Universidade Estadual de Londrina

DC Departamento de Computação

5COP093 - Compilador C: Analisador Semântico C Simplificado

Neste ponto do trabalho, o compilador dever ser capaz de reconhecer erros léxicos, sintáticos e semânticos, sendo que o processo de compilação deve terminar quando o primeiro erro (léxico, sintático ou semântico) for encontrado.

Os arquivos de teste se dividirão em arquivos sem nenhum tipo de erro, arquivos somente com erros léxicos, arquivos somente com erros sintáticos, arquivos somente com erros semânticos e arquivos contendo uma combinação de erros (léxicos, sintáticos e semânticos).

O analisador semântico C deve reconhecer somente os erros semânticos apontados neste texto. Qualquer outra possibilidade de erro semântico que não tenha sido listada neste texto não precisa ser verificada, pois não fará parte do conjunto de testes.

Um novo tipo de saída que deve aparecer agora são os **warnings**, que nada mais são do que avisos sobre alguma condição que foi detectada no código, que embora não seja um erro, pode gerar resultados imprevistos durante a execução do programa. Quando o compilador detectar uma condição que gera um **warning**, ele deve emitir a mensagem apropriada e continuar o processo de compilação, ou seja, um **warning** não é um erro fatal que irá causar o término do processo de compilação.

Quando um erro léxico for detectado, deve-se mostrar a linha e a coluna onde o erro ocorreu.

Exemplo de entrada:

```
void f(){ if(@)
{
   a
}
```

Saída esperada:

```
error:lexical:1:14: @
```

Observe que na saída esperada, a mensagem de erro apresentada foi:

```
error:lexical:1:14: @
```

onde o primeiro número indica a linha onde o erro ocorreu e o segundo número indica a coluna onde o erro ocorreu. No exemplo dado, o erro ocorreu na linha 1, coluna 14. Observe também que deve ser impresso o caractere que causou o erro léxico.

Observe o exemplo a seguir:

```
void f(){ if(1)
{
/* isto eh um
comentario iniciado
e nao terminado
```

Saída esperada:

```
error:lexical:3:1: unterminated comment
```

No exemplo apresentado, o arquivo termina com um comentário de bloco que não foi fechado. Tal tipo de erro deve ser apresentado com a mensagem padrão do exemplo. Observe também que mesmo o arquivo contendo 3 linhas de comentário de bloco não terminado, a mensagem de erro irá mostrar a linha onde o comentário se inicia. No caso do exemplo, o comentário se inicia na linha 3. Mesmo que o comentário possua inúmeras linhas, o erro deve apontar a linha onde o comentário é iniciado. Tal como outros erros léxicos, também deve-se informar a coluna onde o erro ocorreu. No exemplo apresentado, o erro ocorreu na coluna 1.

Observe o exemplo a seguir:

```
void f()
{
    if(struct)
    {
        main();
    }
}
```

Saída esperada:

```
error:syntax:3:8: struct
    if(struct)
       ^
```

No exemplo apresentado ocorreu um erro sintático na linha 3, coluna 8. Observe que o *token* causador do erro também é impresso, no caso do exemplo, o *token* `struct`. Outra particularidade desta mensagem de erro é que na linha seguinte é impresso a linha de código onde ocorreu o erro sintático, sendo que na terceira linha da mensagem de erro o símbolo `^` é impresso indicando a coluna 8, ou seja, o ponto exato que causou o erro sintático.

Observe o exemplo a seguir:

```
int B,Q; void f() { /*comentario*/
```

Saída esperada:

```
error:syntax:1:35: expected declaration or statement at end of input
int B,Q; void f() { /*comentario*/
                                  ^
```

No exemplo apresentado não existe, em princípio, erro sintático, pois trata-se na verdade de um programa que foi iniciado e não terminado. Quando tal tipo de programa for compilado, deve-se exibir a mensagem `expected declaration or statement at end of input` ao invés de se apresentar um *token*. O programa do exemplo possui uma linha e 34 caracteres. Como deveria haver uma declaração ou comando após o último caractere, a coluna apresentada é a posterior ao último caractere do programa. Neste caso como o último caractere do programa fica na coluna 34, a mensagem de erro deve indicar a coluna 35. Observe que o símbolo ^ aponta para a coluna 35 ao imprimir a linha que causou o erro.

Observe o exemplo a seguir:

```
void f() { //comentario de linha
```

Saída esperada:

```
error:syntax:1:12: expected declaration or statement at end of input
void f() { //comentario de linha
           ^
```

Este exemplo também trata-se de um programa iniciado e não terminado que, além deste fato, não apresenta erro sintático anterior. O programa possui apenas uma única linha.

Da mesma forma que no exemplo anterior, deve-se exibir a mensagem `expected declaration or statement at end of input` ao invés de se apresentar um *token*.

Como o programa foi terminado com um comentário de linha, a declaração ou comando que deveria haver, deve aparecer antes do comentário de linha. Desta forma na mensagem de erro sintático, a coluna a ser indicada é a coluna onde se inicia o comentário de linha.

Observe o exemplo a seguir:

```
int main()
{
    int i;
    int i;
    printf("%d\n",i);
    return 0;
}
```

Saída esperada:

```
error:semantic:4:9: variable 'i' already declared, previous declaration in line 3 column 9
    int i;
        ^
```

No exemplo apresentado ocorreu um erro semântico na linha 4, coluna 9. Observe que em caso de erro semântico, deve-se informar a linha e a coluna onde ocorreu o erro acompanhado pela mensagem adequada informando o tipo de erro semântico.

Observe o exemplo a seguir:

```
#define constante 1 << 32

int main()
{
    int i;
    printf("%d\n",i);
    return 0;
}
```

Saída esperada:

```
warning:1:21: left shift count >= width of type
#define constante 1 << 32
                    ^
SUCCESSFUL COMPILATION.
```

No exemplo apresentado ocorreu um **warning** pois o valor do deslocamento à esquerda é maior ou igual que o tamanho (32 bits) do valor que sofrerá o deslocamento, resultando em um valor igual à zero. Observe também que na mensagem do **warning** é informada a linha e a coluna que gerou o mesmo, acompanhado da mensagem adequada.

Caso nenhum erro léxico, sintático ou semântico seja encontrado, o compilador deve imprimir a mensagem:

```
SUCCESSFUL COMPILATION.
```

Observe o exemplo a seguir:

```
void f()
{
    if(1)
    {
        666;
    }
}
```

Saída esperada:

```
SUCCESSFUL COMPILATION.
```

Compatibilidade de tipos entre operadores

A tabela a seguir apresenta as combinações possíveis de tipos que podem ocorrer, bem como cada combinação é tratada por cada um do operadores presentes na linguagem C simplificada. Nessa tabela, considere o seguinte:

✓: Implica que a operação é válida e não gera nenhum erro semântico ou `warning`.

•: Implica que a operação pode ou não ser válida, dependendo das condições, podendo ser gerado um erro semântico ou um `warning`. Informações adicionais aparecem após a tabela de compatibilidade.

Um espaço vazio implica que a combinação de tipos não é compatível para a operação apresentada e deve ser gerado um erro semântico.

Operadores Binários	int/int	char/char	int/char	ponteiro/int	ponteiro/char	ponteiro/ponteiro
+ PLUS	✓	✓	✓	✓	✓	
- MINUS	✓	✓	✓	✓	✓	
* MULTIPLY	✓	✓	✓			
/ DIV	✓	✓	✓			
% REMAINDER	✓	✓	✓			
& BITWISE_AND	✓	✓	✓			
| BITWISE_OR	✓	✓	✓			
^ BITWISE_XOR	✓	✓	✓			
&& LOGICAL_AND	✓	✓	✓	✓	✓	✓
|| LOGICAL_OR	✓	✓	✓	✓	✓	✓
== EQUAL	✓	✓	✓	•	•	•
!= NOT_EQUAL	✓	✓	✓	•	•	•
< LESS_THAN	✓	✓	✓	•	•	•
> GREATER_THAN	✓	✓	✓	•	•	•
<= LESS_EQUAL	✓	✓	✓	•	•	•
>= GREATER_EQUAL	✓	✓	✓	•	•	•
>> R_SHIFT	✓	✓	✓	•	•	
<< L_SHIFT	✓	✓	✓	•	•	
= ASSIGN	✓	✓	✓			•
+= ADD_ASSIGN	✓	✓	✓			•
-= MINUS_ASSIGN	✓	✓	✓			•

Operadores Unários	int	char	ponteiro
+ PLUS	✓	✓	
- MINUS	✓	✓	
* MULTIPLY			✓
++ INC	✓	✓	✓
-- DEC	✓	✓	✓
~ BITWISE_NOT	✓	✓	
! NOT	✓	✓	✓
& BITWISE_AND	✓	✓	✓

- Os operadores binários `==, !=, <, >, <=, >=` devem apresentar erro caso a comparação seja feita entre dois tipos diferentes de ponteiros como entre `int*` e `int**`, por exemplo. Para comparações entre um tipo ponteiro com um tipo `int` ou `char`, a comparação deve gerar um `warning`. Para ponteiros de mesmo tipo, nenhum erro ou `warning` é gerado. O valor retornado por uma comparação é do tipo `int`.
- Os operadores binários `<<` e `>>` quando utilizados com ponteiros, devem necessariamente ter o tipo ponteiro como o operando esquerdo e o tipo `int` ou `char` como o operando direito. A combinação inversa é inválida e deve gerar um erro semântico. O tipo retornado nesta operação é o tipo do ponteiro que está no lado esquerdo da operação.
- Os operadores binários de atribuição `=, +=, -=` quando utilizados para atribuir um valor para um ponteiro, requerem que a expressão do lado direito gere um ponteiro do mesmo tipo que o operando do lado esquerdo. Assim um identificador do tipo `int**` só pode receber uma expressão cujo resultado seja também do tipo `int**`. Atribuições entre tipos diferentes de ponteiros devem gerar erro semântico.

Especificações Semânticas do Compilador C Simplificado

No projeto do compilador `C` simplificado não existe nenhum pré-processador, desta forma a cláusula `#define` deve ser tratada como parte da linguagem. Neste compilador, toda expressão especificada após um identificador nas cláusulas `#define` deve gerar um valor inteiro que possa ser determinado em tempo de compilação. O identificador associado a expressão irá ter a função de uma constante, não podendo ter seu valor modificado pelo programa. Qualquer tentativa de modificação do valor de tal identificador deve gerar um erro semântico.

Uma `string` pode aparecer em uma expressão, sendo que ela irá corresponder ao tipo `char*`. Como a `string` não pode ser modificada em tempo de execução, o tipo `char*` associado a `string` funciona como uma constante, a qual não pode ser modificada, somente lida. Desta forma uma atribuição para tal constante gera um erro semântico, mesmo ela sendo do tipo `char*` cujo valor é o endereço da `string`.

Nas chamadas de função os parâmetros passados devem possuir o mesmo tipo do respectivo paramêtro declarado na função. Embora nas expressões existem operadores que aceitem combinações entre tipos diferentes sem nenhum problema, como por exemplo a soma entre um tipo `int` e `char`, nas chamadas de função essa conversão automática de tipo não acontece. Assim um parâmetro declarado como `int` só aceita expressões `int`; a utilização de uma expressão `char`, por exemplo, para este parâmetro irá causar um erro semântico.

Parâmetros de função declarados como `arrays` podem receber ponteiros que sejam equivalentes ao tipo base do `array`. Por exemplo um parâmetro que seja declarado como `int a[10][10]`, nada mais é que um ponteiro do tipo `int*`. Desta forma um identificador que tenha o tipo `int*` é compatível com o parâmetro `int a[10][10]` de uma função qualquer. É importante ressaltar que o contrário também é verdadeiro, ou seja, uma função que tenha um parâmetro do tipo `int*` pode receber um `array` cujo tipo base seja `int`, não importando quantas dimensões tal `array` possua.

Ainda em relação à `arrays`, um identificador que tenha sido declarado como `int** a;` pode ser utilizado em uma expressão como um `array` através da sintaxe `a[5][5]`, por exemplo. Já um identificador que tenha sido declarado como `int a[5][5]`, por exemplo, também pode ser acessado através da forma `a[1]` ou somente `a`, sendo que em ambos os casos, o tipo retornado é `int*`, pois não foram utilizadas todas as dimensões declaradas.

Os comandos condicionais `if`, `while`, `do-while` têm em comum o fato de utilizar uma expressão que deve ser avaliada como verdadeira ou falsa. Expressões que retornem o tipo `void` não são compatíveis com estes comandos e a sua utilização em uma condição de controle deve gerar um erro semântico.

O operador de `cast` é capaz de converter praticamente de qualquer tipo para qualquer tipo, mesmo que o resultado possa ser potencialmente perigoso, como em um `cast` do tipo `char` para um tipo ponteiro. Cabe ao programador ter consciência do que está fazendo e das devidas implicações. O operador de `cast` deve apenas gerar um `warning` quando a conversão do tipo de origem for para um tipo de menor tamanho, como por exemplo de um tipo ponteiro (32 bits) para `char` (8 bits). O operador `cast` só não conseguirá conveter um tipo `void` para um tipo que seja diferente de `void`, quando deve então gerar um erro semântico.

Os operadores binários `<<` e `>>` quando empregados em expressões válidas, sempre retornam o tipo do operando esquerdo. Assim a expressão `'a'<< 2` retorna o tipo `char`.

Demais operadores binários que utilizarem uma combinação entre os tipos `int` e `char` irão retornar o tipo `int`, como por exemplo nas expressões `'c' + 2` ou `'c' * 'a'`, onde ambas irão retornar o tipo `int`.

Erros Semânticos

Os seguintes erros semânticos devem ser detectados pelo compilador:

Erros em declarações de variáveis/funções/protótipos

```
"redefinition of '<nome-do-identificador>' previous defined in line <numero-da-linha>
 column <numero-da-coluna>"
"variable '<nome-da-variavel>' declared void"
"variable '<nome-da-variavel>' already declared, previous declaration in line <numero-da-linha>
 column <numero-da-coluna>"
"parameter '<nome-do-parametro>' declared void"
"argument '<nome-do-parametro>' does not match prototype"
"prototype for '<nome-da-funcao>' declares fewer arguments"
"prototype for '<nome-da-funcao>' declares more arguments"
"conflicting types for '<nome-da-funcao>'"
"size of array '<nome-do-array>' is negative"
"size of array '<nome-do-array>' is zero"
"incompatible types in initialization when assigning to type '<nome-do-tipo>'
 from type '<nome-do-tipo>'"
```

Erros em chamadas de função

```
"called object '<nome-do-identificador>' is not a function or function pointer"
"'<nome-do-identificador>' undeclared"
"too few arguments to function '<nome-da-funcao>'"
"too many arguments to function '<nome-da-funcao>'"
"incompatible type for argument '<numero-do-argumento>' of '<nome-da-funcao>'
 expected '<nome-do-tipo>' but argument is of type '<nome-do-tipo>'"
```

Erros em retornos de função

```
"no return statement in function returning non-void"
"return with a value, in function returning void"
"return with no value, in function returning non-void"
"incompatible types when returning type '<nome-do-tipo>' but '<nome-do-tipo>' was expected"
```

Erros na verificação de tipos em comandos/expressões

```
"’<nome-do-identificador>’ undeclared"
"lvalue required as increment operand"
"lvalue required as left operand of assignment"
"’<nome-do-identificador>’ initializer element is not constant"
"right shift count is negative"
"left shift count is negative"
"string type is not compatible with define"
"division by zero"
"void value not ignored as it ought to be"
"subscripted value is neither array nor pointer"
"lvalue required as unary ’<operador-unario>’ operand"
"invalid type argument of unary ’<operador-unario>’ (have ’<nome-do-tipo>’)"
"array subscript is not an integer"
"comparison between ’<nome-do-tipo>’ and ’<nome-do-tipo>’ operator ’<operador-binario>’"
"cannot convert from ’<nome-do-tipo>’ to int"
"assignment of read-only location <string>"
"assignment of read-only location ’<caractere>’"
"incompatible types when assigning to type ’<nome-do-tipo>’ from type ’<nome-do-tipo>’"
"invalid operands to binary ’<operador-binario>’ (have ’<nome-do-tipo>’ and ’<nome-do-tipo>’)"
"wrong type argument to unary plus"
"wrong type argument to unary minus"
```

Warnings

As seguintes mensagens de aviso devem ser geradas pelo compilador:

```
"left shift count >= width of type"
"right shift count >= width of type"
"array index out of bounds"
"comparison between ’<nome-do-tipo>’ and ’<nome-do-tipo>’ operator ’<operador-binario>’"
"cast from ’<nome-do-tipo>’ to ’<nome-do-tipo>’ of different size"
"’<nome-do-tipo>’/’<nome-do-tipo>’ type mismatch in conditional expression"
```

Exemplos de Mensagens de Erro e Warnings

Esta parte da especificação irá apresentar exemplos onde são geradas mensagens de erros semânticos bem como `warnings`. A contagem da numeração das linhas de cada código fonte exemplo se inicia sempre no primeiro comando escrito.

```
void f()
{
   int i;
   int i;

   return;
}

error:semantic:4:8: variable 'i' already declared, previous declaration in line 3 column 8
   int i;
       ^
```

```
void f()
{
    int i;
    char i;

    return;
}

error:semantic:4:10: redefinition of 'i' previous defined in line 3 column 9
    char i;
         ^
```

```
void f()
{
    void i;

    return;
}

error:semantic:3:10: variable 'i' declared void
    void i;
         ^
```

```
void f()
{
    int i[0];

    return;
}

error:semantic:3:9: size of array 'i' is zero
    int i[0];
        ^
```

```
#define v1 2
#define v2 5
#define v3 10
#define v4 1
int i[v3-v1*v2-v4];

error:semantic:5:5: size of array 'i' is negative
int i[v3-v1*v2-v4];
    ^
```

```
void f(int a);

void f(int a, int b)
{
    int i[0];

    return;
}

error:semantic:3:6: prototype for 'f' declares fewer arguments
void f(int a, int b)
     ^
```

```
void f(int a);

void f(char a)
{
    int i[0];

    return;
}

error:semantic:3:13: argument 'a' does not match prototype
void f(char a)
            ^
```

```
int f(char* a);

void f(char* a)
{
    int i[0];

    return;
}

error:semantic:3:6: conflicting types for 'f'
void f(char* a)
     ^
```

```
int f(char* a)
{
    int i;

    i = 0;
}

error:semantic:1:5: no return statement in function returning non-void
int f(char* a)
    ^
```

```
void f(char* a)
{
    return 0;
}

error:semantic:1:6: return with a value, in function returning void
void f(char* a)
     ^
```

```
int* f(char* a)
{
    return "SkyNet Online";
}

error:semantic:3:5: incompatible types when returning type 'char*' but 'int*' was expected
    return "SkyNet Online";
    ^
```

```
int* f(char* a)
{
    return;
}

error:semantic:3:5: return with no value, in function returning non-void
    return;
    ^
```

```
int* f(char* a)
{
    return i;
}

error:semantic:3:12: 'i' undeclared
    return i;
           ^
```

```
void f()
{
    666++;
}

error:semantic:3:8: lvalue required as increment operand
    666++;
       ^
```

```
void f()
{
    666 = "SkyNet Online";
}

error:semantic:3:9: lvalue required as left operand of assignment
    666 = "SkyNet Online";
        ^
```

```
int i;
#define constante i+1

error:semantic:2:19: 'i' initializer element is not constant
#define constante i+1
                  ^
```

```
int j;
int i[j];

error:semantic:2:7: 'j' initializer element is not constant
int i[j];
      ^
```

```
#define v1 10
#define v2 15
#define constante 1 << (v1-v2)

error:semantic:3:21: left shift count is negative
#define constante 1 << (v1-v2)
                    ^
```

```
#define constante "Adeus Mundo!"

error:semantic:1:19: string type is not compatible with define
#define constante "Adeus Mundo!"
                  ^
```

```
#define v1 2
#define v2 5
#define v3 10
#define constante 1/(v3-v1*v2)

error:semantic:4:20: division by zero
#define constante 1/(v3-v1*v2)
                   ^
```

```
void f()
{
    int i;
    i = f();
    return;
}

error:semantic:4:7: void value not ignored as it ought to be
    i = f();
      ^
```

```
void f()
{
    int i;
    if(f()) { i++; }
    return;
}

error:semantic:4:8: void value not ignored as it ought to be
    if(f()) { i++; }
       ^
```

```
void f()
{
    int i[10],j;

    j = i[1][1];
}

error:semantic:5:13: subscripted value is neither array nor pointer
    j = i[1][1];
            ^
```

```
void f()
{
    int i[10],j;

    j = i[1](1);
}

error:semantic:5:9: called object 'i' is not a function or function pointer
    j = i[1](1);
        ^
```

```
void f()
{
    "SkyNet Online" = 666;
}

error:semantic:3:21: assignment of read-only location "SkyNet Online"
    "SkyNet Online" = 666;
                    ^
```

```
void foo(int i,int j);

void f()
{
    foo(3);
}

error:semantic:5:5: too few arguments to function 'foo'
    foo(3);
    ^
```

```
void foo(int i,int j);

void f()
{
    char c;
    foo(3,&c);
}

error:semantic:6:5: incompatible type for argument ’2’ of ’foo’ expected ’int’ but
argument is of type ’char*’
    foo(3,&c);
    ^
```

```
void f()
{
    int* i = &666;
    return;
}

error:semantic:3:14: lvalue required as unary ’&’ operand
    int* i = &666;
             ^
```

```
void f()
{
    int* p;

    +p;
    return;
}

error:semantic:5:5: wrong type argument to unary plus
    +p;
    ^
```

```
void f()
{
    int p;

    *p;
    return;
}

error:semantic:5:5: invalid type argument of unary '*' (have 'int')
    *p;
    ^
```

```
void f()
{
    int* p;
    int i[10];

    i[p];
    return;
}

error:semantic:6:6: array subscript is not an integer
    i[p];
      ^
```

```
void f()
{
    int* p;
    int i;
    i << p;
    return;
}

error:semantic:5:7: cannot convert from 'int*' to int
    i << p;
      ^
```

```
void f()
{
    int *p,i;
    i = p;
    return;
}

error:semantic:4:7: incompatible types when assigning to type 'int' from type 'int*'
    i = p;
      ^
```

```
void f()
{
    int i = &i >= "teste";
    return;
}

error:semantic:3:16: comparison between 'int*' and 'char*' operator '>='
    int i = &i >= "teste";
                ^
```

```
void f()
{
    int *i,*j;
    i + j;
    return;
}

error:semantic:4:7: invalid operands to binary '+' (have 'int*' and 'int*')
    i + j;
      ^
```

```
#define constante 1 >> 32

warning:1:21: right shift count >= width of type
#define constante 1 >> 32
                    ^
SUCCESSFUL COMPILATION.
```

```
#define c1 1 >> 32
#define c2 1 << 33

warning:1:14: right shift count >= width of type
#define c1 1 >> 32
             ^
warning:2:14: left shift count >= width of type
#define c2 1 << 33
             ^
SUCCESSFUL COMPILATION.
```

```
#define c1 2
#define c2 5
int v[10];
int i = v[c1*c2] + c1;

warning:4:10: array index out of bounds
int i = v[c1*c2] + c1;
         ^
SUCCESSFUL COMPILATION.
```

```
void f()
{
    int* p,i;

    i = p > i;
}

warning:5:11: comparison between 'int*' and 'int' operator '>'
    i = p > i;
          ^
SUCCESSFUL COMPILATION.
```

```
void f()
{
    char c;
    int i;
    int v[10];

    int* p;


    for(p=&(v[0]);p;p++)
    {
        printf("Conteudo: %d",*p);
    }

    c = (char) i; /*meio certo*/  i << -1; /*errado*/
}

warning:15:9: cast from 'int' to 'char' of different size
    c = (char) i; /*meio certo*/  i << -1; /*errado*/
        ^
error:semantic:15:37: left shift count is negative
    c = (char) i; /*meio certo*/  i << -1; /*errado*/
                                      ^
```

```
void f()
{
    char c;
    int i;
    int* p;
    p = i?&i:&c;
}

warning:6:13: 'int*'/'char*' type mismatch in conditional expression
    p = i?&i:&c;
            ^
SUCCESSFUL COMPILATION.
```

```
void f()
{
    char c;
    int i;
    c = (char)(i?&i:&c);
}

warning:5:20: 'int*'/'char*' type mismatch in conditional expression
    c = (char)(i?&i:&c);
                    ^
warning:5:9: cast from 'int*' to 'char' of different size
    c = (char)(i?&i:&c);
        ^
SUCCESSFUL COMPILATION.
```

```
void f()
{
    char c;
    int i,*p;
    p = (char)(i?&i:&c);
}

warning:5:20: 'int*'/'char*' type mismatch in conditional expression
    p = (char)(i?&i:&c);
                    ^
warning:5:9: cast from 'int*' to 'char' of different size
    p = (char)(i?&i:&c);
        ^
error:semantic:5:7: incompatible types when assigning to type 'int*' from type 'char'
    p = (char)(i?&i:&c);
      ^
```

```
#define valor_negativo -15
#define valor_positivo +15
void f()
{
    int* v[valor_positivo];
    v[valor_negativo] =
    v[-valor_positivo];

    v[valor_negativo]
    =
    2
    <<
    valor_negativo;
}

warning:6:6: array index out of bounds
    v[valor_negativo] =
     ^
warning:7:6: array index out of bounds
    v[-valor_positivo];
     ^
warning:9:6: array index out of bounds
    v[valor_negativo]
     ^
error:semantic:12:5: left shift count is negative
    <<
    ^
```

```
int main()
{
    if(3?(void)4:(void)5)
    {
        666;
    }
    return 0;
}

warning:3:10: cast from 'int' to 'void' of different size
    if(3?(void)4:(void)5)
         ^
warning:3:18: cast from 'int' to 'void' of different size
    if(3?(void)4:(void)5)
                 ^
error:semantic:3:17: void value not ignored as it ought to be
    if(3?(void)4:(void)5)
                ^
```

```
char c = "character";
int l=66,i = (void*) 0;
int z,w=5,x = (void)3;

int main()
{
    return 0;
}

error:semantic:1:8: incompatible types in initialization when assigning to
type 'char' from type 'char*'
char c = "character";
       ^
```

```
int z,w=5,x = (void)3;

int main()
{
    return 0;
}

warning:1:15: cast from 'int' to 'void' of different size
int z,w=5,x = (void)3;
              ^
error:semantic:1:13: void value not ignored as it ought to be
int z,w=5,x = (void)3;
            ^
```

```
int main()
{
    int i = (void) 0;

    c = i;
    i = c;

    return 0;
}

warning:3:13: cast from 'int' to 'void' of different size
    int i = (void) 0;
            ^
error:semantic:3:11: void value not ignored as it ought to be
    int i = (void) 0;
          ^
```

```
void f()
{
    int i;
    i++;
}

int main()
{
    char c = 'c';
    int w = (int)c;

    (void)f();
    (void)((void)f());

    (void) 666;
    (void*) 666;
    (void) 'z';
    (void)(char)(int)(char)77;

    w = (int) f();

    return 0;
}

warning:15:5: cast from 'int' to 'void' of different size
    (void) 666;
    ^
warning:18:22: cast from 'int' to 'char' of different size
    (void)(char)(int)(char)77;
                     ^
warning:18:11: cast from 'int' to 'char' of different size
    (void)(char)(int)(char)77;
          ^
error:semantic:20:9: void value not ignored as it ought to be
    w = (int) f();
        ^
```

```
#define c1 10
#define c2 -c1
#define c3 c1 << 2
#define c4 c2 >> c3*c3
#define NULL 0
#define CALL_ELEVEN 11
int v[c2+c3];

void* f()
{
    v[c4],c1>v,(char)c2,NULL?v:c3;
    return (void*) 0;
}

#define TRUE  1
#define FALSE 0
#define SERA_VERDADE (TRUE || FALSE)?TRUE:FALSE
char* vetor[SERA_VERDADE];

char g()
{
    void* v = (void*) "DDominar o mundo\n" + 1;
    printf("%s",((char*)v) >> 666);
    v=(void*)(v?((void*)((char)((char)(int*)666<(char)(void*)11))):(char*)"Mundo Invertido");
    return (char)vetor[1 << 1];
}

int main()
{
    void* p = f(), *vp = (void*) 0, *ptr = (void*)NULL;
    char  c = g(), ch = '\r',  d = (char) 65; int* i = (int*) &p;
    for(ptr = (void*)((char)((int)666)); ptr;ptr++)
    {
        while(ptr)
        {
            if(ptr<=(void*)0)
            {
                ptr = (void*) &c;
                *i/*666*/; *i/ 666/*Porta para o mundo invertido*/; (char)CALL_ELEVEN;
            }
            else
            {
                do{ ptr++; }while(v);
            }
        }
    }
    return (((int)ptr) + *i);
}
```

```
warning:4:15: right shift count >= width of type
#define c4 c2 >> c3*c3
              ^
warning:11:6: array index out of bounds
    v[c4],c1>v,(char)c2,NULL?v:c3;
     ^
warning:11:13: comparison between 'int' and 'int*' operator '>'
    v[c4],c1>v,(char)c2,NULL?v:c3;
            ^
warning:11:16: cast from 'int' to 'char' of different size
    v[c4],c1>v,(char)c2,NULL?v:c3;
               ^
warning:11:31: 'int*'/'int' type mismatch in conditional expression
    v[c4],c1>v,(char)c2,NULL?v:c3;
                               ^
warning:23:28: right shift count >= width of type
    printf("%s",((char*)v) >> 666);
                           ^
warning:24:33: cast from 'int*' to 'char' of different size
    v=(void*)(v?((void*)((char)((char)(int*)666<(char)(void*)11))):(char*)"Mundo Invertido");
                                 ^
warning:24:49: cast from 'void*' to 'char' of different size
    v=(void*)(v?((void*)((char)((char)(int*)666<(char)(void*)11))):(char*)"Mundo Invertido");
                                                ^
warning:24:26: cast from 'int' to 'char' of different size
    v=(void*)(v?((void*)((char)((char)(int*)666<(char)(void*)11))):(char*)"Mundo Invertido");
                         ^
warning:24:67: 'void*'/'char*' type mismatch in conditional expression
    v=(void*)(v?((void*)((char)((char)(int*)666<(char)(void*)11))):(char*)"Mundo Invertido");
                                                                   ^
warning:25:23: array index out of bounds
    return (char)vetor[1 << 1];
                       ^
warning:25:12: cast from 'char*' to 'char' of different size
    return (char)vetor[1 << 1];
           ^
warning:31:36: cast from 'int' to 'char' of different size
    char  c = g(), ch = '\r',  d = (char) 65; int* i = (int*) &p;
                                     ^
warning:32:23: cast from 'int' to 'char' of different size
    for(ptr = (void*)((char)((int)666)); ptr;ptr++)
                       ^
warning:39:69: cast from 'int' to 'char' of different size
                *i/*666*/; *i/ 666/*Porta para o mundo invertido*/; (char)CALL_ELEVEN;
                                                                    ^
SUCCESSFUL COMPILATION.
```

Observações

Para este trabalho considere os seguintes tamanhos para cada um dos tipos:

- `void` 8 bits
- `char` 8 bits
- `int` 32 bits
- `ponteiro` 32 bits

A mensagem de erro semântico:

```
"comparison between '<nome-do-tipo>' and '<nome-do-tipo>' operator '<operador-binario>'"
```

se aplica somente aos operadores de comparação `==`, `!=`, `<`, `>`, `<=`, `>=`.

A mensagem de erro semântico:

```
"invalid operands to binary '<operador-binario>' (have '<nome-do-tipo>' and '<nome-do-tipo>')"
```

se aplica somente aos operadores binários `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`

A mensagem de erro semântico:

```
"incompatible types in initialization when assigning to type '<nome-do-tipo>'
 from type '<nome-do-tipo>'"
```

se aplica somente na inicialização de variáveis (globais e locais) durante a sua declaração, como no código a seguir:

```
int* i = 1;

void f()
{
    int* j = 2;

    return;
}
```

No código apresentado, tanto a inicialização de `i` e `j` irão gerar a mensagem de erro semântico aqui apresentada.

A mensagem de `warning`:

```
"comparison between '<nome-do-tipo>' and '<nome-do-tipo>' operator '<operador-binario>'"
```

se aplica somente aos operadores de comparação `==`, `!=`, `<`, `>`, `<=`, `>=` e deve ser gerada sempre que um dos operandos for do tipo ponteiro e o outro operando for um `int` ou `char`.

As mensagens de erro semântico:

```
"redefinition of '<nome-do-identificador>' previous defined in line <numero-da-linha>
 column <numero-da-coluna>"

"variable '<nome-da-variavel>' already declared, previous declaration in line <numero-da-linha>
 column <numero-da-coluna>"

"incompatible types in initialization when assigning to type '<nome-do-tipo>'
 from type '<nome-do-tipo>'"

"incompatible type for argument '<numero-do-argumento>' of '<nome-da-funcao>'
 expected '<nome-do-tipo>' but argument is of type '<nome-do-tipo>'"
```

embora sejam apresentadas em duas linhas neste texto, devem gerar uma única linha na saída do compilador. Elas foram quebradas em duas linhas neste texto, pois não couberam em uma única linha na apresentação de sua especificação.

A mensagem de `warning`:

```
"'<nome-do-tipo>'/'<nome-do-tipo>' type mismatch in conditional expression"
```

se aplica somente ao operador ternário `?`. Considere o seguinte exemplo:

```
void f()
{
    char c;
    int i;
    int* p;
    p = i?&i:&c;
}

warning:6:13: 'int*'/'char*' type mismatch in conditional expression
    p = i?&i:&c;
            ^
SUCCESSFUL COMPILATION.
```

No código apresentado a expressão `i` é utilizada para determinar se o operador ternário `?` será verdadeiro ou falso. Caso seja verdadeiro, ele irá retornar o resultado da expressão `&i` que possui tipo `int*`. Caso seja falso, ele irá retornar o resultado da expressão `&c` que possui tipo `char*`.

Como os tipos `int*` e `char*` são diferentes, o `warning` é gerado para avisar de tal diferença. Caso os tipos retornados fossem idênticos, não importando qual seja, nenhum `warning` seria gerado.

No operador ternário caso os tipos que aparecem nas expressões a serem retornadas sejam diferentes, deve-se retornar o tipo "maior". Para comparação de tipos, considere a seguinte relação de ordem:

$$\texttt{ponteiro} > \texttt{int} > \texttt{char} > \texttt{void}$$

Para ponteiros, tanto maior será um tipo, quanto maior a quantidade de dereferências. Desta forma tem-se:

`int**` > `int*`
`char**` > `int*`
`void***` > `int**`

Para ponteiros com a mesma quantidade de dereferências, aplica-se a relação de ordem dos tipos básicos. Desta forma tem-se:

`int**` > `char**`
`char*` > `void*`
`int***` > `void***`

Ordem de Verificação dos Erros/Warnings

O compilador deve finalizar o processo de compilação ao encontrar o primeiro erro, seja ele léxico, sintático ou semântico. Se um `warning` for gerado, o processo de compilação deve continuar.

Declarações de Variáveis e Constantes

Declarações de variáveis globais e constantes (`#define`) são verificadas na ordem em que aparecem no código.

Expressões Matemáticas

Em expressões matemáticas, a verificação deve ser realiza em pós-ordem. As mensagens de `warning` que porventura sejam emitidas para uma mesma expressão matemática, devem ser geradas na ordem em que as condições geradoras forem encontradas durante a verificação seguindo o percurso em pós-ordem. O mesmo acontece com erros semânticos, os quais são gerados durante a verificação em pós-ordem. Desta forma, uma mesma expressão matemática pode gerar um `warning` (ou mesmo vários) e um **erro semântico**, bastando apenas que no percurso em pós-ordem a condição geradora do `warning` apareça primeiro.

Ainda em relação a verificação de expressões matemáticas, quando um erro semântico for detectado, a linha/coluna informada na mensagem de erro devem indicar o nó que detectou o erro. Considere o código a seguir com a respectiva saída:

```
void f()
{
    int i;
    i = i + f();
}

error:semantic:4:11: void value not ignored as it ought to be
    i = i + f();
          ^
```

A função `f` possui tipo de retorno `void` o qual não possui valor a ser utilizado em uma expressão matemática. Durante o percurso em pós-ordem na árvore sintática abstrata, o nó do operador `+` irá verificar se os tipos dos filhos são compatíveis para realizar a adição e irá verificar que o tipo do filho direito é inválido para a operação. Desta forma a mensagem de erro semântico indica a linha/coluna do operador `+` na mensagem de erro.

Funções e Comandos

Na declaração de funções o abre parênteses para os parâmetros inicia um novo escopo, desta forma um parâmetro pode ter o mesmo nome de uma variável global, por exemplo, sem que isso cause um erro. Na declaração de uma função, deve-se seguir a seguinte ordem de verificação:

- Verificar se o identificar da função já foi utilizado.
- Verificar se já existe um protótipo para a função e em caso positivo, se a declaração da função e do protótipo coincidem.
- Verificar o tipo de retorno da função, onde este item irá se adiantar um pouco em relação a outras verificações.
    1. **Função retornando void:** Se a função tem tipo de retorno `void`, então o código da função não deve possuir o comando `return` ou, se o possuir, ele não deve conter nenhuma expressão como parâmetro. Caso o comando `return` contenha uma expressão para ser retornada, o erro semântico adequado deve ser gerado.
    2. **Função retornando algum valor:** Se a função retorna algum tipo de valor, então deve-se verificar se existe o comando `return`. Se o comando `return` não existir, o erro semântico adequado deve ser gerado. Caso exista o comando `return` ele deve obrigatoriamente conter alguma expressão a ser retornada. Se nenhuma expressão for passada como parâmetro para o comando `return` o erro semântico adequado deve ser gerado. Caso o comando `return` contenha uma expressão, nenhum erro é gerado neste momento. A verificação de se o tipo de retorno da expressão do comando `return` e da função coincidem **não** é realizada neste momento. Tal verificação só será realizada posteriormente ao se analisarem os comandos dentro da função.
- Verificar as declarações de variáveis locais na ordem em que aparecem junto com a expressão de inicialização, caso exista.
- Verificar os comandos declarados dentro da função na ordem em que aparecem. Em relação ao comando `return`, é somente nesta etapa que se verifica, se for o caso, que se a função retorna algum valor, o tipo da expressão passada como parâmetro para o comando `return` deve ser idêntico ao tipo de retorno da função, caso contrário o devido erro semântico é gerado.

Recomendações

Por favor, evite escrever código da seguinte forma:

```
for(int i = 0; ...)
{
    ...
}
```

onde uma variável local está sendo declarada dentro de um comando. Se ainda sim quiser utilizar tal estilo de programação, não se esqueça de colocar no `Makefile` as devidas opções para que o compilador aceite tal construção, pois nem todos os compiladores a aceitam por padrão.

Em geral a opção `-std=c99` é o suficiente para que tal construção seja aceita e compilada sem maiores problemas. Sua utilização, em geral, é da seguinte forma:

```
$gcc -std=c99 teste.c -o teste
```

Se você programar utilizando `C++ 11`, por favor, utilize também a opção adequada do compilador para habilitar a compilação de tal versão do `C++`.

IMPORTANTE: Se ficou com alguma dúvida em relação a qualquer item deste texto, não hesite em falar com o professor da disciplina, pois ele está à disposição para sanar eventuais dúvidas, além do que, isso faz parte do trabalho dele.

Especificações de Entrega

O trabalho deve ser entregue no *moodle* em um arquivo .zip com o nome `semantico.zip`. Este arquivo .zip deve conter somente os arquivos necessários à compilação, sendo que deve haver um `Makefile` para a geração do executável.

A entrega deve ser feita exclusivamente no *moodle* até a data/hora especificada. Não serão aceitas entregas atrasadas ou por outro meio que não seja o *moodle*.

Observação: o arquivo .zip não deve conter pastas, para que quando descompactado, os fontes do trabalho apareçam no mesmo diretório do .zip. O nome do executável gerado pelo `Makefile` deve ser `semantico`.

O programa gerado deve ler as suas entradas da entrada padrão do sistema e imprimir as saídas na saída padrão do sistema. Um exemplo de execução para uma entrada chamada `teste.c` seria a seguinte:

```
$./semantico < teste.c
```

Se o programa que for fornecido como entrada estiver correto, a seguinte mensagem deve ser impressa:

```
SUCCESSFUL COMPILATION.
```

Nenhuma linha extra deve ser gerada após a mensagem, ou seja, essa linha seria gerada pelo comando:

```
printf("SUCCESSFUL COMPILATION.");
```

Para erros léxicos, sintáticos, semânticos e warnings também não deve haver linhas extras na mensagem gerada.

IMPORTANTE: Arquivos ou programas entregues fora do padrão receberão nota **ZERO**. Entende-se como arquivo fora do padrão aquele que tenha um nome diferente de `semantico.zip`, que contenha subpastas ou que não seja um `.zip` por exemplo. Entende-se como programa fora do padrão aquele que não contiver um `Makefile`, que apresentar erro de compilação, que não ler da entrada padrão, não imprimir na saída padrão ou o nome do executável for diferente de `semantico`, por exemplo. Uma forma de verificar se seu arquivo ou programa está dentro das especificações é testar o mesmo com o `script` de testes que é fornecido no *moodle*. Se o seu arquivo/programa **não** funcionar com o `script`, significa que ele está **fora** das especificações e, portanto, receberá nota **ZERO**.