UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Compiladores

2020-2021

Projeto 3 – Análise Semântica

Introdução

No 3.º Projeto pretende-se implementar um processo de validação semântica da linguagem ALG, cujo compilador será desenvolvido na cadeira de compiladores ao longo do semestre. A linguagem ALG é baseada na linguagem FIR[1].

Objetivos e requisitos

Partindo do trabalho realizado nos projetos 1 e 2, deverão agora usar a ferramenta ANTLR para fazer a análise semântica da linguagem de acordo com a especificação que se apresenta na secção seguinte. Embora seja recomendada a utilização dos padrões de desenho *listener* e *visitor*, cabe a cada grupo decidir se querem usar ações semânticas e atributos diretamente na gramática ou uma combinação das duas abordagens.

Poderão também, caso o entendam, fazer alterações à gramática submetida no 2.º projeto. No entanto, deverão indicar (em comentário no código) quais as alterações efetuadas com uma breve justificação.

Devem capturar erros semânticos e apresentar ao utilizador do compilador uma informação de erro que seja apropriada e percetível.

Linguagem ALG (especificação da semântica)

Iremos apresentar de seguida as restrições semânticas que se verificam para os vários elementos da linguagem ALG.

Espaço de nomes, declarações e identificadores de variáveis e funções

O espaço de nomes global é único, e um nome (identificador) usado para declarar uma entidade num determinado contexto não pode ser usado para declarar outras entidades no mesmo contexto (mesmo que sejam entidades de natureza diferente[2]).

As variáveis locais a uma função, bem como os seus argumentos[3], existem apenas durante a sua execução. Não é possível definir símbolos globais dentro de uma função[4].

A redefinição de um identificador é possível desde que seja feita dentro de um contexto inferior. Ou seja, uma redeclaração local dentro de uma função pode esconder a declaração global (enquanto se estiver no contexto da função).

Identificadores de variáveis são visíveis desde a sua definição até ao fim do alcance. Isto implica que uma variável tem que estar previamente definida antes de poder ser usada. Uma variável definida dentro de um bloco de uma função é visível nesse bloco e nos blocos seguintes. Por exemplo, uma variável definida

---

[1] A Linguagem FIR foi uma linguagem usada como projeto da cadeira de Compiladores no Instituto Superior Técnico no ano letivo 2020/2021.

[2] Ou seja, após ser declarada uma variável com a forma *int x;* o identificador x não poderá ser usado para declarar outra variável ou função dentro do mesmo contexto.

[3] Os argumentos de uma função podem ser vistos como declarações locais de variáveis

[4] Ou seja, qualquer definição dentro de uma função vai ser sempre uma definição local.

dentro do prólogo é visível no corpo e no epílogo, mas uma variável definida no epílogo não é visível nos anteriores.

Os indentificadores de funções declarados são sempre visíveis independentemente da sua localização. Ou seja, eu posso invocar uma função que só seja declarada mais à frente no ficheiro.

Programa

Um ficheiro deve declarar e definir obrigatoriamente a função principal ***alg*** (irá ser usada para iniciar o programa) com os argumentos e tipo de retorno correto.

Declaração com inicialização

Uma declaração com inicialização declara e inicializa uma única variável, e tem 2 formas: a forma ***tipo identificador = expressão*** e a forma ***tipo identificador* = [*expressão*]**.

A primeira forma tem a semântica equivalente a uma declaração e a uma atribuição, pois na prática corresponde a fazer uma declaração e uma instrução de atribuição ao mesmo tempo.

A segunda forma é usada para reservar memória no *stack* atual para um vetor (*array*) de objetos de um determinado tipo[5]. Isto implica que o tipo só pode ser um tipo ponteiro (não vazio) e a expressão tem que ser uma expressão de tipo inteiro.

Expressões e tipos

O tipo de uma expressão simples que corresponda a um literal é dado pelo tipo do literal. O literal de ponteiros nulo (***null***) é um ponteiro especial, cujo tipo é considerado um ponteiro para vazio, ou seja ***<void>***.

Uma expressão que corresponda a uma invocação de uma função tem como tipo o tipo de retorno da função. Se a função retornar ***void***, então o tipo da expressão é ***void***.

As expressões com operadores devem respeitar as seguintes restrições semânticas:

Nome Operador	*Tokens*	Exemplo Forma	Restrições semânticas sobre tipos
Parênteses	( )	**(***E1***)**	A expressão resultante é do mesmo tipo que E1.
Indexação de ponteiro	[ ]	*E1***[***E2***]**	E1 é qualquer tipo ponteiro (exceto o ponteiro para vazio). E2 é do tipo Inteiro. A expressão resultante é do tipo primitivo do ponteiro E1. Por exemplo, se E1 é do tipo ***<int>***, a expressão resultante será do tipo ***int***.
Identidade, simétrico,	+, -	*-E1*	E1 é do tipo real ou inteiro, a expressão resultante é do mesmo tipo que o operando.
negação,	~	~E1	E1 é do tipo booleano. A expressão resultante é do tipo booleano.
extração de ponteiro	?	?E1	E1 deve ser uma expressão que corresponda ao lado esquerdo de uma atribuição (identificador ou indexação de ponteiro). Para além disto, E tem que ser um tipo primitivo (não pode ser ***void***), e não pode ser um ponteiro[6]. A expressão resultante é do tipo ponteiro para o tipo primitivo de E.
Multiplicação e divisão	*, /	*E1* ***** *E2*	E1 e E2 têm que ser números (inteiros ou reais). Se ambos forem inteiros, o resultado

[5] Por exemplo, ***<float> p = [5]*** reserva memória para um array de 5 números reais.

[6] Uma indexação de ponteiro já garante isto, mas um identificador não nos oferece esta garantia. Se a variável i tiver sido declarada como "***int i***;" podemos fazer "***?i***", mas se tiver sido declarada "***<int> i;***" já não.

			da expressão é também do tipo inteiro. Se algum deles for real, então o resultado da expressão é real.
Divisão Inteira	%	*E1 % E2*	A divisão inteira apenas se aplica a números inteiros (E1 e E2). O resultado é do tipo inteiro.
Soma e subtração	+, -	*E1* **+** *E2*	Existem 2 semânticas alternativas para os operadores + e -. a) operação sobre números Se E1 e E2 forem números (inteiros ou reais), a expressão corresponde a uma operação numérica, e o tipo da expressão resultante segue as mesmas regras que para a multiplicação e divisão. b) deslocamento de ponteiro Se E1 for do tipo ponteiro não vazio, E2 terá que ser do tipo inteiro, e a operação corresponde a um deslocamento de ponteiro. O tipo da expressão será igual ao tipo de E1. Este operador não se pode aplicar se E1 for um ponteiro vazio.
Comparadores	<,>,>=, <=,	*E1 >= E2*	E1 e E2 terão que ser do tipo numérico (inteiros ou reais). O tipo da expressão resultante é booleano.
Igualdade	**==, !=**	*E1 != E2*	Os operadores de igualdade aplicam-se quer a tipos primitivos quer a ponteiros. Caso E1 e E2 sejam tipos primitivos (não vazios) deverão ser do mesmo tipo[7]. Caso sejam ponteiros, ou um deles é o tipo ponteiro para vazio, ou devem ser ambos ponteiros do mesmo tipo primitivo. O tipo da expressão resultante é booleano.
Lógicos	&&, ||	*E1* && *E2*	E1 e E2 deverão ser do tipo booleano. O tipo da expressão resultante é booleano.

Invocação de funções

Quando uma função é invocada, o número de argumentos e o tipo dos parâmetros usados na invocação deve ser igual ao número e tipo dos parâmetros formais da função invocada (poderão existir conversões implícitas[8]), e deve manter a mesma ordem dos argumentos.

As conversões implícitas permitidas são de inteiros passados como argumentos reais, e ponteiros para vazio como argumentos de qualquer tipo de ponteiro.

Corpo de funções, blocos e instrução de retorno

Os diferentes componentes do corpo de uma função (prólogo, bloco principal e epílogo) têm restrições semânticas diferentes. Tal como foi dito anteriormente, variáveis declaradas num bloco são sempre visíveis nos blocos seguintes, mas o contrário não é verdade. Todos os blocos de um método têm acesso aos argumentos declarados desse método.

[7] Embora fosse possível calcular o valor do operador (==, ou !=) quando E1 e E2 são de tipos diferentes, se um programador compara um inteiro com uma *string* é porque se enganou, e vamos avisá-lo.
[8] Uma conversão implícita ocorre quando um tipo é convertido noutro tipo sem um pedido explícito do programador (*cast*).

A utilização da instrução de retorno é opcional para o prólogo e para o epílogo, mas é obrigatória para o bloco principal (uma das instruções do bloco principal tem que ser necessariamente um retorno) quando um método tem um tipo de retorno não vazio. Se o tipo de retorno for da função for ***void***, deixa de ser obrigatório o retorno para o bloco principal.

Quando usada, uma instrução de retorno deve ser consistente com o tipo de retorno especificado para a função. Se o tipo de retorno for ***void***, deve ser usada apenas a instrução de retorno "***return***". Se o tipo de retorno da função for diferente de ***void***, então a instrução de retorno deverá ter obrigatoriamente a forma "***return expressão***" e o tipo da expressão deve ser consistente com o tipo de retorno declarado para a função (considerando se necessário conversões implícitas).

Caso exista uma instrução de retorno diretamente dentro de um bloco esta deverá ser a última instrução do bloco onde está inserida.

Caso exista uma instrução de retorno dentro de um subbloco, esta deve ser a última instrução desse subbloco. No entanto, podem existir outras instruções a seguir fora do subbloco.

Instrução de atribuição

Uma instrução de atribuição é usada quando se pretende atribuir um valor a uma zona de memória, e tem a forma ***lado_esquerdo = expressão***.

O lado esquerdo de uma atribuição tem que corresponder a uma zona de memória, e, portanto, pode apenas ser um identificador ou uma indexação de ponteiros.

O lado esquerdo também tem um tipo, tal como as expressões. O tipo do lado esquerdo tem que corresponder ao tipo da expressão para que a atribuição seja válida. No entanto existem alguns casos onde pode ocorrer conversão implícita:

- É válido atribuir uma expressão de tipo inteiro a um lado esquerdo de tipo real. O contrário não é possível (atribuir um real a um inteiro).
- Se o tipo do lado esquerdo for um ponteiro (de qualquer tipo), podemos atribuir-lhe uma expressão correspondente a um tipo ponteiro vazio. Ex: ***<int> x; x = null;***

A expressão, ou lado direito da atribuição, pode ser qualquer expressão desde que não seja do tipo ***void***. Isto quer dizer que eu não posso fazer uma atribuição usando do lado direito uma função que não retorna nada. Por exemplo, a instrução "***x=f(5);***" deverá gerar um erro semântico quando a função f está declarada como retornando ***void***.

Instrução de terminação e reinício

As instruções ***leave*** e ***restart*** só podem ser usadas dentro de um subbloco de um ciclo, e caso sejam usadas são a última instrução do seu subbloco.

Instruções de impressão

As funções de impressão ***write*** e ***writeln***, estão apenas definidos para receber como argumentos expressões que correspondam a tipos primitivos, não sendo possível a impressão de expressões do tipo ponteiro.

Condições de realização

O projeto deve ser realizado em grupo, de acordo com as inscrições em grupo do laboratório. Projetos iguais, ou muito semelhantes, originarão a reprovação na disciplina. O corpo docente da disciplina será o único juiz do que se considera ou não copiar num projeto.

O 3.º projeto vale 30% da nota final da componente prática. O Corpo Docente está ainda a estudar a possibilidade de avaliar esta parte do projeto de forma automática usando o Mooshak. Informações

adicionais sobre o processo de submissão serão disponibilizadas em breve. O projeto deverá ser entregue até às **23:59** do dia **21/05/2021**.