

ITGRAPHS: uma linguagem para percorrimento em topologias de grafos

Nur Corezzi - 15/0143290

Departamento de Ciência da Computação, Campus Darcy Ribeiro, Asa Norte,
Brasília - DF, 70910-900

Foi levantado que a implementação do grafo como matriz ficaria muito simplificada e sugerido a implementação de grafos dinâmicos. O documento ainda não foi atualizado por falta de tempo, mas pretende-se implementar a sugestão liberando a adição de vértices e possivelmente adicionando a criação de um tipo genérico para que o armazenamento do valor do vértice não seja somente tipos inteiros. Com relação a adição de rótulos para vértices será adicionado caso sobre tempo para a sua adição.

1 Introdução

Neste trabalho será apresentada a especificação de uma linguagem a ser desenvolvida e que se destina a área de processamento de grafos. A mesma tem como objetivo fornecer funcionalidades básicas de uma linguagem impeprativa como comandos iterativos *for* e *while*, condicionais *if*, tratamento de expressões *booleanas* e aritméticas e também algumas funcionalidades orientadas a grafos. Seu diferencial está na introdução de um tipo básico *graph* que possui uma interface para inserção de arestas direcionais e armazenamento de valores em vértices. Também existirá um comando *for* facilitador que permite iterar fazendo percorrimentos *DFS* (Depth First Search) e *BFS* (Breadth First Search) buscando por valores específicos armazenados em cada vértice. A nova primitiva facilita buscas que necessitam ser feitas com frequência em grafos e permite que ações sejam seletivamente executadas de acordo com o percorrimento e os valores dos vértices armazenados. Este tipo básico poupa a necessidade do programador em manter e cuidar do uso de sua própria estrutura, permitindo que algum comportamento seja aplicado de forma prática à topologia de um grafo especificado.

2 Visão geral do tradutor

O tradutor implementado consiste em uma separação de processamento comumente utilizada no desenvolvimento de compiladores para linguagens de programação. As etapas consistem em *análise léxica*, *sintática*, *semântica* e por fim *geração de código intermediário*. Inicialmente será definida uma gramática para a linguagem, em seguida serão determinadas as unidades léxicas para gerar as definições do analisador sintático (gerado em FLEX) que será responsável por fornecer os *tokens* reconhecidos. O analisador sintático criado por *BISON* irá fazer uso dos *tokens* formatados para gerar a árvore de derivação e popular a

tabela de símbolos. Por último o analisador semântico irá verificar erros e gerar as anotações devidas na árvore obtida para que em seguida seja feita a geração de código intermediário.

3 Gramática

3.1 Definição

$\langle init \rangle$	$::= \langle program \rangle$
$\langle program \rangle$	$::= \epsilon$ $\langle program \rangle \langle function \rangle$
$\langle function \rangle$	$::= \langle type \rangle \langle id \rangle ' (\langle params \rangle ') ' \langle block \rangle$
$\langle params \rangle$	$::= \epsilon$ $\langle params \rangle ' , ' \langle declaration \rangle$ $\langle declaration \rangle$
$\langle function-call \rangle$	$::= \langle id \rangle ' (\langle params-call \rangle ') '$
$\langle params-call \rangle$	$::= \epsilon$ $\langle params-call \rangle ' , ' \langle expr-assign \rangle$ $\langle expr-assign \rangle$
$\langle graph-call \rangle$	$::= 'dfs' \langle graph-params-call \rangle$ $'bfs' \langle graph-params-call \rangle$
$\langle graph-params-call \rangle$	$::= ' (\langle id \rangle ' , ' \langle expr-assign \rangle ' , ' \langle expr-assign \rangle ') '$
$\langle statements \rangle$	$::= \epsilon$ $\langle statements \rangle \langle statement \rangle$
$\langle statement \rangle$	$::= \langle statement-prefix \rangle \langle statement-end \rangle$ $\langle statement-prefix \rangle \langle dangling-if \rangle$ $\langle statement-end \rangle$ $\langle dangling-if \rangle$
$\langle statement-no-dangle \rangle$	$::= \langle statement-prefix \rangle \langle statement-end \rangle$ $\langle statement-end \rangle$
$\langle dangling-if \rangle$	$::= 'if' ' (\langle expr-assign \rangle ') ' \langle statement \rangle$
$\langle statement-prefix \rangle$	$::= 'if' ' (\langle expr-assign \rangle ') ' \langle statement-no-dangle \rangle 'else'$ $'while' ' (\langle expr-assign \rangle ') '$ $'for' ' (\langle declaration-or-assign \rangle ' ; ' \langle expr-assign \rangle ' ; ' \langle expr-assign \rangle ') '$ $'for' ' (\langle declaration \rangle ' : ' \langle graph-call \rangle ') '$
$\langle block \rangle$	$::= ' \{ ' \langle statements \rangle ' \}'$

$\langle \text{statement-end} \rangle$	$::= \langle \text{block} \rangle$ $ \text{'>>' } \langle \text{id-or-access} \rangle \text{' ;'}$ $ \text{'<<' } \langle \text{expr-assign} \rangle \text{' ;'}$ $ \langle \text{declaration-or-assign} \rangle \text{' ;'}$ $ \langle \text{expr-assign} \rangle \text{' ;'}$ $ \langle \text{graph-add} \rangle \text{' ;'}$ $ \text{'return' } \langle \text{expr-assign} \rangle \text{' ;'}$
$\langle \text{expr-assign} \rangle$	$::= \langle \text{expr-relational} \rangle \text{' = ' } \langle \text{expr-assign} \rangle$ $ \langle \text{expr-relational} \rangle$
$\langle \text{expr-relational} \rangle$	$::= \langle \text{expr-and} \rangle \langle \text{compare-op} \rangle \langle \text{expr-relational} \rangle$ $ \langle \text{expr-and} \rangle$
$\langle \text{expr-and} \rangle$	$::= \langle \text{expr-or} \rangle \langle \text{and} \rangle \langle \text{expr-and} \rangle$ $ \langle \text{expr-or} \rangle$
$\langle \text{expr-or} \rangle$	$::= \langle \text{expr-add} \rangle \langle \text{or} \rangle \langle \text{expr-or} \rangle$ $ \langle \text{expr-add} \rangle$
$\langle \text{expr-add} \rangle$	$::= \langle \text{expr-sub} \rangle \langle \text{add} \rangle \langle \text{expr-add} \rangle$ $ \langle \text{expr-sub} \rangle$
$\langle \text{expr-sub} \rangle$	$::= \langle \text{expr-mul} \rangle \langle \text{sub} \rangle \langle \text{expr-sub} \rangle$ $ \langle \text{expr-mul} \rangle$
$\langle \text{expr-mul} \rangle$	$::= \langle \text{expr-div} \rangle \langle \text{mul} \rangle \langle \text{expr-mul} \rangle$ $ \langle \text{expr-div} \rangle$
$\langle \text{expr-div} \rangle$	$::= \langle \text{expr-unary} \rangle \langle \text{div} \rangle \langle \text{expr-div} \rangle$ $ \langle \text{expr-unary} \rangle$
$\langle \text{expr-unary} \rangle$	$::= \langle \text{unary} \rangle \langle \text{factor} \rangle$ $ \langle \text{factor} \rangle$
$\langle \text{factor} \rangle$	$::= \text{'(' } \langle \text{expr-assign} \rangle \text{')'}$ $ \langle \text{value} \rangle$ $ \langle \text{function-call} \rangle$
$\langle \text{unary} \rangle$	$::= \text{'!'}$ $ \langle \text{add} \rangle$ $ \langle \text{sub} \rangle$
$\langle \text{and} \rangle$	$::= \text{'\&\&'}$
$\langle \text{or} \rangle$	$::= \text{' '}$
$\langle \text{add} \rangle$	$::= \text{'+'}$
$\langle \text{sub} \rangle$	$::= \text{'-'}$
$\langle \text{mul} \rangle$	$::= \text{'*'}$

$\langle div \rangle$::= '/'
$\langle compare-op \rangle$::= '<' '<=' '>' '>=' '==' '!='
$\langle graph-add \rangle$::= $\langle id \rangle$ '[' $\langle expr-assign \rangle$ 'to' $\langle expr-assign \rangle$ ']'
$\langle declaration-or-assign \rangle$::= $\langle declaration \rangle$ $\langle declaration \rangle$ '=' $\langle expr-assign \rangle$
$\langle declaration \rangle$::= $\langle type \rangle$ $\langle size-specifier \rangle$ $\langle id \rangle$
$\langle size-specifier \rangle$::= ϵ '[' $\langle number-int \rangle$ ']'
$\langle value \rangle$::= $\langle id-or-access \rangle$ $\langle number \rangle$ $\langle boolean-const \rangle$
$\langle id-or-access \rangle$::= $\langle id \rangle$ $\langle id \rangle$ '[' $\langle number-int \rangle$ ']'
$\langle type \rangle$::= 'int' 'boolean' 'float' 'graph' 'void'
$\langle number \rangle$::= $\langle number-int \rangle$ $\langle number-float \rangle$
$\langle number-int \rangle$::= $\langle digit \rangle$ $\langle number-list \rangle$
$\langle number-float \rangle$::= $\langle digit \rangle$ $\langle number-list \rangle$ '.' $\langle digit \rangle$ $\langle number-list \rangle$
$\langle number-list \rangle$::= $\langle number-list \rangle$ $\langle digit \rangle$ ϵ
$\langle digit \rangle$::= '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'
$\langle boolean-const \rangle$::= 'true' 'false'
$\langle id \rangle$::= $\langle letter \rangle$ $\langle letter-or-digit-list \rangle$
$\langle letter-or-digit-list \rangle$::= $\langle letter-or-digit-list \rangle$ $\langle letter \rangle$ $\langle letter-or-digit-list \rangle$ $\langle digit \rangle$ ','
$\langle letter \rangle$::= 'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' 'L' 'M' 'N' 'O' 'P' 'Q' 'R' 'S' 'T' 'U' 'V' 'W' 'X' 'Y' 'Z' 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o' 'p' 'q' 'r' 's' 't' 'u' 'v' 'w' 'x' 'y' 'z' '_'

1. Acrescentados *number-int* e *number-float* para facilitar identificação.
2. Estrutura de *statements* foi modificada devido a ambiguidades. Em *statement-prefix* é forçado o casamento entre um *if* e um *else*, impedindo *if* abertos dentro de uma sessão *if-else* a não ser que o mesmo esteja em uma sessão de bloco.
3. Adição das constantes *false* e *true* que possuem valor semântico 0 e 1 respectivamente.
4. Adição de *size-specifier* para especificação do tamanho do grafo na declaração.
5. Adição de operadores de comparação que haviam sido esquecidos.

3.2 Aspectos informais da gramática

A gramática em questão define uma lista de funções como um programa válido, tendo ponto de execução inicial em uma função definida com identificador *main*. Cada função possui um conjunto de argumentos que são declarações que envolvem um tipo e um identificador e possui também um tipo de retorno. Serão também delimitadas por um conjunto de abre e fecha chaves `{ }` que possui uma lista de declarações.

Uma declaração constitui um comando da linguagem que podem ser qualquer um dos especificados em *statements*. Existe também um comando diferenciado de *for* para o percorrimento dos grafos que contempla também chamadas de funções *BFS* e *DFS* na regra *graph-op*. Esta chamada recebe um identificador do grafo um parâmetro vértice de início e um valor a ser procurado, a função irá retornar os vértices na ordem em que o valor especificado for encontrado no percorrimento (exemplo em código mais abaixo).

Um grafo terá o número de vértices especificado na sua declaração não sendo possível a remoção ou adição de mais vértices. Cada vértice armazena apenas valores inteiros. Esta decisão evita tipos genéricos na definição apenas para simplificar a implementação. Com relação as operações (descritas mais abaixo em código) será possível atribuir valores a um vértice, identificado por um inteiro, e designar arestas entre vértices existentes.

Os comandos de *for*, *while* e *if* contemplam expressões que irão avaliar os fluxos de execução e também uma regra *statement* que permite encadear uma lista de comandos a sua execução.

Expressões podem ser consideradas como operações aritméticas, lógicas e atribuições, que seguem uma precedência nessa mesma ordem. Existe também a possibilidade de definir prioridades distintas a partir do uso de parêntesis definidos na regra *factor*.

Os tipos básicos da linguagem contemplam *boolean*, *int*, *float*, *void* e *graph*. O tipo *graph* é um tipo especial que também é unicamente identificado pelo tamanho especificado entre colchetes [*number-int*].

Por último os numerais da linguagem são definidos pelos terminais numéricos como listas que definem inteiros e reais e os identificadores são definidos pelos caracteres alfabéticos minúsculos e maiúsculos além do underscore.

3.3 Descrição semântica

A seguir segue um pequeno trecho de código explicitando o uso da nova primitiva em um programa básico da linguagem:

```
void main() {
    graph[5] g;
    //Define uma aresta direcional
    g[0 to 2];
    // Designa valor para algum vertice
    g[0] = g[2] = 1;
    for (int vertice : dfs(g, 0, 1)) {
        // Percorre o grafo em dfs iniciando no vertice zero, procurando
        // vrtices que possuam o valor 1 armazenado
        << vertice;
    }
}
```

1. O ponto inicial de execução será sempre a função que possui o identificador de main e sua falta implica em não execução.
2. Parâmetros serão passados sempre por cópia assim como atribuições.
3. Cópia de variáveis do tipo *graph* apenas serão possíveis para aqueles que possuem um mesmo tamanho especificado entre parêntesis.
4. Cast implícito será possível entre *float* e *int*, e de *float* e *int* para *boolean*, caso valor seja diferente de zero será considerado *true*, e *false* caso valor seja zero.
5. Expressões de atribuição possuem um valor de retorno que será o do próprio identificador sendo atribuído.
6. O escopo de variáveis está será delimitado pelas chaves.
7. Grafo inicialmente não irá possuir arestas e possui valores de vértices default em zero.

4 Implementação

Para a realização da análise léxica foi utilizado o gerador de analisador léxico *FLEX*. Foi realizada a definição regular para reconhecimento dos lexemas para que em seguida fosse transportada para as definições em *FLEX*. Os tokens reconhecidos são repassados como um enumerador, a informação adicional sobre o lexema é encontrada nas próprias variáveis de *buffer* que o gerador produz podendo ser acessadas pelo analisador sintático com facilidade.

O gerador de analisador sintático escolhido foi o BISON. Estes gerador cria analisadores *top-down* LALR(1) e facilita o procedimento de criação dos conjuntos e estados LR, gerando também as tabelas de ações e os procedimentos necessários para a derivação da árvore sintática. Seguindo a especificação da documentação, a gramática formalmente apresentada acima foi traduzida para

o formato de produções compreendidas pelo BISON. As definições de tokens e regras da gramática podem ser encontradas no arquivo font *gramatica.y*.

Para cada regra da gramática foram associadas ações semânticas para a criação da árvore abstrata. Cada nó da árvore é representado por uma estrutura genérica *Node* que irá conter um identificador do símbolo da gramática o qual representa para auxiliar em ações posteriores. Foram abstraídas rotinas adicionais para criação destes nós, assim como para a criação e manipulação da tabela de símbolos que serão melhor especificadas abaixo.

4.1 Definição regular

Na Figura 1 é possível identificar as variáveis em negrito/itálico e os símbolos terminais em fonte normal. Não foram agrupados símbolos por funcionalidade como comparadores, operadores, tipos e sim um token por unidade terminal da linguagem. Caso seja necessário agrupamentos para facilitar a análise sintática, posteriormente podem ser definidas funções que determinam pertinência de determinado token em algum grupo. Cada token será identificado por um valor de *enum* gerado pelo analisador sintático. Os tokens são especificados em *gramatica.y* pelas regras prefixadas em *%token*.

1. Estrutura anterior de token foi removida pois se apresentou desnecessária. Agora o valor passado ao analisador sintático é apenas um inteiro representando o token, o valor do lexema é extraído de variáveis produzidas pelo BISON, assim como as linhas e colunas para indicar posição.

4.2 Descrição de funções adicionais

Funções adicionais foram criadas para: auxiliar na criação da tabela de símbolos, gerar os nós da árvore abstrata, fazer liberação de alocações e auxiliar na impressão das estruturas para o console. A seguir se encontram todas as funções adicionais utilizadas pelo analisador sintático:

1. *create_node* e *push_child*: Responsáveis por criar um nó *standalone* e adicionar filhos a algum nó.
2. *stable_create_symbol*, *stable_find* e *stable_add*: Funções auxiliares para criar um símbolo *standalone*, identificar a existência de um símbolo na tabela e adicionar um símbolo qualquer a tabela.
3. *free_node*, *free_tree*, *free_symbol_node* e *free_stable*: responsáveis por liberação de memória alocada para a tabela de símbolos e para a árvore abstrata.
4. *stable_symbol_print* e *stable_print*: São funções auxiliares para imprimir no console símbolos individuais e tabelas de símbolos.
5. *print_node_prefix*, *print_node_sufix*, *print_node* e *print_tree*: Também funções auxiliares para imprimir no console a árvore abstrata formatando os prefixos e separando nós de forma individual para debug.

ws → [\t\n]	read → >>
letter → [A-Za-z]	write → <<
digit → [0-9]	and → &&
	or →
if → if	le → <=
else → else	ge → >=
for → for	less → <
while → while	greater → >
	eq → ==
boolean → boolean	ne → !=
int → int	not → !
float → float	mul → *
void → void	div/ → /
graph → graph	sum → +
to → to	sub → -
	assign → =
true → true	end → ;
false → false	open_brace → {
	close_brace → }
dfs → dfs	it → :
bfs → bfs	separator → ,
return → return	open_p → (
	close_p →)
id → (letter) (letter digit _)*	open_bracket → [
number → (digit)+ (. digit)+?	close_bracket →]

Figura 1. Definições regulares, **number** foi separado em **c_int** e **c_float** para facilitar identificação.

6. *token_to_type*, *stype_to_string* e *gtype_to_string*: Utilizado para conversões entre tokens e tipos da gramática além de auxiliar as rotinas de impressão no console.

4.3 Estrutura da árvore sintática abstrata (AST)

Á árvore gerada é construída por meio de uma abstração de um nó. Cada nó pode conter diversos filhos e possuem um identificador para informar qual símbolo da gramática o gerou. Os nós são criados individualmente a cada redução e a estrutura da árvore é gerada a partir do método de adição de filhos. Cada regra composta por mais de um símbolo, que já possua valor semântico presente (ou seja nó instanciado), será acrescida como filho do símbolo ao qual a regra será reduzida. Ao final, na regra geradora o valor será atribuído á um ponteiro *AST* declarado no analisador.

Cada nó possui uma informação de qual símbolo da gramática o mesmo representa como por exemplo uma função, parâmetro, tipo, identificador e etc. Os nós também possuem ponteiros para seus filhos e além disso uma informação adicional de complemento utilizada por variáveis, funções e constantes numéricas que precisam do valor ao qual se referem.

4.4 Tabela de símbolos

A tabela de símbolos inicialmente é uma estrutura de lista de listas ligada onde cada colisão gera um novo encadeamento em alguma entrada. Os elementos instanciados são declarações de variáveis e de funções. As informações adicionadas se tratam de uma cadeia de caracteres que irá representar o escopo do símbolo (será preenchido posteriormente na análise semântica), o tipo do símbolo informando se é uma função ou variável, um identificador com o nome da variável ou função e para facilitar verificações qual o tipo do símbolo na gramática (*void*, *int*, *graph* e etc) e uma referência para o nó da árvore abstrata caso seja necessário ter acesso a informações adicionais, como por exemplo os parâmetros de funções para verificações semânticas.

5 Tratamento de erros

5.1 Identificação e tratamento de erros léxicos

A nível de análise léxica apenas será necessário verificar erros referentes a caracteres não existentes e padrões que contemplem símbolos do alfabeto da linguagem mas que não se encaixem nas definições regulares informadas. A forma de resposta será sempre informar caractere a caractere do padrão não reconhecido informando linha e coluna. Em seguida todos os caracteres incorretos serão descartados da análise e o analisador irá continuar com o restante da entrada. Não foi necessário a introdução de procedimentos para tratamento de erro uma vez que a regra de captura de padrões não reconhecidos (regra ". {}" em FLEX) irá automaticamente descartar os caracteres não desejados bastando apenas acrescentar uma rotina para informar a ocorrência do erro.

5.2 Identificação e tratamento de erros sintáticos

Por simplicidade nenhum erro identificado será corrigido. O token responsável pelo erro será simplesmente descartado e o analisador por padrão irá desempilhar os estados até encontrar alguma derivação que possua um token *error*. Este token se apresenta em pontos chaves do código sendo eles a regra de produção de um função e a regra de produção de *statements* para que a partir destes pontos o analisador possa retomar apenas para verificar possíveis próximos erros na derivação. Os erros serão descritos identificando a linha e coluna do token o qual gerou o erro, e será informado qual token deveria ser esperado, sempre que possível inferir.

6 Arquivos de teste

6.1 Arquivos de teste léxico

Os arquivos *lexico/correto1.txt*, *lexico/correto2.txt*, *lexico/errado1.txt* e *lexico/errado2.txt* possuem exemplos de programas sem e com erros léxicos. Em *errado1.txt* são apresentados caracteres não reconhecidos pela linguagem, espera-se que sejam informados um a um os caracteres incorretos, e em *errado2.txt* erros com símbolos do alfabeto da linguagem, porém com padrões não reconhecidos onde será apresentado um a um os caracteres que não fazem parte de um padrão reconhecível. Ambos arquivos de erro são descritos na Figura 3, lembrando que os comentários apenas descrevem o erro e não necessariamente o que será informado ao usuário.

```
void main() {
    @"'\çá~^ // Cada símbolo será informado como erro
}

void main() {
    .9 1. // Numeros incorretos
    a | b // Apenas um pipe não reconhecido
    a &&& b // & a mais
    _nome // Underscore no inicio de identificador
}
```

Figura 2. Programas comentados com os respectivos erros léxicos

6.2 Arquivos de teste sintáticos

Os arquivos *sintatico/correto1.txt*, *sintatico/correto2.txt*, *sintatico/errado1.txt* e *sintatico/errado2.txt* possuem exemplos de programas sem e com erros sintáticos. Em *errado1.txt* é apresentada uma função sem tipo de retorno declarado, um for sem declaração de uma variável na primeira parte e uma declaração de variável sem fechamento de ponto e vírgula. Em *errado2.txt* existem erros de comando de leitura sem variável de destino, for iterador sem uma função correta de iteração,

um if sem condicional e um acesso incorreto de um vértice por meio de ponto flutuante. Em cada caso espera-se que o token imediato do erro seja informado e em algumas situações, onde seria possível inferir o token esperado, o mesmo também será informado. Em alguns casos é possível inferir o que seria o token esperado mas em muitas situações existe ambiguidade no que seria a intenção do usuário, portanto algumas mensagens não são precisas e difíceis de se determinar.

```

1  main() {           // Função sem tipo será identificado como ID não esperado
2      int a, b, c;    // Todos as construções já reduzidas que compõem a função serão descartadas
3  }
4
5  void main () {
6      int a = 2        // Seria esperado um ';' porém foi encontrado o token FOR
7      for (; i; i) {}   // For sem declaração de índice para iterar, não é possível determinar um erro exato
8                      // pois neste caso diferentes construções corretas poderiam ser sugeridas
9      while(a) a;
10 }
11
12 void main() {
13     >>;              // ';' não esperado seria esperado algum valor de uma expressão como ID
14     for (int i : vdsf(a, 1, 2)) {    // FOR de iteração deve conter alguma função DFS ou BFS
15     }
16     if () {}          // IF sem expressão para avaliar, final de parentesis não esperado
17     g[1.2];           // Acesso a vértice deve ser por inteiros, porém aqui mais de uma
18                     // construção será possível portanto mensagem talvez não seja significativa
19 }

```

Figura 3. Programas comentados com os respectivos erros sintáticos

7 Dificuldades da implementação

Com relação a parte léxica as definições da linguagem se mostraram simples e não exigiram grande aprofundamento no ferramental fornecido pelo *FLEX*. Na parte sintática já foi necessário maior estudo com relação as funcionalidades fornecidas pelo *BISON*. Ambiguidades presentes na gramática também apresentaram desafios na sua redefinição. Também existiram dificuldades com relação ao que seria necessário para a próxima fase uma vez que não havia muito conhecimento sobre o que seria feito na análise semântica nem na geração de código intermediário. Essa dificuldade já se apresentou na parte léxica uma vez que mudanças foram efetuadas nesta fase devido ao desconhecimento de certas necessidades.

Referências

- [ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, jan 2007.
- [Dega] Jutta Degener. Ansi c grammar, lex specification. <http://www.quut.com/c/ANSI-C-grammar-l-2011.html>. Accessed: 2020-03-17.
- [Degb] Jutta Degener. Ansi c yacc grammar. <http://www.quut.com/c/ANSI-C-grammar-y-2011.html>. Accessed: 2020-03-17.