Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
Brasil

Analisador Semântica e Geração de Código Intermediário

Leandro Hideki Aihara
Thiago Martendal Salvador
Pablo Daniel Riveros Strapasson

Professor
Alvaro Junior Pereira Franco

Florianópolis
Agosto de 2021

Sumário

1. Tarefa ASem

1.1 Introdução

Apresentamos neste trabalho o desenvolvimento das fases finais do processo de compilação, que são a análise semântica e geração de código intermediário.

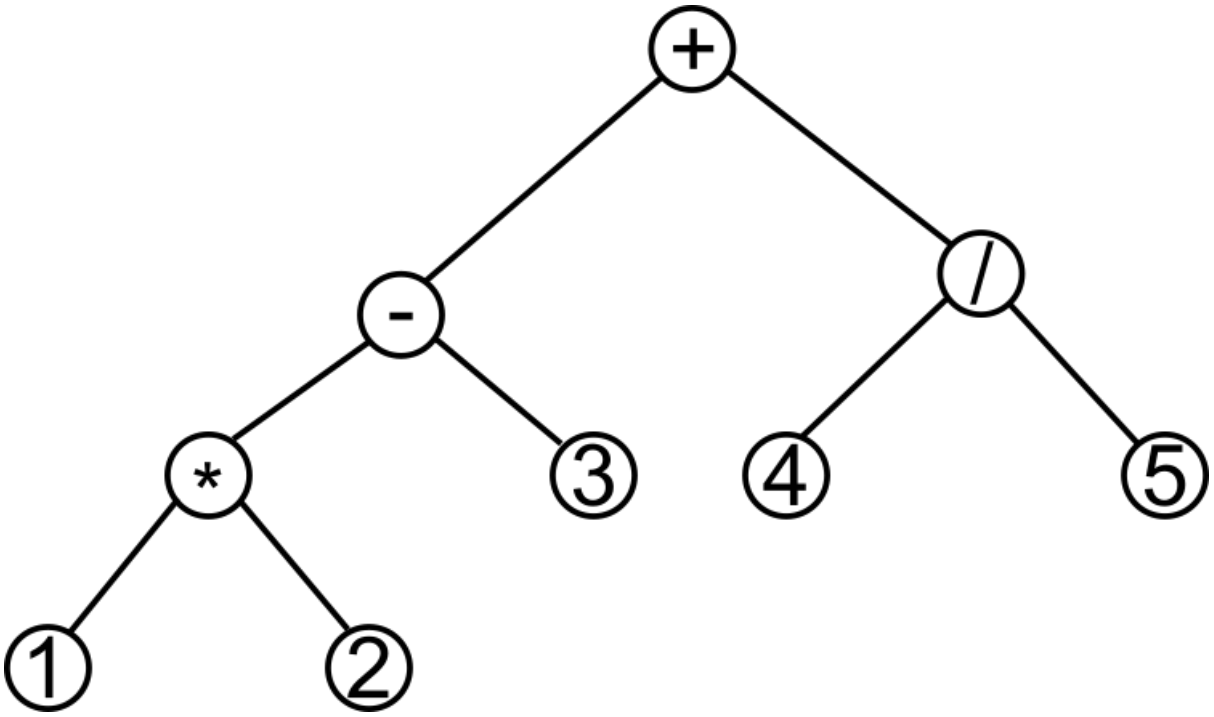
Os arquivos do projeto estão disponíveis no envio do trabalho, juntamente com um makefile de opções de compilação e execução, e um arquivo readme mostrando a interação com o trabalho.

O arquivo README foi enviado nos arquivos do trabalho, mas também pode ser visualizado aqui:
https://github.com/thiagomartendal/Trabalho3Compiladores/blob/main/README.md

As opções de compilação e execução estão disponíveis no arquivo README.

1.2 Construção da Árvore de Expressão

A árvore de expressão é construída utilizando ações semânticas em pontos específicos da gramática. Dada uma expressão: 1*2-3+4/5, a árvore gerada é +-*123/45. A representação na forma gráfica da árvore gerada é:



A seguir é apresentado uma tabela da SDD L-atribuída com a parte da gramática que gera expressões aritméticas:

Produção	**Ação Semântica**

NUMEXPRESSION -> TERM NTERM	if (op != "") {
	  expressao = op+expressao;
	  op = "";
	}
	expressoes.push_back(expressao);
	expressao = "";

NTERM -> ADD TERM NTERM	op += "+";
NTERM -> SUB TERM NTERM	op += "-";
TERM -> UNARYEXPR MUL UNARYEXPR	expressao += "*"+expAux; expAux = "";
TERM -> UNARYEXPR DIV UNARYEXPR	expressao += "/"+expAux; expAux = "";
TERM -> UNARYEXPR PRC UNARYEXPR	expressao += "%"+expAux; expAux = "";
TERM -> UNARYEXPR	expAux = "";
AFACTOR -> ICT	expAux += "+"+std::string(yytext);
AFACTOR -> FCT	expAux += "+"+std::string(yytext);
AFACTOR -> SCT	expAux += "+"+std::string(yytext);
SFACTOR -> ICT	expAux += "-"+std::string(yytext);
SFACTOR -> FCT	expAux += "-"+std::string(yytext);
SFACTOR -> SCT	expAux += "-"+std::string(yytext);
FACTOR -> ICT	expAux += yytext;
FACTOR -> FCT	expAux += yytext;
FACTOR -> SCT	expAux += yytext;
FACTOR -> NL	expAux += “null”

Nesta SDD foi apenas mencionado as variáveis necessárias para a construção da árvore nas ações semânticas. Cada ação semântica listada acima ocorre no fim de cada produção listada, onde as variáveis cooperam entre si para a construção da árvore, logo os atributos são atributos sintetizados. E como cada ação semântica da SDD acima ocorre ao fim

de cada produção, a SDD é L-atribuída. O programa não constrói árvores para expressões com parênteses.

Abaixo podemos ver a SDT correspondente a SDD L-atribuída acima.

Produções

NUMEXPRESSION -> TERM NTERM {if (op != "") {
expressao = op+expressao;
op = "";
}
expressoes.push_back(expressao);
expressao = "";}

NTERM -> ADD TERM NTERM {op += "+";}
NTERM -> SUB TERM NTERM {op += "-";}
TERM -> UNARYEXPR MUL UNARYEXPR {expressao += "*"+expAux; expAux = "";}
TERM -> UNARYEXPR DIV UNARYEXPR {expressao += "/"+expAux; expAux = "";}
TERM -> UNARYEXPR PRC UNARYEXPR {expressao += "%"+expAux; expAux = "";}
TERM -> UNARYEXPR {expAux = "";}
AFACTOR -> ICT {expAux += "+"+std::string(yytext);}
AFACTOR -> FCT {expAux += "+"+std::string(yytext);}
AFACTOR -> SCT {expAux += "+"+std::string(yytext);}
SFACTOR -> ICT {expAux += "+"+std::string(yytext);}
SFACTOR -> FCT {expAux += "+"+std::string(yytext);}
SFACTOR -> SCT {expAux += "+"+std::string(yytext);}
FACTOR -> ICT {expAux += yytext;}
FACTOR -> FCT {expAux += yytext;}
FACTOR -> SCT {expAux += yytext;}
FACTOR -> NL {expAux += “null”;}

1.3 Inserção do Tipo na Tabela de Símbolos

A inserção de tipo é feita na fase de análise léxica. Quando é feita a análise de um identificador, é verificado se este é uma variável ou um nome de função, e se for uma variável, seu tipo é adicionado junto ao nome na tabela de símbolos. Se for uma função, seu tipo é função.

Esta ação é mais simples pois o tipo da variável já acompanha seu nome, o que torna a verificação simples.

Não foi utilizada uma SDD para a inserção de tipo, já que este item não é realizado com uso da gramática da linguagem.

1.4 Verificação de Tipos

A verificação de tipos é realizada comparando-se o tipo que foi atribuído a variável com o tipo dos elementos que são atribuídos. Quando uma produção que reconhece que uma constante ou uma variável é percorrida, é chamado o método checaTipoExpressao passando o tipo da constante ou da variável. No método checaTipoExpressao é feita a comparação do tipo que foi passado com o tipo que foi atribuído para a variável na tabela de símbolos. Também é possível obter o tipo na produção de declaração de variável, e compará-lo ao tipo da expressão. Se alguma destas comparações for falsa, é reconhecido um erro de tipo, pois a variável tem atribuída um valor ou expressão com tipo diferente do tipo declarado. Caso o tipo passado não seja diferente, não há a ocorrência de erro, e o programa segue sua operação.

1.5 Verificação de Identificadores por Escopo

A verificação de identificadores por escopo é feita atribuindo-se o escopo do método onde o identificador foi declarado na tabela de símbolos, junto ao nome e ao tipo do identificador. Se o identificador for uma variável, seu tipo é o tipo da variável. Se o identificador for uma função, seu tipo é função. A verificação então ocorre nas fases de análise léxica e análise sintática.

Na fase de análise léxica a verificação ocorre quando um identificador é reconhecido. Se a tabela de símbolos já contiver um identificador com o mesmo nome, no mesmo escopo, então é sinalizado um erro que diz que uma variável com o mesmo nome foi declarada duas vezes no mesmo escopo. Se for uma função, a mensagem aponta que uma função foi declarada repetidamente.

Na fase de análise semântica, a verificação ocorre quando um escopo é encerrado, dessa forma, é possível verificar o que ocorreu no escopo recém finalizado para se determinar se houveram declarações repetidas. Um escopo é determinado pela ocorrência de chaves - ‘{’ e ‘}’ - e quando o escopo é encerrado, é chamado o método checarVariavelEscopo que realiza a verificação. O método então procura na tabela de símbolos se um identificador com mesmo nome e mesmo escopo ocorre mais de uma vez, e se ocorrer, é retornada a mensagem de erro indicando declaração repetida.

Nos atributos de cada identificador, também é mantido um conjunto de escopos, que armazena os escopos onde o identificador foi encontrado. E o atributo escopoDeclaracao demarca em qual escopo o identificador foi declarado pela primeira vez, e esta variável é utilizada para comparar se uma variável com mesmo nome foi declarada mais de uma vez no mesmo escopo.

O parâmetro escopoFuncao é muito útil para determinar qual a função que está sendo analisada a partir de seu escopo, e também demarca em qual função encontra-se a variável, permitindo então realizar mais verificações de variáveis e funções repetidas.

1.6 Comandos Dentro de Escopo

Nesta seção será descrita a verificação da ocorrência do comando break dentro de um escopo de laço de repetição.

A verificação é feita da seguinte forma: na gramática, quando é percorrido o terminal FOR na produção FORSTAT, a variável escopoLoop é setada como true. Então, quando é percorrida a produção STATEMENT -> BRK PV, é verificado se a variável escopoLoop é igual a true. Se a variável for false, significa que nenhum laço de repetição foi iniciado antes do comando break, o que ocasiona erro e uma mensagem de erro aponta que o comando break foi declarado fora de um escopo correto.

Dessa maneira, é possível verificar se um comando break está ou não dentro do escopo do laço de repetição, basta verificar se um loop foi iniciado anteriormente.

Quando o escopo do laço de repetição termina, a variável escopoLoop é declarada como false novamente para não interferir na verificação em outro loops.

2. Tarefa GCI

2.1 Introdução a Geração de Código

Nesta seção será discutida a implementação da geração de código intermediário. A geração de código é realizada através de ações semânticas que são realizadas em determinados pontos da gramática. Quando uma produção da gramática é percorrida, caso esta tenha uma ação semântica, a ação será executada gerando o código que corresponde a entrada que foi analisada. Abaixo é apresentada uma SDD teórica sobre a geração de código. Esta SDD simboliza a idéia de como o código deve ser gerado, mas a implementação em código segue um caminho um pouco diferente, pois a SDD teórica consegue trabalhar muito bem com nodos, e o código pode ser implementado de uma forma mais direta já que a ferramenta Bison facilita bastante a simulação do processo de uma árvore.

Portanto, tendo a auxílio da ferramenta Bison para a implementação, a SDD abaixo tem sua função de passar uma visão teórica sobre o processo de geração de código, mas sem a necessidade de segui-lo de forma literal. Mas ainda sim, há a similaridade nas regras

semânticas com a implementação, que embora seja difícil de se visualizar devido ao código ter ficado bastante bagunçado, elas estão lá representadas.

Infelizmente, houveram alguns problemas em determinados pontos, onde o código gerado não conseguiu seguir totalmente o padrão requisitado pela codificação em três endereços. Isso ocorre nas expressões aritméticas, na aritmética de ponteiros dos vetores, e na atribuição de funções a variáveis. Outras pequenas inconsistências podem ocorrer, principalmente na aritmética de ponteiros. Mas apesar desses problemas, a ideia central do código de três endereços a da fase de geração de código foi representada.

2.2 SDD L-atribuída para Conv-CC-2021-1

Abaixo podemos ver a SDD L-atribuída para a gramática Conv-CC-2021-1:

Produções	**Regras Semânticas**
PROGRAM -> STATEMENT	PROGRAM.node = STATEMENT.node
PROGRAM -> FUNCLIST	PROGRAM.node = FUNCLIST.node
PROGRAM -> &	PROGRAM.sin = PROGRAM.her
FUNCLIST -> FUNCDEF FUNCLIST’	FUNCLIST.node = FUNCLIST’.sin FUNCLIST’.her = FUNCDEF.node
FUNCLIST’ -> FUNCLIST	FUNCLIST’.node = FUNCLIST.node
FUNCLIST’ -> &	FUNCLIST’.sin = FUNCLIST’.her
FUNCDEF -> def identfunc (PARAMLIST){STATELIST}	FUNCDEF.node = STATELIST.sin STATELIST.her = PARAMLIST.node
PARAMLIST -> int identPARAMLIST’	PARAMLIST’.her = new node(int ident, -, PARAMLIST.her) PARAMLIST.sin = PARAMLIST’.sin
PARAMLIST -> float identPARAMLIST’	PARAMLIST’.her = new node(float ident, -, PARAMLIST.her) PARAMLIST.sin = PARAMLIST’.sin
PARAMLIST -> string identPARAMLIST’	PARAMLIST’.her = new node(string ident, -, PARAMLIST.her) PARAMLIST.sin = PARAMLIST’.sin
PARAMLIST -> &	PARAMLIST.sin = PARAMLIST.her
PARAMLIST’ -> , PARAMLIST	PARAMLIST.her = new node(, , -, PARAMLIST’.her) PARAMLIST’.sin = PARAMLIST.sin

PARAMLIST' -> &	PARAMLIST'.sin = PARAMLIST'.her
STATEMENT -> VARDECL;	STATEMENT.node = VARDECL.node
STATEMENT -> ATRIBSTAT;	STATEMENT.node = ATRIBSTAT.node
STATEMENT -> PRINTSTAT;	STATEMENT.node = PRINTSTAT.node
STATEMENT -> READSTAT;	STATEMENT.node = READSTAT.node
STATEMENT -> RETURNSTAT;	STATEMENT.node = RETURNSTAT.node
STATEMENT -> IFSTAT	STATEMENT.node = IFSTAT.node
STATEMENT -> FORSTAT	STATEMENT.node = FORSTAT.node
STATEMENT -> {STATELIST}	STATEMENT.node = STATELIST.node
STATEMENT -> break ;	STATEMENT.node = new node(break; ,-,-)
STATEMENT -> ;	STATEMENT.node = new node(; , -, -)
VARDECL -> int ident VARDECL'	VARDECL'.her = new node(int ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin
VARDECL -> float ident VARDECL'	VARDECL'.her = new node(floar ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin
VARDECL -> string ident VARDECL'	VARDECL'.her = new node(string ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin
VARDECL' -> NVARDECL	VARDECL'.node = NVARDECL.node
VARDECL' -> &	VARDECL'.sin = VARDECL'.her
NVARDECL -> [int_constant]NVARDECL'	NVARDECL'.her = new node([int_constant], -, NVARDECL.her) NVARDECL.sin = NVARDECL'.sin
NVARDECL' -> NVARDECL	NVARDECL'.node = NVARDECL.node
NVARDECL' -> &	NVARDECL'.sin = NVARDECL'.her
ATRIBSTAT -> LVALUE = ATRIBSTAT'	ATRIBSTAT'.her = new node(=, ATRIBSTAT'.her, LVALUE.node) ATRIBSTAT.sin = ATRBSTAT'.sin

ATRIBSTAT' -> EXPRESSION	ATRIBSTAT'.node = EXPRESSION.node
ATRIBSTAT' -> ALLOCEXPRESSION	ATRIBSTAT'.node = ALLOCEXPRESSION.node
ATRIBSTAT' -> FUNCCALL	ATRIBSTAT'.node = FUNCCALL.node
FUNCCALL -> identfunc(PARAMLISTCALL)	PARAMLISTCALL'.her = new node(identfunc, -, PARAMLISTCALL.her) FUNCCALL.sin = PARAMLISTCALL.sin
PARAMLISTCALL -> identPARAMLISTCALL'	PARAMLISTCALL'.her = new node(ident, -, PARAMLISTCALL.her) PARAMLISTCALL.sin = PARAMLISTCALL'.sin
PARAMLISTCALL -> &	PARAMLISTCALL.sin = PARAMLISTCALL.her
PARAMLISTCALL' -> , PARAMLISTCALL	PARAMLISTCALL.her = new node(, , -, PARAMLISTCALL'.her) PARAMLISTCALL.sin = PARAMLISTCALL.sin
PARAMLISTCALL' -> &	PARAMLISTCALL'.sin = PARAMLISTCALL'.her
PRINTSTAT -> print EXPRESSION	EXPRESSION.her = new node(print, -, PRINTSTAT.her) PRINSTAT.sin = EXPRESSION.sin
READSTAT -> read LVALUE	LVALUE.her = new node(read, -, READSTAT.her) READSTAT.sin = LVALUE.sin
RETURNSTAT -> return	RETURNSTAT.node = new node(return,-,-)
IFSTAT -> if(EXPRESSION) STATEMENT	C.F = IFSTAT.next C.V = new label() testEXPRESSION = new label() STATEMENT.next = testEXPRESSION IFSTAT.code = testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code||
IFSTAT -> ifelse(EXPRESSION) STATEMENT else STATEMENT¹	C.F = new label() C.V = new label() testEXPRESSION = new label()

	STATEMENT.next = testEXPRESSION IFSTAT.code = testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code|| C.F|| STATEMENT¹.code
FORSTAT -> for(ATRIBSTAT; EXPRESSION; ATRIBSTAT¹) STATEMENT	C.F = FORSTAT.next C.V = new label() A.L = new label() testEXPRESSION = new label() STATEMENT.next = testESPRESSION FORSTAT.code = ATRIBSTAT.code|| testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code|| A.L|| ATRIBSTAT¹.code|| goto testEXPRESSION
STATELIST -> STATEMENT STATELIST'	STATELIST.node = STATELIST'.sin STATELIST'.her = STATEMENT.node
STATELIST' -> STATELIST	STATELIST'.node = STATELIST.node
STATELIST' -> &	STATELIST'.sin = STATELIST'.her
ALLOCEXPRESSION -> new ALLOCEXPRESSION'	ALLOCEXPRESSION'.her = new node(new, -, ALLOCEXPRESSION.her) ALLOCEXPRESSION.sin = ALLOCEXPRESSION.sin
ALLOCEXPRESSION' -> int [NUMEXPRESSION]	NUMEXPRESSION.her = new node(int, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin
ALLOCEXPRESSION' -> float [NUMEXPRESSION]	NUMEXPRESSION.her = new node(float, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin
ALLOCEXPRESSION' -> string[NUMEXPRESSION]	NUMEXPRESSION.her = new node(string, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin
EXPRESSION -> NUMEXPRESSION EXPRESSION'	EXPRESSION.node = EXPRESSION'.sin EXPRESSION'.her = NUMEXPRESSION.node
EXPRESSION' -> < NUMEXPRESSION	NUMEXPRESSION.her = new node(<, -, EXPRESSION'.her)

	EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> > NUMEXPRESSION	NUMEXPRESSION.her = new node(>, -, EXPRESSION’.her) EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> <= NUMEXPRESSION	NUMEXPRESSION.her = new node(<=, -, EXPRESSION’.her) EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> >= NUMEXPRESSION	NUMEXPRESSION.her = new node(>=, -, EXPRESSION’.her) EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> == NUMEXPRESSION	NUMEXPRESSION.her = new node(==, -, EXPRESSION’.her) EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> != NUMEXPRESSION	NUMEXPRESSION.her = new node(!=, -, EXPRESSION’.her) EXPRESSION.sin = NUMEXPRESSION.sin
EXPRESSION’ -> &	EXPRESSION.sin = EXPRESSION.her
NUMEXPRESSION -> TERM NUMEXPRESSION’	NUMEXPRESSION.node = NUMEXPRESSION’.sin NUMEXPRESION’.her = TERM.node
NUMEXPRESSION’ -> NNUMEXPRESSION	NUMEXPRESSION’.node = NNUMEXPRESSION.node
NUMEXPRESSION’ -> &	NUMEXPRESSION’.sin = NUMEXPRESSION’.her
NNUMEXPRESSION -> + TERM NNUMEXPRESSION’	NUMEXPRESSION’.her = new node(+, NNUMEXPRESSION.her, TERM.node) NNUMEXPRESSION.sin = NNUMEXPRESSION’.sin
NNUMEXPRESSION -> - TERM NNUMEXPRESSION’	NUMEXPRESSION’.her = new node(-, NNUMEXPRESSION.her, TERM.node) NNUMEXPRESSION.sin = NNUMEXPRESSION’.sin
NNUMEXPRESSION’ ->	NNUMEXPRESSION’.node =

NNUMEXPRESSION	NNUMEXPRESSION.node
NNUMEXPRESSION' -> &	NNUMEXPRESSION'.sin = NNUMEXPRESSION'.her
TERM -> UNARYEXPR TERM'	TERM.node = TERM'.sin TERM'.her = UNARYEXPR.node
TERM' -> * UNARYEXPR	UNARYEXPR.her = new node(*, -, TERM'.her) TERM'.sin = UNARYEXPR.sin
TERM' -> / UNARYEXPR	UNARYEXPR.her = new node(/, -, TERM'.her) TERM'.sin = UNARYEXPR.sin
TERM' -> % UNARYEXPR	UNARYEXPR.her = new node(%, -, TERM'.her) TERM'.sin = UNARYEXPR.sin
TERM' -> &	TERM'.sin = TERM'.her
UNARYEXPR -> + FACTOR	FACTOR.her = new node(+, -, UNARYEXPR.her) UNARYEXPR.sin = FACTOR.sin
UNARYEXPR -> - FACTOR	FACTOR.her = new node(-, -, UNARYEXPR.her) UNARYEXPR.sin = FACTOR.sin
UNARYEXPR -> FACTOR	UNARYEXPR.node = FACTOR.node
FACTOR -> int_constant	FACTOR.node = new node(int_constant,-,-)
FACTOR -> float_constant	FACTOR.node = new node(float_constant,-,-)
FACTOR -> string_constant	FACTOR.node = new node(string_constant,-,-)
FACTOR -> null	FACTOR.node = new node(null,-,-)
FACTOR -> LVALUE	FACTOR.node = LVALUE.node
FACTOR -> (NUMEXPRESSION)	FACTOR.node = NUMEXPRESSION.node
LVALUE -> identLVALUE'	LVALUE'.her = new node(ident, -, LVALUE.her) LVALUE.sin = LVALUE'.sin

LVALUE' -> NLVALUE	LVALUE'.node = NLVALUE.node
LVALUE' -> &	LVALUE'.sin = LVALUE'.her
NLVALUE -> [NUMEXPRESSION]NLVALUE'	NLVALUE.node = NLVALUE'.sin NLVALUE'.her = NUMEXPRESSION.node
NLVALUE' -> NLVALUE	NLVALUE'.node = NLVALUE.node
NLVALUE' -> &	NLVALUE'.sin = NLVALUE'.her

2.3 Verificação se a SDD é L-atribuída

Por cada não-terminal possuir, tanto, atributos sintetizados e atributos herdados podemos afirmar que a SDD acima é L-atribuída, além de que todas as ações ocorrem ao fim de cada produção, corroborando com a afirmação anterior.

2.4 SDT para SDD de Conv-CC-2021-1

Produções
PROGRAM -> STATEMENT {PROGRAM.node = STATEMENT.node}
PROGRAM -> FUNCLIST {PROGRAM.node = FUNCLIST.node}
PROGRAM -> & {PROGRAM.sin = PROGRAM.her}
FUNCLIST -> FUNCDEF FUNCLIST' {FUNCLIST.node = FUNCLIST'.sin FUNCLIST'.her = FUNCDEF.node}
FUNCLIST' -> FUNCLIST {FUNCLIST'.node = FUNCLIST.node}
FUNCLIST' -> & {FUNCLIST'.sin = FUNCLIST'.her}
FUNDEF -> def identfunc (PARAMLIST){STATELIST} {FUNCDEF.node = STATELIST.sin; STATELIST.her = PARAMLIST.node}
PARAMLIST -> int identPARAMLIST' {PARAMLIST'.her = new node(int ident, -, PARAMLIST.her) PARAMLIST.sin = PARAMLIST'.sin}
PARAMLIST -> float identPARAMLIST' {PARAMLIST'.her = new node(float ident, -, PARAMLIST.her)

PARAMLIST.sin = PARAMLIST'.sin}
PARAMLIST -> string identPARAMLIST' {PARAMLIST'.her = new node(string ident, -, PARAMLIST.her) PARAMLIST.sin = PARAMLIST'.sin}
PARAMLIST -> & {PARAMLIST.sin = PARAMLIST.her}
PARAMLIST' -> , PARAMLIST {PARAMLIST.her = new node(, , -, PARAMLIST'.her) PARAMLIST'.sin = PARAMLIST.sin}
PARAMLIST' -> & {PARAMLIST'.sin = PARAMLIST'.her}
STATEMENT -> VARDECL; {STATEMENT.node = VARDECL.node}
STATEMENT -> ATRIBSTAT; {STATEMENT.node = ATRIBSTAT.node}
STATEMENT -> PRINTSTAT; {STATEMENT.node = PRINTSTAT.node}
STATEMENT -> READSTAT; {STATEMENT.node = READSTAT.node}
STATEMENT -> RETURNSTAT; {STATEMENT.node = RETURNSTAT.node}
STATEMENT -> IFSTAT {STATEMENT.node = IFSTAT.node}
STATEMENT -> FORSTAT {STATEMENT.node = FORSTAT.node}
STATEMENT -> {STATELIST} {STATEMENT.node = STATELIST.node}
STATEMENT -> break ; {STATEMENT.node = new node(break; ,-,-)}
STATEMENT -> ; {STATEMENT.node = new node(; , -, -)}
VARDECL -> int ident VARDECL' {VARDECL'.her = new node(int ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin}
VARDECL -> float ident VARDECL' {VARDECL'.her = new node(floar ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin}
VARDECL -> string ident VARDECL' {VARDECL'.her = new node(string ident, -, VARDECL.her) VARDECL.sin = VARDECL'.sin}
VARDECL' -> NVARDECL {VARDECL'.node = NVARDECL.node}
VARDECL' -> & {VARDECL'.sin = VARDECL'.her}
NVARDECL -> [int_constant]NVARDECL' {NVARDECL'.her = new node([int_constant], -, NVARDECL.her) NVARDECL.sin = NVARDECL'.sin}

NVARDECL' -> NVARDECL {NVARDECL'.node = NVARDECL.node}
NVARDECL' -> & {NVARDECL'.sin = NVARDECL'.her}
ATRIBSTAT -> LVALUE = ATRIBSTAT' {ATRIBSTAT'.her = new node(=, ATRIBSTAT'.her, LVALUE.node) ATRIBSTAT.sin = ATRBSTAT'.sin}
ATRIBSTAT' -> EXPRESSION {ATRIBSTAT'.node = EXPRESSION.node}
ATRIBSTAT' -> ALLOCEXPRESSION {ATRIBSTAT'.node = ALLOCEXPRESSION.node}
ATRIBSTAT' -> FUNCCALL {ATRIBSTAT'.node = FUNCCALL.node}
FUNCCALL -> identfunc(PARAMLISTCALL) {PARAMLISTCALL'.her = new node(identfunc, -, PARAMLISTCALL.her) FUNCCALL.sin = PARAMLISTCALL.sin}
PARAMLISTCALL -> identPARAMLISTCALL' {PARAMLISTCALL'.her = new node(ident, -, PARAMLISTCALL.her) PARAMLISTCALL.sin = PARAMLISTCALL'.sin}
PARAMLISTCALL -> & {PARAMLISTCALL.sin = PARAMLISTCALL.her}
PARAMLISTCALL' -> , PARAMLISTCALL {PARAMLISTCALL.her = new node(, , -, PARAMLISTCALL'.her) PARAMLISTCALL.sin = PARAMLISTCALL.sin}
PARAMLISTCALL' -> & {PARAMLISTCALL'.sin = PARAMLISTCALL'.her}
PRINTSTAT -> print EXPRESSION {EXPRESSION.her = new node(print, -, PRINTSTAT.her) PRINSTAT.sin = EXPRESSION.sin}
READSTAT -> read LVALUE {LVALUE.her = new node(read, -, READSTAT.her) READSTAT.sin = LVALUE.sin}
RETURNSTAT -> return {RETURNSTAT.node = new node(return,-,-)}
IFSTAT -> if({C.F = IFSTAT.next; C.V = new label(); testEXPRESSION = new label()}EXPRESSION) {STATEMENT.next = testEXPRESSION} STATEMENT {IFSTAT.code = testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code||}}
IFSTAT -> ifelse({C.F = new label(); C.V = new label(); testEXPRESSION = new label()} EXPRESSION) {STATEMENT.next = testEXPRESSION} STATEMENT else STATEMENT¹ {IFSTAT.code = testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code|| C.F|| STATEMENT¹.code}

FORSTAT -> for({C.F = FORSTAT.next; C.V = new label(); A.L = new label(); testEXPRESSION = new label()}ATRIBSTAT; EXPRESSION; ATRIBSTAT') {STATEMENT.next = testESPRESSION} STATEMENT {FORSTAT.code = ATRIBSTAT.code|| testEXPRESSION|| EXPRESSION.code|| C.V|| STATEMENT.code|| A.L|| ATRIBSTAT'.code|| goto testEXPRESSION}
STATELIST -> STATEMENT STATELIST' {STATELIST.node = STATELIST'.sin STATELIST'.her = STATEMENT.node}
STATELIST' -> STATELIST {STATELIST'.node = STATELIST.node}
STATELIST' -> & {STATELIST'.sin = STATELIST'.her}
ALLOCEXPRESSION -> new ALLOCEXPRESSION' {ALLOCEXPRESSION'.her = new node(new, -, ALLOCEXPRESSION.her) ALLOCEXPRESSION.sin = ALLOCEXPRESSION.sin}
ALLOCEXPRESSION' -> int [NUMEXPRESSION] {NUMEXPRESSION.her = new node(int, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin}
ALLOCEXPRESSION' -> float [NUMEXPRESSION] {NUMEXPRESSION.her = new node(float, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin}
ALLOCEXPRESSION' -> string[NUMEXPRESSION] {NUMEXPRESSION.her = new node(string, -, ALLOCEXPRESSION'.her) ALLOCEXPRESSION'.sin = NUMEXPRESSION.sin}
EXPRESSION -> NUMEXPRESSION EXPRESSION' {EXPRESSION.node = EXPRESSION'.sin EXPRESSION'.her = NUMEXPRESSION.node}
EXPRESSION' -> < NUMEXPRESSION {NUMEXPRESSION.her = new node(<, -, EXPRESSION'.her) EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> > NUMEXPRESSION {NUMEXPRESSION.her = new node(>, -, EXPRESSION'.her) EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> <= NUMEXPRESSION {NUMEXPRESSION.her = new node(<=, -, EXPRESSION'.her)

EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> >= NUMEXPRESSION {NUMEXPRESSION.her = new node(>=, -, EXPRESSION'.her) EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> == NUMEXPRESSION {NUMEXPRESSION.her = new node(==, -, EXPRESSION'.her) EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> != NUMEXPRESSION {NUMEXPRESSION.her = new node(!=, -, EXPRESSION'.her) EXPRESSION.sin = NUMEXPRESSION.sin}
EXPRESSION' -> & {EXPRESSION.sin = EXPRESSION.her}
NUMEXPRESSION -> TERM NUMEXPRESSION' {NUMEXPRESSION.node = NUMEXPRESSION'.sin NUMEXPRESION'.her = TERM.node}
NUMEXPRESSION' -> NNUMEXPRESSION {NUMEXPRESSION'.node = NNUMEXPRESSION.node}
NUMEXPRESSION' -> & {NUMEXPRESSION'.sin = NUMEXPRESSION'.her}
NNUMEXPRESSION -> + TERM NNUMEXPRESSION' {NUMEXPRESSION'.her = new node(+, NNUMEXPRESSION.her, TERM.node) NNUMEXPRESSION.sin = NNUMEXPRESSION'.sin}
NNUMEXPRESSION -> - TERM NNUMEXPRESSION' {NUMEXPRESSION'.her = new node(-, NNUMEXPRESSION.her, TERM.node) NNUMEXPRESSION.sin = NNUMEXPRESSION'.sin}
NNUMEXPRESSION' -> NNUMEXPRESSION {NNUMEXPRESSION'.node = NNUMEXPRESSION.node}
NNUMEXPRESSION' -> & {NNUMEXPRESSION'.sin = NNUMEXPRESSION'.her}
TERM -> UNARYEXPR TERM' {TERM.node = TERM'.sin TERM'.her = UNARYEXPR.node}
TERM' -> * UNARYEXPR {UNARYEXPR.her = new node(*, -, TERM'.her) TERM'.sin = UNARYEXPR.sin}
TERM' -> / UNARYEXPR {UNARYEXPR.her = new node(/, -, TERM'.her) TERM'.sin = UNARYEXPR.sin}
TERM' -> % UNARYEXPR {UNARYEXPR.her = new node(%, -, TERM'.her) TERM'.sin = UNARYEXPR.sin}
TERM' -> & {TERM'.sin = TERM'.her}

UNARYEXPR -> + FACTOR {FACTOR.her = new node(+, -, UNARYEXPR.her) UNARYEXPR.sin = FACTOR.sin}
UNARYEXPR -> - FACTOR {FACTOR.her = new node(-, -, UNARYEXPR.her) UNARYEXPR.sin = FACTOR.sin}
UNARYEXPR -> FACTOR {UNARYEXPR.node = FACTOR.node}
FACTOR -> int_constant {FACTOR.node = new node(int_constant,-,-)}
FACTOR -> float_constant {FACTOR.node = new node(float_constant,-,-)}
FACTOR -> string_constant {FACTOR.node = new node(string_constant,-,-)}
FACTOR -> null {FACTOR.node = new node(null,-,-)}
FACTOR -> LVALUE {FACTOR.node = LVALUE.node}
FACTOR -> (NUMEXPRESSION) {FACTOR.node = NUMEXPRESSION.node}
LVALUE -> identLVALUE' {LVALUE'.her = new node(ident, -, LVALUE.her) LVALUE.sin = LVALUE'.sin}
LVALUE' -> NLVALUE {LVALUE'.node = NLVALUE.node}
LVALUE' -> & {LVALUE'.sin = LVALUE'.her}
NLVALUE -> [NUMEXPRESSION]NLVALUE' {NLVALUE.node = NLVALUE'.sin NLVALUE'.her = NUMEXPRESSION.node}
NLVALUE' -> NLVALUE {NLVALUE'.node = NLVALUE.node}
NLVALUE' -> & {NLVALUE'.sin = NLVALUE'.her}

2.5 Código Intermediário

Como descrito na introdução deste tópico, o código intermediário que foi gerado foi o código de três endereços. O código de três endereços segue um padrão de representação de dois operandos por variável, como segue para a expressão abaixo:

a = x+y-z

Então, a decomposição em três endereços codifica a atribuição para:

a = x+a1

```
a1 = y-z
```

O exemplo acima foi uma breve explicação do processo, mas se tratando de expressões, deve-se avaliar a precedência de operandos.

Um exemplo mais detalhado mostra como ocorre a tradução para o seguinte código:

```
int main() {
    int i;
    int b[10];
    for (i = 0; i < 10; ++i) {
        b[i] = i*i;
    }
}
```

O código traduzido:

```
main():                 # Label da função
i = 0                   # Atribuição
L1:   if i >= 10 goto L2    # Salto condicional
    t0 = i*i            # Valor da atribuição
    t1 = &b             # Endereço do vetor b
    t2 = t1 + i         # Endereço calculado de b[i]
    *t2 = t0            # Atribuição do valor através do ponteiro de b[i]
    i = i + 1           # Incremento da variável i
    goto L1             # Executa mais uma iteração do loop
L2:                     # Label de saída do loop
```

As estruturas If, If else e laços de repetição tem sua construção definida por labels que determinam o começo do bloco básico, e um label que representa a saída da estrutura. Um label pode ser acessado quando a condição lógica da estrutura é válida ou não (a implementação do código determina se o label será alcançado ou não).

Como dito anteriormente, expressões aritméticas são traduzidas sem seguir o padrão de três endereços, portanto uma atribuição:

```
int val;
val = 1+2*3-4/5;
```

É traduzida como:

```
val = 1+2*3-4/5
```

Não foi encontrada uma boa forma para a geração de código para expressões. Apesar da árvore de expressão ter sido bem definida com a concatenação em determinadas ações semânticas, para a geração de código o processo tornou-se complicado devido às atribuições

de operações em variáveis intermediárias. Como o padrão de três endereços propõe dois valores por operação, teria-se a necessidade de se armazenar operandos em variáveis auxiliares para se seguir o padrão.

Portanto, a tradução correta da atribuição em código de três endereços seria:

Expressão:

```
int val;
val = 1+2*3-4/5;
```

Código de três endereços equivalente:

```
t0 = 2*3
t1 = 4/5
t2 = 1+t0
t3 = t2-t1
```

A tradução apresentada acima não foi possível devido a dificuldade em se atribuir corretamente cada operação a uma variável auxiliar, e a chamada dessa mesma variável no momento correto.

2.6 Saídas do Programa

Sobre a saída apresentada, quando ocorre um erro no código, é retornada uma mensagem de insucesso indicando do que se trata o erro. Caso nenhum erro ocorra, são exibidas as seguintes saídas:

1. A tabela de símbolos com os identificadores declarados, seu tipo, e os locais onde se encontram.
2. As árvores de expressão para todas as expressões aritméticas declaradas no programa, com uma mensagem mostrando que as expressões são válidas.
3. Uma mensagem avisando que não existem variáveis declaradas em escopos incorretos.
4. Uma mensagem dizendo que não existe um comando break fora de um escopo de um laço de repetição.

Estas saídas são exibidas apenas se o programa estiver correto, portanto, nas ocorrência de qualquer erro simplesmente é retornada a mensagem de insucesso.

2.7 Conclusão

Neste trabalho mostramos um pouco da ideia das fases de análise semântica e geração de código, aplicando a teoria desses processos na implementação do software por meio da ferramenta Bison.

Para fins de testes foram disponibilizados alguns programas escritos na linguagem LCC, alguns com poucas linhas para mostrar os resultados mais breves, e três programas com 100 linhas para se analisar o resultado obtido e as inconsistências ocorridas.

Ao se testar expressões, é interessante evitar expressões que utilizem parêntesis ou expressões que contenham vetores ou matrizes, pois o software não está lidando muito bem com a ocorrência destes.

Um bom exemplo de expressão para teste seria:

```
int x;
int a;
a = 1;
x = a*2-3+4/5;
```

Finalizamos este trabalho com a entrega de uma boa compreensão das fases finais do processo de compilação (com exceção da geração de código objeto para gerar o executável, que não é estudado nesta disciplina.

Apesar das inconsistências que não puderam ser corrigidas a tempo, foi desenvolvido o máximo possível para se ter uma visão geral dos processos finais.