A Grammar for the Compiler course project

Spring 2017

*Caution: First read this document completely, and then start the project!

1 Introduction

This is the grammar for the Spring 2017 semester's Compiler course project. For the grammar that follows here are the types of the various elements by type font or symbol:

- **Keywords are in this type font.**
- ***TOKEN CLASSES ARE IN THIS TYPE FONT.***
- *Nonterminals are in this type font.*
- The symbol ε means the empty string.

1.1 Some Token Definitions

- letter = a | ... | z | A | ... | Z
- digit = 0 | ... | 9
- **ID** = consists of only letters and it does not have to be a keyword.
- **NUMCONST** = #digit^{+}
- All constant numbers start with # (e.g. #8, #9.15).
- **CHARCONST** = is any representation for a single character by placing that character in single quotes. A backslash is an escape character. Any character preceded by a backslash is interpreted as that character. For example \x is the letter x, \' is a single quote, \\ is a single backslash. There are only two exceptions to this rule: \n is a newline character and \0 is the null character.
- **White space** (a sequence of blanks and tabs) is ignored. Whitespace may be required to separate some tokens in order to get the scanner not to collapse them into one token. For example: "intx" is a single **ID** while "int x" is the type **int** followed by the **ID** x. The scanner, by its nature, is a greedy matcher.
- **Comments** are ignored by the scanner. Comments begin with // and run to the end of the line.
- All **keywords** are in lowercase. You need not worry about being case independent since not all lex/flex programs make that easy.

2 The Grammar

1. *program* → **PROGRAM** ***ID*** *declarations_list procedure_list* **MAIN** *block*
2. *declarations_list* → *declarations* | *declarations_list declarations*
3. *declarations* → *type_specifiers declarator_list;* | *ε*
4. *type_specifiers* → **INT** | **REAL** | **CHAR** | **BOOLEAN**
5. *declarator_list* → *declarator* | *declarator_list, declarator*
6. *declarator* → *dec* | *dec := initializer*
7. *dec* → ***ID*** | ***ID*** *[range]* | ***ID*** *[**NUMCONST**]*
8. *range* → ***ID*** *..* ***ID*** | ***NUMCONST*** *..* ***NUMCONST*** |
   arithmetic_expressions .. arithmetic_expressions
9. *initializer* → *constant_expressions* | { *initializer_list* }
10. *initializer_list* → *constant_expressions , initializer_list* | *constant_expressions*
11. *procedure_list* → *procedure_list procedure* | *ε*
12. *procedure* → **PROCEDURE** ***ID*** *parameters* { *declarations_list block* };
13. *parameters* → (*declarations_list*)
14. *block* → { *statement_list* }
15. *statement_list* → *statement*; | *statement_list statement*;
16. *statement* → ***ID*** *:= expressions* |
    IF *bool_expressions* **THEN** *statement* |
    IF *bool_expressions* **THEN** *statement* **ELSE** *statement* |
    DO *statement* **WHILE** *bool_expressions* |
    FOR ***ID*** *:= counter* **DO** *statement* |
    SWITCH *expressions case_element default* **END** |
    ID *(arguments)* |
    ID *[expressions] := expressions* |
    RETURN *expressions* |
    EXIT WHEN *bool_expressions* |
    block | *ε*
17. *arguments_list* → *multi_arguments* | *expressions* | *ε*

18. *multi_arguments* → *multi_arguments, expressions* | *expressions*

19. counter → ***NUMCONST*** **UPTO** ***NUMCONST*** | ***NUMCONST*** **DOWNTO** ***NUMCONST***

20. *case_element* → **CASE** ***NUMCONST*** : *block* |

    case_element **CASE** ***NUMCONST*** : *block*

21. *default* → *DEFAULT* : *block* | ε

22. *expressions* → *constant_expressions* | *bool_expressions* | *arithmetic_expressions* |

    ID | ***ID*** [*expressions*] | ***ID*** *(arguments)* | (*expressions*)

23. *constant_expressions* → ***NUMCONST*** | ***REALCONST*** | ***CHARCONST*** | ***BOOLCONST***

24. *bool_expressions* → < *pair* | <= *pair* | > *pair* | >= *pair* | = *pair* | <> *pair* |

    AND *pair* | **OR** *pair* | **AND THEN** *pair* | **OR ELSE** *pair* |

    NOT *expressions*

25. *arithmetic_expressions* → + *pair* | - *pair* | * *pair* | / *pair* | % *pair* | – *expressions*

26. pair → *(expressions, expressions)*

3 Semantic Notes

- The numbers are **INT**s or **REAL**s. Be sure to eliminate leading or trailing zeros.
- There can only be one function with a given name. There is no function overloading.
- In if statements the **ELSE** is associated with the most recent **IF**.
- Expressions are evaluated in order consistent with operator associativity and precedence found in mathematics.
- Note that **AND** and **OR** evaluate as short-circuit.
- Array assignment works. Array comparison does not. Passing of arrays is done by reference.
- Code that exits a *procedure* without a **RETURN** returns a zero.
- All variables and procedures have to be declared before use.

4 An Example

```
program example
	int i := #1;
	int j := #2;
	int k := #3;
	real r := #1.1;
	boolean b := true;
	int array[#2] := {#1, #7};
	char chars[i..k] := {'c', 'd', '7'};

	procedure func (int input; boolean which;) {
		int mid := #2;
		int c := +(input, #3);
		{
			if and then(which, true) then return +(c, mid);
			else return –(c, mid);
		}
	};

	procedure abs (int input;) {
		{
			if >(input, #0) then return input;
			else return *(-#1, input);
		}
	}

main {
	r := and(or else(+(*(#2, r), false), true), array[#1] >
#1);
	do {
		i := abs(-(i, k));
		exit when not(>(array[#2], #0))
		switch i
			case #0: {i := +(i, #1)}
			case #1: {k := -(k, #2)}
			case #2: {i := +(i, #5)}
			case #3: {i := -(i, #2)}
			default: {k := #0}
		end
		array[#2] := array[#2] – (chars[#2] = 'd');
	} while k > #0;
	for j := #10 downto #2 do {
		i := +(i, #1);
		k := -(k, #1);
	}
}
```