CS 315 - PROGRAMMING LANGUAGES
FALL 2017

Project Report Part 2
ZUCC

Group 33
Instructor: Halil Altay Güvenir

Group Members

H.Buğra Aydın(21501555) **Section 1**

Ata Gün Öğün(21501187) **Section 1**

1)BNF Description of ZUCC

<program> ::= MAIN LCURLY <stmts> RCURLY

<stmts> ::= <stmt> | <stmt> TERMINATOR <stmts>

<stmt> ::= <if_stmt> | <assignment_stmt> | <init_stmt> | <while_stmt> | <output_stmt> |<predicate_declaration>

<init_stmt> ::= INIT <var_assignment_stmt> | INIT C_IDENTIFIER ASSIGNMENT <boolean_expr> | INIT A_IDENTIFIER ASSIGNMENT LP <boolean_exprs> RP

<assignment_stmt> ::= <var_assignment_stmt> | <arrelement_assignment_stmt>

<arrelement_assignment_stmt> ::= <arrelement> ASSIGNMENT <boolean_expr>

<var_assignment_stmt> ::= V_IDENTIFIER ASSIGNMENT <boolean_expr>

<if_stmt> ::= IF LP <boolean_expr> RP THEN <stmts> END
| IF LP <boolean_expr> RP THEN <stmts> ELSE <stmts> END

<while_stmt> ::= WHILE LP <boolean_expr> RP DO <stmts> END

<input_stmt> ::= INPUT LP RP

<output_stmt> ::= OUTPUT LP <boolean_expr> RP

<predicate_declaration> ::= DEFINE P_IDENTIFIER LP <ids> RP BGN <stmts> RETURN
<boolean_expr> END | DEFINE P_IDENTIFIER LP
<ids> RP BGN RETURN <boolean_expr> END

<predicate_call> ::= P_IDENTIFIER LP <boolean_exprs> RP | P_IDENTIFIER LP RP | <input_stmt>

<boolean_expr> ::= <boolean_expr> <connective> <boolean_exp2> | <boolean_exp2>

<boolean_exp2> ::= NOT <boolean_exp1> | <boolean_exp1>

<boolean_exp1> ::= LP <boolean_expr> RP | <boolean_exp0>

<boolean_exp0> ::= <predicate_call> | <id>

<boolean_exprs> ::= <boolean_exprs> COMMA <boolean_expr> | <boolean_expr>

<ids> ::= <ids> COMMA <id> | <id>

<id> ::= V_IDENTIFIER | <arrelement> | C_IDENTIFIER | BOOL

<arrelement> ::= A_IDENTIFIER INDEX

<connective> ::= AND | OR | EQCHECK

2) Structure of ZUCC

2.1) General Description of the structure of ZUCC

ZUCC's main aim is to provide easy to read and understandable codes to the users. So when deciding on our structure, our purpose was to obtain these criterias. When deciding on the variable names, we got inspired from the Perl language and we restricted the usage of names. For example, an array name has to start with 'a_', a constant has to start with 'c_' and a predicate name has to start with 'p_'. Programs in ZUCC starts with the program terminal that consist the 'main{' opening, statements and a closing with '}'. Most of our reserved words are same as the other programming languages, in order to provide a large scale of understandability to our users. We decided to separate every single statement with the terminator token '~'. This provides ZUCC readability. ZUCC supports array and predicate declarations and these features are really important since they allow coders to do many operations from large to low scale. Predicate declarations are really important since they allow to implement many different functionalities. For example, our language doesn't have an implies sign, but it is possible to define a predicate that accepts two arguments and returns a boolean expression corresponding to their implies equivalent using or/and/not logical operations in the predicate body. This provides our users with orthogonality, usability and extendability. We also provide our users input and output statements, allowing them to write dynamic programs in the future and output their programs' results.

2.2) Description of Important Non-terminals

<program>: This construct is used for describing what a program is. a program needs to start with a MAIN token. Between a LCURLY token and RCURLY token, the program statements are defined. The whole execution starts from this non terminal and ends with a close curly bracket.

Example:

```
main{
init c_tru = true~
init c_fal = false~
init zucc = ture~
output(zucc)~
zucc = false~
output(zucc)~
output(zucc | c_fal)~
output(zucc | false & c_tru)~
output(!zucc)
}
```

<stmts>: This construct defines “statements”. It can be described recursively as a single statement or a single statement followed by a terminator symbol followed by more statements. This construct intends to describe code blocks which are individual statements separated by the terminator symbol. Having this construct is especially helpful for describing selection and looping statements. An important trait to of ZUCC: The terminator symbol should be used after a statement if and only if that statement is followed by another statement.

Example:

```
if(!p_trueCheck(false)) then
output((p_trueCheck(((!((!false)))))))~
zucc = c_fal
else if (true) then
output(false)
else
output(true)
end~
init zucc = true
```

<stmt>: Defines what a single “statement” is. It’s aim is to contain all the different statement constructs under a single name. Note that a statement does not have a terminator at the end; a terminator symbol should be used after a statement if and only if that statement is followed by another statement. It can span if,assignment, init, while, output statements and function declarations.

Example:

```
define p_trueCheck(a)
begin
return a==true
end
```

<init_stmt>: Defines what an “initialization statement” is. Used for the initialization of arrays, constants and variables. Init statement was added in order to differentiate it from assignment statement since constants supposed to be only initialized.

Example:

```
init a_example_array = (true,p_isTrue(false),(
true|false))
```

<assignment_stmt>: Defines what an “assignment statement” is. Used for assigning a value to a variable or assigning a value to an array element.

Example:

```
zucc = false
```

<arrelement_assignment_stmt>: Defines a statement to let the user to change the content of an array index.

Example:

```
a_example_array[0] = false
```

<var_assignment_stmt>: Used for assigning a value to a variable.

Example:

```
zucc = false
```

<if_stmnt>: Defines “if/then” and “if/then/else” statements. This construct is used for selections. The selection is based off of the boolean expression that is written between the parentheses after the IF reserved word. After the THEN reserved word a set of statements are written, followed by the END or ELSE reserved word. If it the set of statements were followed by ELSE, another set of statements are written, followed by the END reserved world.

Example:

```
if(zucc) then
zucc = (!(zucc&true)==!c_fal)
```

```
end
```

<while_stmt>: Defines “while/do” statements. The while loop is based only on propositional truth values.

Example:

```
while(zucc) do
output(p_trueCheck(zucc))~
zucc=false
end
```

<input_stmt>: Defines the “input” statement that is used to get an input value from the user during execution. Input can be either true or false.

Example:

```
mahmut = input()
```

<output_stmt>: Defines the “output” statement that is used to output the boolean value of an identifier.

Example:

```
output(p_trueCheck(zucc))
```

<predicate_declaration>: Defines predicate declarations. Predicates can get one or more arguments and return a single value.

Example:

```
define p_or(k,kk)
begin
if(false) begin
output(false)
end~
kkk=k | kk
return kkk
end
```

<predicate_call>: Defines predicate calls. Predicate calls correspond to boolean values.

Example:

```
if(!p_trueCheck(false)) then
output((p_trueCheck(((!((!false)))))))~
zucc = c_fal
else if (true) then
output(false)
else
output(true)
end
```

<boolean_expr>: Defines all boolean expressions. Complex boolean expressions are also defined such as compound formulas containing multiple function calls, connectives and parentheses. Used in selection, loop, initialization, assignment, output, function declaration and function call statements. The order of precedence is as follows: function calls>parentheses>negation>logical operations.

Example:

```
zucc = true
```

<boolean_exp2>: This connective is used for setting the precedence of negation higher than other logical operations.

Example:

```
zucc = !true
```

<boolean_exp1>: This connective is used for setting the precedence of parentheses higher than negation.

Example:

```
zucc = !((true & false) & false)
```

<boolean_exp0>: This connective is used for setting the precedence of function calls higher than parentheses. It also defines all identities as boolean expressions.

Example:

```
zucc = (true&false)&p_zzzz(bugra)
```

<boolean_exprs>: Defines the comma seperrated boolean expressions to represent the inner elements of arays.

Example:

```
init a_exaple_array = (true,false,c_aConstant,p_function(true),aVariable)
```

<ids>: Defines “comma separated list of identities”. Used in function calls and declarations to describe arguments of functions.

Example:

```
zucc = (true&false)&p_zzzz(bugra,c_ata)
```

<id>: This non-terminal acts as a wrapper for all identifiers, namely variable identifier, constant identifier, array element identifier and function identifier. Note that all these identifiers correspond to a boolean value.

Example:

```
zucc = (true&false)&p_zzzz(bugra,c_ata)
```

<arrayelement>: This non-terminal describes how an array element is referenced.

Example:

```
a_arrayName[5]= !a_arrayName[2]
```

<connective>: This non-terminal acts as a wrapper for all connectives such as "EQCHECK", "AND" and "OR" tokens.

Example:

```
zucc = (true==false)&p_zzzz(bugra,c_ata)
```

3)Descriptions of Nontrivial Tokens in ZUCC

BOOL(true/false): BOOL is either 'true' or 'false'. It is the datatype of our language, making it possible to handle various logic operations.

Motivations and Constraints:

We decided to have a token for booleans in order to match with this project's expectations. Having truth values in our language was required and boolean is a perfect data type for that.

Relations to various language criteria:

Both 'true' and 'false' are both easy to read and write. It is really hard for user to make any mistakes with those tokens and those tokens would perform their roles under any condition. So these tokens are reliable.

AND: AND is one of our connectives that corresponds to the AND logic operation. Reserved keyword for that token is '&'.

Motivations and Constraints:

We decided to have a token for AND logic operation in order to handle various logic operations. It is possible to handle all logic operations with and-or-not operations so those were our selections.

Relations to various language criteria:

Since '&' token corresponds to the 'and' operation in many languages, it is really easy for user to read and write it. Due to that, we choosed '&' token for the and operation and this helped our language to be readable and writable.

OR: OR is one of our connectives that corresponds to the OR logic operation. Reserved keyword for that token is '|'.

Motivations and Constraints:

We decided to have a token for OR logic operation in order to handle various logic operations. It is possible to handle all logic operations with and-or-not operations so those were our selections.

Relations to various language criteria:

Since ‘|’ token corresponds to the ‘or’ operation in almost all programming languages, it is really easy for user to read and write it. Due that that, we choosed ‘|’ token for the or operation and this helped our language to be readable and writable.

EQCHECK: ZUCC programming language checks equality between two boolean expressions by using ‘==’ token. Since our language has booleans as only data types,

Motivations and Constraints:

We decided to have a token for equality in order to check if to logical expressions are equal to each other.

Relations to various language criteria:

Most of the current programming languages uses ‘==’ token for the equality checking. We also decided on this in order for our language to be easy to read and write for new programmers that chooses our language. We considered to match with readability and writability criteria of programming languages.

NOT: In the ZUCC programming language, the corresponding token that represents the negation of a boolean expression is ‘!’. The programmer can use this token in order to use the logical negation of a boolean expression.

Motivations and Constraints:

We decided to have a token for ‘not’ because our language is a logical calculation language and it was needed.

Relations to various language criteria:

Most of the mathematical languages and programming languages uses ‘!’ token for the negation operation. We also decided to use this token in order to not cause any conflicts and make our program easy to read and write.

LP/RP

Motivations and Constraints:

Using ‘(‘ and ‘)’ for left and right parenthesis is a common thing in general. So we also decided to use them in our language, in order to not create any conflicts and misunderstanding by our programmers.

Relations to various language criteria:

Having tokens for left and right parenthesis is really important for our language for usability, since the programmers will need to define some logical operations and some of them will have a higher precedence,

IF/THEN/ELSE/WHILE/DO/BGN/END/INIT/DEFINE/OUTPUT/INPUT/ RETURN:

Motivations and Constraints:

We chose to use the same actual words for those tokens, to make writability and readability much more efficient. Users can easily adapt to ZUCC due to that. Our main focus when selecting those tokens were the ease of usage. They literally do what their name suggests. In the BGN token, we first decided on BEGIN but the C code generated by the yacc parser created a redefinition error because the

code automatically generated was using a function called BEGIN so we changed the token into BGN.

Relations to various language criteria:

Having those tokens improves ZUCC's readability, writability and usability because those tokens let users do various operations without having to learn and get used to new words. When users need a logical programming language, ZUCC will offer them any operation they would need with those various tokens. This makes ZUCC usable.

ASSIGNMENT/TERMINATOR/COMMA: Our assignment operator is '=' our comma token is ',', and our terminator is '~'. ZUCC programming language lets its user to be able to change the values of boolean identifiers. So a assignment operator is needed. A terminator operator is required in order to specify the end of a statement. Otherwise there would be some occurrences of ambiguity in the BNF form of our syntax. Also, a terminator operator makes the language more readable. COMMA token is used in order to separate one boolean expression from others in boolean_exprs.

MAIN/LCURLY/RCURLY: We have 3 tokens that indicates the start of the program and end of the program. MAIN token is straight up 'main' word. LCURLY is '{' and RCURLY is '}'. Program will be between these bracket tokens.(main{...})

C_IDENTIFIER/P_IDENTIFIER/A_IDENTIFIER:/V_IDENTIFIER: ZUCC uses four types of identifiers. an identifier starts with a letter and can continue with a letter or a digit. Except for V_IDENTIFIER which specifies a value identifier, they all have to start with a specific letter and an underscore. C_IDENTIFIER specifies constant variable names, P_IDENTIFIER specifies predicate names, A_IDENTIFIER specifies array names.

Relations to various language criteria:

Using names with specific starting terms will make our language much more readable and easy to understand. However there is a trade-off. Writability of our language is restricted in order to gain readability.

INDEX: INDEX is a token we decided to specify in order to make it easy to understand an array element during the yacc phase. It specifies two brackets and a number between them (e.g. [23]).

COMMENT: ZUCC lets its users to write comments between '/*' and '*/' tokens. Having a commenting feature is really important for a language to be understandable and readable for further use. Explaining a code segment with comments makes it much easier to understand and read. So having this feature in ZUCC was a necessary thing for us to implement. This increases readability and increases the human maintenance of the language.

4) Unresolved Conflicts

*****THERE ARE NO CONFLICTS LEFT UNRESOLVED IN THIS SUBMISSION*****