BILKENT UNIVERSITY

DEPARTMENT OF COMPUTER ENGINEERING

CS315 PROGRAMMING LANGUAGES

PROJECT REPORT 2: *ATALAR* PROGRAMMING LANGUAGE **(APL)**

BERAT BİÇER - SECTION 3, 21503050

19.11.2017

ANKARA

BNF DESCRIPTION OF ATALAR PROGRAMMING LANGUAGE

<program> **->** START <predicates> STOP GO <statements> FINISH | GO <statements> FINISH

<predicates> **->** <predicate> <predicates> | <predicate>

<predicate> **->** <name> LB <parameter_list> RB <parameter_body>

<name> **->** ALPHANUMERIC

<parameter_list> **->** <parameter> | <parameter> COMMA <parameter_list>

<parameter> **->** <var> | TRUTH | <const> | <list_access>

<var> **->** CAPITAL_ ALPHANUMERIC

<const> **->** <number> | ALPHANUMERIC

<number> **->** INTEGER | DOUBLE

<list_access> **->** <var> LSB INTEGER RSB

<predicate_body> **->** DEF_START <statements> RETURN returndis DEF_STOP

<statements> **->** <statement> <statements> | <statement>

<statement> **->** <comment_body> | <yebunu> |<okuyazbunu> | <assignment>

<comment_body> **->** COMMENT_START <comments> COMMENT _END

<comments> **->** <comment> <comments> |

<comment> **->** ALPHANUMERIC | capital_alphanumeric

<yebunu> **->** <if> | <loop>

<if> **->** IF LB <prop> RB TBEGIN <statements> TEND ELSE TBEGIN <statements> TEND

<prop> **->** <prop> <esitlebakam> <operand> | <operand>

<esitlebakam> **->** IMPLY | EQUIVALENCE

<poperand> **->** <poperand> <andor> <toterand> | <toterand>

<toterand> **->** <toterand> SDIF <kokerand> | <kokerand>

<kokerand> **->** EXCLAMATION <kokerand> | <term>

<term> **->** LB <prop> RB | <parameter>

<andor> **->** SAND | SOR

<loop> **->** WHILE LB <prop> RB TBEGIN <statements> TEND

<okuyazbunu> **->** <read> | <write>

<read> **->** READ LB <var> RB

<write> **->** WRITE LB <yazbunu> RB

<yazbunu> **->** <prop> | <predicate_inst>

<predicate_inst> **->** <name> LB <parameter_list> RB

<assignment> **->** <var> SEQU <kullanbunu>

<kullanbunu> **->** <prop> | <predicate_inst> | <list>

<list> **->** LCB <values> RCB

<values> **-> <**tutbunu> COMMA <values> | <tutbunu>

<tutbunu> **->** TRUTH | CONST

<returndis> **->** <prop> | <predicate_inst>

DISCUSSION OF BNF ELEMENTS

Terminals: Terminals in ATL are as follows:

- **Integer**: Integers in ATL are represented as positive or negative integers. For example, 43, -65 and +5412 are both interpreted as integers and Lex returns the token INTEGER if it encounters one.
- **Double**: Doubles are represented with ".", for example 12.7, +15,864 and -167,19 are interpreted as integers and Lex returns the token DOUBLE if it encounters one.
- **Truth**: Boolean values in ALP: false or true. Lex returns the token TRUTH if it encounters one.
- **Start**: Reserved word "start" in APL, Lex returns the token START if it encounters one. The token START points the starting point of predicate definitions in an APL program.
- **Stop**: Reserved word "stop" in APL, Lex returns the token STOP if it encounters one. The token STOP points the end point of predicate definitions in an APL program.
- **If**: Reserved word "if" in APL, Lex returns the token IF if it encounters one. The token IF points the starting point of an if statement in an APL program.
- **Else**: Reserved word "else" in APL, Lex returns the token ELSE if it encounters one. The token ELSE points the starting point of an else statement in an APL program which must follow an if statement.
- **Begin**: Reserved word "begin" in APL, Lex returns the token BEGIN if it encounters one. The token BEGIN points to the starting point of a scope as if it is an opening curly bracket. For example, an if statement's statements are placed between a BEGIN and END token.
- **End**: Reserved word "end" in APL, Lex returns the token END if it encounters one. The token END points to the end point of a scope as if it is a closing curly bracket. For example, an if statement's statements are placed between a BEGIN and END token.
- **While**: Reserved word "while" in APL, Lex returns the token WHILE if it encounters one. The token WHILE is the keyword used for loops.

- **Return**: Reserved word “return” in APL, Lex returns the token RETURN if it encounters one. The token RETURN states which expression is returned from a predicate.
- **Read**: Reserved word “read” in APL, Lex returns the token READ if it encounters one. The token READ represents the built-in function for getting an input from the user.
- **Write**: Reserved word “write” in APL, Lex returns the token WRITE if it encounters one. The token WRITE represents the built-in function for printing out an expression to the console.
- **Go**: Reserved word “go” in APL, Lex returns the token GO if it encounters one. The token GO points the starting point of program statements in an APL program.
- **Finish**: Reserved word “finish” in APL, Lex returns the token FINISH if it encounters one. The token FINISH points the end point of program statements in an APL program.
- **Alphanumeric**: An alphanumeric in APL starts with a lower-case letter. It can include any alphabetical character and any combination of digits after the first character. For example, “ab5S3123kk9193” is a valid alphanumeric whereas “A213l123kö”, “ı1o30123” and “829kdnslamedu92ıo1jk” are not.
- **Capital Alphanumeric**: An capital alphanumeric in APL starts with an upper-case letter. It can include any alphabetical character and any combination of digits after the first character. For example, “Ab5S3123kk9193” is a valid alphanumeric whereas “a213l123kö”, “ı1o30123” and “829kdnslamedu92ıo1jk” are not.
- **“=”**: The *assignment* operator in APL, Lex returns the token SEQU if it encounters one.
- **“&”**: Logical *and* operator in APL, Lex returns the token SAND if it encounters one.
- **“|”**: Logical *or* operator in APL, Lex returns the token SOR if it encounters one.
- **“\”**: Logical *difference* operator in APL, Lex returns the token SDIF if it encounters one.
- **“->”**: Logical *implies* operator in APL, Lex returns the token IMPLY if it encounters one.
- **“<->”**: Logical *equivalent* operator in APL, Lex returns the token EQUIVALENCE if it encounters one.
- **“#%”**: Symbol points the start of a comment, Lex returns the token COMMENT_START if it encounters one.
- **“%#”**: Symbol points the end of a comment, Lex returns the token COMMENT_END if it encounters one.
- **“(“**: Opening parenthesis in APL, Lex returns the token LB if it encounters one.
- **“)”**: Closing parenthesis in APL, Lex returns the token RB if it encounters one.
- **“[“**: Opening square bracket in APL, Lex returns the token LSB if it encounters one.
- **“]”**: Closing square bracket in APL, Lex returns the token RSB if it encounters one.
- **“{“**: Opening curly bracket in APL, Lex returns the token LCB if it encounters one.
- **“}”**: Closing curly bracket in APL, Lex returns the token RCB if it encounters one.
- **“!”**: Logical *negate* operator in APL, Lex returns the token EXCLAMATION if it encounters one.
- **“,”**: Listing symbol in APL, Lex returns the token COMMA if it encounters one. Listing symbol separates the elements of array elements and predicate parameters.
- **“::”**: Symbol representing the starting point of a predicate definition, Lex returns the token DEF_START if it encounters one.
- **“;;”**: Symbol representing the ending point of a predicate definition, Lex returns the token DEF_STOP if it encounters one.

Nonterminals: Nonterminals in APL are as follows:

- **<program>:** Entry point for grammar rules, this nonterminal represents any program APL accepts.
- **<predicates>:** This nonterminal includes predicate definitions in APL. They are written at the beginning of any APL program, and must be placed between START and STOP terminals before program statements, symbolizing the boundary for predicate definitions.
- **<predicate>:** Represent a single predicate definition, and used as the base case of recursion for predicate definitions.
- **<name>:** Names in APL, names of constants and predicates, start with a lower-case letter followed by any alphanumeric. There's no limit in length for names in APL and each character is significant.
- **<parameter_list>:** Represents parameters for a predicates. This nonterminal is used when defining the predicate and invoking a predicate instance.
- **<parameter>:** Parameters in APL represents elements that can be used for logical operations and assignment statements. A parameter can be either of the following: a variable, a truth value as either false or true, a constant written in name format, or a list access which is to get an element of a list structure.
- **<var>:** Variable names in APL start with a capital letter followed by any alphanumeric.
- **<const>:** Constants are either numbers or strings in name format.
- <number>: APL supports integers and double values as numbers.
- **<list_access>:** List access means receiving an element of a list structure with at a given index. This access is trivial, the name of the list structure followed by an index written between an opening square bracket and a closing square bracket, respectively.
- **<predicate_body>:** Predicate body represents any legal language statement that can be written inside a predicate between a DEF_START and DEF_STOP terminal characters. Any APL statement can be written inside a predicate, as functions/methods do in imperative languages. For returning values, only a logical proposition or a predicate instance can be returned from a predicate.
- **<statements>:** Represents any statement that is accepted by APL as valid language construct.
- **<statement>:** Represents a single, legal statement of APL. A statement can be either of the following: a matched if statement, an assignment, a while loop, input/output built-in functions and comments. For reliability and a conflict-free parser, unmatched if statements which do not have an else counterpart are not allowed in APL. This design choice, however, costs in writability; the programmer needs to write more code in order to have an even the simplest if statement. Any language statement can be written inside an if statement. Also, comments are written between the start symbol and end symbol that can spawn multiple lines since APL ignores white spaces. Also, since the order of execution is top-down, APL does not introduce a precedence between statement types of the language since the program text defines which statement should be executed first.
- **<comment_body>:** Represents a single comment in APL. Comments are either in name or variable format in APL, and written between "#%" and "%#".
- **<ye_bunu>:** A nonterminal for grouping if statements and loops.
- <if>: If statements follow the structure of this nonterminal as follows: the "if" keyword, a logical proposition that is used as the condition check, statements between the keywords

"begin" and "end", "else" keyword for providing the matched if structure, and statements between keywords "begin" and "end".

- **<prop>:** Logical propositions in APL are described with this nonterminal. The precedence among operators is as follows, from lowest to the highest: imply-equivalence, and-or, differentiation, negation, parenthesis support and parameters which can be a variable, a Boolean value, a constant or a list access.
- **<loop>:** Looping in APL is provided with a while statement which uses a proposition for loop control. For loops, do-while loops etc. are not allowed in APL. Any language statement can be written inside a loop.
- **<okuyazbunu>:** A nonterminal for grouping input/output statements in APL.
- **<read>:** Built-in function for reading input via console. Accepts a variable as its parameter.
- **<write>:** Built-in function for writing input on the console. Accepts a proposition or a predicate instance as its parameter.
- **<predicate_inst>:** Instantiation of predicates in APL uses a similar pattern to Prolog: name of the predicate in name format, and a parameter list written between parenthesis where each value is separated by a comma.
- **<assignment>:** Assignments in APL uses two operands: one is a variable that is written on the left-hand side and a value. This value can be a proposition, a predicate instance that returns a value, or a list structure.
- **<list>:** Lists are array-like structures in APL. The origin for lists are inspired from Python, which uses heterogeneous, dynamic lists. In APL, a list can contain a truth value or a constant. Empty lists are not allowed in APL in declaration. Also, each list element is separated by a comma.
- **<values>:** Represents elements of lists.
- **<returndis>:** This nonterminal is the group of elements that a predicate can return, which are either a proposition or a predicate instance.

Important Things to Know About APL

Writing an APL program

An APL program can be in one of two types: 1, a program without any custom predicates or 2, a program that has its own predicates and language statements. The first type of programs is used when no custom predicates are necessary. That is, if the user wants to perform some basic logical operations, this type of APL program is optional. Type 1 APL programs are differentiated from Type 2 with its boundary anchors. They start with "go" and finishes with "finish" reserved words. Type 2 APL programs start with "start" and finishes with "stop" reserved words. However, they switch to the structure of Type 1 at the end of predicate definitions section.

Inside if statements, while loops, predicates and body of the program; any legal language statement is allowed. Thus, we can assume these scopes are identical to the main program in practice.

APL does not introduce any symbol for finishing a statement as C# does with ";", semi column. Instead, it's more intuitive. When a legal language statement is read, the program assumes it's reading another language statement &| construct. However, the programmers are advised to pay a close attention on

their indentation and layout of the constructs. This way, the lack of an end-of-the-statement symbol will not be an issue.

Most APL structures and statements are inspired from existing PL, mostly from C# and Python. Thus, we believe most design decisions will be intuitive for the programmers.

Language Statements

APL includes 4 major type of statements: if statements, loops, input/outputs and other operations.

If-Else Statements

Most PL include if statements in the following 2 ways: unmatched or matched. Unmatched if statements correspond to if statements that have no else counterparts. Thus, if consecutive if statements are written inside an unmatched if statements which has else counterparts, it is problematic to match which else statement belongs to which if statement. Therefore, to avoid such complications and provide a higher reliability, APL does not support unmatched if statements. However, this decision causes less writability. For example, assume we want to return a truth value from a predicate if the input from the user is equal to the parameter "*Abs*" of our predicate. Then, if unmatched if statements were supported, we'd be writing the following APL code:

```
read(Vary)
if (equals (Vary, Abs)) begin
      return true
end
```

However, since APL does not support this structure, the actual code should look like this:

```
read(Vary)
if (equals (Vary, Abs)) begin
      return true
end else begin
      return false
end
```

If statements in APL accepts a logical proposition for condition checking. Any other value or statement will cause an error in the program.

Loops

Only loop structure supported in APL is while loops. For loops are not supported in APL since they inherit imperative aspects that are not compatible with propositional calculus. For example, arithmetic operations are not supported in APL, thus indexing with for loops cannot be done via integers. It is

possible to use Boolean values in for loops, but while loops are easier to handle. While loops are preferred over other loop types such as do-while in APL for writability and simplicity.

While loops too, uses propositions as conditional check in APL. Any other statement will result in an error.

Input/Outputs

For inputs, “read” built-in function is used in APL. It accepts a variable as parameter, reads the user input and assigns its value into the parameter. While assignment statements are limited at assigning values into variables, user input into the parameter is treated as a constant and thus has no restrictions. However, this may lead to some unexpected behavior. For example, consider the following APL program:

```
read(Vary)

read(String)

Array = {true, false}

if(Array[Vary]) begin

      write (String & true)

      write (String | false)

end

else begin

      print (equals (String, bat))

end
```

If the input for variable “Vary” is not an integer, the program will try to access a list with a parameter that does not correspond to an index. If the input “String” has non-Latin characters such as Turkish “ş, ü, ğ”, the behavior is undefined since lex and yacc operates on ASCII. If the input “String” is not a propositional value, true or false, a call to “write (String & true)” will result in an undefined behavior. Parameters of write function are restricted to propositions and predicate instance to prevent the problem above, printing directly a string or value that is not applicable (such as a list or empty variable) or is not an ASCII string.

Other Elements

Assignments

Assignments in APL accepts a variable on the LHS and one of the following on the RHS: a proposition, a predicate instance or a list. This means practically every programming unit in APL can be used in assignment statements.

Lists

Lists are inspired from Python's lists: they are heterogeneous and flexible; they can grow and shrink. However, APL does not include a built-in function for inserting an element into the list or removing one from it. Instead, these actions are performed via assignments. For example, consider the following APL program:

```
List = {1,2,3,4,5}

Read (Vary)

If (equals (Vary, List [1]} begin

      List = {1,2,3,4,5,6,7,8}

end else begin

      List = {1,2,3}

End
```

Inside the if statement, the previously defined list instance "List" is grown with new variables and inside the else part, it shrinks. This structure is aimed to provide an easy way to manage dynamic arrays in the program.

Comments

Comments in APL are placed between the start-comment symbol ("#%") and finish-comment symbol ("%#"). The comment body, or comment text can be either in variable or name format. That is, comments cannot contain any combination of characters. Consider the following APL program:

```
go …

#% This is a comment %#

#% şqwlekgkltm21m 1122jh 3k21epi1 2e123i ş3i12.çe.!2ç 123i %#

… finish
```

Such a program will return an error since the tools used for writing the syntax analyzer of APL, lex and yacc, uses ASCII as character set. So, some UNICODE characters are not recognized properly. Also, it is discouraged to use some characters such as comment anchors inside the comment text. Thus, by restricting the body of a comment, we increase the reliability of APL as well as readability.

The reason we chose to use the comment anchors as before is the following: Many languages uses a simpler syntax for comments, usually a "//" or "#" symbol that spawns the rest of the line as a comment. In such cases, it's sometimes inevitable to use the same symbol in the comment body. Also, it might be problematic to set the rest of the line as comment in some PL. By selecting a non-natural, almost-never-used-in-daily-programming symbol pair, we ensure that the user will not accidentally terminate the comment prematurely. Also, using two anchor points for comments allows us to identify comments more accurately.

Propositions

Propositions in APL are the main focus of any program. The following major operations are supported in APL, according to their precedence:

- Parenthesis support
- Negation
- Logical differentiating
- Logical and & or
- Logical implication & equivalence

Propositions can be combined in various ways, for example:

Var = Abc & false -> true & !Abc | false <-> true

Var = false & !true <-> false & false -> !true

Var = false | false -> !true | !false & false <-> true

Parameters are supported as operands in propositions.

Names

Names in APL follow these rules:

- Lower-case alphanumerics are referred as names. They are used for constants and predicates.
- Upper-case alphanumerics are referred as variables. They are used for variable names.

Consider the following names for a variable:

Variable

Berat

Patates

askndmöwJKDnl

BöErsk

12uı3mMSöm

Among the previous ones, names 1 to 3 are valid because they follow the syntax defined: An upper-case letter followed by any string of alphanumeric. On the other hand, the name number 4, 5 & 6 contains non-ASCII characters whereas name 6 starts with a number instead of an upper-case letter. Thus, they are not valid names suitable for variables.

Now consider the following names for a predicate:

Variable

askndmöwJKDnl

12uı3mMSöm

Predicate

preDicaT2e

predicTa2AtsttS

Names number 1,2,3 & 4 are not suitable for a predicate name. However, names number 5 & 6 are suitable since they start with a lower-case letter and followed by an ASCII alphanumeric.