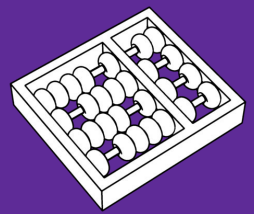
Analisador Semântico

Análise do Código

Hervé Yviquel
herve@ic.unicamp.br

Universidade Estadual de Campinas (Unicamp)
Instituto de Computação (IC)
Laboratório de Sistemas de Computação (LSC)

MC921 • Projeto e Construção de Compiladores • 2022 S2



UNICAMP

Aula Anterior

Resumo

- Análise Ascendente
  - Derivações/Reduções
- Parser LR
  - Usa pilha
  - Ações Shift e Reduce
- Construção do LR(0)
  - Autômato e tabela
- Conflitos
  - Shift-Reduce
  - Reduce-Reduce
- SLR
  - FOLLOW do reduce

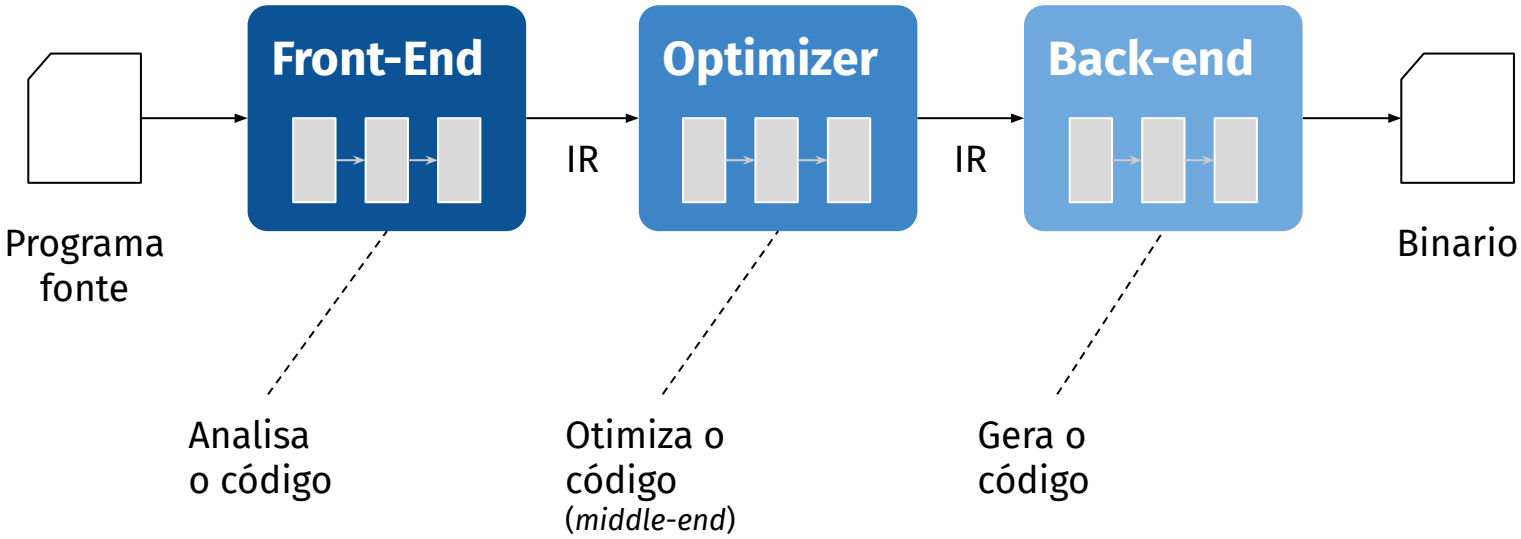
Aula de Hoje

Plano

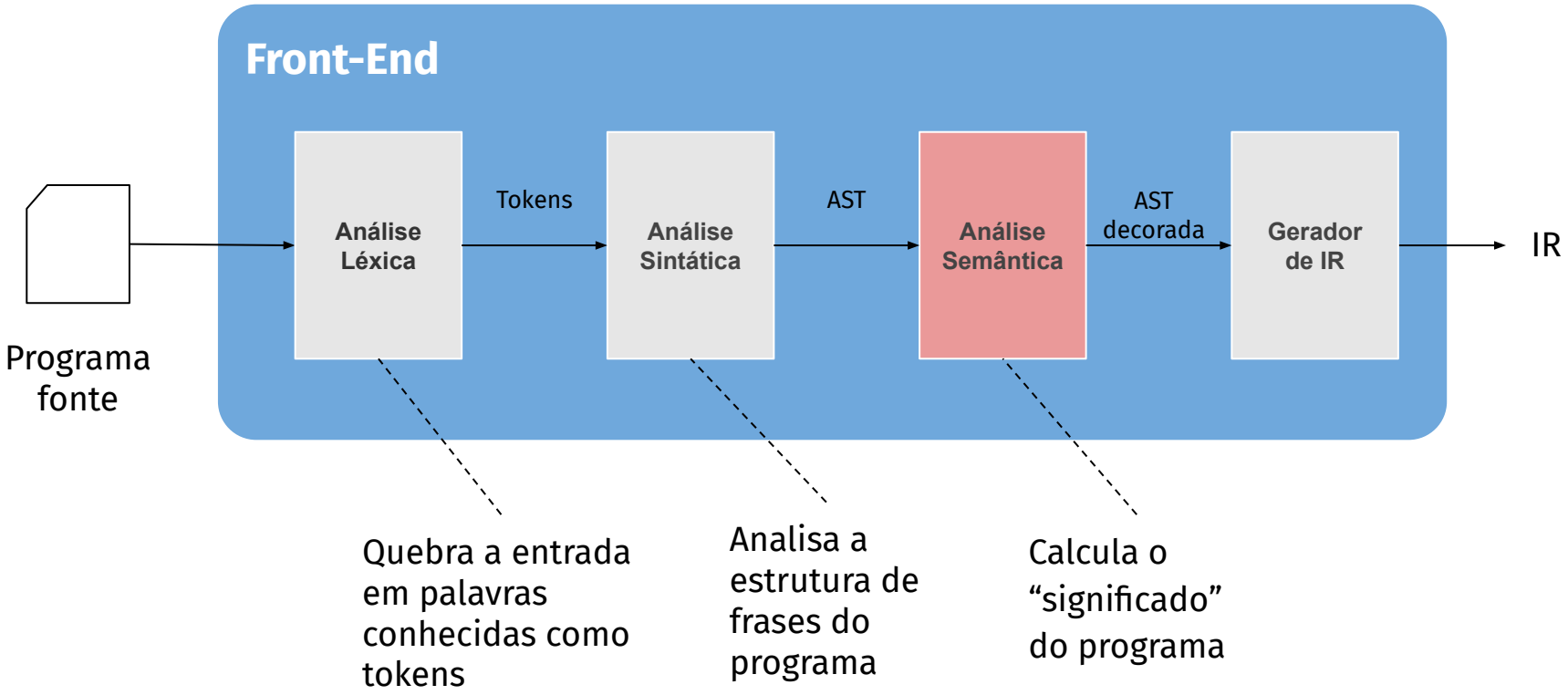
- Visão Geral do Front-end
- Analisador Semântico
- Symbol table
- Type Checking
- Visitor
- Erros semânticos

Visão Geral do Front-end

Estrutura Moderna do Compilador



Front-end do Compilador



Análise Semântica

Análise Semântica

- Ela verifica se o uso dos elementos do programa definidos pelo programador é compatível com a definição dos mesmos
- É necessário manter as informações relativas à declaração desses elementos numa base de dados que é consultada durante a verificação
  - Essa base de dados é chamada de Tabela de Símbolos (ou *Symbol Table*)

Erros Sintáticos x Semânticos

- As gramáticas nos permitem construir analisadores que reconhecem a estrutura sintática das linguagens
  - Qualquer programa que não esteja de acordo com a estrutura descrita pela gramática é rejeitado pelo analisador
  - Chamamos esses erros de “erros sintáticos”
- Erros semânticos são erros no comportamento do programa e não podem ser detectados pelo analisador sintático
  - Programas com erros semânticos geralmente são sintaticamente corretos
  - Uma certa classe desses erros semânticos pode ser capturada pelo interpretador/compilador

Exemplo de Erro Semântico


```
int x = 10;
x++;
int x = 10;
x++;
```

- Aqui estamos re-declarando a variável 'x' que não é possível em muitas linguagens de programação
- Muitos outros erros semânticos não podem ser detectados pelo interpretador/compilador e aparecem como “bugs” no programa

Erros Semânticos

Erros Semânticos


```
int a = x;
```

Erro
x is not defined

Erros Semânticos


```
int arr[3];
int b = arr['p'];
```

Erro
Subscript must be of type(int), not type(char)

Erros Semânticos


```
void f() {
    assert 2;
};
```

Erro
Expression must be of type(bool)

Erros Semânticos


```
void f() {
    int a;
    a = 'c';
}
```

Erro
Cannot assign type(char) to type(int)

Erros Semânticos


```
void f (int a, char b) {
    if (a > b) {}
}
```

Erro
Binary operator > does not have matching LHS/RHS types

Erros Semânticos


```
char c = 'a' + 'b';
```

Erro
Binary operator +
is not supported by
type(char)

Erros Semânticos


```
void f () {
    break;
}
```

Erro
Break statement must be inside a loop

Erros Semânticos


```
int arr[2][];
```

Erro
Array dimension mismatch

Erros Semânticos


```
int var = 'a';
```

Erro
var initialization type mismatch

Erros Semânticos


```
int matrix[3][3] = {{1, 2, 3}, {4, 5}};
```

Erro
Lists have different sizes

Erros Semânticos


```
int arr[4] = {4, 5, 6};
```

Erro
List & variable have different sizes

Erros Semânticos


```
void f () {
    if (2) {
    }
}
```

Erro
conditional expression
is type(int),
not type(bool)

Erros Semânticos


```
void f(int i) {
    i(3);
}
```

Erro
i is not a function

Erros Semânticos


```
int sum (int a, int b)
{
    return a + b;
}

void f() {
    sum(1, 2, 3);
}
```

Erro
No. arguments to call sum function mismatch

Erros Semânticos


```
int sum (int a, int b) {
    return a + b;
}

void f() {
    sum(1, '1');
}
```

Erro
Type mismatch with parameter b

Erros Semânticos


```
void f() {
    int i = 0;
    if (i) {
    }
}
```

Erro
The condition expression must be of type(bool)

Erros Semânticos


```
void f() {
    int i = 1;
    int y[1] = {i};
}
```

Erro
Expression must be a constant

Erros Semânticos


```
void a() {
      return;
}

void f() {
      print(a());
}
```

Erro
Expression is not of basic type

Erros Semânticos


```
void f() {
    int arr[2];
    print(arr);
}
```

Erro
arr does not reference a variable of basic type

Erros Semânticos


```
void f() {
    read("value");
}
```

Erro
"value" is not a variable

Erros Semânticos


```
int f() {
    return '1';
}
```

Erro
Return of type(char) is incompatible with type(int) function definition

Erros Semânticos


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

Erro
Name i is already
defined in this scope

Erros Semânticos


```
char c = !'a';
```

Erro
Unary operator ! is not supported

Symbol Table

Tabela de Símbolos

- Durante a verificação semântica das declarações (de variáveis, funções, tipos, etc.), os símbolos (nomes) declarados são inseridos na Tabela de Símbolos
  - Associa variáveis a tipos (significados)
- Durante a verificação do uso desses nomes, são feitas consultas na mesma tabela
- A Tabela de Símbolos deve refletir as regras de visibilidade de nomes definidas na linguagem
  - O que significa isso?

Escopo e Tabela de Símbolos

- A maioria das linguagens de programação modernas tem alguma noção de escopo
- Escopo define o “tempo de vida” de um símbolo no programa
  - Se um símbolo não estiver mais acessível, dizemos que está “fora do escopo”.
  - O escopo mais simples é o “escopo de bloco”.
- Com escopo, precisamos de uma noção de declaração de variável
  - o que nos permite afirmar em que âmbito a variável é visível ou acessível

Escopo e Tabela de Símbolos

- A tabela de símbolos deve definir o escopo em que a variável é visível
- Parâmetros e variáveis locais de um método **m** somente são visíveis dentro de **m**
- Campos e métodos de uma classe somente são visíveis dentro da classe

Variable Shadowing

Um problema com declarações com escopo é que declarações podem “ocultar” declarações externas

```
void f() {
     int i = 0;
     {
          int i = 10;
          print(i)
     }
}
```

Regras semânticas para Declarações de Variáveis

- Uma declaração de variável insere a variável na tabela de símbolos do escopo atual
- Procurar variável na tabela de símbolos retorna um valor de variável do escopo atual ou dos escopos maiores
- Cada variável precisa ser declarada antes do uso
- Nenhuma variável pode ser declarada mais de uma vez no escopo atual

Como Implementar Escopo?

Implementação da Análise

Verificação em Duas Fases

- Fase 1
  - Construção tabela de símbolos (tipos básicos int e boolean)
  - Outros tipos são adicionados (novas classes, etc.)
- Fase 2
  - Verificação de tipos de statements e expressions
  - Tabela de tipos verificada para cada identificador achado
  - Adiciona finalmente na tabela de símbolos
- Preciso fazer em duas fases
  - Chamada de um método que ainda não foi definido.
  - Classes mutuamente dependentes

Não é necessário para uC que é mais simples

Symbol Table


```
class SymbolTable:
    """Class representing a symbol table.
    """
    __data = dict()

    def add(self, name: str, value: Any) -> None:
        """ Adds to the SymbolTable.
        """
        self.__data[name] = value

    def lookup(self, name: str) -> Union[Any, None]:
        """ Searches `name` on the SymbolTable and returns the value
        assigned to it.
        """
        return self.__data.get(name)
```

TODO: Implementar a gestão do escopo

Sistema de Tipo


```
class uCType:
    '''
    Class that represents a type in the uC language.  Basic
    Types are declared as singleton instances of this type.
    '''
    def __init__(self, name, binary_ops=set(), unary_ops=set(),
                 rel_ops=set(), assign_ops=set()):
        '''
        You must implement yourself and figure out what to store.
        '''
        self.typename = name
        self.unary_ops = unary_ops
        self.binary_ops = binary_ops
        self.rel_ops = rel_ops
        self.assign_ops = assign_ops
```

```
# Create specific instances of basic types. You will need to add
# appropriate arguments depending on your definition of uCType
IntType = uCType("int",
                 unary_ops   = {"-", "+", "*", "&"},
                 binary_ops  = {"+", "-", "*", "/", "%"},
                 rel_ops     = {"==", "!=", "<", ">", "<=", ">="},
                 assign_ops  = {"="}
                 )

CharType = uCType("char",
                  # TODO: Complete
    )

FloatType = …
```

Gestão de Erros


```
def _assert_semantic(self, condition: bool, msg_code: int, coord,
                     name: str = "", ltype="", rtype=""):
    """Check condition, if false print selected error message and exit"""
    error_msgs = {
        1: f"{name} is not defined",
        2: f"subscript must be of type(int), not {ltype}",
        3: "Expression must be of type(bool)",
        …
        25: f"Unary operator {name} is not supported",
    }
    if not condition:
        msg = error_msgs[msg_code]  # invalid msg_code raises Exception
        print("SemanticError: %s %s" % (msg, coord), file=sys.stdout)
        sys.exit(1)
```

25 mensagens de erros padronizados

Visitor


```
class NodeVisitor:
    """ A base NodeVisitor class for visiting uc_ast nodes.
        Subclass it and define your own visit_XXX methods, where
        XXX is the class name you want to visit with these
        methods.
    """

    def visit(self, node):
        """ Visit a node.
        """
        …
        return visitor(node)

    def generic_visit(self, node):
        """ Called if no explicit visitor function exists for a
            node. Implements preorder visiting of the node.
        """
        for _, child in node.children():
            self.visit(child)
```

Implementação inicial


```
class SemanticAnalysis(NodeVisitor):

    def visit_Program(self, node):
        # Visit all of the global declarations
        for _decl in node.gdecls:
            self.visit(_decl)
        # TODO: Manage the symbol table

    def visit_BinaryOp(self,node):
        # Visit the left and right expression
        self.visit(node.left)
        ltype = node.left.uc_type
        self.visit(node.right)
        rtype = node.right.uc_type
        # TODO:
        # - Make sure left and right operands have the same
type
        # - Make sure the operation is supported
        # - Assign the result type to current node
```

```
def visit_Assignment(self, node):
    # visit right side
    self.visit(node.rvalue)
    rtype = node.rvalue.uc_type
    # visit left side (must be a location)
    _var = node.lvalue
    self.visit(_var)
    if isinstance(_var, ID):
        self._assert_semantic(_var.scopeis not None,
1, node.coord, name=_var.name)
    ltype = node.lvalue.uc_type
    # Check that assignment is allowed
    self._assert_semantic(ltype == rtype,4,
          node.coord, ltype=ltype, rtype=rtype)
    # Check that assign_ops is supported by the type
    self._assert_semantic(node.opin ltype.assign_ops,
          5, node.coord, name=node.op, ltype=ltype)
```

AST decorada


```
int a = 3 * 4 + 5, b[1][2];
```

Pretty-print

```
Program:
    GlobalDecl:
        Decl: ID(name=a)
            VarDecl:
                Type: int @ 1:1
            BinaryOp: + @ 1:9
                BinaryOp: * @ 1:9
                    Constant: int, 3 @ 1:9
                    Constant: int, 4 @ 1:13
                Constant: int, 5 @ 1:17
        Decl: ID(name=b)
            ArrayDecl:
                ArrayDecl:
                    VarDecl:
                        Type: int @ 1:1
                    Constant: int, 2 @ 1:25
                Constant: int, 1 @ 1:22
```

```
Program:
    GlobalDecl:
        Decl: ID(name=a)
            VarDecl:
                Type: int @ 1:1
            BinaryOp: ucType: int, + @ 1:9
                BinaryOp: ucType: int, * @ 1:9
                    Constant: ucType: int, 3 @ 1:9
                    Constant: ucType: int, 4 @ 1:13
                Constant: ucType: int, 5 @ 1:17
        Decl: ID(name=b)
            ArrayDecl:
                ArrayDecl:
                    VarDecl: ucType: ArrayType
                        Type: int @ 1:1
                    Constant: int, 2 @ 1:25
                Constant: int, 1 @ 1:22
```

Resumo

- Visão Geral do Front-end
- Analisador Semântica
- Symbol table
- Type Checking
- Visitor
- Erros semânticos
- AST decorada

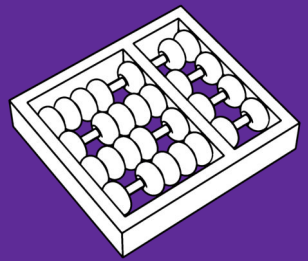
Leitura Recomendada

- Capítulo XX do livro do Cooper.
- Capítulo XX do livro do Appel.

Próxima Aula

- Parser LR(1)

Obrigado!
Merci!



UNICAMP

Pallete

Bubbles

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

BUBBLE
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

DRACULA

Tables

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Table Title	
Column 1	**Column 2**
One	Two
Three	Four

Emojis

Analisador Sintático

- Deve ser capaz de expressar a sintaxe de linguagens de programação
  - Que tal dar nomes à ERs?

expr = ab(c|d)e

aux = c | d
expr = a b aux e

Gramáticas Livre de Contexto

- Exemplo de ER usando nomes:
  - digits = [0-9]+
  - sum = (digits “+”)* digits
- Como isso é implementado?
  - O analisador léxico substitui os nomes das ERs antes de traduzir para um autômato finito
  - sum = ([0-9]+ “+”) * [0-9]+

Gramáticas Livre de Contexto

- É possível usar a mesma ideia para definir uma linguagem para expressões que tenham parênteses balanceados?
  - (1+(245+2))
- Tentativa
  - digits = [0-9]+
  - sum = expr “+” expr
  - expr = “(“ sum “)” | digits