siC: Uma linguagem baseada em C incluindo fila como tipo primitivo

Gabriella de Oliveira Esteves, 110118995

[1]Departamento de Ciência da Computação - Universidade de Brasília

Objetivo

Este trabalho visa projetar e construir uma nova linguagem chamada de siC - Structure in C, baseada na linguagem C. O siC acrescenta a estrutura de dados fila como tipo de dado primitivo e, para manipulá-la, adiciona certas operações próprias para tal.

Introdução

Um compilador é um programa que recebe como entrada um código fonte e o traduz para um programa equilavente em outra linguagem [1]. Ele pode ser dividido em sete fases, ilustrado na Figura 1.

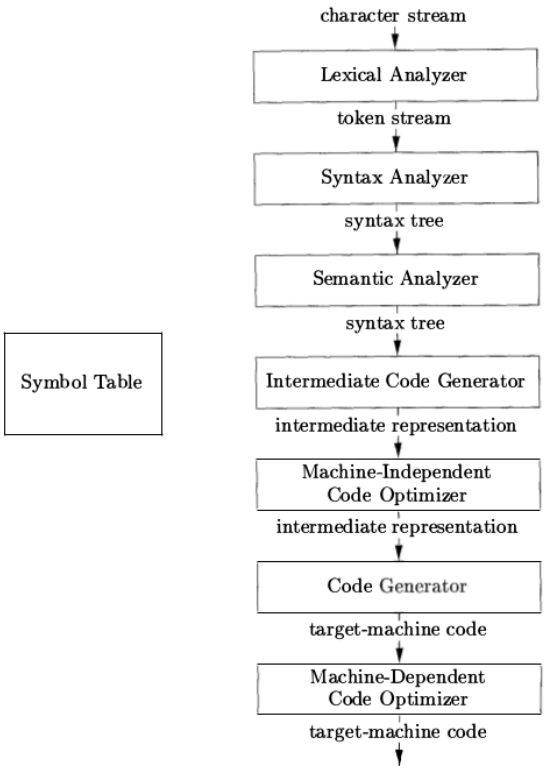

Figura 1. Fases de um compilador

1 **Analisador Léxico:** Lê o código fonte e atribui significado à cada sequência de caracteres, agora chamados lexemas. Cada lexema é mapeado para um token, que por sua vez é um par de nome (símbolo abstrato) e atributo (ponteiro para tabela de símbolos);
2 **Analisador Sintático:** Constrói uma representação gramatical dos tokens em forma de árvore;

3 **Analisador Semântico:** Utiliza a árvore sintática juntamente com a tabela de símbolos para verificar se a consistência semântica é mantida de acordo com a definição da linguagem.

4 **Gerador de Código Intermediário:** Converte árvore sintática anotada em código intermediário, com linguagem parecida com assembly e que possui apenas três operadores por linha de código. Nesse sentido, quebra-se estruturas complexas em estruturas mais simples, nesta fase.

5, 6 **Otimizador de Código Independente/Dependente de Máquina:** Procura aprimorar o código intermediário com o objetivo de melhorar o código-alvo de alguma forma: o deixando mais rápido, mais curto, consumindo menos energia, etc.

7 **Gerador de Código:** Converte o código intermediário no código-alvo, buscando atribuir os registradores às variáveis da maneira ótima.

O foco do projeto será nas fase 1, 2, 3 e 4, porém a princípio serão apresentadas apenas a descrição da linguagem siC, uma breve descrição de sua semântica e o analisador léxico. Como a fila é uma das estruturas de dados mais básicas, é possível dizer que siC se destina a inúmeras áreas de Ciência da Computação, como, por exemplo, sistemas operacionais, onde ela é usada para organizar prioridades dos processos.

Dois grandes motivos sustentam a escolha do tema deste projeto. Primeiro, uma vez que a fila faz parte dos tipos primitivos de uma linguagem, haverá menos manipulação de ponteiros na mesma, portanto erros envolvendo-os são menos prováveis de ocorrer. Segundo, a linguagem siC é mais alto-nível que C devido à abstração desta estrutura de dados básicas, e, de maneira geral, pode ser mais *user-friendly*. Nesse sentido, o usuário (da linguagem) leigo deverá entender como a estrutura funciona, bem como suas vantagens/desvantagens e usabilidade; porém a implementação de cada uma estará a cargo da própria siC.

Gramática

A seguir será apresentada a gramática da linguagem siC, baseada em C [2]. Alguns comentários são feitos ao longo da gramática para facilitar o entendimento das variáveis e nomenclatura utilizada. As palavras reservadas da linguagem são representadas aqui como *tokens*. As variáveis e constantes são representadas como *identifiers*, que por sua vez é uma expressão regular, e a única diferença entre este e *identifier_struct* é que o segundo tem acesso ao início da fila caso este seja o tipo do *identifier*.

Antes de apresentá-la, segue abaixo algumas alterações e correções da versão passada deste arquivo:

1 As variáveis *argument*, *compare_expression* e *type_queue* foram removidas para aumentar a simplicidade da gramática, uma vez que elas possuiam apenas uma regra;

2 As variáveis *arguments*, *identifier_struct_expression*, *statement* e *statements* foram renomeadas para *argList*, *type_struct_expression*, *stmt* e *stmtList* respectivamente, para aumentar a legibilidade da gramática;

3 As variáveis *identifiers*, *identifier*, *identifier_struct*, *caractere*, *letra* e *digito* foram removidas pois estes valores vêm direto do analisador léxico, atribuídos à nova variável chamada *value*;

4 Foi criada uma variável *valueList* que permite, por exemplo, chamadas de função com mais de um parâmetro;
5 A variável *statement* à direita das regras de IF, IF-ELSE e WHILE foram substituídas por *block*. Um block representa apenas um *statement* ou uma lista de *statements* entre chaves;
6 As operações sobre dados do tipo fila foram substituídas de "+"para "SETLAST"e de -"para "RMVFIRST"(Ver exemplo na Seção 6);
7 Foram adiciondos os tokens EQ ("=="), NEQ ("!="), LEQ («="), GEQ (»="), LT («") e GT (») para facilitar a passagem do lex para o bison.
8 Foi criada uma variável nova program_start, para possibilitar a criação e impressão de programas com uma função ou mais.

Segue abaixo a gramática proposta cuja variável inicial é *program_start*, com as alterações acima em vermelho e as características diferenciais da linguagem siC em negrito.

```
token: WHILE, IF, ELSE, RETURN, SETLAST, RMVFIRST
token: QUEUE, FIRST, VOID, FLOAT, INT, CHAR
token: EQ, NEQ, LEQ, GEQ, LT, GT

program_start
   → program

program
   → program function
   | function

function
   → type_struct ID ( argList ) { stmtList RETURN identifier ; }

valueList
→ valueList , value
| value
| ε
```

Existe um novo tipo de dado, *QUEUE*, que será composto por tipos simples de dados apenas (ou seja, não será possível criar uma variável do tipo fila em que seus elementos também são filas). Caso a variável seja do tipo fila, ela poderá obter o primeiro elemento através do comando "ID.FIRST".

```
value
→ NUM_INT
| NUM_FLOAT
| CARACTERE
| ID
| ID . FIRST

type_struct
   → type_simple
    | QUEUE GT type_simple LT

type_simple
```

```
   → VOID | FLOAT | INT | CHAR

argList
   → argList , type_struct ID
   | type_struct ID
   | ε
```

A seguir serão descritas quatro estruturas básicas da linguagem siC: comando com repetição, condicional, expressões matemáticas e expressões com pilhas e filas. A última contempla as operações de adicionar elemento no topo da pilha ou no fim da fila, "SETLAST", e remover do topo ou do início da fila, "RMVFIRST", onde o valor do elemento retirado é armazenado no último operando da expressão.

```
stmtList
   → stmtList stmt
   | ε

stmt
   → type_struct ID ;
    | ID ( valueList ) ;
    | ID = ID ( valueList ) ;
    | IF ( value compare_assignment value ) block
    | IF ( value compare_assignment value ) block ELSE block
    | WHILE ( value compare_assignment value ) block
    | ID = assignment_expression ;
    | type_struct_expression

block
 → stmt
    | { stmtList }

compare_assignment
   → EQ | NEQ | LEQ | GEQ | LT | GT

assignment_expression
   → assignment_expression + term
    | assignment_expression - term
    | term

term
   → term * factor
    | term / factor
    | factor

factor
   → value
    | ( assignment_expression )


type_struct_expression    → ID = ID . SETLAST ARROW value ;
    | ID = ID . RMVFIRST ;
```

Analisador Léxico

A principal tarefa do analisador léxico é examinar cada elemento do código fonte (variáveis, símbolos, números, etc), reconhecê-los com base em certos *tokens* e classificá-los em grupos de lexemas. Além disso, ele pode realizar outras funções como eliminar espaços em brancos, tabulações, quebras de linhas e comentários; armazenar e acompanhar os números da linha e coluna corrente no momento de sua execução; Informar mensagens de erro ou avisos de prevenção de erro diretamente ao usuário da linguagem [1].

Existem três termos distintos bastante relacionados com analisador léxico. O primeiro, já citado anteriormente, é o *token*, um par onde o primeiro elemento é o *token name* (símbolo abstrato que representa um tipo de unidade léxica, como a palavra chave "while", por exemplo) e onde o segundo elemento é um atributo (informação adicional e opcional sobre o *token*). O segundo termo é o *pattern*, uma descrição, em forma de expressão regular, do um lexema de um token. No caso de um número inteiro, por exemplo, o *pattern* seria uma sequência de um ou mais dígitos de 0 à 9. Por fim, o termo lexema significa uma sequência de caracteres no programa fonte que correspondem com o *pattern* de um lexema específico, ou seja, cada lexema é uma instância de um token. No caso de existirem mais de uma correspondência, o lexema será a instância do *token* cujo *pattern* aparece primeiro no arquivo .lex.

Esta seção apresenta o analisador léxico FLEX, utilizado neste trabalho, além de toda a descrição do arquivo .lex contruído a partir da gramática descrita no capítulo anterior.

FLEX: The Fast Lexical Analyzer

Flex é uma ferramenta que gera um programa, chamado de *scanner*, cuja função é identificar *patterns* no código fonte. Ele recebe como entrada um arquivo de entrada especificados pelo usuário que serão reconhecidos a partir de expressões regulares mescladas com código em C (chamadas de descrição) no arquivo .lex [3]. Com o comando *Flex nome_do_arquivo.lex*, um código fonte chamado *lex.yy.c* é criado e nele existe uma função chamada *yylex()*, a qual realiza de fato as operações do *scanner*. Esse código, ao ser compilado corretamente com a flag *-lfl* da biblioteca do flex, gera um arquivo objeto executável que recebe uma entrada qualquer e gera uma saída que depende do código em C que foi escrito no .lex (imprimir o lexema identificado, contabilizar o número de linhas, imprimir mensagem de erro léxico, etc).

Neste projeto foram utilizadas duas variáveis globais muito úteis: *yytext* e *yyin*. A primeira contém o lexema que foi reconhecido como *token*. Esta variável é modificada sempre que um novo lexema é identificado no código fonte. A segunda define como a entrada será lida, que pode ser tanto pela entrada padrão quanto por arquivo.

Arquivo lex

O código fonte de extensão .lex é composto por três partes: definições, regras e código em C do usuário. Na seção de definições é onde são declarados nomes para certas expressões regulares, para facilitar a escrita das regras na próxima seção. Neste projeto foram feitas seis definições, apresentadas na Tabela 2. A primeira representa um dígito apenas, de 0 à 9. A segunda é uma letra maiúscula ou minúscula e o símbolo $, que em

	Pattern	Ação
1	\n	Identifica uma quebra de linha e incrementa variável *lines* para contagem de linhas
2	[\t]+	Identifica um ou mais espaço ou tabulação
3	"//"[^\n]*	Ignora tudo a frente do comentário de uma linha "//"exceto quebra de linha
4	{digito}+{letra}+{digito}*	Gera o erro identificado na linha *lines*: Sufixo inválido no número inteiro
5	{digito}+"."{digito}*{letra}+{digito}*	Gera o erro identificado na linha *lines*: Sufixo inválido no número float
6	{digito}+"."{digito}*	Identifica números float
7	{digito}+	Identifica números inteiros
8	"'"({letra}|{digito})"'"	Identifica valor para uma variável do tipo char, que pode ser uma letra ou dígito
9	"''"	Gera o erro identificado na linha *lines*: Constante de caractere vazia
10	"==", "!=", «=", »=", «", »"	Identifica símbolos de comparação entre dois elementos
11	".", ";", ",", "'", "{", "}", "(", ")"	Identifica pontuação e delimitadores de blocos e valores de char
12	"+", -", "*", "/"	Identifica operadores matemáticos
13	(?i:"VOID")	Identifica palavra chave tipo void com as letras em caixa-alta ou caixa-alta.
14	(?i:"FLOAT")	Identifica palavra chave tipo float com as letras em caixa-alta ou caixa-alta
15	(?i:"INT")	Identifica palavra chave tipo inteiro com as letras em caixa-alta ou caixa-alta
16	(?i:"CHAR")	Identifica palavra chave tipo char com as letras em caixa-alta ou caixa-alta
17	(?i:"QUEUE")	Identifica palavra chave tipo fila com as letras em caixa-alta ou caixa-alta
18	(?i:"FIRST")	Identifica palavra chave que representa primeiro elemento da fila com as letras em caixa-alta ou caixa-alta
19	(?i:"IF")	Identifica palavra chave condicional if com as letras em caixa-alta ou caixa-alta
20	(?i:"ELSE")	Identifica palavra chave condicional else com as letras em caixa-alta ou caixa-alta
21	(?i:"WHILE")	Identifica palavra chave do laço while com as letras em caixa-alta ou caixa-alta
22	(?i:"RETURN")	Identifica palavra chave return, de retorno de função, com as letras em caixa-alta ou caixa-alta
23	{id}	Reconhece um identificador cuja regra é: não é permitido conter símbolos além de $, letras e dígitos e não é permitido começar a palara com dígito
24	.	Gera o erro identificado na linha *lines*: *Token* desconhecido

Tabela 1. Tabela de regras

	Nome	Definição
1	digito	[0-9]
2	letra	[a-zA-Z$]
6	id	[a-zA-Z$][a-zA-Z$0-9]*

Tabela 2. Tabela de definições

C pode compor o nome de um identificador. A terceira representa os símbolos de comparação entre dois números elementos (que serão do tipo char, inteiro e float). A quarta representa delimitadores do código siC, para finalizar um comanto, definir um escopo, etc. A quinta definição apresenta as quatro operações matemáticas básicas. A sexta e última apresenta uma expressão regular que especifica o formato de um identificador: ele deve começar com uma letra ou $ e pode terminar com letras, $s ou dígitos.

A segunda parte do código lex é composto pelas regras que são um par de *pattern* e ação que devem estar na mesma linha. A Tabela 1 apresenta cada regra utilizada no projeto. No código fonte, todos os elementos da entrada que são identificados são imprimidos, exceto quebra de linha, espaços, tabulações e comentários, que são ignorados. Existem três variáveis contadoras que são utilizadas nas ações: *lines*, que começa com um e é incrementada sempre que uma quebra de linha é reconhecida, e *errors*, que começa com zero e é incrementada sempre que um erro léxico é encontrado. Ao longo da execução são imprimidos as descrições dos erros léxicos na tela e, ao final, o número total de erros.

A ordem em que as regras estão é importante para o funcionamento correto do programa, pois se existir mais de uma correspondência de *pattern* para um elemento, ele será identificado pela regra que aparecer primeiro. Neste projeto, as regras das keywords deve vir antes das regras dos identificadores, pois assim, se a seguinte entrada *int x = 2;* for lida, por exemplo, o elemento *int* será reconhecido como identificador e também tipo inteiro, porém a regra de keyword deverá identificá-lo.

Os *patterns* contidos entre chaves foram definidos na seção de descrição. Em relação aos demais, segue abaixo a descrição de algumas expressões regulares.

[^\n] : Reconhece tudo exceto espaço e quebra de linha (Exemplo: regra 3);
{digito} : Reconhece apenas um dígito (Exemplo: regra 8);
{digito}* : Reconhece zero ou mais dígitos (Exemplo: regra 6);
{digito}+ : Reconhece um ou mais dígitos (Exemplo: regra 5);
“abc” : Reconhece a sequência de caracteres "abc"(Exemplo: regra 9);
“a” | “b” : Reconhece o caractere "a"ou o "b"(Exemplo: regra 8);
(?i:"AB") : Reconhece as sequências "AB", "Ab", "aB", "ab"(Exemplo: regra 13);
. : Reconhece qualquer elemento (Exemplo: regra 24).

A terceira parte do código lex é composta por código em C, que define como será lida a entrada (por arquivo ou pela entrada padrão) e define também as ações tomadas por cada *pattern*.

Erros Léxicos

Foram reconhecidos quatro erros léxicos em siC. Caso exista um elemento que comece com dígitos e termine com letras, o usuário será informado de que o sufixo de

letras é inválido para um tipo inteiro (regra 4). Caso exista um elemento que comece com dígitos, tenha depois um ponto, e termine com letras, o usuário será informado de que o sufixo de letras é inválido para um tipo float (regra 5). Caso exista na entrada duas aspas simples, uma seguida da outra, o programa entende que entre eles deveria existir algum caractere que seria o valor que algum char, portanto o usuário será informado de que a constante de caractere está vazia (regra 9). Por fim, caso exista algum elemento não identificado na entrada, o usuário será informado (regra 24).

Para testar o código foram criados dois arquivos de extensão .sic, um de acordo com as normas especificadas neste projeto (teste_correto.sic), outro com todos os quatro tipos de erros léxicos (teste_errado.sic). Cinco erros léxicos são reportados neste último arquivo:

1 **ERROR on line 4 : Invalid suffix on integer “0i0”**
Erro de variável começando começando com dígito, ou número inteiro contendo algum caractere;
2 **ERROR on line 9 : Invalid suffix on floating “0.0a0”**
Erro de número float contendo algum caractere;
3 **ERROR on line 16 : Unknown token ’!’**
Erro de caractere desconhecido;
4 **ERROR on line 17 : Empty character constant “ ‘’ ”**
Nesta linha ocorreu a inserção de um caractere vazio na pilha q;
5 **ERROR on line 21 : Unknown token ’@’**
Outro erro de caractere desconhecido.

Dificuldades enfrentadas

Nesta fase do projeto, as principais dificuldades foram construir as expressões regulares que formam os *patterns*, bem como ordená-las de forma que o analisador respeite as regras de precedência de reconhecimento dos padrões. Além disso, foi um desafio procurar por erros léxicos, uma vez que os principais e mais conhecidos são sintáticos.

Analisador Sintático

O analisador sintático recebe como entrada o retorno de cada elemento aprensentado na tabela 1 do analisador léxico e constrói duas estrutura de dados: a árvore sintática, que representa a hierarquia do código e a tabela de símbolos, que apresenta os identificadores do programa. O parser escolhido para realizar o projeto foi gerado pelo Bison.

GNU Bison: Parser Generator

O Bison é um gerador de Parser do tipo *top-down* LALR(1) (*Left-to-Right scanning, Rightmost derivation with Looahead*), escrito por Robert Corbett e Richard Stallman. A versão utilizada foi a 3.0.4. O Bison é uma implementação do YACC ("*Yet another compiler-compiler*"), gerador de parser LALR com *lookahead* também.

Arquivo y e structs

A partir do projeto do analisador sintático foi necessário criar um *Makefile* para ligar a biblioteca *structs.h* e conectar os analisadores léxicos e sintáticos, portanto basta dar o comando *make* para criar o executável e *make clean* para limpar a pasta.

Para representar cada variável da gramática, fez-se uma *struct* chamada *Variable* apresentada abaixo, onde o atributo (1) *variable_name* é o nome da variável, (2) *variable_tag* é seu identificador, (3) *variable_num_nexts* é a quantidade de filhos cuja regra selecionada da variável possui, (4) *token* é o próprio token passado do léxico para o sintático, (5) *rule_num* é o identificador desta regra selecionada, (6) *id_index* é o índice dos identificadores na tabela de símbolos e (7) *variable_list* é um ponteiro para uma lista de variáveis filhas da regra selecionada. *MAX_WORD* define um tamanho máximo para o nome de uma variável e o token (elementos não dependentes do usuário da linguagem siC) e *MAX_CHILD_RULES* define o número máximo de filhos que uma regra pode possuir.

```
1  #define MAX_WORD 64
2  #define MAX_CHILD_RULES 5
3  typedef struct Variable {
4      char variable_name[MAX_WORD];
5      int variable_tag;
6      int variable_num_nexts;
7      char token[MAX_WORD];
8      int rule_num;
9      int id_index[5];
10     struct Variable *variable_list[MAX_CHILD_RULES];
11 } Variable;
12 }
```

A gramática está localizada no arquivo 110118995.y, agora com ações semânticas sempre ao final de cada regra. As variável foram definidas como sendo do tipo *Variable* e os demais *tokens*, do tipo *string*. Todas as ações semânticas fazer o mesmo: um vetor de x *structs varList* é alocado na memória, onde x é o número de filhos que aquela regra possui. O ponteiro para cada filho (representados por $x). é atribuído para *varList[i]* e, por fim, a função para criar uma nova variável é chamada e seu retorno é atribuído à variável cuja regra está sendo executada (representada por $$). Os parâmetros desta função, *new_variable*, são (1) a tag da variável, (2) número de filhos que a regra possui, (3) a lista de variáveis para as quais ela aponta, (4) um terminal da regra, (5) o identificador da regra para aquela variável e (6) os índices dos identificadores na tabela de símbolos. Quando a regra possui algum token, ele é adicionado à tabela de símbolos na ação semântica também, então a função *add_symbol_on_table* é chamada e seu retorno (índice na tabela de símbolos) é atribuído à *id_index[i]*. A função de criar nova variável está localizada no arquivo structs.h, assim como a *struct Variable* e as funções de mostrar a árvore sintática, adicionar um elemento na tabela de símbolos e mostrar a tabela de símbolos.

A função *new_variable* apenas alloca espaço para uma *struct Variable* e passa os elementos do parâmetro para os atributos da *struct*. Para cada variável, é atribuído dentro de um *switch-case* seu nome, tag e, quando necessário, o *token*. A função *add_symbol_on_table* adiciona um terminal passado por parâmetro à um vetor de strings global, se este terminal ainda não está presente no vetor. A função *show_symbol_table* percorre este vetor ao final do programa, imprimindo na tela todos os terminais.

Árvore sintática

A árvore sintática foi construída ao longo da leitura da entrada, pois cada ação semântica cria novos nós de baixo para cima. Nesse sentido, a impressão da árvore po-

deria ser feita inversamente nas ações semânticas, porém, por questões de estética, foi feito uma função *show_tree* para tal. Ela é chamada pela primeira regra da variável inicial da gramática (topo da árvore) e recebe como parâmetro um nó de *Variable*, o número de *tabs* necessárias para a impressão de uma árvore legível e o índice do primeiro filho daquela regra (para facilitar a impressão na função recursiva). A função verifica se o *token* daquele nó é vazio (identificado pela string "END"), ou algum operador matemático, significando que aquele nó é uma *Variable* com filhos. Se ele é, então seu nome é impresso e em seguida os elementos que compõem aquela regra, buscando identificadores na tabela de símbolos quando necessário. Se ele possui filhos, a função *show_tree* é chamada recursivamente. Se ele não é, ou seja, se o nó é uma folha, seu terminal é impresso.

Erros sintáticos

O tipo de recuperação de erro escolhido foi o Panic Mode: algumas variáveis possuem uma regra cuja ação semântica é *error "x"*, onde x é uma string que representa o estado seguro o qual, a partir dalí, o parser pode prosseguir.

Para testar o código foram utilizados dois arquivos de extensão .sic, um de acordo com as normas especificadas neste projeto (teste_correto.sic), outro com todos os tipos de erros sintáticos (teste_errado.sic). Todos os erros léxicos foram removidos do arquivo *teste_errado*. Três erros sintáticos auto-explicativos podem ser reportados neste último arquivo:

1 (sin) ERROR near line 6 : Erro nos argumentos da funcao
2 (sin) ERROR near line 8 : syntax error
  (sin) ERROR near line 16 : Erro nos argumentos da funcao
3 (sin) ERROR near line 17 : Erro na operacao matematica

Dificuldades enfrentadas

A principal dificuldade foi a criação da estrutura da árvore sintática. A princípio tentei construir uma *struct* para cada variável, o que funcionou até o momento em que foi necessário imprimir a árvore. Como não consegui percorrer uma árvore de várias structs diferentes, criei uma *union* cujos atributos eram todas estas *structs*, porém me deparei com um código gigante com vários pequenos erros e então resolvi começar tudo de novo com apenas uma string chamada *Variable* que possui certos atributos desnecessários para algumas regras, porém aumenta a legibilidade do programa.

Além disso, acredito que a impressão da árvore não esteja correta da forma que está, pois a ordem com que os identificadores aparecem não está óbvia na função *show_tree*. As regras de escopo na tabela de símbolos não foram implementadas.

Não foi implementada ação semântica para a variável *type_struct_expression* devido ao *warning* ainda não resolvido que ocorreu na gramática da Figura 2.

```
110118995.y:268.27-52: aviso: conflito de tipos na ação padrão: <v> != <str> [-Wother]
 type_struct_expression:   ID '.' SETLAST ARROW value  //  x.setlast->y (Add fim da fila)
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
110118995.y:269.29-50: aviso: conflito de tipos na ação padrão: <v> != <str> [-Wother]
                         | ID '=' ID '.' RMVFIRST // x = y.rmfirst;
                           ^^^^^^^^^^^^^^^^^^^^^^
```

Figura 2. Este warning não impediu a criação do arquivo executável

Analisador Semântico

A análise semântica utiliza da árvore sintática para checar a consistência da linguagem. Uma de suas obrigações mais importantes é a checagem de tipo. No caso do siC, existem várias restrições a serem consideradas:

- Para adicionar um elemento A de tipo simples (char, int ou float) no fim da fila de um elemento struct B, a atribuição deve ser do tipo B = B.SETLAST->A, onde A deverá ter tipo compatível com o de B, ou seja, se B for fila de inteiros, A deve ser um inteiro;
- Para remover um elemento A de tipo simples (char, int ou float) do início da fila de um elemento struct B, a atribuição deve ser do tipo B = B.RMVFIRST, onde A deverá ter tipo compatível com o de B, ou seja, se B for fila de inteiros, A deve ser um inteiro;
- Nenhuma operação matemática (*assignment_expression*) pode conter um identificador B do tipo fila, apenas seu início, ou seja, B.FIRST.

Um exemplo de código em siC é aprensentado a seguir. O programa adiciona três elementos numa fila de inteiros e depois eles são somados um a um e armazenados na variável *sum*. Ao final, a variável *lixo*, recém retirada da fila, é adicionada à *sum*. Nesse sentido, o resultado final de sum deve ser 7.

```
1  VOID main () {
2      QUEUE<INT> q;
3      INT sum, INT lixo;
4
5      q = q.setlast->0;
6      q = q.setlast->1;
7      q = q.setlast->2;
8      q = q.setlast->3;
9      sum = 0;
10
11     WHILE (q.FIRST != 0) {
12         sum = (sum + q.FIRST);
13         lixo = q.rmvfirst;
14     }
15     sum = sum + lixo;
16
17     RETURN 0;
18 }
```

Referências

[1] A. V. Abo, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers - Principles, Techniques and Tools* 2nd ed. 1986

[2] ANSI C Yacc grammar, `http://www.quut.com/c/ANSI-C-grammar-y.html`, 18 12 2012.

[3] Flex: The Fast Lexical Analyser, `http://flex.sourceforge.net/`, The Flex Project, 2008