UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

Trabalho 3 - Análise Semântica

Ramna Sidharta (16100742)
Matheus Schaly (18200436)
Letícia do Nascimento (16104595)

Florianópolis, 23 de abril de 2021

Análise Semântica

Análise semântica é a terceira fase da compilação. Nesse momento ocorre a validação de uma série de regras que não podem ser verificadas nas etapas anteriores, por exemplo, fazer a divisão de um número inteiro por outro número do tipo ponto flutuante. Além disso, esta etapa é onde coletam-se as informações necessárias para a próxima fase da compilação - que é a geração de código objeto. Ela verifica e aponta as expressões que quebram qualquer regra determinada pelas regras. Mas o nível de complexidade aumenta quando tratamos de linguagens dependentes de contexto. A análise semântica busca apontar (não resolver) este tipo de erros (dependentes de contexto).

Resolução das Tarefas

Tarefa ASem

1. Construção de uma árvore de expressão T (nós da árvore somente com operadores e operandos);
2. Inserção do tipo de variáveis na tabela de símbolos; Além disso, alguns pontos realizados na análise semântica deverão ser tratados. São eles:
3. A verificação de tipos (em expressões aritméticas);
    a. A verificação de tipos deverá acessar o tipo de uma determinada variável em uma tabela de símbolos.
4. A declaração de variáveis por escopo;
5. A verificação de que um comando break está no escopo de um comando de repetição.

Ponto 1. *Construção da árvore de expressão*

Recapitula-se que na etapa anterior deste trabalho, onde foi implementado o parsing, foi utilizado o yacc (yacc.py). Naquela etapa, cada regra de produção é definida como uma função p_simboloNaoTerminal(p), onde atribuímos o resultado da cauda da produção à sua cabeça, representada com p[0].

Para a construção da árvore de expressão, o yacc não fornece nenhuma função específica. No entanto, existem algumas formas de implementar a árvore sem o uso direto da ferramenta. Uma maneira simples de construir uma árvore é criando uma tupla ou lista em cada função de regra gramatical. Outra abordagem é criar um conjunto de estrutura de dados para os tipos de nós da árvore, e atribuí-los à cabeça da produção em cada regra. Dentre as possibilidades, optamos por representar cada nodo da árvore como uma classe Python, pois assim temos uma representação mais clara da árvore, e a etapa de geração de código, ou qualquer validação extra, fica mais fácil, pois o código é melhor extensível. Cada nó é a representação concreta de uma classe abstrata Node. Node possui ao menos um método, além de seu construtor, que chama-se validate() e que possui um tipo:

```
class Node(ABC):
       """ Base class for the nodes of the parse tree. """
       _type = None

       @abstractmethod
       def validate(self, scope):
           pass

       @property
       def type(self):
           return self._type

       @type.setter
       def type(self, typ):
           self._type = typ
```

Veja um exemplo de declaração de um nó:

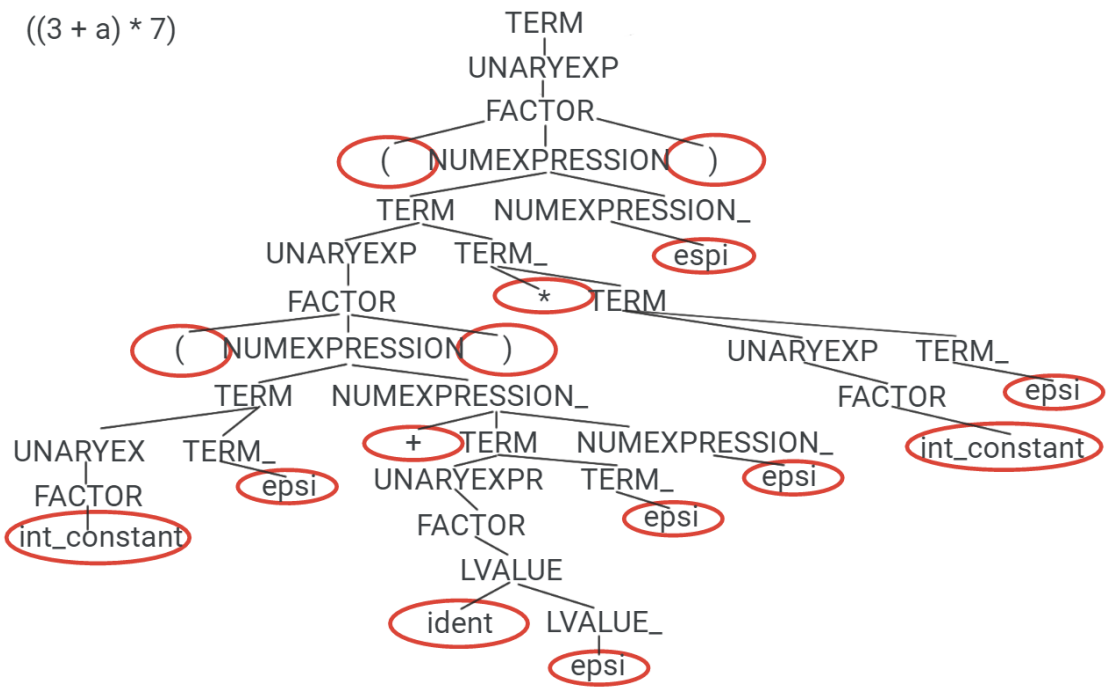
```
def p_funcdef(p):
       """
       funcdef : DEF IDENT LPAREN paramlist RPAREN LCBRACKET statelist RCBRACKET
       """
       p[0] = FuncDecl(p[2], p[7], params=p[4])
```

Detalhes sobre esta implementação são apresentados mais adiante.

EXPA:

```
NUMEXPRESSION -> TERM NUMEXPRESSION_
NUMEXPRESSION_ -> + TERM NUMEXPRESSION_
NUMEXPRESSION_ -> - TERM NUMEXPRESSION_
NUMEXPRESSION_ -> ''
TERM -> UNARYEXPR TERM_
TERM_ -> ''
TERM_ -> * TERM
TERM_ -> / TERM
TERM_ -> % TERM
UNARYEXPR -> + FACTOR
UNARYEXPR -> - FACTOR
UNARYEXPR -> FACTOR
FACTOR -> int_constant
FACTOR -> float_constant
FACTOR -> string_constant
FACTOR -> null
FACTOR -> LVALUE
FACTOR -> ( NUMEXPRESSION )
LVALUE -> ident LVALUE_
LVALUE_ -> ''
LVALUE_ -> NUM_EXP_LIST
```

Árvore de derivação anotada:



SDD-L-Atribuída:

Produções	Regras Semânticas
NUMEXPRESSION -> TERM NUMEXPRESSION_	NUMEXPRESSION.nó = NUMEXPRESSION_.her
NUMEXPRESSION_ -> + TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("+", NUMEXPRESSION_.her, TERM.nó)
	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin
NUMEXPRESSION_ -> - TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("-", NUMEXPRESSION_.her, TERM.nó)
	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin
NUMEXPRESSION_ -> ''	NUMEXPRESSION_.her = TERM.nó
TERM -> UNARYEXPR TERM_	TERM.nó = TERM_.her
TERM_ -> ''	TERM_.her = UNARYEXP.sin
TERM_ -> * TERM	TERM_.her = novo nó("*", TERM.nó, UNARYEXP.sin)
TERM_ -> / TERM	TERM_.her = novo nó("/", TERM.nó, UNARYEXP.sin)
TERM_ -> % TERM	TERM_.her = novo nó("%", TERM.nó, UNARYEXP.sin)
UNARYEXPR -> + FACTOR	UNARYEXP.sin = FACTOR.nó
UNARYEXPR -> - FACTOR	UNARYEXP.sin = FACTOR.nó
UNARYEXPR -> FACTOR	UNARYEXP.sin = FACTOR.nó
FACTOR -> int_constant	FACTOR.nó = novo nó("int_constant", -, -)
FACTOR -> float_constant	FACTOR.nó = novo nó("float_constant", -, -)
FACTOR -> string_constant	FACTOR.nó = novo nó("string_constant", -, -)
FACTOR -> null	FACTOR.nó = novo nó("null", -, -)
FACTOR -> LVALUE	FACTOR.nó = LVALUE.nó
FACTOR -> (NUMEXPRESSION)	FACTOR.nó = NUMEXPRESSION.nó
LVALUE -> ident LVALUE_	LVALUE.nó = LVALUE_.her
LVALUE_ -> ''	LVALUE_.her = novo nó("ident", -, -)
LVALUE_ -> NUM_EXP_LIST	LVALUE_.sin = NUM_EXP_LIST.sin

Para demonstrar que a SDD é L-atribuída precisamos checar, para cada atributo:

1) Se ele for sintetizado, ele não pode ter ciclos.

2) Se ele for herdado, ele deve herdar do pai, dos irmãos à esquerda, ou dele mesmo (desde que não gere ciclos).

Produções	Regras Semânticas	Atributos
NUMEXPRESSION -> TERM NUMEXPRESSION_	NUMEXPRESSION.nó = NUMEXPRESSION_.her	NUMEXPRESSION.nó depende apenas de seu filho
NUMEXPRESSION_ -> + TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("+", NUMEXPRESSION_.her, TERM.nó)	NUMEXPRESSION_.her é herdado mas depende dele mesmo e de um irmão à esquerda
	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin	NUMEXPRESSION_.sin depende apenas de seu filho e não possui ciclo
NUMEXPRESSION_ -> - TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("-", NUMEXPRESSION_.her, TERM.nó)	NUMEXPRESSION_.her é herdado mas depende dele mesmo e de um irmão à esquerda
	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin	NUMEXPRESSION_.sin depende apenas de seu filho e não possui ciclo
NUMEXPRESSION_ -> "	NUMEXPRESSION_.her = TERM.nó	NUMEXPRESSION_.her é herdado mas depende de um irmão à esquerda
TERM -> UNARYEXPR TERM_	TERM.nó = TERM_.her	TERM.nó é herdado mas depende dele mesmo e de um irmão à esquerda
TERM_ -> "	TERM_.her = UNARYEXP.sin	TERM_.her é herdado mas depende de um irmão à esquerda
TERM_ -> * TERM	TERM_.her = novo nó("*", TERM.nó, UNARYEXP.sin)	TERM_.her é herdado mas depende dele mesmo e de um irmão à esquerda
TERM_ -> / TERM	TERM_.her = novo nó("/", TERM.nó, UNARYEXP.sin)	TERM_.her é herdado mas depende dele mesmo e de um irmão à esquerda
TERM_ -> % TERM	TERM_.her = novo nó("%", TERM.nó, UNARYEXP.sin)	TERM_.her é herdado mas depende dele mesmo e de um irmão à esquerda
UNARYEXPR -> + FACTOR	UNARYEXP.sin = FACTOR.nó	UNARYEXP.sin depende apenas de seu filho e não possui ciclo
UNARYEXPR -> - FACTOR	UNARYEXP.sin = FACTOR.nó	UNARYEXP.sin depende apenas de seu filho e não possui ciclo
UNARYEXPR -> FACTOR	UNARYEXP.sin = FACTOR.nó	UNARYEXP.sin depende apenas de seu filho e não possui ciclo

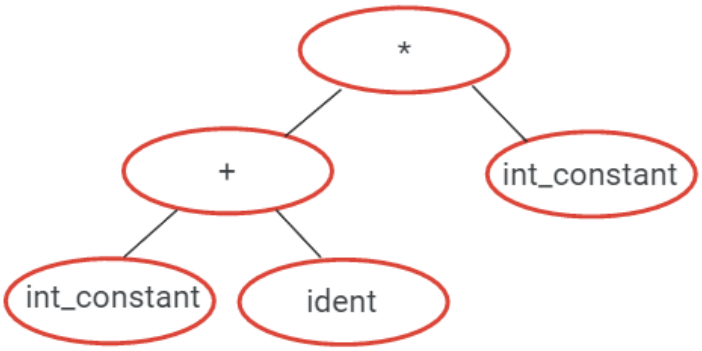
FACTOR -> int_constant	FACTOR.nó = novo nó("int_constant", -, -)	FACTOR.nó depende apenas de seu filho
FACTOR -> float_constant	FACTOR.nó = novo nó("float_constant", -, -)	FACTOR.nó depende apenas de seu filho
FACTOR -> string_constant	FACTOR.nó = novo nó("string_constant", -, -)	FACTOR.nó depende apenas de seu filho
FACTOR -> null	FACTOR.nó = novo nó("null", -, -)	FACTOR.nó depende apenas de seu filho
FACTOR -> LVALUE	FACTOR.nó = LVALUE.nó	FACTOR.nó depende apenas de seu filho
FACTOR -> (NUMEXPRESSION)	FACTOR.nó = NUMEXPRESSION.nó	FACTOR.nó depende apenas de seu filho
LVALUE -> ident LVALUE_	LVALUE.nó = LVALUE_.her	LVALUE.nó depende apenas de seu filho
LVALUE_ -> ''	LVALUE_.her = novo nó("ident", -, -)	LVALUE_.her é herdado mas depende de um irmão à esquerda
LVALUE_ -> NUM_EXP_LIST	LVALUE_.sin = NUM_EXP_LIST.sin	LVALUE_.sin depende apenas de seu filho e não possui ciclo

Em uma SDT, insere-se ações semânticas no corpo das produções com base nas regras semânticas

Produções	Regras Semânticas	Ações semânticas
NUMEXPRESSION_ -> + TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("+", NUMEXPRESSION_.her, TERM.nó)	NUMEXPRESSION_ -> + TERM {NUMEXPRESSION_.her = novo nó("+", NUMEXPRESSION_.her, TERM.nó)} NUMEXPRESSION_ {NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin}
	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin	
NUMEXPRESSION_ -> - TERM NUMEXPRESSION_	NUMEXPRESSION_.her = novo nó("-", NUMEXPRESSION_.her, TERM.nó)	NUMEXPRESSION_ -> - TERM {NUMEXPRESSION_.her = novo nó("-", NUMEXPRESSION_.her, TERM.nó)} NUMEXPRESSION_ {NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin}

	NUMEXPRESSION_.sin = NUMEXPRESSION1_.sin	
NUMEXPRESSION_ -> ''	NUMEXPRESSION_.her = TERM.nó	NUMEXPRESSION_ -> '' {NUMEXPRESSION_.her = TERM.nó}
TERM_ -> ''	TERM_.her = UNARYEXP.sin	TERM_ -> '' {TERM_.her = UNARYEXP.sin}
TERM_ -> * TERM	TERM_.her = novo nó("*", TERM.nó, UNARYEXP.sin)	TERM_ -> * TERM {TERM_.her = novo nó("*", TERM.nó, UNARYEXP.sin)}
TERM_ -> / TERM	TERM_.her = novo nó("/", TERM.nó, UNARYEXP.sin)	TERM_ -> / TERM {TERM_.her = novo nó("/", TERM.nó, UNARYEXP.sin)}
TERM_ -> % TERM	TERM_.her = novo nó("%", TERM.nó, UNARYEXP.sin)	TERM_ -> % TERM {TERM_.her = novo nó("%", TERM.nó, UNARYEXP.sin)}
LVALUE_ -> ''	LVALUE_.her = novo nó("ident", -, -)	LVALUE_ -> '' {LVALUE_.her = novo nó("ident", -, -)} {LVALUE_.sin = NUM_EXP_LIST.sin}
LVALUE_ -> NUM_EXP_LIST	LVALUE_.sin = NUM_EXP_LIST.sin	

Árvore de expressão:



Ponto 2. *Inserção do tipo na tabela de símbolos*

Para a checagem de tipos, é necessário que as regras tenham acesso aos tipos das variáveis, que geralmente comumente são obtidas na tabela de símbolos. Contudo, aproveitando o design da árvore com orientação a objetos, este analisador semântico não precisou acessar a tabela de símbolos. Isso foi possível por duas razões:

1. Fizemos um mapeamento direto dos tipos da linguagem LCC para a linguagem em que o compilador foi implementado, o Python.
2. Antes de executar cada checagem de erro, um nó faz a validação de seus filhos, e na validação dos filhos, será definido seu tipo, sendo assim, um nó já tem acesso ao tipo de seus filhos e ao seu próprio tipo.

De fato esta abordagem pode não funcionar sempre, dependendo dos tipos que a linguagem sendo implementada suporta. Neste caso, poderíamos facilmente acessar a tabela de símbolos que fica disponível globalmente em lexer.py (e que foi construída e populada nas etapas anteriores de compilação). Esse acesso é possível, pois cada nó tem seu nome, o qual é a chave na tabela de símbolos.

Veja o exemplo do nó BinOp:

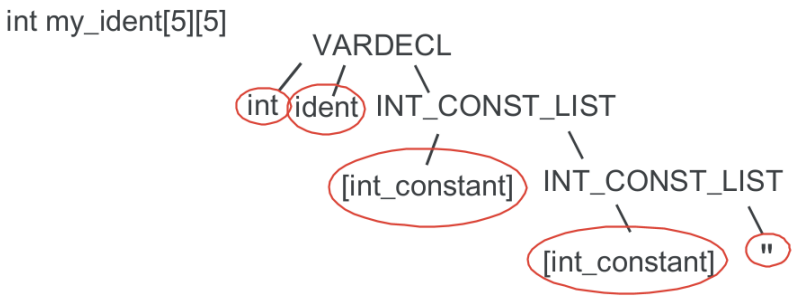
```
class BinOp(Node):
       def __init__(self, left, op, right):
           super(BinOp, self).__init__()
           self.left = left
           self.op = op
           self.right = right

       def validate(self, scope):
           self.left.validate(scope)
           self.right.validate(scope)

           a_type = self.left.type
           b_type = self.right.type
           if self.left.type != self.right.type:
              raise IncompatibleTypesException(f'TypeError: incompatible types '
                                               f'{a_type} {self.op} {b_type}!')
           if a_type == str or b_type == str:
            if not self.valid_str_op():
             msg = f'TypeError: unsupported operand "{self.op}" for type ' \
                   f'STRING_CONSTANT!'
             raise UnsupportedOperandException(msg)

           self.type = a_type

       def valid_str_op(self):
           return self.op in ['+', '*', '==']
```

Veja que as linhas iniciais executam a validação dos lados esquerdo e direito da operação, ou seja, filhos do nó BinOp. Em seguida, o nó já acessa o tipo de seus filhos. E ao final, ele define seu próprio tipo baseado no tipo de seus filhos (a_type == b_type).
Imagine que self.left seja um nó do tipo Literal (um inteiro). O validate() daquele nó definirá seu tipo como int, estando disponível para o nó pai de Literal.

DEC:

```
VARDECL -> int ident INT_CONST_LIST
VARDECL -> float ident INT_CONST_LIST
VARDECL -> string ident INT_CONST_LIST
INT_CONST_LIST -> [int_constant] INT_CONST_LIST
INT_CONST_LIST -> ''
```

Árvore de derivação anotada:



SDD-L-Atribuída:

Produções	Regras Semânticas
VARDECL -> int ident INT_CONST_LIST	VARDECL .sin = novo nó (int.tipo, "ident", INT_CONST_LIST.val)
VARDECL -> float ident INT_CONST_LIST	VARDECL .sin = novo nó (float.tipo, "ident", INT_CONST_LIST.val)
VARDECL -> string ident INT_CONST_LIST	VARDECL .sin = novo nó (string.tipo, "ident", INT_CONST_LIST.val)
INT_CONST_LIST -> [int_constant] INT_CONST_LIST	INT_CONST_LIST.sin = novo nó ([int_constant], INT_CONST_LIST.val, -)
INT_CONST_LIST -> ''	INT_CONST_LIST.her = INT_CONST_LIST.val

Para demonstrar que a SDD é L-atribuída precisamos checar, para cada atributo:

3) Se ele for sintetizado, ele não pode ter ciclos.
4) Se ele for herdado, ele deve herdar do pai, dos irmãos à esquerda, ou dele mesmo (desde que não gere ciclos).

Produções	Regras Semânticas	Atributos
VARDECL -> int ident INT_CONST_LIST	VARDECL .sin = novo nó (int.tipo, "ident", INT_CONST_LIST.val)	VARDECL .sin depende apenas de seu filho e não possui ciclo
VARDECL -> float ident INT_CONST_LIST	VARDECL .sin = novo nó (float.tipo, "ident", INT_CONST_LIST.val)	VARDECL .sin depende apenas de seu filho e não possui ciclo
VARDECL -> string ident INT_CONST_LIST	VARDECL .sin = novo nó (string.tipo, "ident", INT_CONST_LIST.val)	VARDECL .sin depende apenas de seu filho e não possui ciclo
INT_CONST_LIST -> [int_constant] INT_CONST_LIST	INT_CONST_LIST.sin = novo nó ([int_constant], INT_CONST_LIST.val, -)	INT_CONST_LIST .sin depende apenas de seu filho e não possui ciclo
INT_CONST_LIST -> ''	INT_CONST_LIST.her = INT_CONST_LIST.val	INT_CONST_LIST.her é herdado mas depende dele mesmo

Em uma SDT, insere-se ações semânticas no corpo das produções com base nas regras semânticas

Produções	Regras Semânticas	Ações Semânticas
VARDECL -> int ident INT_CONST_LIST	VARDECL .sin = novo nó (int.tipo, "ident", INT_CONST_LIST.val)	VARDECL -> int ident INT_CONST_LIST {VARDECL .sin = novo nó (int.tipo, "ident", INT_CONST_LIST.val)}
VARDECL -> float ident INT_CONST_LIST	VARDECL .sin = novo nó (float.tipo, "ident", INT_CONST_LIST.val)	VARDECL -> float ident INT_CONST_LIST {VARDECL .sin = novo nó (float.tipo, "ident", INT_CONST_LIST.val)}
VARDECL -> string ident INT_CONST_LIST	VARDECL .sin = novo nó (string.tipo, "ident", INT_CONST_LIST.val)	VARDECL -> string ident INT_CONST_LIST {VARDECL .sin = novo nó (string.tipo, "ident", INT_CONST_LIST.val)}
INT_CONST_LIST -> [int_constant] INT_CONST_LIST	INT_CONST_LIST.sin = novo nó ([int_constant], INT_CONST_LIST.val, -)	INT_CONST_LIST -> [int_constant] INT_CONST_LIST {INT_CONST_LIST.sin = novo nó ([int_constant], INT_CONST_LIST.val, -)}

INT_CONST_LIST -> "	INT_CONST_LIST.her = INT_CONST_LIST.val	INT_CONST_LIST -> " INT_CONST_LIST.her = INT_CONST_LIST.val

Ponto 3. *Verificação de tipos*

A explicação no início da seção anterior é a base para a verificação de tipos. Como cada nó tem seu tipo, e como cada nó possui o método validate(), fazer a checagem e gerar erros se torna simples e claro. Ainda utilizando o nó BinOp como exemplo, vejamos como erros podem ser gerados:

```
def validate(self, scope):
     ...
     a_type = self.left.type
     b_type = self.right.type
     if self.left.type != self.right.type:
         raise IncompatibleTypesException(f'TypeError: incompatible types '
                                          f'{a_type} {self.op} {b_type}!')
     if a_type == str or b_type == str:
         if not self.valid_str_op():
             msg = f'TypeError: unsupported operand "{self.op}" for type ' \
                   f'STRING_CONSTANT!'
             raise UnsupportedOperandException(msg)

     self.type = a_type
```

Aqui, além da verificação de que os tipos dos dois lados da operação devem ser iguais, também verificamos se, caso os tipos sejam string, a operação é permitida. Por exemplo, optamos por determinar que uma operação "-" para o tipo string não é definida, gerando um erro "TypeError: unsupported operand "-" for type STRING_CONSTANT".
Veja abaixo exemplos de código fonte e dos erros gerados pelo compilador de LCC:

```
def main() {
   int a;
   a = 1 + 0.1;
}

->
vlant.errors.IncompatibleTypesException: TypeError: incompatible types 'int' + 'float'!
```

```
def main() {
   int a;
   float b;
   a = a + b;
}

->
vlant.errors.IncompatibleTypesException: TypeError: incompatible types ‘int’ + 'float'!

def main() {
   string x;
   x = “Hello “ / “ world!”;
}

->
vlant.errors.UnsupportedOperandException: TypeError: unsupported operand "/" for type STRING_CONSTANT!
```

Ponto 4. *Declaração de variáveis por escopo*

Ainda aproveitando o design orientado a objetos, a implementação de escopo foi simples. Definimos uma classe responsável por armazenar todos os escopos, ela define métodos para recuperar variáveis que estariam ou não definidas no escopo atual, por exemplo. Os escopos são armazenados como conjuntos (tipo Set do Python) em uma pilha. Cada vez que um novo escopo deve ser criado, o nó inicia um escopo, o qual é adicionado ao topo da pilha. O escopo atual, utilizado para checar se uma nova variável declarada já foi declarada anteriormente no mesmo escopo, por exemplo, fica disponível através do método current().

Segue a classe Scopes:

```
class Scopes(object):
       def __init__(self):
           super(Scopes, self).__init__()
           self.scopes = [{}]

       def create(self):
           self.scopes.append({})
```

```
    def pop(self):
        return self.scopes.pop()

    @property
    def current(self):
        return self.scopes[-1]

    def get(self, name):
        for s in reversed(self.scopes):
           if name in s:
             return s[name]
        return None

    def __contains__(self, item):
        return item in self.current

    def __getitem__(self, key):
        return self.current[key]

    def __setitem__(self, key, value):
        self.current[key] = value

    @contextmanager
    def __call__(self):
        self.create()
        yield self
        self.pop()
```

Exemplo da criação de um novo escopo:

```
class FuncDecl(Node):
    def validate(self, scope):
      ...
      # Validate function block and params within new scope
      with scope() as scop:
          for arg in self.params:
          arg.validate(scop)
          for statement in self.block:
          statement.validate(scope)
```

Uma função deve definir um novo escopo e validar seus parâmetros e comandos, portanto aqui, a validação dos parâmetros e statements ficam dentro de “with scope() as scop:”. Note que o

escopo é passado para a função validate de cada um dos demais nós. Assim, ao definir uma nova variável, por exemplo, o validate() do nó VarDecl já tem acesso ao escopo:

```
class VarDecl(Node):
    def validate(self, scope):
      if self.name in scope.current:
        raise VarDeclException(f'Variable "{self.name}" already declared!')

       scope[self.name] = self
```

Veja que caso o nome da variável sendo declarada já exista no escopo atual (if self.name in scope.current), então um erro ocorre, caso contrário, a variável é inserida no escopo.

Seguem exemplos de erros de escopo:

```
def main() {
   int a;
   string a;  # a already declared
}

->
vlant.errors.VarDeclException: Variable "a" already declared!


def main() {
   int a;
   if (1 == 1) {
       a = 3;  # this is fine
   }
}

->
<No error>

def main() {
   if (1 == 1) {
       int a;
   }
   a = 3;  # var does not exist in this scope
}
->
vlant.errors.VarDeclException: NameError: variable "a" not declared!
```

Tratamento de ambiguidade

O Ply apenas apresenta exemplos com gramáticas ambíguas por questões de simplicidade, uma vez que, de acordo com a própria documentação do ply, gramáticas não-ambíguas podem ser escritas de uma forma muito complexa. Portanto, nossa gramática foi modificada, permitindo ambiguidade. O yacc.py permite fazer tratamento de ambiguidade definindo precedentes, atribuindo tokens individuais a um nível de precedência e associatividade. Por exemplo:

```
precedence = (
     ('left', 'PLUS', 'MINUS'),
     ('left', 'TIMES', 'DIVIDE'),
 )
```

Esta declaração especifica que PLUS/MINUS:

- Têm o mesmo nível de precedência;
- São associativos à esquerda.

E também que TIMES/DIVIDE:

- Possui a mesma precedência;
- São associativos à esquerda.

Com a declaração de precedência, os tokens são ordenados da precedência mais baixa para a mais alta. Assim, esta declaração específica que TIMES/DIVIDE têm precedência superior a PLUS/ MINUS (uma vez que aparecem posteriormente na especificação de precedência).

```
precedence = (
    ('left', 'LTE', 'LT', 'GTE', 'GT', 'EQUALS', 'NOT_EQUAL'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'MULTIPLY', 'DIVIDE', 'MOD')
)
```

Tarefa GCI

Não conseguimos completar esta parte. A ideia seria adicionar mais um método à classe abstrata Node, chamado generate(). Estes métodos seriam chamados após toda a validação da árvore, e seriam executados recursivamente assim como os métodos validate(). Os métodos generate() utilizariam a lib LLVMLITE, que auxilia muito na geração de código intermediário LLVM. Com isto, seria possível executar a linguagem LCC inclusive pelo GCC.

- Note que para a execução de validate() e generate(), basta fazer a chamada do nó raiz da árvore. Isso está sendo feito no main.py.

Referências

https://www.dabeaz.com/ply/ply.html#ply_nn27
https://github.com/numba/llvmlite