TITLE
Asst0: Walking in GCC's Footsteps

DUE
23:55 26 Sep 2019

NUMBER OF RESUBMISSIONS ALLOWED
Unlimited

ACCEPT RESUBMISSION UNTIL
23:55 26 Sep 2019

MODIFIED BY INSTRUCTOR
22:44 18 Sep 2019

Walking in GCC's Footsteps:

Note: This project is individual. Be sure to do your own work.

ABSTRACT

Of all the tasks you will need to complete in time or behaviors to encode, scanning and parsing tend to crop up over and over again in some form.

When scanning you take a sequence of characters and break them into tokens and when parsing you evaluate the tokens based on some rule set in order to determine if they are not only used correctly, but also what computations they represent.

In this assignment you'll get some practice with C by scanning and doing some simple parsing.

This should also give you some insight in to interpreting what the C compiler tells you when it finds problems.

INTRODUCTION

The language you are working with consists of **two kinds** of **operators**: **logical** and **arithmetic**.

The **logical operators** are: { `AND`, `OR`, `NOT` }
and are applied to the **values**: { `true`, `false` }.

The **arithmetic operators** are: { `-`, `+`, `*`, `/` }
and are applied to the **values**: { `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `0` }.

You'll **receive all your tokens** in a **single command line argument**,
and **your code should take exactly one argument**.

You can count on whitespace (e.g. `' '`) to divide tokens, however it may behoove you to make your delimiter parametrizable for future use.

The null terminator (`'\0'`) will end your input and the line you are evaluating.

Once you have detected all the tokens, you next should make sure the tokens make sense.
This language is an **infix language**,
meaning **all binary operators appear between their operands**:

Legal expressions:	**Illegal expressions:**
`1 + 2`	`1 +2`
`4 – 9`	`4 –`
`true AND false`	`True AND false`
`false OR true`	`fals OR 1`

Logical `NOT` is your only unary operator.
Its **argument** should **appear directly after it**: `NOT false`.

If there are multiple expressions per line, they should be separated by a semicolon (`';'`):

```
1 + 2; 9 * 2
true AND true; 9 / 3
```

Output should consist of:

For a **correct expression**:

```
Found <X> expressions: <N> logical and <M> arithmetic.
OK.
```

For an **incorrect expression**:

```
Found <X> expressions; <N> logical and <M> arithmetic.
Error: <error type> in expression <expression number>: <description> in:
       <expression error was found in, or expression fragment/token>
```

For instance:

```
> ./check "1 +2"
Found 1 expressions: 0 logical and 0 arithmetic.
Error: Parse error in expression 0: unknown identifier
       "+2"
Error: Parse error in expression 0: missing operator
       "1 +2"
```

```
> ./check ""
Found 1 expressions: 0 logical and 0 arithmetic.
Error: Scan error in expression 0: incomplete expression
       ""
```

```
> ./check "true AND false OR true"
Found 1 expressions: 1 logical and 0 arithmetic.
Error: Parse error in expression 0: expression was not ended
       "true AND false OR"
Error: Parse error in expression 0: unexpected operator
       "OR true”
```

(Note: 'expression was not ended' is not one of the canonical errors listed below in the FAQ, however, since it is here, I am I fine if you use it. Truly, this error ought to be an 'incomplete expression', since it is missing the semicolon required to end the expression)

You should **continue past the first error** and **detect/log each subsequent error**.
You may very well have **multiple incorrect and correct expressions in a single line.**

METHODOLOGY

Do not use `sort()`, `strtok()`, or anything from the `ctype.h` or `string.h` libraries. Do not use regular expressions or any other built-in tokenizer.

The object of this assignment is not to hunt through the installed libraries and engage in some heavy shoehorning, but to write some code working with strings and characters and gain some insight in to how to interpret error messages.

This can seem daunting, but be careful to plan first. The range of things you can get is indeed large, however only certain things are allowed.

All you need to do is enforce the structure of what is allowed, and that is not a very long list of possibilities.

I'd recommend you go by this strategy:

Identification: (find everything important)

0. find all **tokens**
   - make them in to strings and print them out

1. find all **expressions**
   - assemble tokens until you see `';'`

2. find all **operators**
   - scan through expressions for something in your operator list
   (there are only **7 possibilities**) { -, +, *, /, AND, OR, NOT }

3. find all **operands**
   - scan through expressions for something in your operand list
   (there are only **12 possibilities**) { false, true, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, }

Once you are here, you have pretty much all you need.

Computation: (abstract basic things as **functions)**

0. Write a **function** to check if:
   - a **character** is in your **arithmetic operand list**
   - a **string** is in your **logical operand list**

1. Write a **function** to check if a **character/string** is in your **operator list**
2. Write a **function** to take a **whole line** and give you **one expression at a time**
3. Write a **function** to take a **whole expression** and give you **one token at a time**

Evaluation: (assemble your **functions** for fun and profit!**)**

Feed your **input line** to your **expression finder**,
and as long as it doesn't run out of expressions:

> Feed your **next expressions** to your **token finder**,
> **and as long as it doesn't run out of tokens**:
> (**no/too few tokens**? output an **error**)
>
> Feed the **first token** to your **token classifier**
> …it should be your **first operand**, or NOT
> - if an **error**, output
>
> Feed the **next token** to your **token classifier**
> …it should be your **operator** or NOT's **operand**
> - if an **error**, output
>
> If not NOT, feed the **next token** to your **token classifier**
> …it should be your **last operator**
> - if an **error**, output
>
> (**too many tokens**? output an **error**)

You are free not to follow this strategy, however do not sit down and try to start writing code immediately. **Without some planning you can easily get very tangled up.**

RESULTS

Note: This project is **individual**. Be sure to **do your own work**.

Do:

- Make sure to **put quotes around your outputted error fragments**.
- Make sure you **free all dynamic memory you use**.
- Make sure you **never Segmentation Fault**.
- No matter what your code outputs, **Segmentation Fault makes it instantly incorrect**.
- Make sure you can **handle odd or obviously broken input** and that you always **check the number of arguments you have** before you start using them.
- If you **don't get the correct number of arguments**, print an **error** and **stop**.

Do not:

- **Do not use any Makefiles**, even if you know how to write them and how they work. **We won't run them**.
- **Do not use any extra compiler flags** or **versions** of C.
  - For instance; **do not use** `fsanitize` or C99. Just **compile** with `gcc`.
- **Do not use** `sort()`, `strtok()`, or anything from the `ctype.h` or `string.h` libraries.
- **Do not use regular expressions** or **any other built-in tokenizer**.

You should submit **text** and **one file** on **Sakai**:

file:

`check.c`

`check.c` should be compilable to an executable with:

```
$   gcc -o check.c check
```

text:

your **test plan**

Your **test plan** should **detail how you tested your code**; **what inputs did you use** and **why**.

EVALUATION:

Grading will be based on:

- your **code's function** when run against **a variety of test inputs**
- the **completeness** of your **test plan**
- your **code's readability** and **modularity**

FAQ:
Based on recent questions, here is an **error inventory** with some examples and rules:

General Rules

- scan/parse **left to right (don't backtrack)**
- **first operator found** sets **expression type**
- **first token must be an operand** unless it is `NOT`
- **only valid characters after last operand** in an **expression** are '`;`' or '`\0`'
- **exactly one whitespace between each token** (**a space after a space** is a **blank token**)
- seeing an **unknown identifier** at the **start of an expression resets your evaluation**, so you **presume the next token** is the **beginning** of the **expression.**

(examples below are only descriptive, **not necessarily all errors are catalogued**)
(**more than one error** may be caused by the **same token/identifier/whatever (see below!)**)

operator:

unknown: `"1 a 2"`
unexpected: `"+ 2"`
missing: `“1"`
> type mismatch (if you want to use it…see below)

operand:

unknown: `"1 + a"`
unexpected: `"1 + 2 1"`
missing: `"1 + “`
type mismatch: `"1 + true"` `"1 AND 2”`

~~(any type mismatch is only the operands' faults because the operator sets the expression type)~~
> the above is truly only a semantic issue, since the number of errors will be the same and for the same reason, however since I talked about this in class as an error being caused by a different operator type, I'll count them the same way.

If you throw two operand type mismatches or one operator type and one operand type mismatch for this error type, I'll count both the same way.

if your first token is broken/unknown:
unknown identifier: `"a NOT true"` `"a + 2"`
(resets parse, presume next token is the first of the expression)
(yes, this could (and likely should!) generate more errors)

expression:

wasn't ended (expected ";" found ...): `"1 + 2 3 4 5 6 7 8 9"`
incomplete (expected stuff, found ...): `"1 + 3;"`

(All errors in the lines below are listed,
although these are not correctly formatted error messages)

`"1 a 2"`
Unknown operator

`"+ 2"`
Unexpected operator

`"1"`
Missing operator

`"1 AND 2"`
Operand type mismatch
Operand type mismatch

`"true + 2"`
Operand type mismatch

`"1 + a"`
Unknown operand

`"a + 1"`
Unknown identifier
Unexpected operator

`"a NOT true"`
Unknown identifier

`"1 + 2 1"`
Expression wasn't ended
Unexpected operand

`"1 + "`
Missing operand

`"1 + 2 3 4 5 6 7 8 9"`
Expression wasn't ended
Unexpected operand
Unexpected operand
Unexpected operand
Unexpected operand
Unexpected operand
Unexpected operand
Unexpected operand

`"1 + 3;"`
Expression incomplete

Ladies and Gentlemen,

My apologies for the chug and clunk in Asst0.

Firstly; Asst0 firstly is new.
I wrote it up about a week before it was originally released
and I haven't had a chance to have students work on it yet.

There are always things that people find that I didn't expect.
I was expecting some questions, as there are always some.

Secondly; a good number of the TAs are new to TAing and nearly all are new to the course.
This made take-up time for the assignment lag more than normal.

I also had the TAs assigned quite late in the semester, so I couldn't work with them on fundamentals or get the newer ones up to speed on administrative and technical issues.

There were also some administrative and technical difficulties early on, in particular some of the TAs still do not have an office as the scheduling apparatus the University used was down for about 2 weeks.

Thirdly; I tried to make things simpler by limiting the scope of the assignment after it was released, but a couple definitions and productions slipped through when I was going back over the assignment list.

I had one of the TAs meet with me and we both went through the assignment description and the proposed FAQ for about 3 hours, although there were a few leaks. I also sat down last Sunday with one of the TAs to answer Piazza questions for about 4 hours, and that seems to have set things more or less in order, but there may have been some questions answered earlier that were inconsistent.

Even more of your favorite assignment!
As you may notice, I extended the deadline for Asst0 through the weekend.
While this puts us quite late for the rest of the assignments, I did not want to release Asst1 until after the weekend given the recent difficulties, so I found no utility in keeping Asst0 due earlier than that. Since I will be holding Asst0 open that long however, I will also release Asst1 earlier. While it is usually a two-week assignment, I will make it approx. two and a half, so that it will be due in before the midterm.

Re-examination, once again..
I've looked through the assignment description and I did find two issues, one of which was highlighted by students (thanks! 350 different bug-checkers/proofreaders are much more likely to find errors than one or two). I marked them with red text and added notes in blue. In particular; I

originally had said there were no operator type mismatches since mismatches are the operands' faults. In lecture, I spoke of operators not matching operands (and the opposite as well), so I can understand the confusion.

Honestly, it is a semantic difference because in both cases the same kind of error is raised for pretty much the same reason (`type(func) != type(data)`).

So, I will be willing to accept both interpretations.

To be concrete:

```
"1 AND 2"
```

can be either:
operand type mismatch
operator type mismatch

or:
operand type mismatch
operand type mismatch

or:
type mismatch
type mismatch

(make sure to print out the part of the expression you are looking at
when you declare a mismatch in all cases)

While no one did catch it, in class I mentioned 'missing expression',
while in the error catalogue in the FAQ I have only
'expression incomplete' and an 'expression wasn't ended'.

If you want to add/use 'missing expression' to denote that when you looked for an expression, you found nothing whatsoever (`""`), that is fine.

If you want to keep to the FAQ as it was and declare a missing expression ‘incomplete',
I will accept that too.

Concretely:

```
"1 + 2;”
```

can be:
Expression incomplete
(or incomplete expression)

or
Expression missing
(or missing expression)

(make sure to print out the part of the line you are looking at
when you declare an expression to be incomplete/missing)

Another issue that prompted a lot of questions is 'unknown identifier'.
An unknown identifier is something that you can not identify
(it isn't any of your operators or operands)
and it appears somewhere where you have no expectations about what it ought to be.

```
"1 A 2"
```

> the 'A' is not an unknown identifier.

The only way for the expression to reasonable is if the token after '1' is an operator, and there is a token there, but it isn't in your list of operators. Since it is not in your list of operands either, go with your expectation and declare it an unknown operator.

```
"1 + A"
```

> I don't think anyone has a problem with this one, but for completeness; 'A' here is an unknown operand.

You were expecting an arithmetic operand for your last token, but instead you got something not in your operand list. Not only that, it isn't in your operator list, so go with your expectation and declare it an unknown operand.

`"A 1 + 2"`
> In this case '`A`' could be anything; operator or operand.

It is at the front of an expression, so you have no expectations about what kind of thing it should be, but it isn't in your operand list or your operator list…so you can throw up your hands and declare it an unknown identifier.

You're willing to say you found it, but that's pretty much all you can claim.
An unknown identifier is a big, fat monkey wrench thrown in to the works because it is something you can't reason about whatsoever.

Since you can't reason about it, it does not close the current expression (if there is one open/ continuing), however you should expect a new expression to start.

You expect an expression to start because unknown identifier can only occur if you have no expectations, so it must occur either before an expression has started (your first token) or after an expression ought to have ended (after the last operand, but no '`;`' or '`\0`' yet).

`"1 + 2 A 3 + 3"` vs "`1 + 2  3 + 3"`:
`"1 + 2 A 3 + 3"`
> Things are fine until the unknown identifier 'A' pops up:

Expression was not ended: "2 "
Unknown Identifier: "A"
1 expression: 1 arithmetic, 0 logical

(not going to complain about finding `"3 + 3"` even though the expression was NOT ended, because you were expecting one due to the unknown identifier)

`"1 + 2  3 + 3"`
> things are fine until after '`2`':

Expression was not ended: "2 "
Unexpected Operand: "3"
1 expression: 1 arithmetic, 0 logical

(no more complaining, though.. once I found that arithmetic operand, the only thing that ought to follow it is an arithmetic operator, which there was! Then following that an arithmetic operand, which there was! then following that an expression ender (`'\0'` ends everything) .. and there was!)

One student asked a pretty good question about expression format vs things out of place:
`"1 b 2 + 3”`

His question was that things make sense until the 'b', but then what?
Should you say the '+' was unexpected, or would you expect it because you saw a '2', and the correct thing to follow a '2' is an arithmetic operator?

In this case, you should declare the '+' unexpected.
The reason is that if you declare 'b' an unknown operator, you are expecting an operand afterward. Every operator you have must be followed by an operand.

This operator, even though you can't determine which it is, is followed by an operand, and the expression ought to have been ended after it, so the '+' is in the wrong spot:
unknown operator: "b"
expression wasn't ended: "2 "
unexpected operator: “+"

Likewise, you don't complain about the '3'. The "+" was indeed in the wrong place, but it is an operator, and an operand is expected to follow every operator.

I've been asked a couple questions that I put in the FAQ. This has puzzled me a bit. If you don't see an 'FAQ' at the bottom of the Sakai assignment, you should refresh Sakai. Sakai is a website, so it will cache old data. You might need to clear cache/restart your browser.

In particular I have been asked what you do with multiple spaces:
`"1  + 2”`

In the FAQ I note that one space is require after a token, as it is a delimiter.
So, you can find the '1' above, but afterward a space happens immediately,
so the token after '1' is empty: ".

You were expecting an operator, so you have an unknown operator, then the '+' is unexpected, because you already found an operator and you were expecting an operand.

The '2' causes no complaining, though. Even though the '+' was out of place, it IS an operator and ought to be followed by an operand.

(P.S. according to the Interwebs it is ~40 mins to first light and ~68 mins to dawn ... made it! At least the traffic will be light going South .. here's to easy commutes! :^D)

-JAF

Ladies and Gentlemen,

If you print your expression count after your errors, that is fine.

This can make it easier to proceed through your string to parse,
as you can emit errors as you find them,
rather than storing them just to print them out after the expression count.

If you have written significant code already to maintain the expression count
as the first printed output (i.e. before the error output),
that's fine, you can do it that way, too.

-JAF