Compiladores

Trabalho Final

Cristiano Daitx Ribeiro (111150558)
Marcos Vinícius Treviso (121150107)
Matheus Beniz Bieger (131100019)
Sergio Arthur Crespo Neto (111151985)

1 – Compilação e Execução

Para compilar e executar foi disponibilizado dois arquivos diferentes, um para *linux (compile.sh)* e outro para *windows (compile.bat)*.

Para compilar no *linux* é só executar o arquivo e passar como argumento o texto de entrada:

```
./compile.sh inputs/input1.txt
```

Para compilar no *windows* é só executar o arquivo e passar como argumento o texto de entrada:

```
compile.bat inputs/input1.txt
```

Os arquivos de entrada podem ser encontrados no diretório *inputs/.*

2 – Gramática

Em ordem de deixar o trabalho mais robusto, foram criadas novas produções para as gramáticas e modificada outras. Todas as modificações não afetaram a gramática original, de modo que apenas foram acrescentadas novas configurações ou foram eliminados conflitos pertinentes ao *parser* LALR(1) usado pelo Bison.

As adições estão listadas abaixo:

- Adição do tipo `void` para funções.
- Adição do tipo `char`.
- Adição da operador lógico `!=`.
- Adição de regra para atribuição logo na declação do identificador.
- Adição de regra para operações lógicas permitirem trabalhar com identificadores e valores *booleanos*.
- Adição de regra para operador unário de subtração.

As outras modificações feitas na gramática são muito simples, como por exemplo a troca de recursão à direita por recursão à esquerda, para facilitar as ações de cada produção. Foram eliminados também alguns não-terminais, visto que seu uso podia ser

mesclado com outros, como por exemplo a quebra da gramática para os não terminais *ExpAritmetica, T1, F* para uma unica produção com todos os operadores em *ExpAritmetica*, pois o Bison se encarrega de cuidar das precedências e associatividades dos operadores através de suas definições no início do programa. Outra alteração importante foi a inclusão da produção *DeclVarsAux* na produção *CorpoFuncao*, pois desse modo é eliminado a possibilidade de declarar uma função dentro da outra.

A gramática final está dada abaixo (em vermelho as alterações feitas):

```
Programa       → Declaracoes FuncaoMain

Declaracoes    → DeclVariaveis Declaracoes | DelcFuncao Declaracoes
                 | ε

DeclVariaveis  → Tipo ListaItens ;

DeclVarsAux    → ε | declVarsAux declVariaveis

DeclFuncao     → Tipo id ( Parametros ) { CorpoFuncao } ;
                 | void id ( Parametros ) { CorpoFuncao } ;

FuncaoMain     → Tipo main ( ) { CorpoFuncao } ;

ListaItens     → ListaItens , Item | Item | Item = Expressao

Item           → id | id [ int_num ] | * Item

Parametros     → ε | Tipo Item | Parametros , Tipo Item

Tipo           → int | float | bool | char

CorpoFuncao    → DeclVarsAux Comandos

BlocoComandos  → { Comandos }

Comandos       →  ε | Comando ; Comandos

Comando        → Atribuicao | Retorno | SeEntao | Impressao |Enquanto

Atribuicao     → id = Expressao | id [ int_num ] = Expressao

Expressao      → ExpAritmetica | ExpLogica

ExpAritmetica  → ExpAritmetica + ExpAritmetica
                 | ExpAritmetica – ExpAritmetica
                 | ExpAritmetica * ExpAritmetica
                 | ExpAritmetica / ExpAritmetica
                 | T1
```

```
T1              → ( ExpAritmetica ) | id | id ( ListaExp )
                | id [ int_num ] | - int_num | - float_num | int_num
                | float_num | caractere | booleano

ListaExp        → ExpAritmetica | ListaExp , ExpAritmetica

ExpLogica       → ExpLogica & ExpLogica
                | ExpLogica opou ExpLogica
                | ExpLogica < ExpLogica
                | ExpLogica > ExpLogica
                | ExpLogica = ExpLogica
                | ExpLogica <= ExpLogica
                | ExpLogica >= ExpLogica
                | ExpLogica != ExpLogica
                | T2 & T2 | T2 opou T2

T2              → booleano | id | id [ int_num ]

SeEntao   → if ( ExpLogica ) then BlocoComandos else BlocoComandos
          | if ( T2 ) then BlocoComandos else BlocoComandos

Enquanto  → while ( ExpLogica ) BlocoComandos
          | while ( T2 ) BlocoComandos

Impressao → print ExpAritmetica

Retorno   → return ExpAritmetica
```

3 – Tokens

Os tokens propostos foram bem simples e podem ser vistos na tabela abaixo. Para os terminais que não tem uma entrada abaixo, pode ser considerado que foi criado um token com seu nome específico, visto que são constantes na análise léxica.

Nome	Expressão Regular	Descrição
letra	[A-Za-z]	Todas letras de *a* até *z* maiúsculas e minúsculas.
digito	[0-9]	Todos os dígitos de *0* até *9.*
id	{letra}(_{letra}|_{digito}|{letra}|{digito})*	Começa por letra e em seguida pode conter *sublinhado* entre letras ou números alternativamente e sucessivamente.

numint	{digito}+	Sequência de dígitos com no mínimo um dígito.
numfloat	{digito}+((\.)?{digito}+)?	Um inteiro seguido da possíbilidade de ter um ponto para definir a casa dos milhares e em sequência a possibilidade de ter mais dígitos.
caractere	\'.\'	Qualquer símbolo entre aspas.
aspas	[\"\']	Aspas duplas ou simples.
iderro1	{letra}(_{letra}|_{digito}|{letra}|{digito})*_	Identificador que termina com *sublinhado*.
iderro2	{digito}(_{letra}|_{digito}|{letra}|{digito})*	Identificador que começa por com dígito.
nferro	{digito}+((,)?{digito}+)?	Número de ponto flutuante delimitado por vírgula.

Como foram adicionadas novas regras para permitir novas produções, alguns tokens novos tiveram que ser criados, como o BOOLEANO para os lexemas *true* e *false.* Também foram criados tokens novos chamados de NEQ, CHAR, BOOL, VOID, TRUE, FALSE que representam repectivamente o novo operador lógico *!=,* o tipo *char, bool, void* e as palavras reservadas *true* e *false*. Os demais tokens são extremamente simples e seguem a mesma lógica de raciocínio, eles podem ser encontrados no arquivo `trabalho.lex`.

4 – Detecção de Erros

Não foi implementado um mecanismo para recuperação de erro, portanto quando é encontrado um erro léxico, sintático ou semântico é feita a saída imediata do programada através do comando `exit(0)`.

4.1 – Erros Léxicos

Foram detectados diretamente no arquivo compilado pelo Flex, eles podem ser encontrados no arquivo `trabalho.lex`.

Os erros detectados na análise léxica são:

- Identificador não pode ter mais do que 32 caracteres, exclusive.
- Identificador não pode terminar com sublinhado (token *iderro1*).
- Identificador não pode começar com um dígito (token *iderro2*).

- Ponto flutuante não poe ser delimitado por vírgula (token *nferro*).
- Não pode haver aspas simples (se não for definição de *caractere*) e nem aspas duplas.
- Símbolos não pertencente a linguagem.

Erros durante a análise léxica são simples e poucos, dificilmente podemos identificar muitos erros nessa parte, pois nesse momento não há um entendimento maior do significado do programa.

4.2 – Erros Sintáticos

Para os erros sintáticos foi criada uma classe especial chamada *SyntaxError* que é usada durante toda a parte do projeto do compilador para imprimir diferentes tipos de erros sintáticos encontrados. Essa classe pode ser encontrada no arquivo `SyntaxError.cpp`

Muitos erros sintáticos são detectados pelo próprio *parser* e para especificar os detalhes dos erros, foi usada a deritiva `%define parse.error verbose` no arquivo `Trabalho.yacc.`

Outros erros sintáticos foram detectados através da adição de novas produção à gramática, especialmente para detectar esses eventos errôneos. Todos eles podem ser categorizados em:

- *missReturnType*: A função não tem um retorno declarado.
- *missReturnExpression*: A Função tem um retorno declarado mas não tem expressão de retorno
- *missClose:* Um bloco não é fechado por '}'
- *indexType*: O tipo de índice de um vetor não é inteiro.
- *missLogicExpression*: Falta uma expressão lógica dentro da parte de verficação
- *missAritmeticExpression*: Falta uma expressão aritmética dentro de um bloco
- *undefinedMain*: A função *main* não foi declarada
- *multiplesMain*: Mais de uma função main foi declarada
- *unkownOperator*: Operador não conhecido pelo compilador

4.3 – Erros Semânticos

Para os erros sintáticos foi criada uma classe especial chamada Semantic*Error* que é usada durante toda a parte do projeto do compilador para imprimir diferentes tipos de erros sintáticos encontrados. Essa classe pode ser encontrada no arquivo Semantic*Error.cpp*

Os erros semânticos verificados nesse trabalho são simples e utilizam a tabela de símbolos para (Seção 5) a maioria de suas verificações. Eles podem ser categorizados em:

- *redeclaration*: Identificador (variável ou função) já foi declarado.
- *undeclaration*: Identificador (variável ou função) não foi declarado.
- *incompatible*: Tipos (tipo ou dimensão) incompátiveis entre duas expressões.
- *arrayBound*: Acesso a uma posição do vetor fora dos limites declarados.
- *unexpectedReturnType*: Tipo de retorno diferente do tipo de retorno declarado para a função.
- *unexpectedReturnDimension*: Dimensão diferente do tipo de retorno declarado para a função.
- *incompatibleParameters*: Tipo dos parâmetros passados como argumento na chamada da função são diferentes com os declarados para ela.

5 – Tabela de Símbolos

Para a tabela de símbolos foi implementada duas classes que estão disponíveis no arquivo *SymbolTable.cpp*.

A primeira classe é chamada de *Node* e armazena as informações da tabela de símbolo, ou seja, ela é o equivalente a uma linha na tabela de símbolos. Seus métodos são auto-explicativos. Seus atributos seriam as colunas da tabela e é aonde ficam armazenadas as informações específicas dos identificados, são eles:

- string **name**: guarda o nome do identificador (var. ou função)
- unsigned int **line**: guarda o número da linha onde o id foi declarado
- int **scope**: vai de 1 até n (n = maior escopo declarado).
- ii **type**: tupla com o tipo do id, primeiro ítem indica se o id é uma função, variável, constante ou comando, e a segunda o seu tipo (int, float, char, etc).
- int **dim**: indica a dimensão do id (0 é ponteiro, 1 é variável e n é o tamanho do vetor declarado)
- map<string, *Node> **params**: se o id é uma função, seus parâmetros estão ficam mapeados por essa estrutura de dicionário, onde a chave é o nome do id e o valor é um ponteiro para um nodo na tabela de símbolos que corresponde ao nodo do parâmetro declarado.
- double ***value**: Informação adicional colocada para pré-calcular valores de expressões simples.
- Node ***pt**: Informação adicional colocada para pré-calcular valores de expressão caso trabalho com ponteiros.
- bool **setted**: Índica se algum valor foi setado para aquele valor

A outra classe é chamda de *SymbolTable*, seu funcionamento baseia-se no uso de um dicionário implementado com o <map> da biblioteca STL do C++. A sua implementação é simples e consiste de apenas um atributo: map<si, Node> **tab**; Onde a chave é uma tupla contendo o nome do identificador (string) e um escopo (int). Desse modo, a busca por um identificador em um determinado escopo pode ser feito com complexidade logarítmica.

Os métodos para adicionar e buscar os nodos na tabela de símbolo são:

Node* **get**(string name, int scope): responsável por buscar um elemento na tabela de símbolos usando o método do *map* chamado find(). Caso seja encontrado, devolve-se a instância daquele nodo, caso contrário, é devolvido NULL.

bool **add**(Node* n): Se um nodo *n* já estiver na tabela de símbolos nada é feito e retorna-se *false,* indicando que não foi feita a adição do nodo na tabela. Caso contrário, o nodo é adicionado na tabela de símbolos e é retornado *true.*

Além desses métodos responsáveis por adicionar e buscar elementos na tabela de símbolos, foi criado um destrutor virtual para a classe. Desse modo, quando é liberada a memória da instância da classe o C++ se encarrega de liberar a memória de todos os seus atributos recursivamente, portanto todo o nodo dentro do atributo *tab* é liberado.

Por fim, para imprimir as inforamações na tela foi criada um método chamado print que imprime informações correspondente ao nome, tipo, dimensão, escopo, valor e linha de declaração de cada identificador colocado na tabela de símbolos. Seu funcionamento é simples e auto-explicativo.

6 – Helpers

As funções *helpers* foram criados com o objetivos de simplificar a implementação do compilador no arquivo do *yacc.* Todas os *helpers* estão no arquivos *helpers.cpp* com documentação sobre o que cada um faz. Abaixo está um lista com a descrição de cada função.

void **assignOperation**(Node *a, T v1, T v2, string op)
Realiza a atribuição para o nodo *a* entre dois operandos (*v1* e *v2*) com tipo genérico *T* para uma operação *op.*

void **assignValue**(Node *a, Node *b, Node *c, string op)
Realiza a atribuição para o nodo *a* entre dois operandos que são nodos já declarados (*b* e *c*) para uma operação *op.* Essa função identifica erros de tipos no momentos da atribuição. Os erros são identificados pelos tipos e pelas dimensões desses nodos. Caso deseja-se cancelar a questão de calcular os valores para cada expressão, basta então comentar as linhar onde a função *assignOperation* é chamada.

void **appendVector**(vector<Node> *v, vector<Node> *t)
Realiza a concatenação do vetor *v* e do vetor *t.*

void **addListaVars**(vector<Node> *v)
Adiciona os nodos que estiverem dentro do vetor *v* para a tabela de símbolos caso os mesmos ainda não foram declarados, caso contrário, gera um erro de redeclaração de variável.

void **addFunction**(string name, int type)
Adiciona uma função com chamada *name* com tipo *type* para a tabela de símbolos caso os mesmos ainda não foi declarada, caso contrário, gera um erro de redeclaração de função. Além disso, é incrementado a variável global *nscopes* que indica o escopo atual da execução.

void **addParams**(string fname, vector<Node> *v)
Adiciona os parâmetros que estão no vetor *v* para a função *fname* na tabela de símbolos caso eles ainda não tiverem sido declarados. Caso contrário, é gerado um erro semântico de redeclaração de parametro.

void **closeFunction**()
Ao fechar uma função o escopo atual é colocado como sendo o escopo global. Além disso, esse *helper* verifica se a função tem um tipo de retorno, onde caso não tiver é gerado um erro sintático indicando que não há retorno na função sendo analisada.

void **addItem**(vector<Node> *v, Node *x)
Adiciona um nodo *x* no fim de um vetor de nodos *v*.

vector<Node>* **createItemVector**(Node *x)
Cria o vetor de nodos *v* e já adiciona esse nodo *x* nele. É retornado esse vetor *v.*

vector<Node>* **createItemVector**()
Cria e retorna um vetor vazio de nodos.

Node* **getNode**(string name)
Busca um identificador na tabela de símbolos utilizando o escopo atual *nscopes*. Caso não encontre, é buscado no escopo *global*, caso não encontrem nem no *global* então retornado *NULL.*

Node* **getPointerOf**(Node *x)
Busca o ponteiro do nodo *x,* esse *helper* só é utilizado para questões de atribuição de valores.

void **assign**(Node *x, Node *y)
Realiza a atribuição de um nodo *y* para um nodo *x (x = y).* Isso é feito apenas se os tipos e as dimensões de ambos nodos forem iguais. Caso contrário é gerado um erro semântico indicando o erro de incompatibilidade de tipo ou dimensão.

void **assign**(Node *x, unsigned int index, Node *y)
Realiza a atribuição de um nodo *y* para um nodo *x (x[index] = y).* Isso é feito apenas se os tipos e as dimensões de ambos nodos forem compatíveis. Caso contrário é gerado um erro semântico indicando o erro de incompatibilidade de tipo ou dimensão.

void **assignExpression**(Node *x, Node *y)
Realiza a atribuição de uma expressão *y* para uma expressão *y.*

void **assignExpression**(string name, Node *y)
Busca a instância do identificador *name* na tabela de símbolos e realização a atribuição de *y* para esse identificador. Caso a instância do identificador seja nula, significa que aquela variável ainda não foi declarada e é gerado um erro semântico indicando isso.

void **assignExpression**(string name, unsigned int index, Node *y)
Busca a instância do identificador *name* na tabela de símbolos e realização a atribuição de *y* para esse identificador na posição *index* (isso significa que a instância retornada deve ser um nodo com dimensão maior ou igual a 1) caso o valor do índice seja compatível com as dimensões do identificador *name*. Caso essa dimensão seja incompatível, significa que o está tentando acessar uma posição do vetor que vai além dos limites do vetor e então é gerado um erro semântico indicando isso. Assim como o *helper* anterior, essa atribuição é feito apenas se o identificador já foi declarado na tabela de símbolos.

Node* **createItem**(string name, int dim)
Cria a instância de um nodo com nome *name,* dimensão *dim*, tipo como a tupla (variável e tipo atual sendo analisado), escopo indefinido e linha de declaração. Essa instância é então retornada.

Node* **getItem**(string name, string x)
Busca um nodo na tabela de símbolos e o retorna caso não seja NULL, caso contrário gera um erro semântico indicando que o identificador não foi declarado.

Node* **getItem**(string name, unsigned int index, string x)
Busca um nodo na tabela de símbolo e retorna um novo nodo com o valor que corresponde sendo esse nodo na posição *index*. Um erro semântico é gerado caso a dimensão do nodo buscado não seja compátivel com o nodo buscado. Além disso, um erro semântico é gerado caso o retorno da tabela de símbolos seja *NULL,* indicando que o identificador não foi declarado.

Node* **createConstNode**(int type, double value)
Cria um nodo constante de dimensão 1, tipo *type* e valor *value*. Esse nodo é então retornado.

void **verifyReturn**(Node *f, Node *x)
Verifica se o tipo e a dimensão do retorna da função *f* é igual ao da expressão de retorno *x*. Caso seja, a variável global *tiporetorno* é definido como sendo o tipo da expressão de retorno. Caso dê incompatibilidade no tipo ou na dimensão, é gerado um erro semântico indicando essa incompatibilidade.

string **typeToString**(int type)
Função que auxilia no momentar de imprimir a tabela de símbolos, ela recebe um tipo *type* e retorna uma *string* correspondente para aquele tipo, indicando qual tipo é. Essa *string* é então retornada.

string **functionToString**(Node *f)
Concatena a assinatura da função: *tipo nome*. Essa concatenação é feita numa string, que é então retornada.

string **functionToString**(Node *f, vector<Node> *params)
Utiliza a função anterior e aproveita concatena os parâmetros que estão no vetor *params* na string de saída. De modo que fique: *tipo nome* (*tipo1 param1*, *tipo2 param2*, ...).

void **verifyParameters**(Node *f, vector<Node> *v)
Verifica se os tipos dos parâmetros (tipo e dimensão) de uma chamada da função *f* são compatíveis com os tipos dos parâmetros da declaração dessa função. Caso não seja, é gerado um erro semântico indicando o erro.