Syntactical Analyzer Documentation — CS 323

Jared Dyreson
Chris Nutter
California State University, Fullerton

Contents

1 Problem Statement

This project aims to create a syntax analyzer that ensures code conforms to a set of grammar rules defined in the rubric.

2 Credit

Synthetic is built upon the works of **ezaquraii**, with his beautiful work for Flex, GNU Bison integration and the ability to change file streams to make testing go by MUCH faster. If this project did not exist, we would not be here at this point. Here is the link to the original work, all other work is original.

3 Notes and Caveats

- Given there are so many different functions that would need to be created for each individual grammar rule, GNU Bison is was used to generate these trees
- Since this project is so interlaced with Flex (Fast Lexer), the use of **Lexi** (our original lexer built for project #1), could not be utilized.

4 Types Allowed

- bool
- long long int (unsigned 64-bit integers)
- float (for division and other computation requiring floating point arithmetic)

5 Tokens Defined

5.1 Separators

- LEFTPAR → (
- RIGHTPAR →)
- LEFT_CURLY → {
- RIGHT_CURLY → }

5.2 Delimiters

- SEMICOLON → ;
- COMMA → ,

5.3 Operators

- ASSIGN → =
- GEOMETRIC_OP → * /
- ARITHMETIC_OP → + -
- RELATIONAL_OP → > < <= >= && ||
- ID_INC → ++
- ID_DEC → −−

5.4 Reserved Words — Primitive Types

- bool
- int
- float

5.5 Reserved Words — Control Flow

- if
- else
- then
- endif
- for
- forend
- while
- whileend
- do
- doend

6 Rules

- assignment
- statements → statement
  - if
  - for
  - while
  - do while
- expression (arithmetic computation using long long integers)
- term (geometric computation using floating point integers)

7 Process

- Source file read in (all tests are inside the **inputs** directory)
- Gets lexxed using Flex
- Token stream is fed into GNU Bison
- Each individual token is then checked against grammar rules defined and intermediate code generation takes place here
- AST is then created after the code generation occurs

8 Grammars

Some of these rules do have immediate left recursion and GNU Bison is able to account for these

8.1 Expressions

⟨*expression*⟩ ::= NUMBER
| ⟨*expression*⟩ ARITHMETIC_OP ⟨*expression*⟩
| LEFTPAR ⟨*expression*⟩ RIGHTPAR

```
1  ! Testing addition and subtraction rules  !
2  int func() {
3      ! No parens !
4      1 + 1
5      1 - 1
6
7
8      ! Parens !
9      (1 + 1)
10     (1 - 1)
11 }
```

8.2 ID

⟨*id*⟩ ::= ID

```
! Identifiers !

void func(){
    ! Matches numbers and variable names !
    1
    2

    a
    ab
    abc
}
```

8.3 Assignment

⟨*assignment*⟩ ::= PRIMITIVE_TYPE ID SEMICOLON
| PRIMITIVE_TYPE ID ASSIGN ⟨*expression*⟩ SEMICOLON
| ID ASSIGN ⟨*expression*⟩ SEMICOLON
| PRIMITIVE_TYPE ID ASSIGN ⟨*term*⟩ SEMICOLON
| ID ASSIGN ⟨*term*⟩ SEMICOLON

```
! Assignments !

void func(){
    ! This is a test for assignment !

    int value = 10;
    int a  =  15;
    int b = 20;

    int c = 10 * 20;
    int d = 50 / 10;
}
```

8.4 Term

⟨*term*⟩ ::= ⟨*factor*⟩
| ⟨*term*⟩ GEOMETRIC_OP ⟨*factor*⟩
| LEFTPAR ⟨*term*⟩ RIGHTPAR
| ⟨*id*⟩

```
void func(){
    10 * 20 * 30;
    50 / 10 / 5;
    (10 * 10 * 10);
    exampleVar;
}
```

8.5 Factor

⟨*factor*⟩ ::= ⟨*id*⟩ ID
| NUMBER
| LEFTPAR ⟨*expression*⟩ RIGHTPAR

```


void func() {
    ! Gets the value of exampleVar and places it onto a temporary stack to be
    computed !

    exampleVar

    ! Does the same but puts 10 directly, no lookup required !
    10

    ! Computes 10 + 10 and places 20 onto the stack !

    (10 + 10)
}
```

8.6 Statements

⟨*statements*⟩ ::= ⟨*statement*⟩ ⟨*statements*⟩
| ⟨*statement*⟩

```

! Mix and match statements !

void func(){
    int value = 10;
    if (true) then
        value = 100;
    else
        for(int i = 0; i < 10; ++i)
            ! pass !
        endfor
    endif
}
```

8.7 Statement

⟨*statement*⟩ ::= ⟨*assignment*⟩
| ⟨*if_statement*⟩
| ⟨*for_statement*⟩
| ⟨*while_statement*⟩

8.8 If Statement

⟨*if_statement*⟩ ::= IF ⟨*condition*⟩ THEN ⟨*statements*⟩ ENDIF
| IF ⟨*condition*⟩ THEN ⟨*statements*⟩ ELSE ⟨*statements*⟩ ENDIF

```

void func() {
    if (10 < 12) then
        int value = 10;
    else
        int var = 10;
    endif
}
```

8.9 For Loop

⟨*for_statement*⟩ ::= FOR LEFTPAR PRIMITIVE_TYPE ⟨*ID*⟩ ASSIGN ⟨*expression*⟩ SEMICOLON ID RELATIONAL_OP ⟨*expression*⟩ SEMICOLON ID_INC RIGHTPAR ⟨*statements*⟩ FOREND

```
for (int i = 0; i < 10; i++)
    int first = 0;
    for (int i = 0; i < 10; i++)
        int second = 0;
    forend
forend


for (int i = 0; i < 10; ++i)
    int first = 0;
    for (int i = 0; i < 10; ++i)
        int second = 0;
    forend
forend


for (int i = 0; i < 10; --i)
    int first = 0;
    for (int i = 0; i < 10; --i)
        int second = 0;
    forend
forend
```

8.10 While Loop

⟨*while_statement*⟩ ::= WHILE ⟨*condition*⟩ ⟨*statements*⟩ WHILEEND
| DO ⟨*statement*⟩ WHILE ⟨*condition*⟩ DOEND
| DO ⟨*statements*⟩ WHILE ⟨*condition*⟩ DOEND

```

while(value < 10)
    value--;
whileend

do
  value++;
while(value < 10)
doend
```