

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

Gabriella de Oliveira Esteves, 110118995

¹Departamento 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 equivalente em outra linguagem [1]. Ele pode ser dividido em sete fases, ilustrado na Figura 1.

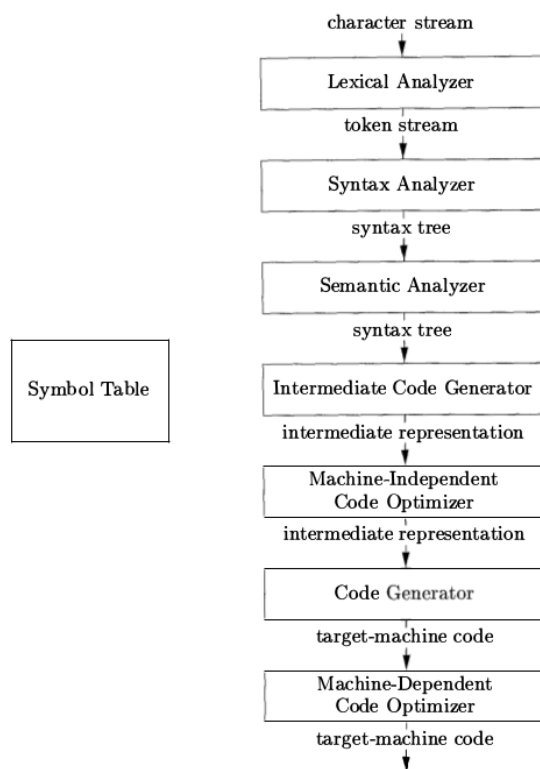


Figura 1. Fases de um compilador

- 1 **Analizador 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 **Analizador Sintático:** Constrói uma representação gramatical dos tokens em forma de árvore;

- 3 **Analizador 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 do analisador léxico para o analisador sintático. **Nenhuma alteração foi feita na gramática do analisador sintático para o semântico, nem do semântico para a geração de código intermediário.**

- 1 As variáveis *argument*, *compare_expression* e *type_queue* foram removidas para aumentar a simplicidade da gramática, uma vez que elas possuíam 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 adicionados 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 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 ;

```

Analizador 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

| | <i>Pattern</i> | <i>Ação</i> |
|----|---|--|
| 1 | <code>\n</code> | Identifica uma quebra de linha e incrementa variável <i>lines</i> para contagem de linhas |
| 2 | <code>[\t]+</code> | Identifica um ou mais espaço ou tabulação |
| 3 | <code>"/"[\^\\n]*</code> | Ignora tudo a frente do comentário de uma linha <code>"/"</code> exceto quebra de linha |
| 4 | <code>{dígito}+{letra}+{dígito}*</code> | Gera o erro identificado na linha <i>lines</i>: Sufixo inválido no número inteiro |
| 5 | <code>{dígito}+."{dígito}*{letra}+{dígito}*</code> | Gera o erro identificado na linha <i>lines</i>: Sufixo inválido no número float |
| 6 | <code>{dígito}+."{dígito}*</code> | Identifica números float |
| 7 | <code>{dígito}+</code> | Identifica números inteiros |
| 8 | <code>""({letra} {dígito})""</code> | Identifica valor para uma variável do tipo char, que pode ser uma letra ou dígito |
| 9 | <code>""</code> | Gera o erro identificado na linha <i>lines</i>: Constante de caractere vazia |
| 10 | <code>"==", "!=", "<=", ">=", "<", ">"</code> | Identifica símbolos de comparação entre dois elementos |
| 11 | <code>".", ";", ",", ":", "{", "}", "(", ")", "</code> | Identifica pontuação e delimitadores de blocos e valores de char |
| 12 | <code>"+", "-", "*", "/"</code> | Identifica operadores matemáticos |
| 13 | <code>(?i:"VOID")</code> | Identifica palavra chave tipo void com as letras em caixa-alta ou caixa-alta. |
| 14 | <code>(?i:"FLOAT")</code> | Identifica palavra chave tipo float com as letras em caixa-alta ou caixa-alta |
| 15 | <code>(?i:"INT")</code> | Identifica palavra chave tipo inteiro com as letras em caixa-alta ou caixa-alta |
| 16 | <code>(?i:"CHAR")</code> | Identifica palavra chave tipo char com as letras em caixa-alta ou caixa-alta |
| 17 | <code>(?i:"QUEUE")</code> | Identifica palavra chave tipo fila com as letras em caixa-alta ou caixa-alta |
| 18 | <code>(?i:"FIRST")</code> | Identifica palavra chave que representa primeiro elemento da fila com as letras em caixa-alta ou caixa-alta |
| 19 | <code>(?i:"IF")</code> | Identifica palavra chave condicional if com as letras em caixa-alta ou caixa-alta |
| 20 | <code>(?i:"ELSE")</code> | Identifica palavra chave condicional else com as letras em caixa-alta ou caixa-alta |
| 21 | <code>(?i:"WHILE")</code> | Identifica palavra chave do laço while com as letras em caixa-alta ou caixa-alta |
| 22 | <code>(?i:"RETURN")</code> | Identifica palavra chave return, de retorno de função, com as letras em caixa-alta ou caixa-alta |
| 23 | <code>{id}</code> | Reconhece um identificador cuja regra é: não é permitido conter símbolos além de \$, letras e dígitos e não é permitido começar a palavra com dígito |
| 24 | <code>.</code> | Gera o erro identificado na linha <i>lines</i>: 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);
- (?:“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 na **versão do analisador léxico**, 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 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 apreendido na tabela 1 do analisador léxico e constrói duas estruturas 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 Lookahead*), 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
2 #define MAX_WORD 64
3 #define MAX_CHILD_RULES 5
4 typedef struct Variable {
5     char variable_name[MAX_WORD];
6     int variable_tag;
7     int variable_num_nexts;
8     char token[MAX_WORD];
9     int rule_num;
10    int id_index[5];
11    struct Variable *variable_list[MAX_CHILD_RULES];
12 } Variable;
```

A gramática está localizada no arquivo 110118995.y, agora com ações semânticas sempre ao final de cada regra. As variáveis foram definidas como sendo do tipo *Variable* e os demais *tokens*, do tipo *string*. Todas as ações semânticas fazem 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 aloca 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-

Erros sintáticos

Para testar o código na **versão do analisador sintático** 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:

- ### Dificuldades enfrentadas

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

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

Analizador 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 $B = B.SETLAST \rightarrow A$, 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.

Apesar de não ter sido feitas mudanças na gramática do analisador sintático para o semântico, várias alterações foram feitas no código. Nesta etapa do projeto, o objetivo foi delimitar os escopos das variáveis, identificar seus tipos, diferenciar na tabela de símbolos variável de função e de número ou caractere. Para isso foi feito:

- A tabela de símbolos não é mais um vetor de strings. Agora ela é uma tabela hash da biblioteca *uthash*, indexada pelo nome do símbolo concatenado com o identificador da função e com o identificador dos escopos. A impressão da tabela de símbolos apresenta o iterador para apresentar o número de símbolos, o identificador da função, o escopo superior ao escopo corrente, o escopo corrente em que a variável/função foi declarada, o tipo da variável e o nome.

```
1
2 typedef struct symbol_hash_table {
3     char key[MAX_WORD+4]; // Chave da tabela hash
4     char str_id[MAX_WORD]; // Nome do ID, num ou caractere
5     char type[MAX_WORD]; // Seu tipo
6     int scope; // Seu escopo
7     int higher_scope; // Seu escopo alcançavel anterior
8     int func_scope; // Seu escopo de funcao
9     UT_hash_handle hh; // Handle da hash
10 } symbol_hash_table;
```

- O numero máximo de escopos é 9 (MAX_SCOPES) e ele é contabilizado do mais global (HIGHEST_SCOPE = 0) para o mais específico. O número máximo de funções é 9 (MAX_FUNCS);
- A variável *id_index* que indexava os símbolos na tabela de símbolos foi trocada por *symbol_ids* que é um vetor das strings (IDs) contidos em cada regra da gramática (utilizado para imprimir árvore na tela);
- Foram adicionados alguns atributos a mais na struct Variáveis, para reconhecimento os tipos dos elementos lidos (NUM_INT, NUM_FLOAT, CARACTERE, ID). A struct final ficou:

```

1 typedef struct Variable {
2     char variable_name[MAX_WORD]; // Nome da variavel
3     int variable_tag; // Identificador da variavel
4     int variable_num_nexts; // Numero de filhos da variavel
5     char token[MAX_WORD]; // Identificador operandos e operadores
6     int rule_num; // Identificador da regra da variavel
7
8     // Numero de simbolos na regra escolhida da variavel
9     char symbol_ids[MAX_SYMBOLS_FOR_RULE][MAX_WORD];
10
11     // Filhos da variavel
12     struct Variable *variable_list[MAX_CHILD_RULES];
13
14     // Analisador semantico: Tipo sintetizado do noh
15     char type_syn[MAX_WORD];
16 } Variable;

```

Este trabalho **não permite ainda** a operação com variáveis do tipo fila.

A impressão final é dada no terminal e apresenta quatro elementos: lista de erros, árvore sintática, tabela de símbolos e matriz de escopo. A lista de erros apresentará apenas os erros citados na subseção 6.1. A árvore sintática é exatamente a mesma do analisador sintático. A tabela de símbolos já foi descrita acima. Por fim, a matriz de escopo representa nos eixos x e y os escopos do código inteiro, sendo a função 0 possui os escopos 0, 1, 2, ... n; a função 1 possui os escopos 0, 1, 2, ... m; porém ao final os escopos são contabilizados como 0, 1, 2, ..., n, n+1, n+2, ..., n+m. Se o elemento matriz_de_escopos[i][j] é igual à 1, significa que o escopo i enxerga as variáveis declaradas no escopo j. Se elemento matriz_de_escopos[i][j] é igual à 0, então o escopo i não enxerga as variáveis da coluna j. A linha 0 não aparece na impressão pois não existem variáveis globais. A imagem abaixo mostra um exemplo.

| ===== MATRIZ DE ESCOPO ===== | | | | | | | |
|------------------------------|---|---|---|---|---|---|---|
| Escopos: | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| Escopo 1: | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Escopo 2: | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Escopo 3: | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Escopo 4: | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Escopo 5: | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Escopo 6: | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

Figura 3. As operações que ocorrem no escopo 3 (linha 3) podem utilizar variáveis dos escopos 3, 2, 1 e 0 (globais e funções). Já as operações que ocorrem no escopo 5 (linha 5) podem utilizar variáveis dos escopos 5 e 0 apenas.

Um exemplo de código em siC é apresentado 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 6.

```

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

```

Erros semânticos

Foram reconhecidos 4 erros semânticos em siC. Para simplificação e fácil visualização, este trabalho apresenta apenas os erros observados na etapa do analisador semântico.

Para testar o código na **versão do analisador semântico**, 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). Quatro erros léxicos são reportados neste último arquivo:

- 1 (sem) **ERROR on line 6 : 'a' redeclared as different kind of symbol:** Erro ocorre quando variável está sendo declarada pela segunda vez (ou mais);
- 2 (sem) **ERROR on line 12 : 'x' undeclared:** Erro ocorre quando variável está sendo utilizada, porém não foi declarada nem no escopo corrente, nem nos escopos alcançáveis pelo escopo corrente;
- 3 (sem) **ERROR on line 15 : variable or field 'f' declared void:** Erro ocorre quando variável é declarada como *void*;
- 4 (sem) **ERROR on line 33 : void value not ignored as it ought to be:** Erro ocorre quando algum ID tenta receber valor de retorno de uma função do tipo *void*, ou quando existe alguma operação matemática envolvendo algum operando do tipo *void*;

Dificuldades enfrentadas

A tarefa mais difícil desta etapa foi a definição dos escopos das variáveis, uma vez que levei bastante tempo para manipular bem as ações semânticas. Para contabilizar o escopo corretamente dentro dos blocos, foi necessário aplicar uma ação chamada *Mid-Rule Actions* [4]. Abaixo está a seção do código em que foi usado este tipo de ação, na variável *block*, que é utilizado sempre ao final das regras IF, IF-ELSE e WHILE, que sempre terminam ou com uma lista de *statments* delimitada por chaves ou por apenas um *statement* sem delimitação por chaves. O escopo, porém, deve ser contabilizado nas regras

entre o ')' e block (como é feito em C), ou dentro das regras IF, IF-ELSE e WHILE, entre '(' e block (como é feito em java onde é possível fazer *for(int i = 0; i < n; i++)*). Então a maneira mais fácil que encontrei para isto foi aplicar a *Mid-Rule* antes dos *statements*, na regra do bloco.

```

1 // ...
2 stmt: IF '(' value compare_assignment value ')' block
3
4 // ...
5
6 block: { higher_scope++; scope++; } stmt { // ...
7       | '{' { higher_scope++; scope++; } stmtList '}' { // ...
8 // ...

```

Geração de código intermediário

A fase de geração de código tem por objetivo diminuir o esforço de tradução de um código na linguagem escolhida para *assembly* e, em seguida, código de máquina. A partir da árvore sintática anotada obtida após a análise semântica, é possível criar um código de três endereços, próximo ao *assembly*. É possível, também, otimizar a representação dos dados em expressões utilizando grafo (grafo acíclico direcionado) ao invés de árvore, onde os nós internos são os operadores e as folhas são os operandos. Caso um operando seja utilizado duas vezes na expressão, haverá duas setas apontando para ele. Assim, a economia de espaço além gerar um código mais sucinto, também o faz mais eficiente.

O grafo descrito acima gera código de três endereços utilizando variáveis temporárias. Para o armazenamento dessas instruções, existem três estruturas (*records*) que realizam esta tarefa [1]: quadrupla, tripla e tripla indireta. A quadrupla é uma matriz de 4 colunas, operação, argumento 1, argumento 2 e resultado. Observe na Figura 4 que cada linha do código gerado é representado facilmente por uma linha na tabela. Essa estrutura, porém, deixa a desejar na questão de economia de espaço uma vez que instruções com menos de três endereços não utilizando o segundo argumento. Além disso, a coluna de resultados poderia ser facilmente eliminada.

| | <i>op</i> | <i>arg₁</i> | <i>arg₂</i> | <i>result</i> |
|---|-----------|------------------------|------------------------|----------------|
| 0 | minus | c | | t ₁ |
| 1 | * | b | t ₁ | t ₂ |
| 2 | minus | c | | t ₃ |
| 3 | * | b | t ₃ | t ₄ |
| 4 | + | t ₂ | t ₄ | t ₅ |
| 5 | = | t ₅ | | a |
| | | ... | | |

(a) Three-address code

(b) Quadruples

Figura 4. Criação de *record* quadrupla.

A estrutura tripla resolve estes problemas de otimização utilizando ponteiros para as linhas da tabela, que representam os próprios resultados. Observe na Figura 5 que

cada linha do código de três endereços representa uma tripla. Isto é bom no sentido de economia de espaço, porém pode não gerar um código tão eficiente, uma vez que algumas instruções seriam mais apropriadas se mudassem de lugar no código. Isto não é possível nas triplas devido às referências diretas às linhas com ponteiros. Por exemplo, pode ser mais apropriado a declaração de todas as variáveis antes do programa de fato começar.

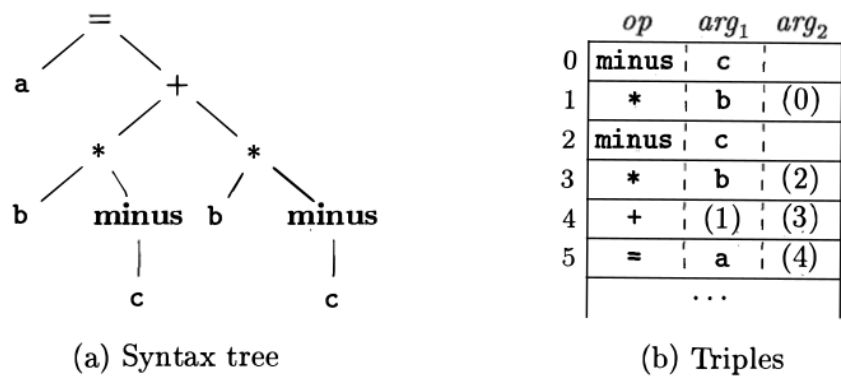


Figura 5. Criação de *record tripla*.

A estrutura tripla indireta resolve este problema ao utilizar uma outra tabela separadamente para referenciar as linhas na ordem de execução, deixando a outra tabela (tripla) como estava originalmente, conforme a Figura 6 apresenta.

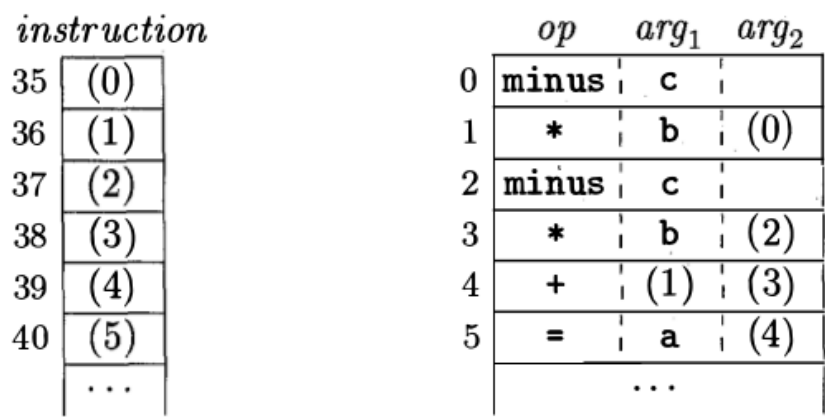


Figura 6. Criação de *record tripla indireta*.

Para a representação dos tipos das variáveis, são usados estruturas chamadas **expressões de tipos**. Essas são árvores construídas dentro das ações semânticas da gramática com o objetivo de estabelecer checagem de tipo no código. Duas expressões de tipos somente são equivalentes se possuem o mesmo tipo básico, são construídas aplicando-se a mesma estrutura e se um denota o outro a partir de do nome do seu tipo.

Arquivo `codigo_gerado.inter`

Nesta fase foi implementada após certas regras da gramática algumas funções para impressão do código intermediário em um arquivo chamado `codigo_gerado.inter`.

Foi feito esforço para que ele apresentasse o mesmo código dos programas em ".sic" em três endereços, entretanto, o código não foi gerado corretamente. Devido à falhas de compreensão das tarefas nesta etapa, não foi possível definir ações fortes o suficiente para geração eficiente e completa de código de três endereços.

Para gerar o código executável, é necessário apenas executar *make*.

Já para gerar o código intermediário, é necessário executar:
`./sic_compiler < teste_correto.sic`

Trabalhos Futuros

Para completar este trabalho, é necessário implementar operações com variáveis do tipo fila; implementar expressões de tipo para o tipo fila, uma vez que além deste tipo só existe tipo simples (void, float, int e char); Criação de estrutura quadrupla, tripla ou tripla indireta, para o armazenamento de código de três endereços. De preferência, tripla indireta, uma vez que o código gerado pode ser otimizado mais facilmente. Por fim, é necessário criar um arquivo intermediário de qualidade que se assemelhe com *assembly* para a implementação da tradução do programa sic.

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
- [4] Using Mid-Rule Actions, https://www.gnu.org/software/bison/manual/html_node/Using-Mid_002dRule-Actions.html visitado em 04-11-2016