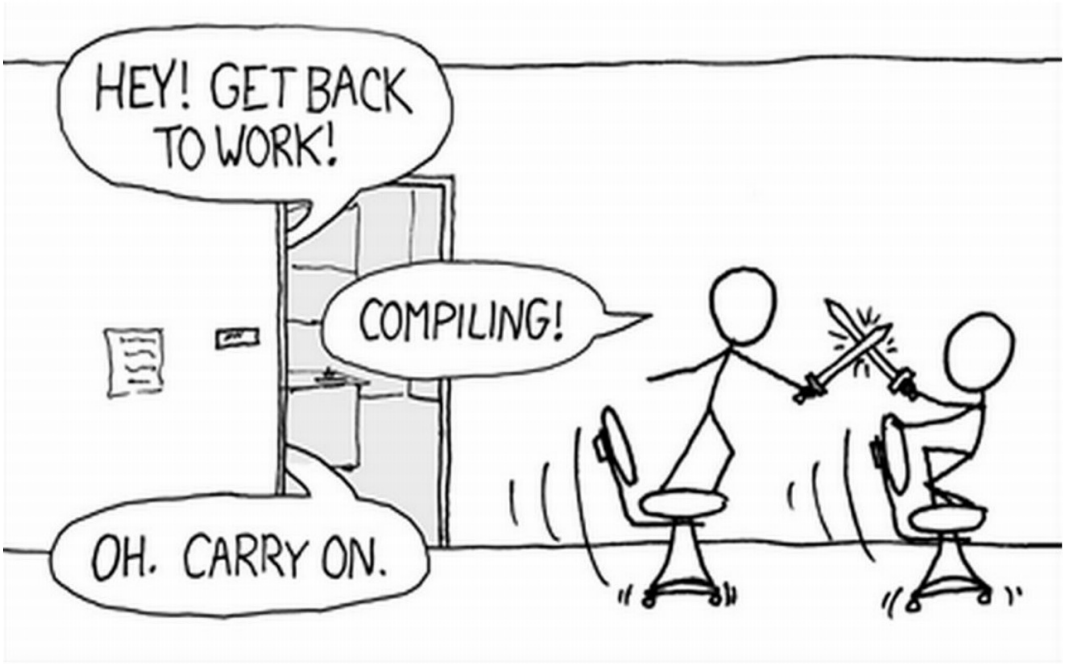
Small Pascal Compiler

João Ferreira & Milton Sêco
Departamento de Engenharia Informática
Universidade de Coimbra
jpbat@student.dei.uc.pt | mseco@student.dei.uc.pt
2009113274 | 2009116156

Junho 2012



Conteúdo

1 Introdução

No âmbito da cadeira de Compiladores, pertencente ao 2º semestre do 3º ano da Licenciatura em Engenharia Informática, foi-nos proposta a implementação de um compilador de um *subset* da linguagem `Pascal` (apelidado de `Small Pascal`) para um `subset` de C. Esse *subset* tem como limitações apenas possuir saltos incondicionais (`goto`), `labels` e a instrução de selecção `if` (sem o correspondente `else`, além de apenas poder ter uma `statement`). Tal permite uma maior semelhança à compilação para código de três endereços.

Este relatório permite esclarecer algumas das decisões tomadas relativamente à sintaxe e semântica, bem como explicar o mecanismo de geração de código utilizado e referir todos os obstáculos que enfrentámos na execução do projecto.

De referir que o login do grupo no `Mooshak` é JoaoMilton e que foram obtidas as seguintes pontuações nos concursos:

- Meta1 (Análise Sintáctica) → 100/100
- Meta2 (Análise Semântica e Interpretador) → 1000/1000
- Meta3 (Geração de Código) → 94/100

Cada uma das três metas e respectivas abordagens será apresentada de seguida.

2 Análise Sintáctica

Tal como nos foi aconselhado pelos docentes recorreu-se ao `lex` para retirar os tokens do ficheiro recebido através do `stdin` e mais tarde ao `yacc` para, usando a gramática dada no enunciado e alterada por nós, criar o analisador sintáctico para a linguagem.

Como a linguagem é *case insensitive* criou-se uma regra para que qualquer letra fosse igual à sua minúscula ou maiúscula[1]. Foi ainda necessário criar um estado para tratar os comentários. Tal como nos foi pedido no enunciado todo o texto que se encontra dentro de comentários é ignorado, com a excepção do `EOF`, sendo que nesse caso é impresso o erro respectivo[2]. Todo o restante texto é processado, podendo dar aso a um erro de caractér inválido[3].

Alguns dos nossos problemas de `shift/reduce` que existiam foram resolvidos dando diferentes associatividades aos operadores. Encontram-se apresentados na seguinte lista:

- nonassoc OP2
- left OP3 OR
- left OP4 AND
- right NOT
- right IF THEN ELSE
- left LPAREN

De referir por fim que foi necessário tomar algumas decisões em termos de implementação, por questões de incompatibilidade com o `Mooshak`, tais como a criação de uma função `mystrdup`, uma vez que a função `strdup` do `string.h` não era suportada (por não ser `ANSI C`) e a criação de uma variável externa (line), que conta as linhas já que a ferramenta `yyline` do `lex` também não é suportada.

Foram implementadas todas as funcionalidades pedidas no enunciado do projecto.

[1] Ex.: A [aA] (excerto do analizador lexical)
[2] Line d: unexpected EOF inside comment. (sendo d o número da linha)
[3] Line d: illegal character (c). (sendo d o número da linha e c o caractér inválido)

2.1 Especificação da Gramática Utilizada

```
start
        : PROGRAM ID ';' block '.'
        ;

block
        : VAR variableDeclarationList functionDecOrDefList BEGIN statementList END
        | functionDecOrDefList BEGIN statementList END
        ;

variableDeclarationList
        : variableDelaration
        | variableDeclarationList variableDeclaration
        ;

variableDelaration
        : idList ':' ARRAY '[' DIGSEQ '..' DIGSEQ ']' OF ID ';'
        | idList ':' ID ';'
        ;

idList
        : ID
        | idList ',' ID
        ;

functionDecOrDefList
        : functionDecOrDefList functionDeclaration
        | functionDecOrDefList functionDefinition
        ;

functionDeclaration
        : functionHeading ';' FORWARD ';'
        ;

functionDefinition
        :functionHeading ';' functionBlock ';'
        ;

functionHeading
        : FUNCTION ID '(' formalArgsList ')' ':' ID
        | FUNCTION ID '(' ')' ':' ID
        ;

formalArgsList
        : formalArgs
        | formalArgsList ';' formalArgs
        ;

formalArgs
```

```
        : VAR idList ':' ARRAY OF ID
        | VAR idList ':' ID
        | idList ':' ARRAY OF ID
        | idList ':' ID
        ;

functionBlock
        : VAR variableDeclarationList BEGIN statementList END
        | BEGIN statementList END
        ;

statementList
        : statement
        | statementList ';' statement
        ;

statement
        : BEGIN statementList END
        | ID '[' expression ']' ASSIGN expression
        | ID ASSIGN expression
        | IF expression THEN statement ELSE statement
        | IF expression THEN statement
        | WHILE expression DO statement
        | FOR ID ASSIGN expression TO expression DO statement
        | FOR ID ASSIGN expression DOWNTO expression DO statement
        | VAL '(' PARAMSTR '(' expression ')' ':' ID ')'
        | WRITELN '(' expression ')'
        | /* empty */
        ;

expression
        : expression AND expression
        | expression OR expression
        | expression OP2 expression
        | expression OP3 expression
        | expression OP4 expression
        | OP3 expression
        | '(' expression ')'
        | DIGSEQ
        | REALNUMBER
        | ID '[' expression ']'
        | ID
        | NOT expression
        | ID '(' expressionList ')'
        | ID '(' ')'
        ;
```

```
expressionList
        : expression
        | expressionList ',' expression
        ;
```

2.2 Árvore Abstracta de Sintaxe

As estruturas de dados usadas para construir a AST foram as mesmas que haviam sido utilizadas para construir a árvore pedida na primeira meta (árvore de derivação). Assim optou-se por ao invés de se adaptar todas as estuturas e tudo o que disso decorria (funções de inserção e shows), por alterar apenas os shows para que a nossa árvore, quando impressa fosse equivalente à AST.

Em relação ao código utilizado na primeira meta foram alteradas algumas regras de gramática. A título de exemplo na primeira meta dois `empty statement` faziam uma `statement list`, enquanto que na segunda considerava-se que a `statement list` estava vazia.

Tal como pedido no enunciado será impressa a árvore abstracta de sintaxe, sempre que o executável for chamado com a flag -t, sendo que o programa termina depois disso.

3 Análise Semântica

A análise semântica encontra-se dividida em duas partes fulcrais: procura e detecção de erros, e a criação da tabela de símbolos. Cada uma destas será detalhadamente explicada de seguida.

Esta parte do compilador é sempre executada (sem necessidade de recorrer a flags), uma vez que após a análise semântica o código pode vir a ser interpretado, ou compilado (escrito para código de três endereços), e aí é necessário garantir-se a não exitência de erros.

Para fazer a análise semântica percorreu-se a árvore abstracta de sintaxe.

3.1 Tabela de Símbolos

De forma a fazer uma correcta análise semântica, é necessária a criação de tabelas de símbolos e ambientes que suportem a informação, para que quando esta seja utilizada seja também possível verificar se um determinado conjunto de variáveis ou métodos foram declarados correctamente por um ambiente que lhe seja acessível.

As estruturas usadas para preencher a tabela são semelhantes às que nos foram fornecidas na ficha 7, para que caso existissem dúvidas estas pudessem mais facilmente resolvidas, assim como quaisquer erros de raciocínio.

A tabela é impressa apenas caso de na chamada do executável com a flag -s. Existem alguns elementos que são sempre adicionados à tabela:

- integer;
- real;
- boolean;
- true;
- false;

É ainda adicionado também sempre a tabela um inteiro (paramcount).

São então inseridas na tabela as variáveis globais caso existam, e de seguida as funções (aquando da sua declaração). Por último são criadas novas tabelas para as funções declaradas, e são inseridas na tabela correspondente todas as variáveis locais.

3.2 Detecção de Erros

Não existe muito que possa ser dito sobre esta parte de um compilador. Não é mais que percorrer toda a árvore verificando se uma série de regras estão a ser cumpridas. Para que isto seja possível, antes de esta análise começar são inseridas na tabela de símbolos todas as funções (ainda que apenas exista a sua declaração), assim como todas as variáveis.

A função `main()` do `YACC` chama a função `check_start()` para começar a fazer a verificação de erros, logo após fazer a verificação da sintaxe do código fonte (`yyparse()`).

Para fazer a análise semântica optou-se por fazer um analisador que tem em conta uma série de regras indicadas pelos professores. São elas:

- O erro apresentado deve ser mostrado à medida que é detectado;
- No caso de existir mais do que um erro na mesma linha, deve-se considerar que esta é analizada da esqerda para a direita.
- Caso seja detectado um erro o compilador deve terminar imediatamente a sua execução.

Assim como outras dicas que nos foram dadas pelos monitores ao longo do semestre.

4 Interpretador

Nesta última parte da 2ª meta, foi criada uma estrutura (`_value`) que possui uma `union` que nos indica qual o tipo de dados guardado e qual o seu valor.

Toda a interpretação foi feita de um modo recursivo, para que todas as expressões fossem resolvidas e se soubesse qual o valor de retorno que uma determinada expressão iria dar. A título de exemplo, no código:

```
program report;
var a : integer;
begin;
    a := 1 + 5 * 4 / 2;
    writeln(a);
end.
```

seria calculado o valor de retorno de 5 * 4, de seguida o valor de 20 / 2 e por fim o valor de 1 + 10. Assim a teria o valor de 11, ou seja o valor de toda a expressão e este seria impresso.

5 Geração de Código

Feita a análise semântica e excluída a existência de erros, procede-se então à tradução do código fornecido para código C de três endereços.

Relativamente às expressões, para estas serem traduzidas, todos os valores são colocados em variáveis temporárias para posteriormente serem realizadas as operações sobre estas. De forma a saber o tipo das variáveis temporárias, recorreu-se ao tipo guardado nas estruturas. As variáveis temporárias vão sendo sempre criadas à medida que são necessárias. É usada a função `getTemp()`, para traduzir o typecast para a váriável temporária correta.

Para a tradução de funções, if's, else's e eventuais breaks, recorreu-se ao uso de goto e de labels. De forma a garantir a distinção de labels e o correcto fluxo durante a execução, recorreu-se ao uso de contadores para identificar a label em que o código se encontra, e para, em caso de break ou término do ciclo, saber para onde voltar.

A nível de funções os argumentos foram passados usando o outgoing da função que chama, e o valor de retorno é devolvido usando o campo returnValue.

Apenas nesta parte do compilador não conseguimos ter nota máxima, uma vez que com a submissão do Mooshak que nos valeu os 94% pontos resolvemos todos os testes a que submetemos o nosso compilador. Dentro de eles encontram-se problemas como o algoritmo `big mod`, e implementações recursivas do gerador da sequência de `fibonacci`, e testes para a passagem de parâmetros por referência e por valor.

6 Conclusão

Hoje em dia, a grande maioria dos IDE usados, fornecem informação sintáctica e semântica em tempo real, dando a possibilidade ao programador, de reparar possíveis erros que estejam a ser cometidos. Concluída a implementação deste projecto, é então possível perceber como estes erros são detectados, o porquê de muitas restrições das linguagens de programação e das mensagens de erro durante a compilação.

Este projecto possibilitou então assimilar os conceitos teóricos fornecidos na aula de uma forma clara mas de algum modo trabalhosa. Para o continuar da cadeira, deixamos aqui a nossa opinião para que esta possa melhorar de ano para ano. Embora este projecto seja de facto interessante, o excessivo trabalho repetitivo é algo que poderia dar lugar por exemplo para à optimização de código. Além disto pensamos que existe necessidade de não haver ambiguidade nos enunciados, como se verificou na análise semântica (segunda meta) deste projecto, onde não se tornava óbvio para os alunos qual o erro que estes deveriam imprimir em determinadas situações.