

COP4020 Programming Assignment 2 – Spring 2011

Consider our familiar augmented LL(1) grammar for an expression language (see “Syntax” lecture notes on the LL(1) expression grammar):

GRAMMAR PRODUCTION	SEMANTIC RULES
<code><expr> -> <term> <term_tail></code>	<code>term_tail.subtotal := term.value;</code> <code>expr.value := term_tail.value</code>
<code><term> -> <factor> <factor_tail></code>	<code>factor_tail.subtotal := factor.value;</code> <code>term.value := factor_tail.value</code>
<code><term_tail1> -> '+' <term> <term_tail2></code>	<code>term_tail2.subtotal :=</code> <code>term_tail1.subtotal+term.value;</code> <code>term_tail1.value := term_tail2.value</code>
<code> '-' <term> <term_tail2></code>	<code>term_tail2.subtotal :=</code> <code>term_tail1.subtotal-term.value;</code> <code>term_tail1.value := term_tail2.value</code>
<code> empty</code>	<code>term_tail1.value := term_tail1.subtotal</code>
<code><factor1> -> '(' <expr> ')'</code>	<code>factor1.value := expr.value</code>
<code> '-' <factor2></code>	<code>factor1.value := -factor2.value</code>
<code> num</code>	<code>factor1.value := num.value</code>
<code> id</code>	<code>factor1.value := lookup(id.name)</code>
<code><factor_tail1> -> '*' <factor> <factor_tail2></code>	<code>factor_tail2.subtotal :=</code> <code>factor_tail1.subtotal*factor.value;</code> <code>factor_tail1.value := factor_tail2.value</code>
<code> '/' <factor> <factor_tail2></code>	<code>factor_tail2.subtotal :=</code> <code>factor_tail1.subtotal/factor.value;</code> <code>factor_tail1.value := factor_tail2.value</code>
<code> empty</code>	<code>factor_tail1.value := factor_tail1.subtotal</code>

where `lookup(id.name)` returns the value of the identifier (assuming it is a variable with a value) from a name-value store, which could be an associative array or hashmap etc. in the implementation of an interpreter that uses this grammar to parse input. Note that we use nonterminals with indexes, so `factor1` and `factor2` are just the same `factor` nonterminal, but indexed so we can distinguish the nonterminals in the productions in the semantic rules. The `empty` in the productions shown above denotes ϵ .

To goal of this project is to implement an interpreter that evaluates arithmetic expressions and that supports the use of variables in local scopes. The local scope is similar to the `let` construct in functional languages:

```
let x = 2 in
  x*x
end
```

which evaluates to 4,

```
3 * ( let x = 2 in
      x*x
    end )
```

which evaluates to 12, and

```
3 * ( let x = 2 in
      ( let y = x*x+1 in
        y/2
      end )
    + 1
  end )
```

evaluates to 9 (integer division rounds towards zero).

To implement the scoping of variable in our interpreter, we need to store name-value pairs. Note that when the interpreter executes a `let` it defines a new variable that is only used within the body of the `let`. Therefore, the natural choice for a name-value store would be a stack, where a new name-value pair is pushed on the stack when the interpreter enters the `let` and pops it off when it leaves the `let`. When the variable is used in the body of the `let` the interpreter searches the stack from the top (the last pair added) to the bottom to find the variable. This also allows for “shadowing” in scope rules when scopes are nested:

```
let x = 1 in
  ( let x = 3 in
    2*x
  end )
+ x
end
```

which evaluates to 7. Here the scope of the `x` in the outer `let` is hidden in the inner body of the inner `let`.

When the interpreter needs the value of a variable in an expression it looks it up in the stack from the top to the bottom to find the most recently added variable, which will be the one of the inner scope.

To do this, we extend this grammar by adding a new production:

GRAMMAR PRODUCTION

```
<expr1> -> let id = <expr2> in <push> <expr3> end <pop>
```

SEMANTIC RULES

```
push.name = id.name;
push.value = expr2.value;
expr1.value = expr3.value
```

and we use “dummy” nonterminals that represent the push and pop operations that need to be performed before and after the evaluation of the `let` body expression `expr3` as follows:

GRAMMAR PRODUCTION

```
<push> -> empty
```

```
<pop> -> empty
```

SEMANTIC RULES

```
stack.push(pair(push.name, push.value))
```

```
stack.pop()
```

Note that in this way the interpreter performs the push and pop operations cleverly while parsing the input, as these are embedded in the augmented grammar.

More specifically, the `stack` object we use is a global stack that is updated with push and pop operations and can be searched top-down with a `lookup` function.

In this project we will implement an interpreter for this language using a recursive-descent parsing technique. The recursive descent parser for the `let` construct has the following outline:

```

/* keywords recognized by the scanner */
#define LET 256
#define IN 257
#define END 258
/* the <expr> parser */
int expr()
{
    int result;
    if (lookahead() == LET)
    {
        char *name;
        int value;
        match(LET);           // let
        name = id();         // id
        match('=');          // =
        value = expr();      // <expr>
        match(IN);          // in
        push(name, value);   // <push>
        result = expr();     // <expr>
        match(END);         // end
        pop();              // <pop>
    }
    else
    {
        int value = term();  // <term>
        result = term_tail(value); // <term_tail>
    }
    return result;
}

```

You need to complete the recursive descent parser implementation for the augmented grammar. For this assignment you can write the code in C or C++.

You need to write a scanner first to tokenize the input. The scanner has two functions: `lookahead()` and `getnext()`. The `lookahead()` returns an integer that represents the current token. The `getnext()` moves to the next token on the input. The `match()` function can be written based on these:

```

void match(int token)
{
    if (lookahead() == token)
        getnext();
    else
        ... // report syntax error and exit
}

```

Note that if all tokens were single ASCII we can simply use `getchar()` for `getnext()` to read the next ASCII token:

```

int current = 0;
// getnext() gets the next token (simple version)
void getnext()
{
    do current = getchar(); while (isspace(current)); // skip space until next char
}
// lookahead() returns current token
int lookahead()
{
    if (current == 0) // first time around
        getnext();
    return current;
}

```

However, we also need to recognize the `let`, `in`, and `end` keywords, identifier names, and integer constants. For identifier names (variables), we need to store the current variable name and set `current` to `ID` or to `NUM` for integer constants:

```
#define ID 259
#define NUM 260
int current = 0;
int number;
char name[80]; // max identifier name length is 79
// getnext() improved version
void getnext()
{ int c;
  do c = getchar(); while (isspace(c));
  if (...)
  {
    // identifier:
    current = ID;
    // ... fill name[] with id name
  }
  else if (...)
  {
    // integer:
    current = NUM;
    // ... number = decoded value
  }
  else
    current = c;
}
```

You should also check for the keywords `let`, `in`, and `end` to set `current` to `LET`, `IN`, or `END`. Note that `EOF` is returned by `lookahead()` upon end of file, which will be used to stop the expression parser and print the calculated value.

Now implement the interpreter using the suggested scanner code shown above. You should test your scanner implementation first by calling `getnext()` in a loop and printing the `lookahead()` token:

```
while (lookahead() != EOF)
{ switch (lookahead())
  { case ID:
    printf("ID=%s\n", name);
    break;
    case NUM:
    printf("NUM=%d\n", number);
    break;
    default:
    printf("%c\n", lookahead());
  }
  getnext();
}
```

This should tokenize the `stdin` stream and print the results to `stdout`. When you are confident that the scanner works properly, you can start working on the next phase, the parser which also performs interpreter operations to evaluate expressions. The parser should use recursive descent. Implement a stack to push and pop name-value pairs and search the stack top-down for the value of a name. The input to the interpreter is read from the `stdin`. The value should be printed to `stdout`.

BONUS ASSIGNMENT

You can earn 10% extra credit as follows. Modify the grammar as follows to parse comma-separated expressions:

GRAMMAR PRODUCTION	SEMANTIC RULES
<code><comm> -> <expr> <rest></code>	<code>rest.total := expr.total; comm.value := rest.value</code>
<code><rest> -> , <comm></code>	<code>rest.value := comm.value</code>
<code> empty</code>	<code>rest.value := rest.total</code>

The new start symbol of the grammar is `comm` (this is where the parser will start). Note that `rest` has two attributes: `total` (inherited) and `value` (synthesized).

This allows parsing of comma-separated expressions, where the value of the list of expressions is the value of the last expression in this list. So the first values are simply ignored. This is similar to the comma operator in C. For example:

```
1, 2
```

evaluates to 2, and

```
3*(1, 2)
```

evaluates to 6.

Next, we add variable assignments to our language, similar to C. This way we can modify the value of a variable in a `let` construct:

```
let x = 0 in
  (x = x + 1, x)
end
```

which evaluates to 1, since `x` is assigned before the comma and the value of the comma expression is the value of `x`. Like in C, assignments can be used within expressions:

```
let x = 0 in
  x = 2 * (x = x + 1)
end
```

which evaluates to 2, since the value of the nested assignment is 1.

The productions for `factor` should be modified to handle the new comma operator within parenthesis and the assignment operator:

GRAMMAR PRODUCTION	SEMANTIC RULES
<code><factor1> -> '(' <comm> ')'</code>	<code>factor1.value := comm.value</code>
<code> '-' <factor2></code>	<code>factor1.value := -factor2.value</code>
<code> num</code>	<code>factor1.value := num.value</code>
<code> id</code>	<code>factor1.value := lookup(id.name)</code>
<code> id = <expr></code>	<code>assign(id.name, expr.value); factor1.value = expr.value</code>

Note that the last two productions have common prefixes, which invalidates the LL(1) property of the grammar. You will have to resolve this in your recursive descent parser. The `assign` operation changes the value of the variable on the stack. That is, it searches top-down in the stack for the variable name and modifies its value stored there.

– End