



CS 315 - PROGRAMMING LANGUAGES
Project Report Part 1
ZUCC

Section 1 Group 33
Instructor: Halil Altay Güvenir

Group Members
H.Buğra Aydın(21501555)
Ata Gün Öğün(21501187)

1) BNF Description of ZUCC

<program> ::= <stmts>

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

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

<init_stmt> ::= INIT IDENTIFIER ASSIGNMENT <boolean_expr> | INIT C_IDENTIFIER
ASSIGNMENT <boolean_expr>

<assignment_stmt> ::= 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

<function_declaration> ::= DEFINE F_IDENTIFIER LP <ids> RP BEGIN <stmts> RETURN
<boolean_expr> END | DEFINE F_IDENTIFIER LP
<ids> RP BEGIN RETURN <boolean_expr> END

<function_call> ::= F_IDENTIFIER LP <ids> RP | F_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> ::= <function_call> | <id>

<ids> ::= <ids> ',' <id> | <id>

<id> ::= IDENTIFIER | C_IDENTIFIER | BOOL

<connective> ::= AND | OR | EQCHECK

2) Explanation of Language Constructs in ZUCC

<program>: This construct is used for describing what a program is.

Example:

```
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(!f_trueCheck(false)) then
output((f_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 f_trueCheck(a)
begin
return a==true
end
```

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

Example:

```
init c_tru = true
```

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

Example:

```
zucc = false
```

<if_stmt>: 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 word.

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(f_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(f_trueCheck(zucc))
```

<function_declaration>: Defines function declarations. Functions can get one or more arguments and return a single value.

Example:

```

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

```

<function_call>: Defines function calls. Function calls correspond to boolean values.

Example:

```

if(!f_trueCheck(false)) then
  output((f_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)&f_zzzz(bugra)

```

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

Example:

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

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

Example:

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

<connective>: This connective acts as a wrapper for all connectives such as “EQCHECK”, “AND” and “OR” tokens.

Example:

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

<comment>: This connective acts as a wrapper for comments. Users can specify their comments between the ‘/*’ and ‘*/’ tokens and end it with a new line. This connective will understand that this code segment between them which is an alphanumeric won’t be executed.

Example:

```
if(!f_trueCheck(false)) then
output((f_trueCheck(((!!(!false))))))~
zucc = c_fal
/* ata buğra */
else if (true) then
output(false)
else
output(true)
end
```

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 chose '&' 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 to that, we chose '|' 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 two 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/BEGIN/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.

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: Our assignment operator 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.

IDENTIFIER/C_IDENTIFIER/F_IDENTIFIER: ZUCC uses three types of identifiers. Identifier starts with a letter and can continue with a letter or a digit. Players will be able to change the current value of these. Our constants starts with 'c_' and continues with an identifier. Users won't be able to change the values of these. Function names will start with 'f_' and continue with an identifier. Values of them will be identified during the function call.

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.

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.