Análise Semântica e Geração de Código Intermediário

Henrique Buss

Setembro 2021

1 Tarefa ASem

1.1 Construção da árvore de expressão

A seguir é apresentada a gramática $EXPA$, que é o subconjunto de produções da gramática $ConvCC-2021-1$ (após ser fatorada à esquerda, conforme exercício programa anterior) que derivam expressões aritméticas. Os símbolos terminais da gramática são representados em vermelho.

$$
\begin{aligned}
NUMEXPRESSION &\rightarrow TERM \quad NUMEXPRESSION' \\
NUMEXPRESSION' &\rightarrow + \quad TERM \quad NUMEXPRESSION' \\
&\quad | \quad - \quad TERM \quad NUMEXPRESSION' \mid \varepsilon \\
TERM &\rightarrow UNARYEXPR \quad TERM' \\
TERM' &\rightarrow * \quad UNARYEXPR \quad TERM' \\
&\quad | \quad / \quad UNARYEXPR \quad TERM' \\
&\quad | \quad \% \quad UNARYEXPR \quad TERM' \mid \varepsilon \\
UNARYEXPR &\rightarrow + \quad FACTOR \mid - \quad FACTOR \mid FACTOR \\
FACTOR &\rightarrow int\_constant \mid float\_constant \mid string\_constant \\
&\quad | \quad null \mid LVALUE \mid (NUMEXPRESSION) \\
LVALUE &\rightarrow ident \; LVALUE' \\
LVALUE' &\rightarrow [NUMEXPRESSION] \; LVALUE' \mid \varepsilon
\end{aligned}
$$

Em sequência, é apresentada uma SDD L-atribuída para $EXPA$:

Produção	**Regras Semânticas**
1) NUMEXPRESSION → TERM NUMEXPRESSION'	NUMEXPRESSION'.her = TERM.val NUMEXPRESSION.val = NUMEXPRESSION'.val
2) NUMEXPRESSION' → + TERM $NUMEXPRESSION'_1$	$NUMEXPRESSION'_1$.her = TERM.val NUMEXPRESSION'.val = NUMEXPRESSION'.her + $NUMEXPRESSION'_1$.val
3) NUMEXPRESSION' → - TERM $NUMEXPRESSION'_1$	$NUMEXPRESSION'_1$.her = TERM.val NUMEXPRESSION'.val = NUMEXPRESSION'.her - $NUMEXPRESSION'_1$.val
4) NUMEXPRESSION' → ε	NUMEXPRESSION'.val = NUMEXPRESSION'.her
5) TERM → UNARYEXPR TERM'	TERM'.her = UNARYEXPR.val TERM.val = TERM'.val
6) TERM' → * UNARYEXPR $TERM'_1$	$TERM'_1$.her = UNARYEXPR.val TERM'.val = TERM'.her × $TERM'_1$.val
7) TERM' → / UNARYEXPR $TERM'_1$	$TERM'_1$.her = UNARYEXPR.val TERM'.val = TERM'.her / $TERM'_1$.val
8) TERM' → % UNARYEXPR $TERM'_1$	$TERM'_1$.her = UNARYEXPR.val TERM'.val = TERM'.her % $TERM'_1$.val
9) TERM' → ε	TERM'.val = TERM'.her
10) UNARYEXPR → + FACTOR	UNARYEXPR.val = FACTOR.val
11) UNARYEXPR → - FACTOR	UNARYEXPR.val = -1 × FACTOR.val
12) UNARYEXPR → FACTOR	UNARYEXPR.val = FACTOR.val
13) FACTOR → int_constant	FACTOR.val = int_constant.valLex
14) FACTOR → float_constant	FACTOR.val = float_constant.valLex
15) FACTOR → string_constant	FACTOR.val = string_constant.valLex
16) FACTOR → null	FACTOR.val = null
17) FACTOR → LVALUE	FACTOR.val = LVALUE.val
18) FACTOR → (NUMEXPRESSION)	FACTOR.val = NUMEXPRESSION.val
19) LVALUE → ident LVALUE'	LVALUE'.her = ident.valLex LVALUE.val = LVALUE'.val
20) LVALUE' → [NUMEXPRESSION] $LVALUE'_1$	$LVALUE'_1$.her = LVALUE.her[NUMEXPRESSION.val] LVALUE'.val = $LVALUE'_1$.val
21) LVALUE' → ε	LVALUE'.val = LVALUE'.her

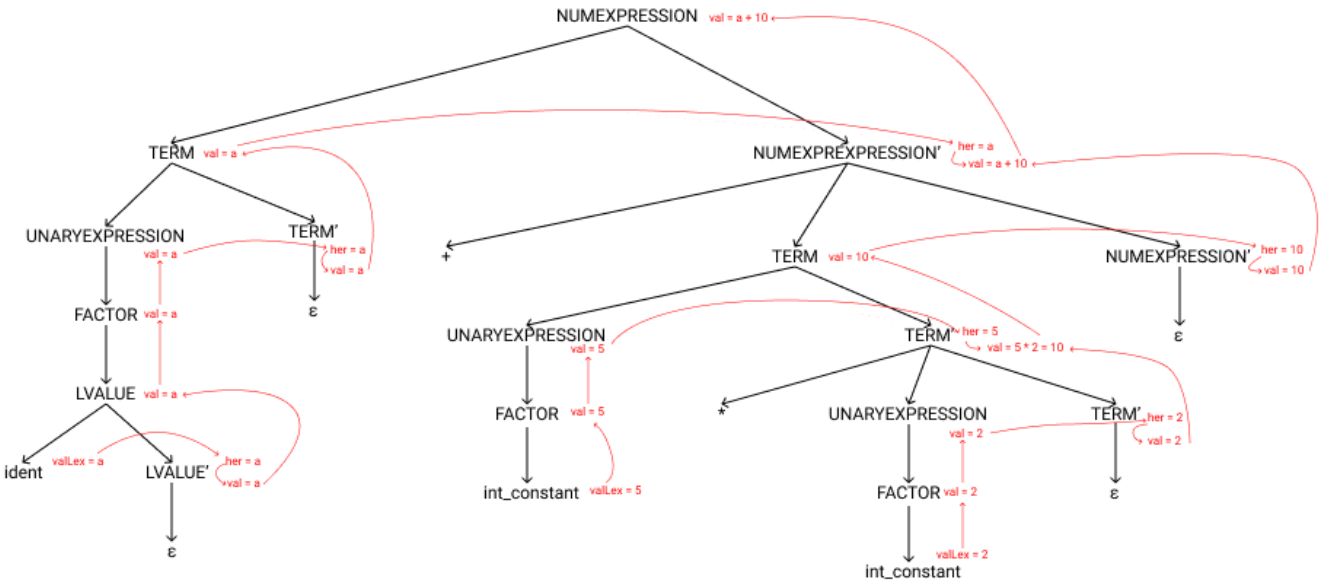
Para que uma SDD seja considerada L-atribuída, ela tem que seguir duas regras:

1. Cada atributo deve ser sintetizado;
2. Cada atributo deve ser herdado do pai, dos irmãos à esquerda, ou dele mesmo, desde que não gere ciclos.

Analisando a tabela acima, podemos ver que a SDD apresentada segue estas duas regras, e portanto, é L-atribuída. Agora apresentamos a SDT que representa $EXPA$:

$$
\begin{aligned}
NUMEXPRESSION &\rightarrow TERM\{NUMEXPRESSION'_1.her = TERM.val)\}NUMEXPRESSION'_1 \\
&\quad \{NUMEXPRESSION.val = NUMEXPRESSION'_1.val\} \\
NUMEXPRESSION' &\rightarrow +TERM\ \{NUMEXPRESSION'_1.her = TERM.val\}NUMEXPRESSION'_1 \\
&\quad \{NUMEXPRESSION'.val = NUMEXPRESSION'.her + NUMEXPRESSION'_1.val\} \\
&\quad \{NUMEXPRESSION.val = NUMEXPRESSION'_1.val\} \\
NUMEXPRESSION' &\rightarrow +TERM\ \{NUMEXPRESSION'_1.her = TERM.val\}NUMEXPRESSION'_1 \\
&\quad \{NUMEXPRESSION'.val = NUMEXPRESSION'.her - NUMEXPRESSION'_1.val\} \\
NUMEXPRESSION' &\rightarrow \varepsilon\{NUMEXPRESSION'.val = NUMEXPRESSION'.her\} \\
TERM &\rightarrow UNARYEXPR\{TERM'.her = UNARYEXPR.val\}TERM'\{TERM.val = TERM'.val\} \\
TERM' &\rightarrow *UNARYEXPR\{TERM'_1.her = UNARYEXPR.val\}TERM'_1 \\
&\quad \{TERM'.val = TERM'.her * TERM'_1.val\} \\
TERM' &\rightarrow /UNARYEXPR\{TERM'_1.her = UNARYEXPR.val\}TERM'_1 \\
&\quad \{TERM'.val = TERM'.her/TERM'_1.val\} \\
TERM' &\rightarrow \%UNARYEXPR\{TERM'_1.her = UNARYEXPR.val\}TERM'_1 \\
&\quad \{TERM'.val = TERM'.her\%TERM'_1.val\} \\
TERM' &\rightarrow \varepsilon\{TERM'.val = TERM'.her\} \\
UNARYEXPR &\rightarrow +FACTOR\{UNARYEXPR.val = FACTOR.val\} \\
UNARYEXPR &\rightarrow -FACTOR\{UNARYEXPR.val = -1 * FACTOR.val\} \\
UNARYEXPR &\rightarrow FACTOR\{UNARYEXPR.val = FACTOR.val\} \\
FACTOR &\rightarrow int\_constant\{FACTOR.val = int\_constant.valLex\} \\
FACTOR &\rightarrow float\_constant\{FACTOR.val = float\_constant.valLex\} \\
FACTOR &\rightarrow string\_constant\{FACTOR.val = string\_constant.valLex\} \\
FACTOR &\rightarrow null\{FACTOR.val = null\} \\
FACTOR &\rightarrow LVALUE\{FACTOR.val = LVALUE.val\} \\
FACTOR &\rightarrow (NUMEXPRESSION\{FACTOR.val = NUMEXPRESSION.val\}) \\
LVALUE &\rightarrow ident\{LVALUE'.her = ident.valLex\}LVALUE'\{LVALUE.val = LVALUE'.val\} \\
LVALUE' &\rightarrow [NUMEXPRESSION\{LVALUE'_1.her = LVALUE.her[NUMEXPRESSION.val]\}] \\
&\quad LVALUE'_1\{LVALUE'.val = LVALUE'_1.val\} \\
LVALUE' &\rightarrow \varepsilon\{LVALUE'.val = LVALUE'.her\}
\end{aligned}
$$

Agora apresentamos uma árvore de derivação que representa a expressão $a + 5 * 2$:



1.2 Inserção do tipo na tabela de símbolos

A seguir é apresentada a gramática DEC, que é o subconjunto de produções da gramática $ConvCC - 2021 - 1$ que derivam declarações de variáveis.

$$
\begin{aligned}
VARDECL &\rightarrow int\ ident\ VARDECL' \mid float\ ident\ VARDECL' \mid string\ ident\ VARDECL' \\
VARDECL' &\rightarrow [int\_constant]\ VARDECL' \mid \varepsilon \\
FUNCDEF &\rightarrow def\ ident(\ PARAMLIST)\ \{\ STATELIST\ \} \\
PARAMLIST &\rightarrow int\ ident,\ PARAMLIST \mid float\ ident,\ PARAMLIST \mid string\ ident,\ PARAMLIST \\
&\quad \mid int\ ident \mid float\ ident \mid string\ ident \mid \varepsilon
\end{aligned}
$$

Por simplicidade, a produção $STATELIST$ foi omitida. O comportamento da SDT seria basicamente percorrer a árvore de declarações, executando um comando *adicionarTipo*, passando o tipo da variável (*int*, *float* ou *string*) toda vez que ver uma nova declaração.

1.3 Verificação de tipos

Dado que temos uma árvore de expressão, podemos "caminhar" pela árvore, definindo uma função $verificarTipo$, que recebe como entrada uma árvore e retorna como saída um tipo. As regras dessa função são:

- Se o nodo for uma constante (como 1, 3.1415 ou "*algumastring*"), sabemos que o tipo desse nodo é o tipo da constante, e simplesmente retornamos esse tipo.
- Se o nodo for uma variável (como a, *segredoDoUniverso* ou *pi*), buscamos o tipo da variável na tabela de símbolos. Se a variável não existir, criamos um erro e paramos a execução do algoritmo. Caso contrário, simplesmente retornamos o tipo da variável.
- Se o nodo for um operando (como $+$, $*$ ou $/$), verificamos o tipo do filho à esquerda (usando a função $verificarTipo$) e o tipo do filho à direita (também usando $verificarTipo$). Se os tipos dos dois filhos são iguais, retornamos esse tipo. Caso contrário, criamos um erro e paramos a execução do algoritmo.
- Podemos também verificar operações inválidas. Por exemplo, uma *string* não pode ser multiplicada ou dividida por outra *string*. Neste caso, se estivermos visitando um nodo de multiplicação (ou divisão, ou qualquer outra operação que não seja válida), e um dos filhos tiver o tipo *string*, criamos um erro.

Podemos então obter o tipo de uma expressão aplicando $verificarTipo$, passando como argumento o nodo raiz da árvore.

1.4 Verificação de identificadores por escopo

Para ter nomes únicos de variável por escopo, podemos guardar uma pilha de escopos. Toda vez que entramos em um escopo diferente (comando if, comando for, etc.), empilhamos o escopo nessa pilha (pode ser simplesmente um número, ou o nome do escopo). Então, cada vez que criamos uma variável, anotamos o escopo no nome da variável. Por exemplo, uma variável definida como a poderia ser chamada de $\$escopo\$0\$a$ (note o uso de \$, para não termos problemas com nomes gerados pelo compilador serem iguais a nomes gerados pelo usuário). E toda vez que sairmos de um escopo, desempilhamos um elemento da pilha e removemos os símbolos deste escopo da tabela de símbolos.

Além disso, toda vez que formos declarar uma variável a, checamos se existe alguma variável terminando com $\$a$ na tabela de símbolos. Se sim, pegamos a variável com maior número de escopo. Se o número do escopo atual for maior do que o número de escopo da variável, podemos criar a variável. Caso contrário, já existe uma variável com o mesmo nome no mesmo escopo, e declaramos um erro.

1.5 Comandos dentro de escopos

Toda vez que entramos em um escopo, podemos empilhar um *contexto*. Um contexto pode ser um $contextoDeFuncao$, $contextoDeIf$, $contextoDeElse$, $contextoDeFor$ ou $contextoDeBloco$. Quando aparecer um comando de *break*, checamos se existe um $contextoDeFor$ na nossa pilha de contextos. Se não acharmos um $contextoDeFor$ na pilha, reportamos um erro. O mesmo pode ser feito para o comando *return*, que só pode ser usado dentro de funções.

2 Tarefa GCI

Para a geração de código intermediário, foi usada uma abordagem baseada em Estados (vista no arquivo $src/Emit/State.elm$), onde percorremos a árvore gerada após a leitura do código fonte, e geramos o código intermediário enquanto percorremos esta árvore. Os arquivos responsáveis por isso podem ser encontrados na pasta $src/Emit$. Conforme geramos o código intermediário, vamos incluindo variáveis definidas pelo usuário e variáveis temporárias definidas pelo compilador em uma

tabela de símbolos, que contém o nome de cada variável, o seu tipo e seu contexto (ver seções 1.4 e 1.5). Temos também uma tabela de funções, que guarda o nome de cada função, e os tipos dos parâmetros que ela recebe.

O estado também é responsável por conter os possíveis erros de compilação. O programa foi arquiteturado de uma maneira em que os arquivos que são responsáveis por emitir o código intermediário não precisam se preocupar se houve algum erro previamente ou não. Eles apenas precisam emitir o erro quando for necessário, e o resto das instruções podem ser como se não houvessem erros.

Para poder mostrar a linha e coluna dos erros, foram adicionados novos campos em alguns dos elementos resultantes do *parsing* do código fonte, que definem o começo e final da definição de alguns termos. Deste modo, podemos indicar onde um erro começou, e onde ele terminou.

Para manipular o estado, existem algumas funções fundamentais, como $initialState$, $addLine$, $createLabel$, $enterContext$, $getVariableType$ e $raiseError$. Com essas funções, os outros módulos podem ser bastante desacoplados da implementação do estado, de modo que, se decidissemos mudar a implementação interna do estado no futuro, não precisaríamos mexer na implementação dos emissores de *statements* e expressões.

Os emissores de código intermediário ($src/Emit/Statement.elm$, $src/Emit/Expression.elm$ e $src/Emit/Program.elm$) são construídos incrementalmente, ou seja, existem pequenos blocos, que são juntados para formar os emissores finais. Por exemplo, existe um emissor para constantes, que é usado para emitir expressões unárias, que é usado para emitir termos, e assim por diante.