TOC Assignment 2

Akshat Oke 2020A7PS0284H
Aaditya Rathi 2020A7PS2191H
Sankalp Kulkarni 2020A7PS1097H
Shreyas Dixit 2020A7PS2079H
Chirag Gadia 2020A7PS1721H

December 2022

Contents

1 Grammar for Basic C

1.1 Grammar syntax

The grammar is mostly in BNF, with the following qualifiers that may follow a grouping or a terminal/non-terminal.

+	One or more of target
*	Zero or more of target
?	Zero or one of target

A grouping of symbols is enclosed by brackets '[' and ']' or '(' and ')'. Additionally, '-' represents negation, which shall *not* match the symbol following it.

Although the qualifiers mentioned above can be converted to BNF fairly easily, negation is problematic and lengthy. For example, to represent `-'as' <word>`, that is, all words except 'as', we can write

⟨*Word-not-as*⟩ ::= <notA> <letter>*
| <notS> <letter>*
| 'a' 's' <letter>+

⟨*notA*⟩ ::= [b..z]

⟨*notS*⟩ ::= [a..r] | [t..z]

⟨*letter*⟩ ::= [a..z]

[0..9] represents the range of characters 0,1,2,3,4,5,6,7,8,9, similarly for [A..Z] and [a..z].

1.2 The Grammar

Start symbol is 'MainProgram'

⟨*MainProgram*⟩ ::= ⟨*Declarations*⟩
| ⟨*Program*⟩
| ⟨*Declarations*⟩ ⟨*Program*⟩

⟨*Declarations*⟩ ::= '`int`' ⟨*identifier*⟩ [',' ⟨*identifier*⟩]* ';'

⟨*Program*⟩ ::= ⟨*statement*⟩+

⟨*statement*⟩ ::= (⟨*assignment*⟩
| ⟨*read-statement*⟩
| ⟨*write-statement*⟩

| ⟨*for-statement*⟩
) '`;`'

⟨*assignment*⟩ ::= ⟨*identifier*⟩ '`=`' ⟨*expression*⟩

⟨*expr-or-assign*⟩ ::= ⟨*expression*⟩
| -⟨*binary-expr-strict*⟩ ⟨*expression*⟩ '`=`' ⟨*expression*⟩

⟨*binary-expr-strict*⟩ ::= '`(`' ⟨*binary-expr-strict*⟩ '`)`'
| ⟨*primary-id*⟩ ⟨*op*⟩ ⟨*binary-expr*⟩

⟨*binary-expr*⟩ ::= ⟨*primary-id*⟩ [⟨*op*⟩ ⟨*binary-expr-strict*⟩]?

⟨*primary-id*⟩ ::= ⟨*identifier*⟩ | '`(`' ⟨*binary-expr*⟩ '`)`' | ⟨*integer*⟩

⟨*op*⟩ ::= '`+`' | '`-`' | '`==`' | '`*`' | '`/`' | '`>`'

⟨*expression*⟩ ::= ⟨*equality*⟩

⟨*equality*⟩ ::= ⟨*comparison*⟩ ['`==`' ⟨*comparison*⟩]?

⟨*comparison*⟩ ::= ⟨*term*⟩ ['`>`' ⟨*term*⟩]?

⟨*term*⟩ ::= ⟨*factor*⟩ [('`+`' | '`-`') ⟨*factor*⟩]*

⟨*factor*⟩ ::= ⟨*unary*⟩ [('`*`' | '`/`') ⟨*unary*⟩]*

⟨*unary*⟩ ::= ⟨*primary*⟩

⟨*primary*⟩ ::= ⟨*integer*⟩ | ⟨*identifer*⟩ | '`(`' ⟨*expression*⟩ '`)`'

⟨*read-statement*⟩ ::= '`read`' ⟨*identifier*⟩

⟨*write-statement*⟩ ::= '`write`' (⟨*integer*⟩ | ⟨*identifer*⟩)

⟨*for-statement*⟩ ::= '`for`' '`(`' ⟨*expr-or-assign*⟩? '`;`' ⟨*expression*⟩? '`;`' ⟨*expr-or-assign*⟩?
'`)`' ⟨*block*⟩

⟨*block*⟩ ::= '`{`' ⟨*statement*⟩* '`}`'

⟨*identifier*⟩ ::= - ⟨*keyword*⟩ ⟨*alpha*⟩+

⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*digit*⟩ ::= [0..9]

⟨*alpha*⟩ ::= [a..z]

⟨*keyword*⟩ ::= '`for`' | '`int`' | '`read`' | '`write`'

1.3 Comments on the grammar

It is possible to skip the <equality> rule as it only forwards <expression>, but makes the meaning clearer.

1.3.1 Left Associativity

The grammar is not left-associative as required, for it would require it to be left-recursive as well. Since the parsing algorithm used is recursive descent, the additive and multiplicative productions are flattened out and interpreted left-recursively by the interpreter.

1.3.2 Expression or Assignment

The 'for' clauses 1 and 3 were specified to be either expressions or assignment statements. In most programming languages today, the assignment is part of the expression, with the lowest priority, whose return value is the expression on its RHS. This in grammar, it looks like

⟨*expression*⟩ ::= ⟨*assignment-expression*⟩

⟨*assignment-expression*⟩ ::= ⟨*equality*⟩
| ⟨*identifier*⟩ '`=`' ⟨*assignment-expression*⟩

The problem with this due to the fact that <equality> may also begin with an <identifier> rule, failing the pairwise-disjointness test. This may be then left-factored to allow LL(1) parsers to succeed. (Otherwise a lookaheda of 2 tokens is required)

Since expressions cannot have assignments, a new rule <expr-or-assign> had to be made to rule out ambiguity in choosing which branch to take. If the LHS <expression> was <identifier> as required by the specification, it would not be possible to parse it as an expression (once '=' was matched)

without backtracking. Thus, it an parses expression, looks for a '=', and if present, attempts to negate <binary-expr-strict> (not match binary expressions involving identifiers) and then proceed to parse the RHS. (This rule prevents the grammar from generating expressions for assignment targets)

Just replacing the Kleene star by '+' in <comparison> and <equality> would have resulted in Python-style expressions such as `a>b>c`!

2 About the implementation

The parser is an LL(1) recursive descent parser.

2.1 Types

Since the document mentions nothing about types, this implementation assumes 'Basic C' to be strongly typed, with two types.

There are two types of expressions: number and boolean. Only a boolean expression can be used in the condition clause of the for statement.
Printing a boolean variable will print 'true' or 'false' correspondingly. Booleans cannot be operated upon by arithmetic operations.

2.2 Uninitialized variables

The <Declarations> rule declares variables that can be used in the program. Variables have been implemented with a hashmap. Referencing a variable before assigning a value to it causes an 'Uninitialized variable' runtime error.

It was possible to implicitly give variables an initial value of 0, but this may lead to unclear code.

2.3 Ghost nodes

The for statement grammar may not include all three clauses, for example `for(;a<4;)`. In such cases the parser inserts a 'ghost node' which evaluates to 'true' but does not get printed by the AST printer code.

This ghost node is also used in <expr-or-assign> in order to provide easy access to the lvalue identifier lexeme.

3 Remarks

Before the correction in the for statement syntax, all three clauses were expressions and had no side effects. This means they could be executed any number of times and in any order during the loop execution. Although it seemed that we could get away easily by not using the AST for interpreting, the additional "rewind" functionality required in the scanner was better left alone.