<', '=', '>']
PRIM_EXP    [num, name, false, true, '-', '(', '~']
PRIM_OP     ['&', '|']
EXP         [num, name, false, true, '-', '(', '~']
GRCOM       [num, name, false, true, '-', '(', '~']
GRCOM_LIST  [num, name, false, true, '-', '(', '~']
DO_STMT     [do]
```

```
IF_STMT     [if]
ACT_PARAM   [num, name, true, false, '=', '(', '~']
ACT_PLIST   [num, name, true, false, '=', '(', '~']
PROC_STMT   [call]
VACS_LIST   [name]
ASC_STMT    [name]
EXP_LIST    [num, name, true, false, '=', '(', '~']
WRITE       [write]
READ        [read]
EMPTY       [skip]
STMT        [skip, read, write, call, if, do, name]
STMT_PART   [epsilon, skip, read, write, call, if, do, name]
PARAM_DEF   [var, integer, Boolean, float]
FORM_PLIST  [var, integer, Boolean, float]
PROC_BLOCK  [ '(' , begin]
PROC_DEF    [proc]
VAR_LIST    [name]
TYPE_SYM    [integer, Boolean, float]
REC_SEC     [integer, Boolean, float]
CONST_DEF   [const]
DEF         [const, proc, integer, Boolean, float, record]
VAR_DEF     [integer, Boolean, float, record]
DEF_PART    [epsilon, const, proc, integer, Boolean, float, record]
BLOCK       [begin]
PROGRAM     [begin]
VPRIME      [name, array]
```

2.7.3 Follow Sets

```
# Not fully updated for new rules #

BLOCK       ['.']
DEF_PART    [end, skip, read, write, name, call, if, do]
STMT_PART   [end, fi, od, '[]']
DEF         [';']
CONST_DEF   [';']
VAR_DEF     [';']
PROC_DEF    [';']
CONST_NAME  ['=']
TYPE_SYM    [name, array]
VAR_LIST    [';', ']']
VAR_NAME    [';', '[', ',']
PROC_NAME   [begin]
STMT        [';']
EMPTY_STMT  [';']
READ_STMT   [';']
WRITE_STMT  [';']
ASSGN_STMT  [';']
PROC_STMT   [';']
IF_STMT     [';']
DO_STMT     [';']
VAC_LIST    [';', ':=']
```

```
VACS        [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
EXP_LIST    [';']
EXP         [',', ';', '->', ')', ']']
PROC_NAME   [';']
GRCOM_LIST  [fi, od]
GRCOM       [fi, od, '[]']
PRIM_EXP    [',', ';', '->', ')', ']', '&', '|']
PRIME_OP    ['-', num, false, true, name, '(', '~']
SIMP_EXP    [',', ';', '->', ')', ']', '&', '|', '<', '=', '>']
REL_OP      ['-', num, false, true, name, '(', '~']
TERM        [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-']
ADD_OP      [num, name, false, true, '(', '~']
FACTOR      [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
MULT_OP     [num, name, false, true, '(', '~']
CONST       [';', ']']
VARNAME     [',', ';', '->', ')', '[', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
NUM         [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
BOOL_SYM    [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
CONST_NAME  [',', ';', '->', ')', ']', '&', '|', '<', '=', '>', '+', '-', '*', '/', '\']
```