DEPARTMENT OF INFORMATION & COMPUTER ENGINEERING

Compilers Workshop

Spring Semester 2023-24

Import

The Uni-C language is a small subset based on the famous general purpose programming language C programming language . C is a language procedural (procedural) programming language.

Uni-C is designed for use solely within the Compiler's Lab and its features are described in detail in the following two parts of this document.

PART 1: Verbal Analysis

A Uni-C source program is read by a lexical analyzer (lexical analyzer) as a long input string. LA separates one by one the lexemes contained within the input string and recognizes them as word units each time the editor sends it a request syntactic analyzer – SA (parser).

This chapter describes how the parser performs the separation and recognition of source code lexicons.

1.1 The alphabet

The alphabet of the language consists of the following ASCII characters:

Lowercase and uppercase Latin characters: az AZ

Numeric digits: 0–9

Special characters: ! " # % & ' ( ) * + , - [. / : ; < = > ? [\ ] ^ _ { | } ~

Whitespace characters: space (space \s), tab (\t), new-line (NEWLINE \n), end-of-file (EOF)

The language is case sensitive between lowercase and uppercase (case sensitive language).

1.2 Separators between words

Except the beginning of a line, or a string, the separator characters, i.e. the blank character, the tab and the semi-colon, can be used in any combination to separate words (lexemes) within the source program. The gap is necessary between two lexemes only when they could be another word (e.g. ab is a word but a and b are two lexemes while while month <= 12 can also be written without its intervening spaces as month<=12).

1.3 Verbal units (tokens)

When the source words are separated and then successfully recognized, by satisfying a recognition pattern, they identify a lexical unit (token) described by the word identifier and other information about it (see theory).

Basic classes of identified in pattern recognition in Uni-C are the following: identifiers or names (identifiers or names), keywords (keywords), literal strings (strings) and operators (operators). The string '\n' is recognized as NEWLINE, while the end-of-file is identified with EOF. All source words are returned

Separators (sequences of spaces and sequences of tabs) used to separate lexicons and comments are recognized by templates but not returned as tokens, while other separator characters used to separate syntax structures are recognized as normal and returned as tokens with an identifier name such as the *delimiter,* **e.g. the character ';' in code a = (b+c); could be returned as** *delimiter(?).*

Where there is ambiguity when splitting words, the split includes the longest possible string that forms an acceptable token when the input string is read (while splitting it) from left to right.

1.3.1 Identifiers (names)

Uni- *C* **identifiers or names are** *described* **by an identification template based on the following verbal definitions.**

They are recognized and returned as tokens with the identifier name :

The lexemes in this category consist of one or more uppercase and/or lowercase Latin letters a to z and A to Z, and/or the underscore _ and, in addition to the initial character, the digits 0 until 9.

Identifiers are theoretically unlimited in length. In practice, however, you can arbitrarily define a reasonable length that serves you in your implementation (e.g. 32 characters)

1.3.2 Keywords

In Uni-C, the following names are used as keywords *(reserved words* **or** *keywords)* **and do not require an identification pattern because they are already registered in the symbol table, but they are normally returned as tokens with an identifying *keyword name,* e.g. keyword(if).**

```
break case                func        const continue
do          double else float for
if int long return short sizeof struct switch void while
```

1.3.3 Verbal literals

String literals **or strings are** *enclosed* **in double " apostrophes and include any character other than backslash \, newline \n, or double apostrophe " that require an escape sequence to be used.**

They are recognized and returned as tokens with the identifier name *string.*

Recognized escape sequence combinations within strings are:

Escape Sequence Meaning		Notes
\\	backslash (\)	
\"	double quote (")	
\n	ASCII Linefeed (LF) NEWLINE	

Examples:

```
""

"Test"

"Mark said, \"Boo!\"\n"
```

1.3.4 Numerical literals

Numerical literals *contain integers* **and floating** *point numbers.*

Note that the numbers do not include a sign: the expression -1 is essentially a combination of the '-' operator and the number 1.

1.3.4.1 Integers

Integers include decimal, hexadecimal, and octal integers and are described as follows: A decimal integer begins with a NON-zero digit (1-9) and is optionally followed by any number of digits 0-9. A hexadecimal integer starts with 0x or 0X followed by one or more hexadecimal (0-9, AF) digits. An octal integer starts with 0 and is followed by one or more octal (0-7) digits.

Note that 0 is the only exception to a decimal integer starting with 0.

They are recognized and returned as tokens with the identifier name *integer* e.g. **integer(3).**

Examples of decimal, hexadecimal and octal integers:

0 3	214748	0x4F
0XFF42 063	0147	00

1.3.4.2 Floating-point numbers

Floating point literals **are described as follows: A decimal number consists of the integer part and the decimal part separated by a dot '.'. Both the integer part and the decimal part are defined according to the rules described above for the simple integers of the decimal number system. It is also possible to define forces using the character 'e' or 'E'. In this case the integer and/or decimal part is raised to the integer power following the character 'e' or 'E'.**

They are recognized and returned as tokens with the identifier name *float.*

Some examples of floating point numbers:

3.14	10.0	0.001	1e100	3.14e-10	0e0

1.3.5 Operators

The following tokens are recognized in Uni-C as operators:

ÿ **numeric: +, -, *, /, %** ÿ
assignment: =
ÿ **incremental assignment: +=, -=, *=, /=** ÿ
boolean logic: !, &&, || ÿ
equality check: ==, != ÿ
increment and decrement by one: ++, -- ÿ
order relations: <, <=, >, >= ÿ
memory address: &

+	-	*	/	%	=	+=	-=
*=	/= <	!	&&	||	==	!=	++
--		>	<=	>=	&		

They are listed within the symbol table without needing an identification pattern. They are returned as tokens with the identifier name *operator,* e.g. operator(<=).

1.4 Comments

In Uni-C there are two kinds of comments: single-line comments and multi-line comments. A single-line comment begins with two consecutive '/' characters (ie //), which are not part of the comment string, and ends with the end-of-line (\n). A multi-line comment begins with /* characters and ends only with */ characters, regardless of the number of intervening lines.

They are recognized by template with the ***comment*** **identifier but are NOT returned as tokens (ignored by the parser).**

1.5 White_spaces characters

Sequences of separator characters (spaces, tabs or combinations thereof), used to separate lexemes, are recognized by templates with the identifier name *white_spaces* but are not returned as tokens (they are ignored by the parser).

Part 2: Editorial Analysis

The role of the syntax analyzer-parser is to determine the syntactic correctness of the source code structures based on the grammar of the language. The input to the syntactic analyzer - SA is a sequence of lexical units *(tokens),* produced by the lexical analyzer - LA (lexical analyzer-scanner), after separating and recognizing the lexicons of the input string. The tokens are returned by the LA one by one after continuous requests from the SA during the checks it performs. Many times, instead of verbal units, the SA asks the LA to identify specific words that in the grammar followed by the SA are called by the general term *atoms.* An atom example is the string '<=' when the SA asks the LA to find the <= character sequence instead of asking the next word to be the token operator(<=). Usually the grammar uses atoms instead of tokens when it comes to some word whose recognition does not require a template (keywords, operators, separators, etc.).

**This chapter describes the grammar for structuring a source code and writing *expressions (expressions)* of the Uni-C language. Expressions within the source code are used to declare an identifier, calculate a value, or call a function. The expressions that will
described are a small subset of the C language expressions in a simplified form.**

2.1 Source Code Structure

2.1.1 Logical lines

The code of a Uni-C program consists of logical lines. Each program expression must be confined within a logical line. Each logical line is terminated with the semi-colon character (?) and is composed of one or more natural lines linked together based on syntactical linking rules.

2.1.2 Physical lines

A physical line of the source file consists of a sequence of characters terminated by the NEWLINE special character.

2.1.2.1 Physical Line Connection Rules

Two natural lines are joined together by the special backslash character (\) at the end of the first line.

A line containing a comment must not contain the backslash either at the beginning or the end of the comment because the comment itself will disconnect the contained line from the next.

It is forbidden to use the backslash anywhere else on a line unless it appears within a string.

Example of a logical line with 4 physical lines:

```
if (year < 1900 && month <= 12 \ && day <= 31 &&
      hour < 24 \ && minute < 30 && second <
      60) return 1;
```

2.1.3 Blank lines

A logical line containing only spaces, tabs and possibly a comment but no language statement is ignored, meaning the parser returns nothing to the parser.

2.2 Variable declarations **Variable**

declarations are made by specifying the data type (double, int, long, short, float) followed by one or more variable names.

Examples:

```
int a;

double var;

float c;
```

2.3 Arrays

An array contains elements separated by commas and enclosed in square brackets [ ]. Elements within an array must strictly have the same data type. A reference to a specific element of an array is made by using a numeric index inside brackets.

Examples of tables:

```
pin1 = [1, 2, 3, 4, 5 ];

pin2 = ["a", "b", "c", "d"];

print(pin1[0]); // Will print the first element ("a") of pin1
```

2.4 Built-in simple functions

In Uni-C the built-in functions are defined: scan, len, cmp and print.

2.4.1 The scan function

The scan function reads a value from the keyboard and assigns it to a variable

Its syntax includes the identifier in parentheses.

Examples:

```
scan(x);
scan(MyVariable);
```

2.4.2 The len function

The len function returns the length of an array or string that must be enclosed in parentheses (). Syntax may contain identifiers.

Examples:

```
len([10, 20, 30, 40, 50]); len("This is a
string"); len(StringVariable);
```

2.4.3 The cmp function

The cmp function compares two strings.

Syntax may contain identifiers.

Examples:

```
cmp("test", "best");

cmp(str1, str2);
```

2.4.4 The print function

In Uni-C the print function is used to print a message to the screen. When more than one item is to be printed with the same command, they should be separated by the special comma character ','.

Examples:

```
print("Hello World"); print(x, "=",
100);
```

There is also the possibility of directly printing the result of calling another function.

Examples:

```
print(cmp(str1, str2)); print(len("This
is a string"));
```

2.5 Declaration of User Functions

Here are the simple rules to define a user function in Uni-C.

Each function is a structural unit and consists of its header and body. The header contains the reserved word func, the function name, and its parameters in parentheses. Each parameter includes a data type and a name while multiple parameters are separated by commas. The function body is enclosed in square brackets. To call the function, it is sufficient to use its name together with the parameters in parentheses.

Example:

```
func myfunc(int varA, int varB) {

            print("a = ", varA); print("b = ",
            varB);
}
```

```
// Call myfunc myfunc(10, 20);
```

2.6 Declarations of simple expressions

A simple statement is contained in a single logical line. Several simple statements can occur on a single line separated by semicolons. Arithmetic expressions and assignments are considered simple expressions.

2.6.1 Numerical expressions

The syntax of numeric expressions in Uni-C involves two identifiers or numbers with some operator (+ - * /) between them and optionally the + signs -.

Examples:

```
a1 * a2
a3 + a4
-5 + 10
15 + a5 - 9
```

2.6.2 Value assignments to identifiers

Assigning values to identifiers is done using the '=' special character for single or group assignments. Group assignment is done by separating identifiers with the ',' character, while the same character is also used to separate assignment values.

Examples of statements are the following:

```
x1=0;

x1, x2 = 0, 1;

x1, list1 = 0, [A, B, C];

x1, list1, string = 0, [A, B, C], "HELLO"?
```

2.6.3 Comparisons

Comparing values in identifiers is done using the >, <, <=, >=, ==, and != operators.

Examples of comparisons are the following:

```
x1 > x2

a <= b

myvar == 52
```

2.6.4 Joining Tables

Two arrays can be joined using the + operator .

Examples:

```
[1, 2, 3] + [4, 5, 6]          -->     [1, 2, 3, 4, 5, 6]
[1, 3] + [5, 7, 9, 11]         -->     [1, 3, 5, 7, 9, 11]
```

2.7 Compound Statements

Compound statements contain other statements. In general, compound statements span multiple lines, although in simple implementations an entire statement can be contained on a single line. Well-known complex statements are if, while and for that implement traditional control flows.

2.7.1 The if statement

The if statement is used to execute code statements if some condition (given in parentheses after the if keyword) is true.

Examples:

```
if (var == 100) print("Value of expression is 100"); if (x < y < z) { print(x); print(y);
print(z); }
```

2.7.2. The While Statement

The while statement is used for repeated execution if some condition (given within it parentheses after the reserved word while) is true.

Example:

```
// Print i as long as i is less than 10 int i; i = 1; while (i < 10) {



        print("i is: ", i); i += 1;

}
```

2.7.3. The for Statement

The for statement is used for repeated execution. In parentheses after the reserved word for there are three arguments separated by semi-colons: The first is an assignment of a value to a variable that occurs only the first time the program execution reaches the for. The second argument is a condition that is checked in each iteration and that as long as the for statement is valid the execution continues. Finally, the third argument is a value assignment that is executed after each iteration of the for.

Example:

```
// Print i, starting from 0 as long as i is less than 10 int i; for (i = 0; i < 10; i++) {


        print("i = ", i);
}
```

Uni-C code examples

Below are two simple examples of writing Uni-C source code programs:

```
func main() {

        int a; int
        b;
        scan(a);
        scan(b);
        myAdd(a,b);
}
```

```
func myAdd(int num1, int num2) {

        int sum;
        sum = a + b;
        print("Sum is: ", sum)
}
```

Last update: 2024.3.9