FRONT-END COMPILER

Julen Indias Garcia

1 Compilation - 2020/2021

Index

Introduction

In thi paper it’s explained how a front-end of compiler was developed. Two different teqniques were used to develop this, upwards analysis and syntax-oriented translate scheme.

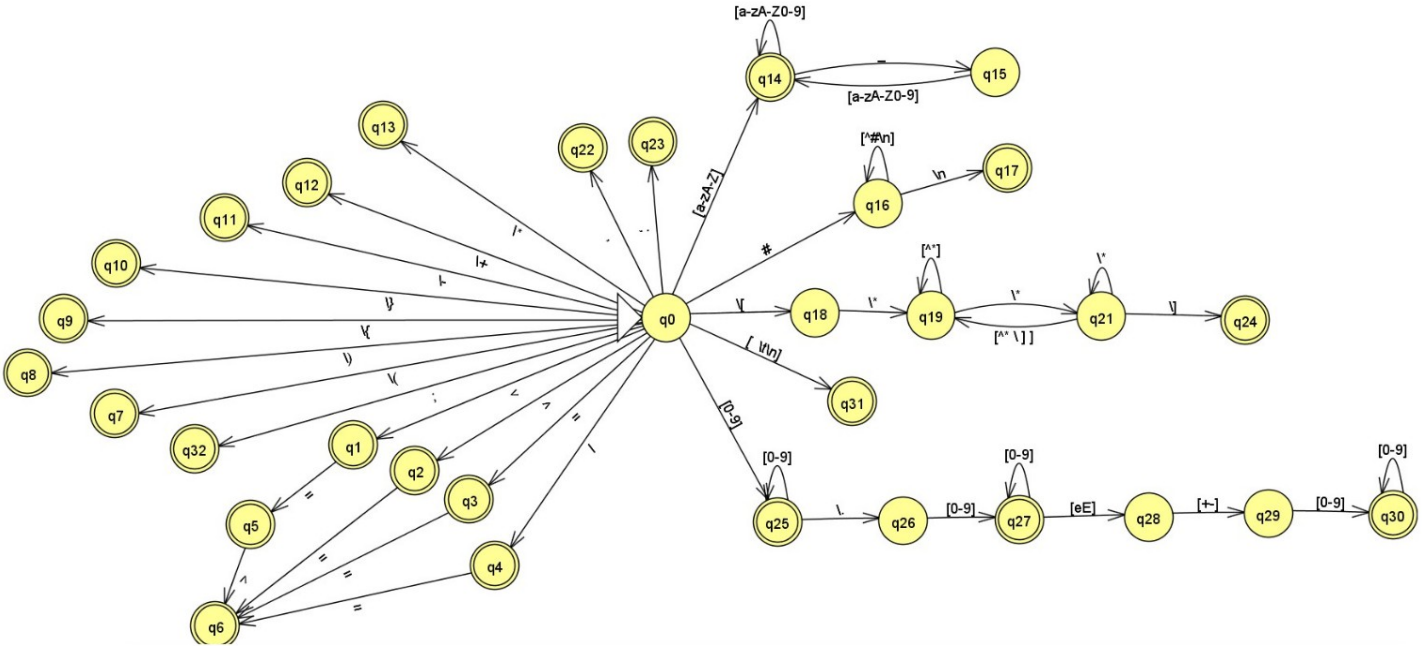
The input (source) language is a high-level programming language, and the output a mor simple language which is later used to translate into the machine or low-level language. C++, Flex and Bison were used to develop this compiler.

Lexical Analysis

Lexical specification

Token type	**Regular expression**
id	[a-zA-Z](_?[a-zA-Z0-9])*
int_const	[0-9]+
real_const	[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?
One-line comments	#[^#\n]*\n
Multiple-line comments	\[*([^*]|*+[^*\]])**+\]
Key-words	program|proc|int|float|if|while|forever|do|until|else|skip|exit|read|println
Whitespaces	[\t\n]
Separators	,|;|\(\)|{|}
Operators	\+|\-|*|\/|==|\>|\<|\>=|\<=|\<=>|=

Automaton



Syntactical definition of the input language:

program → **program id**
declarations
method_declarations
{ statement_list **}**

declarations → type id_list **;** declarations
| ξ

id_list → **id** id_list_others

id_list_others → **, id** id_list_others
| ξ

type → **int** | **float**

method_declarations → method_declaration method_declarations
| ξ

method_declaration → **proc id** arguments
declarations
method_declarations
{ statement_list **}**

arguments → **(** par_list **)**
| ξ

par_list → type par_type id_list par_type_others

par_type → => | <= | <=>

par_type_others → **;** type par_type id_list par_type_others
| ξ

statement_list → statement statement_list
| ξ

statement → variable = statement **;**
| **if** expression **{** statement_list **} ;**
| **while forever {** statement_list **} ;**
| **do {** statement_list **} until** adierazpena **else {** statement_list **} ;**
| **skip if** expression **;**
| **exit ;**
| **read (** variable **) ;**
| **println (** variable **) ;**

variable → **id**

expression → expression + expression
| expression - expression
| expression ***** expression
| expression / expression
| expression == expression
| expression > expression
| expression < expression
| expression >= expression
| expression <= expression
| expression /= expression
| variable
| **integer_const**
| **real_const**
| **(** expression **)**

Atributes:

- Lexicals:

id.name,integer_const.name,real_const.name:corresponding character chain

- Sintetized:

variable.name: character chain corresponding to the variable name
id_list.names,id_list_others.names:list of the character chains corresponding to the names of all ids in the list
expression.true,expression.false:list of all the uncompleted branch references
expression.name: character chain corresponding to the constant or variable in the expression
type.type: character chain corresponding to the type: “ent” (int) or “real”
par_type.type: character chain corresponding to the type of the parameter: “in”, “out”, or “inout”
M.ref: reference of the next statement that’s going to be written

statement.skip: if the current statement is a “skip if” type statement, list of the brach reference that’s going to be completed
statement_list.skip:list of all the “skip if” type branches that are going to be completed
statement.exit:if the current statement is a “exit” type statement, list of the brach reference that’s going to be completed
statenent_list.exit:list of all the “exit” type branches that are going to be completed

Functional abstractions:

The code is a general variable, so we’re not going to put it in the parameters. All the methods related to the code have been pre-made, and are explained in the Code.h file.

add_declarations: code x list x type → code
for each input id, starting from the beggining to the end adds a statement like this: type name

add_ param_declarations: code x type x par_type x list → code
for each input id, starting from the beggining to the end adds a statement like this:
par_type name (if the parameter is “in” type)
ref_type name (if the parameter is “in” or “inout” type)

add: list x element → list
adds the input element into the beggining of the given list and returns the list with the new element on it

new_list: element → list
starts and returns a new list with only the input element on it

new_id: → name
returns a new aux variable with an unused name untill the moment

empty_list: → list
returns an empty list

wrap: list x list → list
given two input lists, returns a new list with the elements of both lists on it

Translation scheme:

program	→	**program id**	{ start_code(); add_statement(prog || id.name); }
		declarations	
		method_declarations	
		{ statement_list **}**	{ add_statement(halt); write_code();}
declarations	→	type id_list ;	{ add_declarations(type.type, id_list.names);}
		declarations	
		| ξ	

```
id_list → id id_list_others
                { id_list.names := add(id_list_others.names, id.name); }
id_list_others → , id id_list_others
                { id_list_others.names := add(id_list_others.names, id.name); }
              | ξ
                { id_list_others.names := empty_list(); }

type        →  int          { type.type := ent; }
             | float        { type.type := real; }

method_declarations→ method_declaration method_declarations
              | ξ

method_declaration → proc id  { add_statement(proc || id.name); }
                     arguments
                     declarations
                     method_declarations
                     { statement_list }   { add_statement(endproc); }

arguments → ( par_list )
          | ξ

par_list →  type par_type id_list
          { add_param_declarations(type.type, id_list.names, par_type.type); }
          par_type_others

par_type    →  =>          { par_type.type:=in; }
             | <=          { par_type.type:=out; }
             | <=>         { par_type.type:=in out; }

par_type_others → ; type par_type id_list
                  { add_param_declarations(type.type, id_list.names, par_type.type);}
                  par_type_others
                  | ξ

statement_list → statement statement_list
                  { statement_list.skip := wrap(sentence.skip, statement_list 1.skip);
                    statement_list.exit := wrap(sentence.exit, statement_list 1.exit);}
              | ξ
                  { statement_list.skip := empty_list();
                    statement_list.exit := empty_list();}

statement  → variable = statement      { add_statement(variable.name || := || expression.name);
                                         sentence.skip := empty_list();
                                         sentence.exit := empty_list();}

           | if expression M{ statement_list } M;
                                       { complete_statement(expression.true, M1.ref);
                                         complete_statement(expression.false, M2.ref);
                                         sentence.skip := sentence_list.exit;
                                         sentence.skip := sentence_list .skip;}
```

| **while forever M{** statement_list **} M;**
{ add_statement(goto || M_1.erref);
complete_statement (sententzia_zerrenda.exit, M_2.erref+1);
sentence.skip := sentence_list.skip;
sentence.exit := empty_list();}

| **do M{** statement_list **} until M** expression **else M{** statement_list **}M;**
{ complete_statement(expression.true, M_3.ref);
complete_statement(expression.false, M_1.ref);
complete_statement(sentece_list$_1$.skip, M_2.ref);
complete_statement(sentece_list$_1$.exit, M_4.ref);
complete_statement(sentece_list$_2$.exit, M_4.ref);
sentence.skip := sentece_list$_2$.skip;
sentence.exit := empty_list();}

| **skip if** expression **; M** { complete_statement(expression.false, M.ref);
sentence.skip := expression.true;}

| **exit ;** { sentence.exit := new_list(get_ref());
add_statement(goto);}

| **read (** variable **) ;** { add_statement(read || variable.name);
statement.skip := empty_list();
statement.exit := empty_list();}
| **println (** expression **) ;** { add_statement(write || expression.name);
add_statement(writeln);
sentence.skip := empty_list();
sentence.exit := empty_list();}

M → ξ { M.ref := get_ref();}

variable → **id** {variable.name := id.name;}

expression → expression + expression {expression.name := new_id();
add_statement(expression || := || expression$_1$.name || + || adierazpena$_2$.izena);}
| expression – expression {expression.name := new_id();
add_statement(expression || := || expression$_1$.name || - || expression $_2$.name);}
| expression * expression {adierazpena .izena := id_berria();
add_statement(expression || := || expression$_1$.name || * || expression $_2$.name);}
| expression / expression {adierazpena .izena := id_berria();
add_statement(expression || := || expression$_1$.name || / || expression $_2$.name);}
| expression == expression { expression.name := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||=||expression$_2$.name||goto);
add_statement(goto);}
| expression > expression {expression.name := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||>||expression$_2$.name||goto);
add_statement(goto);}

| expression < expression { expression.izena := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||<||expression$_2$.name||goto);
add_statement(goto);}

| expression >= expression { expression.izena := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||>=||expression$_2$.name||goto);
ag_gehitu(goto);}
| expression <= expression { expression.izena := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||<=||expression$_2$.name||goto);
ag_gehitu(goto);}
| expression /= expression { expression.izena := new_id();
expression.true := new_list(get_ref());
expression.false := new_list(get_ref() + 1);
add_statement(if||expression$_1$.name||!=||expression$_2$.name||goto);
ag_gehitu(goto);}

| variable {expression.name := variable.name;}
| **integer_const** { expression.name := integer_const.name;}
| **real_const** { expression.name := real_const.name;}
| (expression) { expression.name := expression$_1$.izena;
expression.true := expression $_1$.true;
expression.false := expression $_1$.false;}

Test cases

- **test1: one-line comments, multiple-line comments and assignations**

```
./parser <../testcases/test1.in
started...
1: prog test1 ;
2: a := 0.4756 ;
3: halt  ;
finished...
```

- **test2: method declaration, do-until-else, skip if and if statement**

```
./parser <../testcases/test2.in
started...
1: prog test2 ;
2: ent a ;
3: ent b ;
4: ent c ;
5: real d ;
6: real e ;
7: proc sum ;
8: val_ent x ;
9: val_ent y ;
10: ref_ent result ;
11: ent aux ;
```

```
12: ent iters ;
13: aux := y ;
14: result := x ;
15: if result < 1000 goto 17 ;
16: goto 33 ;
17: iters := 0 ;
18: _t1 := result + 1 ;
19: result := _t1 ;
20: if result > 100000 goto 26 ;
21: goto 22 ;
22: _t2 := aux - 1 ;
23: aux := _t2 ;
24: _t3 := iters + 1 ;
25: iters := _t3 ;
26: if aux = 0 goto 28 ;
27: goto 18 ;
28: if result < 0 goto 30 ;
29: goto 31 ;
30: goto 33 ;
31: write iters ;
32: writeln ;
33: endproc ;
34: read a ;
35: read b ;
36: _t4 := 1 / b ;
37: d := _t4 ;
38: _t5 := c * d ;
39: _t6 := c * _t5 ;
40: _t7 := _t6 + e ;
41: c := _t7 ;
42: write c ;
43: writeln ;
44: halt  ;
finished...
```

- **test3: while_forever + do_until_else + exit test**

```
./parser <../testcases/test3.in
started...
1: prog test3 ;
2: ent a ;
3: ent b ;
4: ent i ;
5: ent j ;
6: read a ;
7: read b ;
8: i := 0 ;
9: j := 0 ;
10: _t1 := a * 2 ;
11: a := _t1 ;
12: _t2 := b * 2 ;
13: b := _t2 ;
14: _t3 := j + 1 ;
15: j := _t3 ;
16: if j = 100 goto 18 ;
17: goto 10 ;
18: write a ;
19: writeln ;
20: write b ;
21: writeln ;
22: if i = 100 goto 24 ;
23: goto 25 ;
```

```
24: goto 28 ;
25: _t4 := i + 1 ;
26: i := _t4 ;
27: goto 9 ;
28: write i ;
29: writeln ;
30: write j ;
31: writeln ;
32: halt  ;
finished...
```

- **test4: do_until_else test and if statement**

```
./parser <../testcases/test4.in
started...
1: prog test4 ;
2: real a ;
3: real b ;
4: real result ;
5: real diff ;
6: proc duplicate_bigger ;
7: val_real a ;
8: val_real b ;
9: ref_real result ;
10: if a > b goto 12 ;
11: goto 14 ;
12: _t1 := a * 2 ;
13: result := _t1 ;
14: if b > a goto 16 ;
15: goto 18 ;
16: _t2 := b * 2 ;
17: result := _t2 ;
18: endproc ;
19: read a ;
20: read b ;
21: result := 5 ;
22: write result ;
23: writeln ;
24: _t3 := result + 5 ;
25: result := _t3 ;
26: if result <= 1000 goto 28 ;
27: goto 22 ;
28: _t4 := result - 1000 ;
29: diff := _t4 ;
30: write diff ;
31: writeln ;
32: halt  ;
finished...
```

- **test5: do-until-else + skip-if test**

```
./parser <../testcases/test5.in
started...
1: prog test5 ;
2: ent a ;
3: ent b ;
4: ent c ;
5: ent i ;
6: a := 1 ;
7: b := 2 ;
8: c := 3 ;
9: i := 0 ;
10: _t1 := a * 2 ;
11: a := _t1 ;
12: _t2 := b + 1 ;
13: b := _t2 ;
14: _t3 := c + 20 ;
15: c := _t3 ;
16: if b > 10 goto 22 ;
17: goto 18 ;
18: write i ;
19: writeln ;
20: _t4 := i + 1 ;
21: i := _t4 ;
22: if a > 30 goto 24 ;
23: goto 10 ;
24: write a ;
25: writeln ;
26: halt  ;
finished...
```

- **test6: while_forever + exit test**

```
./parser <../testcases/test6.in
started...
1: prog test6 ;
2: ent x ;
3: ent i ;
4: x := 0 ;
5: i := 0 ;
6: _t1 := x + 5 ;
7: x := _t1 ;
8: _t2 := i + 1 ;
9: i := _t2 ;
10: if i >= 20 goto 12 ;
11: goto 13 ;
12: goto 16 ;
13: write 1 ;
14: writeln ;
15: goto 6 ;
16: write x ;
17: writeln ;
18: write i ;
19: writeln ;
20: halt  ;
finished...
```

- **bad test 1: *] in the middle of a multiple-line comment**

```
./parser <../testcases/badtest1.in
started...
line 5: syntax error at 'in'
finished...
```

- **bad test 2: istead of separating the parameters with ; use ,**

```
./parser <../testcases/badtest2.in
started...
line 7: syntax error at 'int'
finished...
```

- **bad test 3: using if-else when it’s not in our language**

```
./parser <../testcases/badtest3.in
started...
line 13: syntax error at 'else'
finished...
```

- **bad test 4: do-until-else without the else**

```
./parser <../testcases/badtest4.in
started...
line 22: syntax error at '}'
finished...
```

- **bad test 5: use of the += operator in the 20th line to increment a variable, which is unsupported**

```
./parser <../testcases/badtest5.in
started...
line 20: syntax error at '+'
finished...
```

- **bad test 6: variable with two '_'**

```
./parser <../testcases/badtest6.in
started...
Unknown token: _
line 4: syntax error at '_'
finished...
```

- **bad test 7: regular while structure (with an expression) instead of the while forever which is the one that is in our language**

```
./parser <../testcases/badtest7.in
started...
line 18: syntax error at 'result'
finished...
```