Projeto de Compilador
E4 para **Análise Semântica**

Prof. Lucas Mello Schnorr
schnorr@inf.ufrgs.br

1 Introdução

A quarta etapa do trabalho de implementação de um compilador para a linguagem consiste em verificações semânticas. Elas fazem parte do sistema de tipos com regras simples. A verificação de tipos é feita em tempo de compilação. Todos os nós AST (E3) terão agora um campo que indica o tipo de dado. O tipo de dado de um determinado nó da AST é definido a partir da natureza do nó ou a partir dos tipos de dados dos nós filhos usando as regras de inferência da linguagem.

2 Funcionalidades Necessárias

2.1 Implementar uma tabela de símbolos

A tabela de símbolos guarda informações a respeito dos símbolos (identificadores e literais) encontrados na entrada. Cada entrada na tabela de símbolos tem uma chave e um conteúdo. A chave única identifica o símbolo, e o conteúdo deve ter os campos:

- natureza (literal, identificador ou função)
- tipo do dado do símbolo (`int` ou `float`)
- argumentos e seus tipos (no caso de funções)
- dados do valor do token pelo `yylval` (veja E3)

A implementação deve prever que várias tabelas podem coexistir ao mesmo tempo, uma para cada escopo. As regras de escopo são delineadas a seguir.

2.2 Verificação de declarações

Todos os identificadores devem ter sido declarados no momento do seu uso, seja como variável, seja como função. Todas as entradas na tabela de símbolos devem ter um tipo associado conforme a declaração, verificando-se se não houve dupla declaração ou se o símbolo não foi declarado. Caso o identificador já tenha sido declarado, deve-se lançar o erro `ERR_DECLARED`. Caso o identificador não tenha sido declarado no seu uso, deve-se lançar o erro `ERR_UNDECLARED`. A verificação de declaração de tipos deve considerar o escopo da linguagem. O escopo pode ser global, local da função e local de um bloco, sendo que este pode ser recursivamente aninhado. Uma forma de se implementar estas regras de escopo é através de uma pilha de tabelas de símbolos. Para verificar se uma variável foi declarada, verifica-se primeiramente no escopo atual (topo da pilha) e enquanto não encontrar, deve-se descer na pilha (sem desempilhar) até chegar no escopo global (base da pilha, sempre presente). Caso o identificador não seja encontrado após este procedimento, temos a evidência que ele não foi declarado e portanto emitimos um erro semântico. Para a declaração de um símbolo, basta inseri-lo na tabela de símbolos do escopo que encontra-se no topo da pilha. O grupo deve identificar na gramática os locais adequados para inserir a criação, empilhamento, desempilhamento e destruição de uma tabela de símbolos. Não há necessidade de manter as tabelas em memória até o final do processo de compilação.

2.3 Uso correto de identificadores

O uso de identificadores deve ser compatível com sua declaração e com seu tipo. Variáveis somente podem ser usadas em expressões e funções apenas devem ser usadas como chamada de função, isto é, seguidas da lista de argumentos possivelmente vazia. Caso o identificador dito variável seja usado como uma função, deve-se lançar o erro `ERR_VARIABLE`. Enfim, caso o identificador dito função seja utilizado como variável, deve-se lançar o erro `ERR_FUNCTION`.

2.4 Verificação de tipos

Uma declaração de variável permite ao compilador definir o seu tipo de dado. Os tipos de dados corretos devem ser inferidos onde forem usados, em expressões (aritméticas, lógicas e relacionais) e comandos. Para simplificar esse procedimento, os nós da AST devem ser anotados com um tipo de dado definido de acordo com as regras de inferência de tipos. Um nó da AST deve ter portanto um novo campo que registra o seu tipo de dado. O tipo de dado do comando de atribuição é determinado pelo tipo de quem recebe o valor atribuído e este deve ser compatível com o tipo do dado sendo atribuído. O tipo de dado do comando `if` (com else opcional) e de um `while` é o tipo de dado da expressão de teste. No comando `if` especificamente, os tipos de dados do bloco do if e do bloco do else devem ser compatíveis (quando este está presente). O tipo de dado de uma chamada de função é o tipo da função sendo chamada. O tipo do comando de retorno da função deve ser compatível com o tipo da função. O tipo de dado de um comando de retorno é o tipo do valor de retorno. O tipo de dado de um comando de inicialização de variável no momento da sua declaração é o tipo da variável. As regras de inferência de tipos da linguagem, determinando também a compatibilidade de tipos, são as seguintes. A partir de `int` e `int`, infere-se `int`. A partir de `float` e `float`, infere-se `float`. A partir de `float` e `int`, ou o inverso, a inferência deve falhar com o erro `ERR_WRONG_TYPE`. Este mesmo erro deve ser utilizado caso as regras acima listadas não forem respeitadas.

2.5 Argumentos e parâmetros de funções

A lista de argumentos fornecidos em uma chamada de função deve ser verificada contra a lista de parâmetros formais na declaração da mesma função. Cada chamada de função deve prover um argumento para cada parâmetro, e ter o seu tipo compatıvel. Tais verificações devem ser realizadas levando-se em conta as informações registradas na tabela de sımbolos, registradas no momento da declaração/definição da função.
Na hora da chamada da função, caso houver um número menor de argumentos que o necessário, deve-se lançar o erro `ERR_MISSING_ARGS`. Caso houver um número maior de argumentos que o necessário, deve-se lançar o erro `ERR_EXCESS_ARGS`. Enfim, quando o número de argumentos é correto, mas os tipos dos argumentos são incompatıveis com os tipos registrados na tabela de sımbolo, deve-se lançar o erro `ERR_WRONG_TYPE_ARGS`.

2.6 Verificação de alocação de memória

Adicione a *flag* `-fsanitize=address` na variável `CFLAGS` do seu `Makefile`. Em seguida, tenha certeza que essa flag é passada para o `gcc` na compilação de todos os arquivos do seu projeto. No final da execução do seu compilador, o mesmo reportará a existência de vazamento de memória (*memory leak*). Garanta que seu compilador não reporte nenhum vazamento de memória, fazendo o gerenciamente adequado de memória.

2.7 Mensagens de erro

Mensagens de erro significativas devem ser fornecidas. Elas devem descrever em linguagem natural o erro semântico, as linhas envolvidas, os identificadores e a natureza destes de uma maneira que o usuário do seu compilador compreenda o erro semântico.

A Códigos de retorno

Os seguintes códigos de retorno devem ser utilizados quando o compilador encontrar erros semânticos. O programa deve chamar `exit` utilizando esses códigos imediamente após a impressão da linha que descreve o erro. Na ausência de qualquer erro, o programa deve retornar o valor zero.

```
#define ERR_UNDECLARED       10 //2.2
#define ERR_DECLARED         11 //2.2
#define ERR_VARIABLE         20 //2.3
#define ERR_FUNCTION         21 //2.3
#define ERR_WRONG_TYPE       30 //2.4
#define ERR_MISSING_ARGS     40 //2.5
#define ERR_EXCESS_ARGS      41 //2.5
#define ERR_WRONG_TYPE_ARGS  42 //2.5
```

Estes valores são utilizados na avaliação objetiva.

B Arquivo `main.c`

Utilize o mesmo `main.c` da E3.

Cuide da alocação dinâmica das tabelas.